EUROPEJSKA UCZELNIA W WARSZAWIE

MAREK J. GRENIEWSKI

INFORMATYKA A LOGIKA FORMALNA

WRAZ Z PODSTAWAMI WALIDACJI PROGRAMÓW KOMPUTEROWYCH

WARSZAWA, 2015

Informatyka a logika formalna

– wraz z podstawami walidacji programów komputerowych

Motto:

„*Tests can't assure the absence of bugs* – but formal *proofs can certify the absence of bugs*."

Przedmowa

Wielu logików zajmujących się również podstawami informatyki twierdzi, że „*logika formalna ma niemniejsze lub wręcz większe znaczenie dla informatyki, niż analiza matematyczna dla fizyki*". Należy zauważyć, że logika formalna nie opisuje rzeczywistych rozwiązań wykorzystywanych przez informatykę, ale ich uproszczone modele, które bez zastrzeżeń można jednak uważać za abstrakcyjne twory interesujące dla rozważań dotyczących także informatyki. Mimo tego ograniczenia, logika formalna może dostarczać wiele niezwykle ważnych wniosków - związanych z informatyką. I tak, logika formalna ma dziś szeroki wpływ: np. na budowę składni języków, weryfikację zgodności wymagań na system aplikacyjny z zbudowanymi na podstawie tych wymagań programami komputerowymi, walidację poprawności programów. Innym przykładem są takie obszary jak: teoria złożoności obliczeniowej, teoria automatów skończonych, programowanie obiektowe i tzw. programowanie w logice (np. język PROLOG) — to tylko przykłady działów informatyki, w których metody logiki formalnej stały się standardowym narzędziem informatyki badawczej jak i praktyków.

W ostatnich jednak latach, w pewnym sensie miało miejsce również zjawisko odwrotne. Wniosek wypływający z praktyki informatycznej - ma istotne znaczenie dla klasycznej logiki. A bardziej konkretnie, na początku XXI wieku pojawił się nowy dział informatyki, nazwany Big Data (a dział ten dotyczy wielkich zbiorowisk danych, z reguły słabo uporządkowanych, jak również praw nimi rządzących). Wielkie firmy informatyczne, operujące zbiorowiskami danych, np. takich jak zbiór logowań do przeglądarki Google'a, mogą wyciągać wnioski wynikające z zależności korelacyjnych - odkrywanych w tych danych. Przykładowo, w roku 2008 informatycy firmy Google stwierdzili, że istnieje 45 rodzajów zapytań kierowanych do przeglądarki tej firmy, które towarzyszą zachorowaniom na grypę. Zapytania te, nie mają żadnego bezpośredniego związku z grypą lub objawami grypy. W roku 2009 w czasie zagrożenia pandemią grypy, ta korelacja (ze względu na niemal natychmiastowe tworzenie statystyki takich logowań przez firmę Google) pozwoliła służbom nadzoru sanitarnego USA, uzyskiwać na bieżąco informację o liczbie i miejscach pojawiania się nowych chorych na grypę. Warto podkreślić, że spływ danych o zachorowalności na grypę, zbieranych z placówek medycznych, wymagał blisko tygodnia. Takie związki w Big Data, jak przedstawiona zależność, nie są związkami przyczynowo skutkowym, ale korelacją, o której nie wiemy, dlaczego występuje. Czyli jest to dokładnie odpowiednik implikacji, tak jak ją rozumieją logicy od tysięcy lat. Można uznać, że Big Data potwierdza słuszność przyjętej definicji funkcji implikacji w klasycznej logice formalnej.

W okresie ostatnich sześćdziesięciu lat nastąpił bardzo gwałtowny rozwój informatyki i jej zastosowań. Kluczowym problemem stała się, jakość oprogramowania komputerów. Komputery są urządzeniami obecnymi niemal w każdej dziedzinie naszego życia. Komputery sterują: maszynami oraz urządzeniami produkcyjnymi, systemami diagnostyki medycznej; zarządzają cyfrowymi systemami: telekomunikacji, bankowości i ubezpieczeń, itd. Trudno wyobrazić sobie współczesny świat bez informatyki. Ale dzisiaj następuje wymuszony - nieusuniętymi błędami oprogramowania, powrót do podstaw, czyli do stosowania metod formalnych. Mam tu na myśli – zmiany w podejściu do oprogramowania komputerów, a dokładniej mówiąc wymagań -

zapewnienia poprawności działania programów komputerowych. Można z pewną nonszalancją przyjąć, że w okresie pierwszych ponad pięćdziesięciu lat rozwoju informatyki, tworzenie oprogramowania było w znacznej mierze dziedziną sztuki a nie nauki.

Nie oznacza to, że logika formalna nie miała swojego istotnego wkładu w rozwój metod oraz technik tworzenia oprogramowania komputerów. Należy jednak zauważyć, że stosowanie podejścia formalnego, czyli podejścia bazującego na logice formalnej, dzisiaj ciągle w małym stopniu dotyczyło praktyki projektowania i oprogramowywania systemów informatycznych, a w szczególności systemów aplikacyjnych. Było w tym zakresie kilka chlubnych wyjątków, ale dotyczących raczej nauki niż praktyki, np. przypadek, które zawdzięczamy stosunkowo nielicznej grupie uczonych z Programming Research Group – Oxford University, polegającym na zastosowaniu w latach osiemdziesiątych ubiegłego wieku - NOTACJI Z (zresztą autorstwa tegoż zespołu – będącą językiem komputerowej realizacji języka tzw. logiki pierwszego rzędu), do napisania nowych wymagań na system CICS/ESA VERSION 3 – wprowadzonego na rynek w czerwcu 1989 (CICS - Customer Information & Control System) opracowywanie IBM UK; czy np. również interesujący przypadek (oparty o wyniki prac z lat osiemdziesiątych ubiegłego wieku, uzyskane przez Clarke'a, Emerson'a i Sifakis'a) - podjęcia prac badawczych przez zespół Gerarda J. Holzmanna z Computer Sciences Research Center at Bell Laboratories, kontynuowanych w Laboratory for Reliable Software at NASA's Jet Propulsion Laboratory / California Institute of Technology - dotyczących opracowania praktycznych metody sprawdzania poprawności zachowań programu - we wszystkich osiągalnych przez program stanach, metody zapewniającej w wyniku poprawność działania całości reaktywnego współbieżnego oprogramowania systemu informatycznego w środowisku rozproszonym.

W tym okresie gwałtownego rozwoju zastosowań informatyki, powszechna praktyka dotycząca walidowania oprogramowania, mająca na celu zapewnienia poprawności działania programów i zbudowanych z nich systemów komputerowych - bazowała na testowaniu oprogramowania (*Software UnitTest*). Testowanie programów, w wielu przypadkach okazało się nieskuteczne, ponieważ w swoim założeniu ograniczało się do sprawdzenia pewnej liczby założonych (przez projektantów) scenariuszy - działania opracowywanego oprogramowania w przypadkach zmienności warunków (przyjętych w scenariuszu testów). Ze względu nas złożoność funkcjonalności współczesnego oprogramowania i działania nie na pojedynczych komputerach, ale w rozbudowanych sieciach komputerowych, często rzeczywiste scenariusze użytkowania systemu informatycznego, (czyli oprogramowania składającego się na dany system) – wykracza daleko poza wyobraźnię projektantów systemów informatycznych. W tej sytuacji - jedynym, co może pozwolić na zapewnienie pełnej weryfikacji, jest opracowanie formalnego dowodu poprawnego działania poszczególnych programów składowych, jak również oprogramowania systemu informatycznego, jako całości. Jako wynik potrzeb praktyki oraz uzyskanych wyników prac badawczych, formalne metody weryfikacji oprogramowania stały się jedną z najważniejszych zastosowań logiki formalnej w informatyce.

Logika działania systemów to jedno, a kartezjańska tradycja widzenia działania systemu, jako całości - jedynie przez pryzmat działania części składowych systemu (w myśl błędnej zasady: „*całość jest sumą części*") to drugie, i to ma często wpływ na brak zrozumienia złożoności efektów współbieżności współpracujących modułów systemu. Ten brak wiedzy o zachowaniu złożonych systemów, doprowadził już wielokrotnie do tragicznych skutków. A oto wiele mówiący przykład.

14 września 1993 roku na lotnisku im. Fryderyka Chopina w Warszawie, doszło do katastrofy[1] *w czasie lądowania samolotu pasażerskiego linii lotniczej Lufthansa Airbus A300-200 z 72 osobami na pokładzie. Był to w tym czasie chyba, najnowocześniejszy samolot pasażerski, pierwszy, w którym niebyło mechanicznego połączenia wolantu z układem sterowania. Samolot pilotowany przez doświadczonego pilota i instruktora pilotażu, lądował w trudnych warunkach pogodowych, przy ulewnym deszczu i silnych podmuchach wiatru wiejącego z tyłu lądującego samolotu. Koła samolotu nie dawały dostatecznego tarcia z nawierzchnią pasa startowego, żeby wyhamować, ale pilot był przekonany, że odwrócony ciąg z głównych silników będzie w stanie zatrzymać samolot. Ale okazało się, że wiara w uruchomienie odwróconego ciągu nie potwierdziła się w praktyce. Samolot był w konfiguracji do lądowania, ale nawierzchnia pasa pokryta była warstwą wody deszczowej, a porywy wiatru z tyłu – powodowały, że koła podwozia nie obracały się, ale ślizgały się po warstewce wody. Większość samolotów posiada zabezpieczenia uniemożliwiające uruchomianie odwróconego ciągu w czasie lotu. Ponieważ koła nie toczyły się, program komputera pokładowego samolotu Airbus A300-200, mimo konfiguracji samolotu do lądowania, nie pozwalał na uruchomienie odwróconego ciągu przez dziewięć sekund od momentu pierwszego kontaktu kół podwozia samolotu z podłożem nawierzchni pasa, w konsekwencji samolot uderzył w wał ziemny ograniczający pas startowy, a dwie osoby zginęły. W scenariuszach testowania systemu - zabrakło sytuacji, która wystąpiła w rzeczywistych warunkach.*

Nie darmo nieżyjący już twórca podstaw teoretycznych informatyki E. Dijkstra mawiał: „*Testowanie dowodzi jedynie istnienia błędów w programie, ale nie dowodzi braku błędów*."

Literatura przedmiotu dostępna w języku polskim, nie zawiera dotychczas pozycji dotyczących praktycznego zastosowania metod formalnych, w tym - dowodzenia poprawności programów (czyli tzw. *walidacji*). Celem tej książki, jest między innymi, pokazanie przykładowych – narzędzi walidacji programów (tzw. *Logic Model Checking*) na przykładzie szeroko stosowanego, opracowanego przez zespół *Gerarda J. Holzmanna*, ogólno dostępnego języka PROMELA (*PROCES META-LANGUAGE*) oraz procesora - czekera SPIN (*SIMPLE PROMELA INTERPRETER*). Żeby jednak, móc w świadomy sposób stosować tą nową technologię - zapewnienia poprawności działania oprogramowanie, trzeba opanować podstawy logiki formalnej stosowanej w informatyce.

Na przełomie lat pięćdziesiątych i sześćdziesiątych ubiegłego stulecia prowadziłem długie dyskusje z moim ojcem *Henrykiem Greniewskim*, na temat granic możliwości komputerów, dokładniej mówiąc komputerów cyfrowych opartych o koncepcję *Johna von Neumana*. Obydwaj znaliśmy zarówno konsekwencje twierdzenia *Kurta Gödla* - o nierozstrzygalności w złożonych systemach logicznych, jak również ówczesne podstawy teorii algorytmów, wynikające zarówno z teorii Maszyny Turinga, jak również z rachunku lambda *Alonso Churcha*, z których to wynikało, że warunkiem koniecznym dla możliwości zastosowania komputera do rozwiązywania jakiegoś problemu, jest istnienie algorytmu. Za to sam fakt nie istnienia algorytmu, może mieć wiele różnych przyczyn (jest np. konsekwencją nierozstrzygalności danej teorii).

Niewątpliwie istnienie algorytmu jest warunkiem koniecznym, aby dany problem dał się rozwiązać z pomocą komputera. Powstawało jednak pytanie, które algorytmy dają się zrealizować z pomocą programów komputerowych, czyli jaki jest warunek wystarczający. Na powyższe pytanie - nie można było wówczas odpowiedzieć. Dzisiaj, dzięki rozwojowi teorii algorytmów, jesteśmy nieco bliżsi odpowiedzi. Nie oznacza to jednak, że potrafimy już

[1] Szczegóły katastrofy są opisane na stronie www.crashdatabase.com

precyzyjnie udzielić odpowiedzi na tak sformułowane pytanie. Bliższe wyjaśnienie tej sprawy, znajdzie czytelnik w części trzeciej książki - zatytułowanej „*Formalne podstawy informatyki*".

W 1999 roku, po przejściu na emeryturę, wróciłem do zawodu nauczyciela akademickiego, a ściślej mówiąc wykładowcy przedmiotów informatycznych. Największym zaskoczeniem dla mnie był fakt, że zasadnicze rozwiązania logiczne dotyczące nauczania informatyki (podkreślam - logiczne), właściwie nie uległy większym zmianom, czyli trudno mówić o istotnym postępie, w stosunku do stanu osiągniętego w późnych latach siedemdziesiątych dwudziestego wieku. Zmiany, jakie zaszły i znalazły swoje miejsce w dydaktyce informatyki, po pierwsze dotyczą głównie technologii obwodów scalonych, o której dzisiaj możemy powiedzieć, jako o bardzo wielkiej skali integracji; a po drugie dotyczą rozwinięcia technologii cyfrowej w łączności przewodowej i bezprzewodowej. Zmiany te, doprowadziły łącznie do powstania miniaturowych komputerów o mocy obliczeniowej, – co najmniej porównywalnej z komputerami średniej mocy z lat siedemdziesiątych ubiegłego wieku, jak również szeroko pasmowych łączy światłowodowych i bezprzewodowych – podstawy wydajnych rozległych sieci komputerowych. Obok postępu technologicznego, nastąpił również zasadniczy rozwój teorii algorytmów, – co też znalazło swoje miejsce w dydaktyce informatyki, łącznie z wydzieleniem się nowej specjalności nazwanej geometrią obliczeniową. Nie można również zapominać, o niesłychanych – rewolucyjnych zmianach, dotyczących zakresu aplikacji komputerów oraz powszechności ich stosowania, o czym była już mowa wcześniej.

Po kilkunastu latach wznowionej działalności dydaktycznej, prowadzonej w kolejnych trzech prywatnych szkołach wyższych, doszedłem do wniosku, że jest to ostatni moment, w którym mogę uporządkować wiedzę na temat wpływu logiki formalnej na informatykę, i to takiej logiki, jakiej nauczano w latach czterdziestych i pięćdziesiątych ubiegłego wieku. Bo to ta właśnie logika leży u podstaw współczesnej informatyki, a szczególności u podstaw metod dowodzenia poprawności oprogramowania, – czyli jego walidacji. Jesienią 2011 roku podjąłem wysiłek napisania niniejszej książki. Użyłem tu terminu „napisania książki", ale nie chcę żeby z tego terminu wynikało, że uważam się za jedynego autora niniejszej książki. W rzeczywistości, w odniesienia do większości tekstu jestem jedynie redaktorem, który skompilował fragmenty tekstów wielu autorów, kompletując z nich książkę, sam jedynie modyfikując lub aktualizując poszczególne wybrane fragmenty i pisząc łączniki – spinające w miarę sensownie fragmenty w finalny tekst niniejszej książki. I tak na przykład autorami fragmentów poświęconych logice są: *Henryk Greniewski*, *Leszek Pacholski*, trójka logików z UW (*Jerzy Turin*, *Jerzy Tyszkiewicz* i *Paweł Urzyn*) oraz *Mordechaj Ben-Ari*, a istotny wpływ na formę prezentacji tej tematyki, miał niewątpliwie mój mistrz z czasów studiów na UW, *Andrzej Mostowski*. Podobnie autorem tekstu dotyczącego Maszyny Turinga i teorii obliczeń, jest przede wszystkim *Michael Sipster*. Autorem opisu notacji Z jest *J. Mike Spivey*, *Jim Woodcock* i *Jim Davies*. Autorami tekstu o dyskretnej walidacji programów reaktywnych – współbieżnych są: *Edmund Clarke*, *Allen Emerson*, *Joseph Sifakis* oraz *Gerald Holzmann* z zespołem swoich współpracowników, *Mordechaj Ben-Ari* i *Amir Pnueli*. Oczywiści wymieniłem przykładowo najważniejszych autorów, zarówno ze względu na brak miejsca, jak i zawodną pamięć.

Niewątpliwym ułatwieniem w zbieraniu materiałów do niniejszej książki, było korzystanie z Internetu (np. przeczytałem w oryginale najważniejszą publikację *Amira Pnueli*, przełomową dla zastosowania logiki temporalnej w walidacji oprogramowania), a w szczególności z Wikipedii, – co daje możliwość zarówno szybkiego dostępu do źródeł, jak i sprawdzenie wielu szczegółów historycznych.

Przyjąłem, że współczesna logika formalna, która uległa istotnemu rozwojowi w drugiej połowie dwudziestego wieku, częściowo w wyniku pojawienia się informatyki jak np. jak np. logika temporalna (rozwinięcie logiki modalnej) wprowadzona przez *A. N. Priora* w 1957 roku, częściowo wynikająca z pomysłów dotyczących rozważań nad gramatykami języków naturalnych czy zdaniowa logika dynamiczna - zaproponowana przez *V. Pratta* w 1976 roku – będąca konsekwencją metodyk programowania komputerów.

Jako wykład podstaw logiki formalnej, wybrałem zapomniany współcześnie podręcznik logiki, zatytułowany „*Elementy Logiki Formalnej*", napisany przez mojego ojca Henryka Greniewskiego i wydany w 1955 roku, przez Państwowe Wydawnictwo Naukowe. Ze względu na rozwój logiki, jaki miał miejsce od połowy ubiegłego stulecia, musiałem dokonać daleko idącej modernizacji tekstu wykorzystanych dalej fragmentów tego podręcznika. Tak naprawdę, podręcznik ten został napisany w latach 1951 – 1952 po raz drugi (rękopis pierwszej wersji książki pisanej w czasie okupacji niemieckiej spłonął w Powstaniu Warszawskim). Warto powiedzieć, że po ukończeniu pisania - blisko dwa lata jeszcze trwało opiniowanie napisanego podręcznika wraz z sformułowaniem kilkunastu zaleceń dotyczących usunięcia wątków, uznanych przez redaktora naukowego z ramienia PWN oraz cenzurę, za ideologicznie niepoprawne lub wręcz wrogie socjalizmowi w wydaniu sowieckim. Wręcz zabawne były zalecenia dotyczące sposobu wykładu logiki formalnej - wynikające z rzekomo naukowych publikacji: *Karola Marksa*, *Fryderyka Engelsa*, *Włodzimierza Lenina* i *Józefa Stalina*, a w szczególności tych dwu ostatnich.

W maju 1954 roku, wspólnie z ówczesnym mgr inż. *Zdzisławem Pawlakiem* (nieżyjącym już dzisiaj, późniejszym bardzo wybitnym polskim uczonym, członkiem rzeczywistym PAN, wówczas doktorantem *Henryka Greniewskiego*), cięliśmy maszynopis podręcznika i wklejaliśmy uzupełnienia, napisane w odpowiedzi na zalecenia - dostarczone przez redaktora naukowego. Bez wprowadzenia tych zmian, publikacja książki przez PWN – byłaby niemożliwa. Ostatecznie maszynopis książki, został przekazany do składu w czerwcu 1954 roku.

Prace nad niniejszą monografią, rozpocząłem od zeskanowania tekstu „*Elementów Logiki Formalnej*", usunięcia z tegoż tekstu wszystkich uzupełnień wprowadzonych przez ówczesnego redaktora naukowego wydawnictwa. Niestety pierwotnych tekstów niektórych usuniętych fragmentów, nie mogłem przywrócić, bo się nie zachowały. Usunąłem również zadania, które były niezbędnym elementem podręcznika, ale które raczej nie pasowały do monografii. Kolejnym krokiem, była modernizacja symboliki logicznej, użytej w zeskanowanym tekście. Symbolika używana w publikacjach amerykańskich, która dzisiaj stała się praktycznie międzynarodową symboliką, istotnie różni się od symboliki używanej w Polsce w latach pięćdziesiątych. Ostatecznie, zdecydowałem się na skorzystanie z symboliki używanej przez *Alfreda Tarskiego* (polsko – amerykańskiego wybitnego logika) - uzupełnionej konwencjami, które zaczerpnąłem z języka XML.

Z oryginalnego tekstu „*Elementów Logiki Forma*lnej"- nie wziąłem między innymi pod uwagę wykorzystania: (1) części piątej, zatytułowaną „*Zarys historii logiki*", – ponieważ tematycznie nie pasował mi do planowanej książki, oraz (2) ostatni z rozdziałów z części drugiej, zatytułowany „*2.3. Zastosowanie logiki zdań do sieci elektrycznych*", – ponieważ całkowicie się zdezaktualizował i miałby tylko znaczenie historyczne. Ostatecznie wykorzystałem mniej niż jedną trzecią tekstu, czterech pierwszych części, wzmiankowanej książki autorstwa mojego ojca.

Dokonałem podziału tematyki niniejszej książki, na pięć części:

1. *Część wstępną* napisana jest od podstaw i zatytułowana: „*Komputer – maszyna logiczna*", zawiera dziesięć rozdziałów poświęconych głównie architekturze, organizacji i systemom operacyjnym współczesnych komputerów. W której to części wykorzystałem dwa fragmenty (podrozdziały) z części „5. *Zarys historii logiki - Elementów Logiki Formalnej*", a mianowicie: „5.4.6. *Ramon Lull – prekursor rachunku logicznego*" oraz fragment „5.5.1. *Klasycy literatury pięknej przeciw formalistyce scholastycznej*" – łączą powyższe dwa fragmenty w podrozdział „1.0.1 *Ramon Lull – twórca pierwszej maszyny logicznej*"; pozostałe rozdziały i podrozdziały części 1 - zostały napisane, w oparciu o materiały zaczerpnięte z wielu publikacji.
2. *Cześć druga* jest zatytułowana: „*Co wynika ze współczesnej logiki formalnej?*", zawiera osiem rozdziałów poświęconych głównie różnym działom współczesnej logiki, między innymi zawierających rozdział będącą rozwinięciem wybranych fragmentów części „1*. O językach i teoriach - Elementów Logiki Formalnej*". Rozdział wstępny, zawierający między innymi XIX-wieczne podstawy logiki współczesnej, opracowałem na podstawie części „5. *Zarys historii logiki - Elementów Logiki Formalnej*". W pozostałych rozdziałach wykorzystane zostały fragmenty dotyczące tautologii dwuwartościowego rachunku zdań oraz trójwartościowemu rachunkowi zdań wg Jana Łukasiewicza z „*Elementów Logiki Formalnej*"; pozostałe rozdziały i podrozdziały części 2 - zostały sformułowane w oparciu o materiały z wielu publikacji.
3. *Część trzecia* z kolei, jest zatytułowana: „*Formalne podstawy informatyki*", zawiera dziewięć rozdziałów, z których pięć „3.0. *Rodzaje języków*", „3.1. *O wyrażeniach*", „3.2. *Rodzaje wyrażeń występujących w językach naturalnych*", częściowo rozdział „3.3. *Rodzaje wyrażeń występujących w językach sztucznych i mieszanych*", „3.4. *Definiowanie*", częściowo rozdział „3.6. *Wnioskowanie*", o treści będącej powtórzeniem z aktualizacjami tekstów z „*Elementów Logiki Formalnej*", do której dodałem rozdziały i podrozdziały napisane od podstaw na podstawie dostępnych publikacji. Na treść tej części, bardzo istotny wpływ mają również wyniki prac *Michaela Sipser'a* (zaprezentowane w czterech rozdziałach).
4. *Część czwarta* jest zatytułowana: „*Informatyka przegląd tematyczny*", zawierająca łącznie dziesięć rozdziałów i została napisana od podstaw, w oparciu o materiały publikowane, jak i własne moje doświadczenia.
5. *Część piąta* jest zatytułowana: „*Walidacja oprogramowania*", zawiera tylko sześć rozdziałów i została napisana, w oparciu o materiały publikowane - autorstwa *Gerarda J. Holzmanna* i jego współpracowników oraz *Edmunda Clarke*, *Allena Emerson'a*, *Josepha Sifakis'a, Mordechaja Ben-Ari'ego* i *Amira Pnueli*.

Chciałbym podkreślić, że z braku miejsca nie pokazałem wszystkich związków pomiędzy pojęciami logiki formalnej a informatyką zostawiając te odpowiedniości dociekliwości czytelnika. Wydaje mi się, że ~ 500 stron tekstu formatu A4, to bardzo dużo i brak mi było odwagi, żeby tekst dalej powiększać.

Na zakończenie przedmowy, chciałbym podziękować: *dr Jerzemu G. Isajewowi* za szereg istotnych uwag merytorycznych, oraz mojemu wieloletniemu przyjacielowi, współpracownikowi z lat siedemdziesiątych ub. wieku - *Jackowi Moszczyńskiemu* (znanemu w USA jako *Jack Mosh*), za skrupulatne przeczytanie tekstu książki oraz wprowadzenie niezbędnych poprawek.

Warszawa, wrzesień 2015.

Marek J. Greniewski

Spis treści

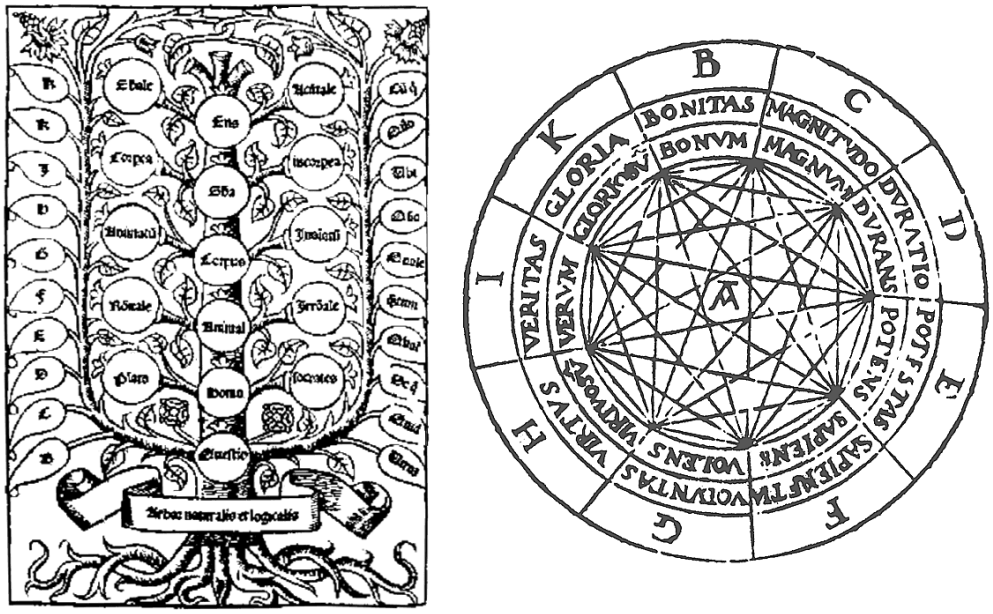
Część 1.
Komputer – Maszyna Logiczna

1.0. Poprzednicy współczesnych komputerów

1.0.1. Ramon Lull – twórca pierwszej maszyny logicznej

Ramon Lull (*Rajmundus Lullus*) urodził się w roku 1232 na wyspie Majorka, zmarł w roku 1316. Kataloński poeta („ojciec literatury katalońskiej"), uczony i misjonarz, długie lata spędził na dworze w służbie Jakuba I Aragońskiego, następnie zajął się niemal wyłącznie sprawą nawracania muzułmanów na chrześcijaństwo, studiując w tym celu język i filozofię arabską.[2]

W roku 1274 zaczyna pisać swoje zasadnicze dzieło logiczne *Ars magna et maxima*. Przedstawiony jest w nim projekt urządzenia (maszyny logicznej), w której podmioty i orzeczenia stwierdzeń teologicznych układały się w okręgi, kwadraty, trójkąty i inne figury geometryczne, a po poruszeniu dźwigni, przekręceniu korby albo obróceniu koła stwierdzenia tworzyły zdania prawdziwe lub fałszywe, same się dowodząc. Projekt ten rozwijany był też w jego kolejnych pracach. Jest to pierwszy jakby przebłysk idei rachunku logicznego. Warto podkreślić, że pomysły *Lulla* wywarły wpływ na *Giordana Bruno* (1548 - 1600) i na *Gottfrieda Wilhelma Leibniza* (1646 - 1716).



Rysunek 1.0.1.00. „Młynek" Lulla

Jak już wspomnieliśmy, swój rachunek logiczny *Lull* wykonywał jednak nie za pomocą napisów na papierze, lecz za pomocą przyrządu mechanicznego, tak zwanego „młynka Lulla" - prostej, naiwnej, lecz pierwszej na świecie maszyny logicznej. Podajemy tu rysunek jednego z jego „młynków". Dalej, objaśnimy użycie „młynka" na przykładzie aparatu prostszego niż tutaj reprodukowany, mianowicie na przykładzie aparatu złożonego z dwu tylko kół współśrodkowych, z których każde jest podzielone na trzy równe łuki.

[2] Piśmiennictwo; na końcu paragrafów zawiera odnośniki do literatury źródłowej poruszanej tematyki.

Przypomnijmy sobie teraz język dwuwartościowego rachunku zdań[3]; wzbogacimy ten język sześcioma zdaniami:

(1) $P_1, P_2, P_3,$

(2) $Q_1, Q_2, Q_3,$

takimi, że

1.0.1.01 $\vdash (P_1 \vee P_2 \vee P_3)$

1.0.1.02 $\vdash (Q_1 \vee Q_2 \vee Q_3)$

1.0.1.03 $\vdash [(\neg P_1 \wedge \neg P_2) \vee (\neg P_2 \wedge \neg P_3) \vee (\neg P_3 \wedge \neg P_1)]$

1.0.1.04 $\vdash [(\neg Q_1 \wedge \neg Q_2) \vee (\neg Q_2 \wedge \neg Q_3) \vee (\neg Q_3 \wedge \neg Q_1)]$

Za pomocą dwuwartościowego rachunku zdań, stosując jako dyrektywy wnioskowania- dyrektywę podstawiamy (każdego z naszych sześciu zdań za zmienną zdaniową) i znaną nam dobrze dyrektywę odrywania, łatwo teraz otrzymać z postulatów 1.0.1.01 i 1.0.1.02 twierdzenie poniższe:

1.0.1.10
$$\vdash [(P_1 \wedge Q_1) \vee (P_1 \wedge Q_2) \vee (P_1 \wedge Q_3) \vee$$
$$\vee (P_2 \wedge Q_1) \vee (P_2 \wedge Q_2) \vee (P_2 \wedge Q_3) \vee (P_3 \wedge Q_1) \vee (P_3 \wedge Q_2) \vee (P_3 \wedge Q_3)]$$

Analogicznie z, postulatów 1.0.1.03 i 1.0.1.04 otrzymujemy twierdzenie:

1.0.1.11
$$\vdash [(\neg P_1 \wedge \neg P_2 \wedge \neg Q_1 \wedge \neg Q_2) \vee (\neg P_2 \wedge \neg P_3 \wedge \neg Q_1 \wedge \neg Q_2) \vee$$
$$\vee (\neg P_3 \wedge \neg P_2 \wedge \neg Q_1 \wedge \neg Q_2) \vee (\neg P_2 \wedge \neg P_3 \wedge \neg Q_2 \vee \neg Q_3) \vee$$
$$\vee (\neg P_2 \wedge \neg P_3 \wedge \neg Q_2 \wedge \neg Q_3) \vee (\neg P_3 \wedge \neg P_2 \wedge \neg Q_2 \wedge \neg Q_3) \vee$$
$$\vee (\neg P_1 \wedge \neg P_2 \wedge \neg Q_3 \wedge \neg Q_1) \vee (\neg P_2 \wedge \neg P_2 \wedge \neg Q_3 \wedge \neg Q'_1) \vee$$
$$\vee (\neg P_3 \wedge \neg P_1 \wedge \neg Q_3 \wedge \neg Q_1)].$$

Wprowadźmy skrót:

$$\bigvee_{\substack{i \neq k \\ j \neq l}} (p_{ij})$$

którego będziemy używali zamiast alternatywy ośmioczłonowej złożonej z wszystkich zdań mających postać: „p_{ij}", gdzie $k = 1, 2, 3$ oraz $l = 1, 2, 3$; oprócz zdania postaci „p_{kl}".

Mamy więc na przykład:

$$\vdash [\bigvee_{i \neq l \; j \neq l} (p_{ij}) \equiv (p_{12} \vee p_{13} \vee p_{21} \vee p_{22} \vee p_{23} \vee p_{31} \vee p_{32} \vee p_{33})].$$

$$\vdash [\bigvee_{i \neq l \; j \neq l} (p_{ij}) \equiv (p_{11} \vee p_{12} \vee p_{13} \vee p_{21} \vee p_{22} \vee p_{31} \vee p_{32} \vee p_{33})].$$

Wprowadźmy skrót:

$$\bigvee_{k,l} (p_{kl})$$

którego będziemy używali zamiast alternatywy dziewięcioczłonowej złożonej ze wszystkich zdań postaci: „p_{ij}" gdzie $i = 1, 2, 3$ oraz $j = 1, 2, 3$; mamy więc:

[3]Patrz rozdział 2.2. Wprowadzenie do klasycznego rachunku zdań.

$$\vdash [(\bigvee p_{kl}) \equiv (p_{11} \vee p_{12} \vee p_{13} \vee p_{21} \vee p_{22} \vee p_{23} \vee p_{31} \vee p_{32} \vee p_{33})].$$

Możemy stosując powyższe skróty udowodnić w oparciu o 1.0.1.10 i 1.0.1.11 twierdzenie:

1.0.1.20 $$\vdash \{\bigvee_{k,l\ i \neq k\ j \neq l} \neg[\bigvee(P_i \wedge Q_j)]\}.$$

Zarówno twierdzenia 1.0.1.10 i 1.0.1.11, jak zwłaszcza twierdzenie 1.0.1.20 są już nieco zawiłe można się pomylić przy ich formułowaniu. Ułatwia nam zadanie nasz uproszczony „młynek". Do każdego z trzech łuków koła zewnętrznego przypiszmy po jednym ze zdań (1), do każdego z trzech łuków koła wewnętrznego przypiszmy po jednym ze zdań (2). Obracając teraz koło wewnętrzne, za każdym razem o 1/3 kąta pełnego, otrzymamy wszystkie dziewięć położeń par łuków odpowiadających dziewięciu członom alternatywy 1.0.1.10; zarazem stanie się dla nas bardziej intuicyjne twierdzenie 1.0.1.11 a nawet twierdzenie 1.0.1.20.

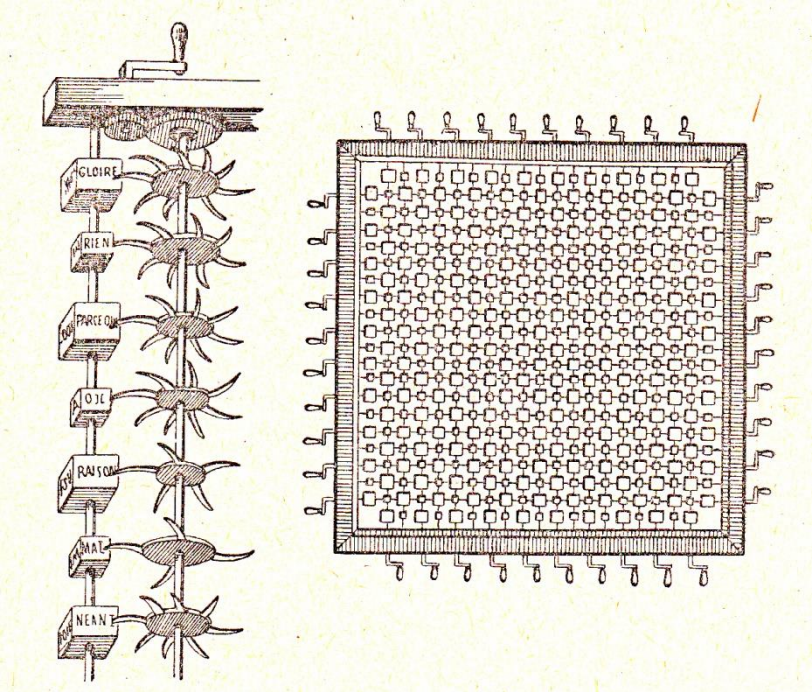
Lull błędnie przypisywał swemu „młynkowi" olbrzymie możliwości odkrywcze; w rzeczywistości użyteczność „młynka" jest dość skromna - ułatwia on tylko kolejne wynajdywanie wszystkich możliwych ewentualności, jak to widzieliśmy na powyższym przykładzie.

Zacytujemy jeszcze *Jonathana Swifta* (1667 - 1745) – irlandzkiego pisarza, autor licznych utworów satyrycznych, natrząsającego się z maszyny logicznej *Lulla*, czy jego następców (*Guliwer* relacjonuje swój pobyt w *Balnibarbi*):

„Potem udaliśmy się na drugą stronę Akademii, przez projekcistów w wiadomościach spekulacyjnych zajmowaną. Pierwszy profesor, którego ujrzałem, znajdował się w wielkim pokoju, otoczony przez czterdziestu uczniów. Po wzajemnym przywitaniu się, gdy spostrzegł, że bardzo uważnie oglądam wielką machinę zabierającą większą część pokoju, zapytał, czy nie budzi we mnie zdziwienia, że trudni się udoskonaleniem wiadomości spekulacyjnych za pomocą operacji mechanicznych. Pochlebia sobie, że świat uzna ważność jego wynalazku i że wznioślejsza myśl nigdy w głowie człowieka nie powstała. Wiadomo, jak trudno przychodzi każdemu człowiekowi nauczyć się kunsztów i umiejętności, lecz dzięki jego wynalazkowi człowiek najbardziej nawet nieukształcony potrafi niewielkim kosztem i po lekkim ćwiczeniu ciała pisać książki filozoficzne, poetyczne, rozprawy o polityce, teologii i matematyce bez najmniejszej po mocy naturalnych zdolności lub nauk. Zaprowadził mnie do warsztatu, przy którym uczniowie stali ustawieni w szeregach. Była to wielka rama, mająca dwadzieścia stóp kwadratowych; powierzchnia jej składała się z małych kawałków drzewa wielkości kostki, niektóre jednak z nich były większe od drugich, a wszystkie połączone ze sobą przez cienkie druty. Na powierzchni sześcianków przylepione były kawałki papieru, na których napisano wszystkie wyrazy języka krajowego w różnych odmianach, koniugacjach, deklinacjach, ale bez żadnego porządku. Profesor prosił mnie, ażebym uważał, bo chce machinę poruszyć. Na jego rozkaz każdy uczeń ujął jedną z czterdziestu antab w ramie będących i obróciwszy je odmienił rozkład wyrazów. Rozkazał potem trzydziestu sześciu chłopcom, ażeby wiersze powoli czytali. Kiedy znajdowali ciąg kilku wyrazów mogących stanowić zdanie, dyktowali je czterem innym chłopcom, którzy to pisali. Ta operacja powtórzona została kilka razy, za każdym obróceniem sześcianki naokoło się obracały i wyrazy coraz inne zajmowały miejsca.

Sześć godzin dziennie pracowali uczniowie przy tej nauce; profesor pokazał mi wiele foliałów powstałych z ułamków zdań, obejmujących, jak zapewniał, skarb wszystkich kunsztów i umiejętności, które ułożyć i wydać zamyśla. Lecz zamiar ten wtedy dopiero może przyjść do skutku, a dzieło do wielkiego stopnia doskonałości, jeżeli publiczność zechce dostarczyć potrzebnych funduszów na założenie pięciuset takich machin i jeżeli dyrektorowie ich obowiązani zostaną

przykładać się wspólnie do wydania tak wielkiego i powszechnie użytecznego dzieła. Zapewnił mnie, że ten wynalazek był owocem wszystkich jego myśli od wczesnej młodości, że użył całego dykcjonarza do tych ram i obliczył ściśle proporcje, jakie są w księgach między rodzajnikami, imionami, czasownikami i innymi rodzajami mowy.
Podziękowałem sławnemu profesorowi za łaskawe pokazanie i objaśnienie mi tego wszystkiego i zapewniłem, że jeżeli bym wrócił kiedy do mej ojczyzny, to uznam go za pierwszego i jedynego wynalazcę cudownej maszyny, której kształt dla lepszej pamięci na papier przeniosłem i na dowód tutaj załączam. Powiedziałem mu także, że zwyczajem jest u uczonych w Europie przywłaszczać sobie wzajemnie cudze wynalazki i dlatego będzie miał przynajmniej tę korzyść, że gdyby powstał spór, kto istotnie jest pierwszym wynalazcą, ja swoim świadectwem sprawię, ze jemu jednemu zostanie przyznany honor pierwszeństwa"[4].



Rysunek 1.0.1.30. Maszyna logiczna wg Swifta.

Piśmiennictwo: *Greniewski* H. G.2.1. , *Swift* S.13.1.

1.0.2. POCZĄTKI AUTOMATYCZNEGO LICZENIA

Pojęci procesu automatycznie wykonywanego w środowisku sztucznie stworzonym przez człowieka, wywodzi się z siedemnasto-osiemnastowiecznych pomysłów technicznych realizowanych w automatycznych maszynach tkackich – tkających tkaniny o skomplikowanym splocie (np. tkaniny żakardowej) i wykonywanych na krosnach tkackich sterowanych programem - tkania splotu, zapisanym w postaci perforowanych sekwencji otworów na karcie papierowej. Program taki był wielokrotnie wykonywany, jedno przejście karty sterującej, to kolejny splot tkanego wzoru tkaniny.

Dopiero w pierwszej połowie wieku dziewiętnastego genialny uczony statystyk i konstruktor brytyjski *Charles Babbege* (1791 - 1871), w konstruowanej przez siebie maszynie liczącej *Analytical Engine,* użył karty perforowanej (z czytnikiem używanym w przemyśle tkackim) do zapisania programu i danych.

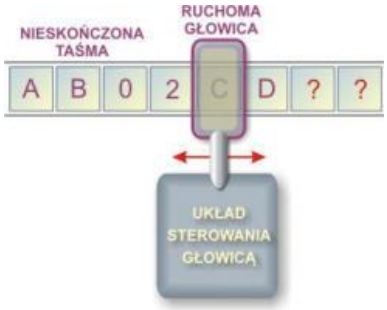
[4] Swift, S.13.1, s. 266 - 269.

W latach osiemdziesiątych dziewiętnastego wieku, udoskonalone karty perforowane stały się nośnikiem danych w opracowanym przez *Hermana Hollerith'a* (1860 - 1929) amerykańskiego inżyniera-wynalazca system maszyn licząco-analitycznych. Maszyny licząco-analityczne, a dokładniej mówiąc czytniki kart perforowanych i tzw. reproducery kart perforowanych, były w latach pięćdziesiątych – siedemdziesiątych ubiegłego wieku - powszechnie używane jako urządzenia wejścia i wyjścia komputerów.

Piśmiennictwo: *Wikipedia*. W.2.5. i W.2.11.

1.0.3. MASZYNA TURINGA

Maszyna Turinga - opracowany w 1936 roku, przez *Alana Turinga*, abstrakcyjny model maszyny logicznej (jakbyśmy dzisiaj powiedzieli – *rodzaju wirtualnego komputera*) służącej do wykonywania algorytmów. Maszyna Turinga składająca się z układu sterowania, głowicy czytająco-piszącej i nieskończenie długiej taśmy podzielonej na pola, powstała - jako notacja umożliwiająca opisywanie algorytmów i badanie ich wykonywania. Taśma może być nieskończona jednostronnie lub obustronnie. Każde pole taśmy, może zawierać: znak pusty albo zawierać jeden wpisany znak alfabetu ∑. Głowica maszyny zawsze jest ustawiona nad jednym z pól (patrz rys. 1.0.3.10).



Rysunek 1.0.3.10. Maszyna Turinga

Stan maszyny określa słowo złożone ze znaków alfabetu i należące do zbioru stanów Q. Zależnie od kombinacji:

a. Aktualnie wykonywanego rozkazu;
b. Stanu maszyny oraz
c. Zawartości pola, nad którym znajduje się głowica;

Maszyna Turinga zapisuje nową wartość w polu, a następnie może przesunąć głowicę:

a) jedno pole w prawo;
b) albo w lewo,
c) albo pozostaje na dotychczasowym miejscu.

Taka operacja nazywana jest rozkazem funkcji przejścia δ. Rozkazy Maszyny Turinga tworzą program działania maszyny. Maszyna rozpoczynając działanie programu od stanu początkowego, który ma na taśmie zapisany – w formie tzw. słowa wejściowego q_1 – na pierwszych *m* polach taśmy, w pozostałych polach taśmy wpisany jest znak ⊔. Maszyna kończy działanie każdego programu w jednym z dwu stanów: stan akceptujący $q_{akcetuj}$ albo stan odrzucający $q_{odrzuć}$. więcej informacji na temat Maszyny Turinga, znajdzie czytelnik w podrozdziale 3.8.4.

Piśmiennictwo: *Sipser M*. S.7.1., *Turing A.* T.6.1., *Wikipedia* W.2.30.

1.0.4. ENIAC – KOLOS Z LAMP ELEKTRONOWYCH

ENIAC (*Electronic Numerical Integrator And Computer* – Elektroniczny, Numeryczny Integrator i Komputer) – automatyczny kalkulator skonstruowany w latach 1943-1945 przez *J.P. Eckerta* i *J.W. Mauchly'ego* na Uniwersytecie Pensylwanii w USA. Zaprzestano jego używania w 1955. ENIAC był innowacyjnym urządzeniem: posiadał właściwość przetwarzania równoległego i oddzielne funkcjonalnie moduły jednostki arytmetycznej i pamięci danych, nie miał jednak możliwości składowania w pamięci programu i wykonywania go.

Początkowo program pracy ENIAC'a był ustawiany na pulpicie z pomocą kabelków łączących gniazda poszczególnych operacji z numerami kroków obliczeniowych, później umożliwiono sterowanie za pomocą kart perforowanych (podobnie jak w projektowanym *Analytical Engine - Charlesa Babbege*). Maszyna używana była głównie do obliczeń związanych z balistyką, wytwarzaniem broni jądrowej, prognozowaniem pogody, projektowaniem tuneli aerodynamicznych i badaniem promieniowania kosmicznego. Wykorzystywano ją także do badania liczb losowych i analizowania błędów zaokrągleń. ENIAC miał masę ponad 27 ton, zawierał blisko 18 tys. lamp elektronowych i zajmował powierzchnię ok. 140 metrów kwadratowych.

Rysunek 1.0.4.10. ENIAC – widok maszyny.

Jak podaje Wikipedia: „Budowa ENIACA związana była z potrzebą sporządzania dla wojska tzw. tablic artyleryjskich. W tym celu sprowadzono z Princeton, na stanowisko szefa projektu, wybitnego matematyka norweskiego *Oswalda Veblena*, który prowadził podobne obliczenia w 1917 roku; ponadto zatrudniono dalszych 7 matematyków, 8 fizyków i 2 astronomów. Ich doradcą był genialny Węgier, *John (Janos) von Neumann*.

Do wojska wcielono w charakterze rachmistrzów około 100 młodych matematyczek, zarekwirowano na potrzeby armii cały, nadający się do wykorzystania sprzęt obliczeniowy. Było jednak jasne, że tą drogą potrzeb artylerii w pełni się nie zaspokoi. W tym samym czasie spotkali się ze sobą: doktor fizyki *John W. Mauchly* (ur. 1907), inżynier elektronik *John Presper Eckert* (ur. 1919) oraz doktor matematyki, porucznik armii USA, *Herman Heine Goldstine* (ur. 1913). *J. Mauchly* już w roku 1940 mówił o możliwości zastosowania elektroniki do budowy maszyny liczącej; wpadł na ten pomysł w związku z ogromem obliczeń, jakie musiał wykonać, gdy zainteresował się zastosowaniami statystyki matematycznej w meteorologii. Kiedy wstąpił na zorganizowane przez Uniwersytet Pensylwanii specjalne kursy, przygotowujące wysokiej klasy specjalistów dla armii, spotkał *J. P. Eckerta*, który był utalentowanym konstruktorem i wykonawcą.

Obaj słuchacze kursu w wolnych od nauki chwilach zaprojektowali wielki kalkulator, uniwersalną maszynę liczącą. Przekazali go oficjalnie, w formie odpowiedniego pięciostronicowego memorandum, *J. G. Brainerdowi*, członkowi Zarządu Uniwersytetu Pensylwanii, zajmującemu się służbowo kontaktami z rządem USA. Ten jednak odłożył ten dokument do szuflady biurka (znaleziono go tam w 20 lat później – był nietknięty) nie przekazując go dalej, co spowodowałoby zamknięcie sprawy, gdyby nie trzeci współtwórca ENIAC'a, *dr H. H. Goldstine*'a.

Goldstine pracował we wspomnianym wyżej ośrodku obliczeniowym armii USA (*Ballistic Research Laboratory*, BRL) i gwałtownie poszukiwał rozwiązania znanego już nam problemu tablic balistycznych. Prowadząc w marcu 1943 rutynową kontrolę pracującego dla wojska ośrodka obliczeniowego Uniwersytetu Pensylwanii, opowiedział o swoich kłopotach pewnemu słuchaczowi kursu przygotowującego specjalistów dla wojska. Tym studentem był *Mauchly*, jeden z autorów wspomnianego memorandum. W kilkanaście dni później *Goldstine* i *Mauchly* zostali przyjęci przez kierownictwo BRL. *Oswald Veblen* nie miał wątpliwości: nakazał natychmiast udostępnić niezbędne pieniądze na budowę maszyny. W ostatnim dniu maja 1943 roku ustalono nazwę *ENIAC*. Piątego czerwca podpisano uruchomienie najściślej tajnego ≪Projektu PX≫, którego koszty ustalono na 150 tys. dolarów (faktycznie wyniosły 486.804 dolary i 22 centy). Oficjalnie pracę rozpoczęto 1 lipca, dwa pierwsze akumulatory[5] uruchomiono w czerwcu następnego roku, całą maszynę oddano do prób laboratoryjnych jesienią 1945 roku, pierwsze eksperymentalne obliczenia przeprowadzono w listopadzie roku 1945. Jak wspomniano, 30 czerwca1946 roku przekazano ENIAC'a armii USA, która pokwitowała odbiór ≪Projektu PX≫.

… W chwili, kiedy ujawniono konstrukcję ENIAC'a, opinii publicznej wydawało się, iż nigdy dotąd nie zbudowano urządzenia tej wielkości i stopnia skomplikowania, w każdym razie w dziedzinie elektroniki. Ustawione w prostokącie 12 na 6 m w kształcie litery U czterdzieści dwie pomalowane na czarno szafy z blachy stalowej – każda miała 3 m wysokości, 60 cm szerokości i 30 cm głębokości – mieściły 18 800 lamp elektronowych szesnastu rodzajów; zawierały ponadto 6.000 różnego typu złączy, 1.500 przekaźników, 50.000 oporników. Całość – jak powiedziano przedstawicielom prasy – wymagała ręcznego wykonania 0,5 mln lutowań. Maszyna ważyła 30 ton i pobierała 140 kW mocy. Jej system wentylacyjny miał wbudowane dwa silniki Chryslera o łącznej mocy 24 KM; każda szafa była wyposażona w termostat, który wstrzymywał pracę komputera, jeśli temperatura wewnątrz którejkolwiek z jego części przekraczała 48 °C. W pomieszczeniu przeznaczonym dla maszyny były jeszcze trzy dodatkowe – również wypełnione elektroniką i większe od pozostałych - szafy przesuwne na kółkach, dołączane w miarę potrzeb w odpowiednim miejscu do zestawu. Stanowiły uzupełnienie czytnika i dziurkarki kart perforowanych."

Piśmiennictwo: *Wikipedia* W.2.9.

[5] *Akumulator* – w tym przypadku, *rejestr* służący do przechowywania argumentu lub wyniku działania arytmetycznego.

1.1. HARDWARE – SPRZĘTOWA CZĘŚĆ KOMPUTERA

1.1.0. KOMPUTER JOHNA VON NEUMANNA

Idea elektronicznego komputera wewnętrznie programowanego, czyli komputera przechowującego wykonywany program, czy też programy, w swojej pamięci wewnętrznej, według danych literaturowych, wywodzi się od *Johna von Neumanna*. W rzeczywistości pomysłodawców było kilku, obok *von Neumanna*, który w 1945 roku opublikował swój pomysł, podobne pomysły przedstawiali: *Alan Mathis Turing* w Wielkiej Brytanii oraz *Konrad Zuse* w Niemczech. Być może, że niezależnie, od wcześniej wymienionych, do podobnej koncepcji mógł dojść również *Siergiej Aleksiejewicz Lebiediew* w Kijowie (ZSRR).

John von Neumann, jak podają liczne encyklopedie (w tym Wikipedia), wniósł znaczący wkład do szeregu dziedzin matematyki m.in. logiki matematycznej, teorii mnogości, analizy matematycznej, udowodnił twierdzenie min-max. W 1944 roku napisał razem z *Oskarem Morgensternem - The Theory of Games and Economic Behavior* stając się twórcą teorii gier. *Von Neumann* był też autorem pierwszej, matematycznie poprawnie sformułowanej książki z mechaniki kwantowej. Oprócz tego był przede wszystkim, jednym z pionierów informatyki. Od 1943 roku uczestniczył w projekcie Manhattan, w ramach którego zbudowano pierwszy efektywnie funkcjonujący reaktor atomowy oraz pierwszą bombę atomową. Z tego czasu pochodzą takie odkrycia jak powstanie pierwszej metody numerycznej rozwiązania hiperbolicznych równań różniczkowych cząstkowych i koncepcja architektury komputera zwanej architekturą *von Neumanna*, która została opisana w 1945 w publikacji *First Draft of a Report on the EDVAC*.

Należy podkreślić, że *von Neumann*, znał dobrze koncepcje *Maszyny Turinga*, podobno prowadził z *Alanem Turingiem* obszerną korespondencję dotyczącą rozważań związanych z tworzeniem architektury pierwszych komputerów cyfrowych. Niewątpliwie, wyniki prac *Turinga* miały istotny wpływ na rozwiązania przyjęte dla komputera EDVAC.

1.1.0.01. **Wyjaśnienie**. Pojęcie architektury komputera odnosi się do tych atrybutów systemu, które są widziane przez programistę. Innymi słowy, atrybuty te mają bezpośredni wpływ na logikę wykonywanego programu. Przykładami atrybutów architektury są:

- Lista rozkazów
- Liczba bitów wykorzystywanych do prezentacji różnych typów danych (np. liczb czy znaków)
- Mechanizmy układów wejścia-wyjścia
- Metody adresowania pamięci.

1.1.0.02. **Wyjaśnienie**. *Architektura von Neumanna* – była pierwszą opublikowaną architekturą komputera, opracowaną przez *Johna von Neumanna*, *Johna W. Mauchly'ego* oraz *Johna Prespera Eckerta* w 1945 roku. Cechą charakterystyczną tej i wszystkich dalszych architektur jest to, że dane przechowywane są wspólnie z instrukcjami, co sprawia, że są kodowane w ten sam sposób. W architekturze tej komputer składa się z czterech głównych komponentów:

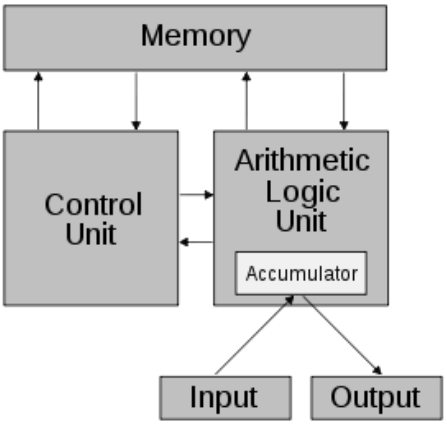
1) pamięci komputerowej przechowującej dane programu oraz instrukcje programu; każda komórka pamięci ma unikatowy identyfikator nazywany jej adresem
2) jednostki sterującej odpowiedzialnej za pobieranie danych i instrukcji z pamięci oraz ich sekwencyjne przetwarzanie

3) jednostki arytmetyczno-logicznej odpowiedzialnej za wykonywanie podstawowych operacji arytmetycznych.
4) urządzeń wejścia/wyjścia służących do interakcji z operatorem.

Jednostka sterująca wraz z jednostką arytmetyczno-logiczną tworzą procesor.

1.1.0.03. **Wyjaśnienie**. Architekturę von Neumanna można scharakteryzować krótko:

1. Wspólna pamięć do przechowywania zarówno rozkazów jak i danych.
2. Pamięć jednowymiarowa, złożona z kolejno ponumerowanych komórek o jednakowej wielkości.
3. Brak jawnego rozróżniania rozkazów i danych.
4. Brak jawnej specyfikacji typów danych.
5. Praca sekwencyjna - przed rozpoczęciem wykonywania kolejnego rozkazu musi zostać zakończone wykonywanie rozkazu chronologicznie poprzedniego.
6. Każdy rozkaz określa jednoznacznie adres następnego.



Rysunek 1.1.0.00. Schemat ideowy komputera o architekturze von Neumanna.

Podane warunki pozwalają przełączać system komputerowy z wykonania jednego zadania (programu) na inne bez fizycznej ingerencji w strukturę systemu, a tym samym gwarantują jego uniwersalność. System komputerowy von Neumanna nie posiada oddzielnych pamięci do przechowywania danych i instrukcji. Instrukcje jak i dane są zakodowane w postaci liczb. Bez analizy programu trudno jest określić czy dany obszar pamięci zawiera dane czy instrukcje. Wykonywany program może się sam modyfikować traktując obszar instrukcji, jako dane, a po przetworzeniu tych instrukcji – danych – zacząć je wykonywać.[6]

1.1.0.04. **Wyjaśnienie**. Architektura von Neumana przewidywała, że komputer wyposażony jest w następujące rejestry:

[6] Na wzór organizacji komputera von Neumana, opracowano abstrakcyjną Maszynę RAM (*Random Access Memory*) – równoważny obliczeniowo maszynie Turinga. Jest to komputer z nieograniczoną pamięcią RAM (Sheperdson & Sturgis). Maszyna RAM realizuje programy w języku zbliżonym do popularnych imperatywnych języków programowania. Składa się z licznika rozkazów oraz pamięci, która jest tablicą rozmiaru $\aleph_0$.

Przykładowa lista instrukcji Maszyny RAM				
Kod instrukcji	Kod adresów	Oznaczenie	Semantyka	Uwagi
0	n	Z(n)	z[n]:=0	Licznik rozkazówzwiększ o 1
1	n	S(n)	z[n] := z[n]+1	Licznik rozkazówzwiększ o 1
2	π(m, n)	T(m,n)	z[n] := z[m]	Licznik rozkazówzwiększ o 1
3	β(m, n, q)	I(m, n, q)	if z[m] = z[n] then goto q	Licznik rozkazów zwiększ o 1, gdy z[m]= z[n], w przeciwnym przypadku umieść w nim q

1) MBR – rejestr buforowy pamięci (*Memory Buffer Register*)
2) MAR – rejestr adresowy pamięci (*Memory Address Register*)
3) IR – rejestr rozkazów czyli wykonywanej instrukcji (*Instruction Register*)
4) PC – licznik programu zwany również licznikiem adresów (*Program Counter*)
5) AC – akumulator - rejestr wyników operacji arytmetycznych oraz logicznych (*Accumulator Counter*)
6) MQ – rejestr mnożenia i dzielenia (*Multiplier-Quote Register*).

Uwaga: w dalszym ciągu, będziemy używać tych nazw rejestrów wprowadzonych przez von Neumanna, ponieważ rejestry te, obok innych, występują we współczesnych komputerach.

Pierwszy działający komputer, zbudowany w oparciu o architekturę von Neumanna powstał jednak, nie w USA, ale w Wielkiej Brytanii, był nim EDSAC, z blisko dwa lata później dopiero uruchomiony został komputer EDVAC w USA.

Rysunek 1.1.0.10. Widok ogólny stojaków lampowych komputera EDSAC.

1.1.0.11. **Wyjaśnienie**. EDSAC (akronim od ***E**lectronic **D**elay **S**torage **A**utomatic **C**alculator*) – komputer oparty na architekturze von Neumanna, skonstruowany przez zespół *Maurice'a Wilkesa* z *University of Cambridge Mathematical Laboratory*, na którym pierwszy program uruchomiono 6 maja 1949 r. EDSAC był pierwszym komputerem wykorzystywanym w praktyce – wykonywano na nim badania uniwersyteckie. Długość instrukcji EDSAC wynosiła 17 bitów, czyli tzw. słowo krótkie, zaś długość słowa arytmetycznego wynosiła 34 bity, czyli tzw. słowo długie. 17 bitowa instrukcja dzieliła się na trzy grupy bitów: (1) kod operacji 5 bitów, (2) wskaźnik adresu argumentu (słowo krótkie to wartość 0 albo słowo długie wartość 1); (3) kod adresu o długości 11 bitów (adres słowa długiego jest zawsze liczbą parzystą) lub argumentem bezpośrednim. Wykorzystywano w nim pamięć na rtęciowych liniach opóźniających (pamięć dynamiczną) - o pojemności 1024 słowa 17 bitowe (czyli 512 słowa 34 bitowe) i elektronowe lampy próżniowe dla układów logicznych. Jak widać, jeden z bitów adresowych nie był wykorzystany dla adresacji pamięci, ponieważ wystarczyło do celów adresacji użycie 10 bitów[7]. W 1953 r. *David Wheeler*, wykorzystał ten wolny bit 17 bitowej instrukcji zaprojektował rejestr indeksowy, jako rozszerzenie oryginalnej konstrukcji EDSAC (instrukcje, których część adresowa miała być powiększona o zawartość rejestru indeksowego, miała na tym bicie wartość 1, pozostałe instrukcje, których adres nie był sumowany z zawartością rejestru indeksowego, miała na tym bicie wartość 0). Była to jedna z pierwszych i bardzo istotna innowacja dotycząca architektury komputerów.

Komórki pamięci są numerowane kolejnymi liczbami naturalnymi i mogą zawierać, tak jak licznik rozkazów, dowolną liczbę naturalną. W czasie obliczeń prawie wszystkie komórki pamięci zawierają liczbę 0. Oznaczenie z[x] oznacza zawartość komórki o numerze x. Uwaga: Instrukcje typu 0, 1, 2 nazywamy arytmetycznymi, a typu 3 warunkowymi.

[7]The Preparation of Programs for an Electronic Digital Computer by Maurice Wilkes, David Wheeler, and Stanley Gill; (original 1951); reprinted with new introduction by Martin Campbell-Kelly; 198 pp.; illus; biblio; bios; index; ISBN 0-262-23118-2. Available through Charles Babbage Institute

1.1.0.21. **Wyjaśnienie**. EDVAC - ***E**lectronic **D**iscrete **V**ariable **A**utomatic **C**omputer*. Maszyna zbudowana według pomysłu *von Neumanna* w kwietniu 1952 w *Moore School of Engineering* przy Uniwersytecie Pensylwanii na potrzeby armii USA. Stworzyli ją *John Mauchly* i *J. Presper Eckert*. Zainstalowana w *Ballistic Research Laboratories* w Aberdeen (stan Maryland) w roku 1949 kosztem ok. 500.000 dolarów (przy budżecie pięciokrotnie mniejszym). EDVAC zajmowała ponad 111 metrów kwadratowych powierzchni, zawierała ok. 6000 lamp elektronowych oraz 12000 diod. Częstotliwość zegara 996,75 kHz. Pamięć operacyjna ultradźwiękowa z rtęciową rurą opóźniającą na 1000 słów; wejście/wyjście z zastosowaniem taśmy perforowanej i kart dziurkowanych systemu IBM; do kontroli sterowania używano zwykłego oscyloskopu; w roku 1953 dodano do maszyny pamięć zewnętrzną w postaci bębna magnetycznego. Zapotrzebowanie na moc 56 kilowatów. EDVAC osiągnął użyteczność obliczeniową już w roku 1951, pracował do grudnia roku 1962.

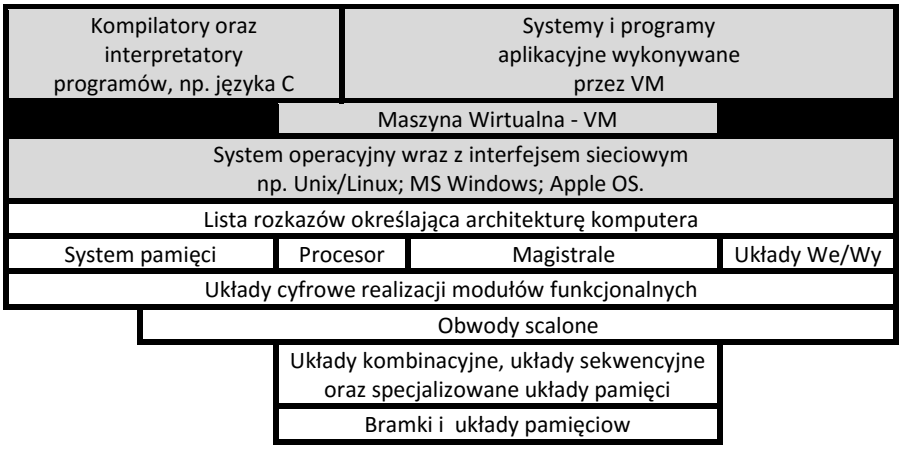
Piśmiennictwo*: Wikopedia.* W.2.4., W.2.19., *Wilkes M*. W.4.1.

1.1.1. KOMPUTERY WEKTOROWY I SUPER-SKALARNY

Współczesny komputer składa się z części tzw. sprzętowej (*hardware*), oraz oprogramowania systemowego, zwanego częścią miękką (*software*) – patrz rys. 1.1.1.00. W wyniku rozwoju architektury von Neumanna, powstało kilka typowych architektur współczesnych komputerów.

1.1.1.01. **Wyjaśnienie.** Przez komputer super-skalarny rozumiemy komputer o najpowszechniej dziś występującej architekturze, będący daleko posuniętym rozwinięciem architektury von Neumanna, w którym nie zawsze jest spełniony warunek sekwencyjnego działania, oraz komputer taki wyposażony jest siedem poniżej wymienionych grup funkcjonalności:

1. Układ obsługi przerwań z licznikami czasu (tzw. czasomierz) i stosem
2. Adresacja z użyciem rejestrów indeksacji
3. System wielopoziomowy pamięci, w tym pamięci masowe i funkcjonalność pamięci wirtualnej
4. Magistrale i układ DMA
5. Sterowanie mikroprogramowe
6. Arytmetyka FPU
7. Potok i instrukcja *„do-nothing”*.



Rysunek 1.1.1.00. Podział architektury komputera super-skalarnego na części: softwarową (miękką), i sprzętową (twardą).

1.1.1.02. **Wyjaśnienie**. Komputer super-skalarny składa się z modułów, współpracujących pomiędzy sobą za pośrednictwem magistrali. Są to następujące moduły:
1. Procesor, obecnie na ogół wielordzeniowy[8];
2. Pamięć RAM (*Random Access Memory*), zwana również pamięcią operacyjną;
3. Modułów interfejsowych (np. SCSI, USB)[9], umożliwiających podłączenie urządzeń zewnętrznych oraz sieci komputerowej;
4. Modułu komunikacji z operatorem (np. monitora i klawiatury).

1.1.1.03. **Wyjaśnienie.** Procesor składa się z następujących modułów funkcjonalnych, współpracujących pomiędzy sobą za pośrednictwem magistrali. Są to następujące moduły:
1. Jednostki arytmetyczno-logicznej ALU (na ogół 64 albo 32 bitowej), realizującej operacje arytmetyczne binarne stałoprzecinkowe oraz dziesiętne;
2. Zespołu rejestrów;
3. Pamięć operacyjna;
4. Pamięci podręcznej (obecnie z reguły trzypoziomowej)- zwanej również pamięcią asocjacyjną, bufora pomiędzy pamięcią RAM a rejestrami. Pamięć skojarzeniowa, na każdym z trzech poziomów, dzieli się na część przeznaczoną do przechowywania fragmentów programów i na część przeznaczoną do przechowywania danych,
5. Jednostka arytmetyki zmiennopozycyjnej FPU;
6. Wewnętrznej magistrali systemowej – umożliwiającej z jednej strony przesyłanie instrukcji i danych wewnątrz procesora, jak również;
7. Układ sterowania – zarówno procesorem, jak również całym komputerem super-skalarnym.

1.1.1.11. **Wyjaśnienie.** Przetwarzanie potokowe (patrz podrozdział 1.1.8). Podstawową techniką pozwalającą przyspieszyć przetwarzanie jest zastosowanie potoku wykonawczego. Podział wykonania pojedynczej instrukcji na etapy pozwala przyspieszyć taktowania procesora oraz zwiększyć efektywność wykorzystania logiki procesora.

1.1.1.12. **Wyjaśnienie.** Pamięć podręczna procesora (*CPU cache*) - jest pamięcią typu SRAM (pamięć RAM statyczna) o krótkim czasie dostępu. Zlokalizowana jest często bezpośrednio w jądrze procesora. Zastosowanie wielopoziomowej hierarchii pamięci podręcznej pozwala, korzystając z zasady lokalności przestrzennej i czasowej na zapewnienie złudzenia posiadania szybkiej i pojemnej pamięci głównej, a więc zmniejsza średni czas dostępu do pamięci głównej. Współcześnie stosuje się 2 i 3-poziomowe pamięci podręczne. Najważniejszymi parametrami funkcjonalnymi pamięci podręcznych są: pojemność i czas dostępu. Pod względem budowy można wyróżnić 3 podstawowe typy organizacji pamięci: - pamięć całkowicie skojarzeniowa (*fully associative*) - pamięć z odwzorowaniem bezpośrednim (*direct-mapped*) - pamięć wielodrożna (*set-associative*).

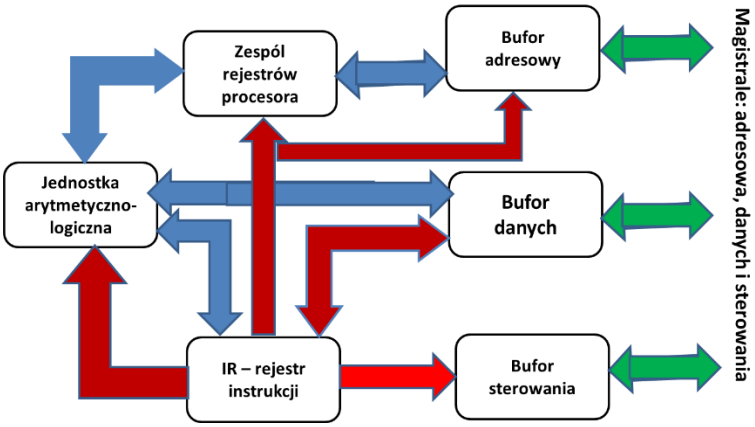
1.1.1.13. **Wyjaśnienie.** *Cache* czyli *pamięć podręczna*. Zastosowanie złożonej hierarchii pamięci podręcznych procesora (*cache*) pozwala na zmniejszenie średniego czasu dostępu do pamięci i znaczne przyspieszenie przetwarzania.

---

[8] *Rdzeń* (*core*)– używane obecnie określenie na jeden zestaw rejestrów i układów wykonania operacji arytmetyczno-logicznych składających się łącznie na odpowiednik jednostkowego procesora. Procesor *wielordzeniowy* to processor zdolny do wykonania tzw. potoku operacji arytmetyczno-logicznych.

[9] *Interfejs* (*interface*) – układ pośredniczący – dopaowujący sygnały, czyli umożliwiające współpracę dwóch urządzeń np. procesora i drukarki.

1.1.1.15. **Wyjaśnienie.** L-1 cache. Zlokalizowana we wnętrzu procesora pamięć podręczna pierwszego poziomu przyspiesza dostęp do bloków pamięci wyższego poziomu, który stanowi zależnie od konstrukcji RAM. Z uwagi na ograniczenia rozmiarów i mocy procesora L-1 zawsze ma stosunkowo mała pojemność. Umieszczona jest blisko głównego jądra procesora i umożliwia szybką komunikację z procesorem. Typowe pamięci L-1 współczesnych procesorów są 2-drożne, posiadają rozdzieloną pamięć danych i kodu, a długość linii wynosi 64 bajty.



Rysunek 1.1.1.10. Poglądowy - uproszczony schemat procesora

1.1.1.16. **Wyjaśnienie.** L-2 cache. Pamięć drugiego poziomu, o rozmiarze od 64KB do 12MB, jest 2, 4 lub 8-drożna, o długości linii od 64 do 128 B, i jest wykorzystywana jako bufor pomiędzy stosunkowo wolną pamięcią RAM a jądrem procesora i pamięcią cache L1. Obecność pamięci drugiego poziomu powoduje duży wzrost wydajności w wielu aplikacjach - dzieje się tak, ponieważ dane poddawane obróbce muszą być pobierane z pamięci RAM, która ma opóźnienia rzędu kilkudziesięciu-kilkuset nanosekund. Dzisiejsze procesory są wyposażone w złożone układy przewidywania, jakie dane będą potrzebne - dane te są pobierane z wyprzedzeniem do pamięci cache, która ma opóźnienia rzędu kilku do kilkunastu nanosekund, co znacznie skraca czas oczekiwania procesora na dane do obliczeń.

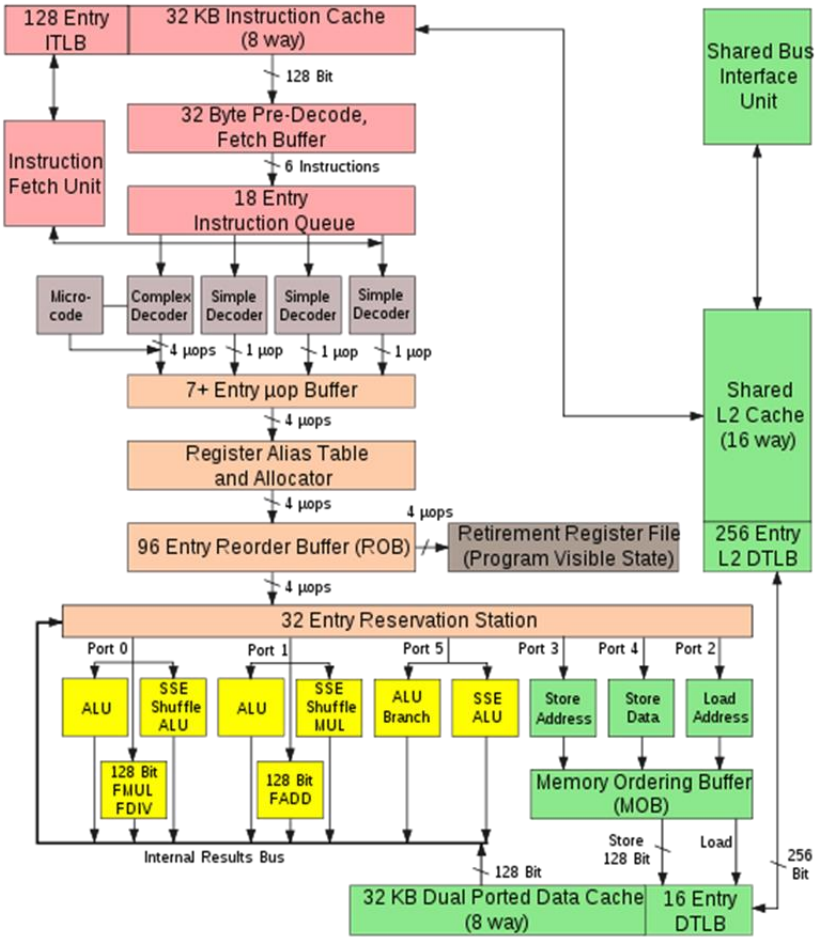
1.1.1.17. **Wyjaśnienie.** L-3 cache. Pamięć podręczna procesora trzeciego poziomu jest wykorzystywana, kiedy pamięć L-2 jest niewystarczająca aby pomieścić potrzebne dane. Najczęściej spotykana jest w procesorach dedykowanych do zastosowań serwerowych. Obecność cache trzeciego poziomu pozwala na znaczącą poprawę wydajności w stosunku do procesorów o konstrukcji pamięci cache dwupoziomowej w wielu aplikacjach i programach. Jest "ratunkiem" przed sięgnięciem po dane do powolnej pamięci RAM, aczkolwiek gdy w pamięci L-3 brakuje miejsca komputer szuka "pomocy" w powolnej pamięci RAM. W systemach z wieloma procesorami lub rdzeniami, pamięć cache trzeciego poziomu najczęściej jest współdzielona przez wszystkie rdzenie i ma od kilku do kilkunastu megabajtów (co jest niewielką ilością w porównaniu do pamięci RAM której rozmiar w nowszych komputerach oscyluje w granicach kilku - kilkunastu gigabajtów). Jej wysoka wydajność wynika z droższych i lepszych komponentów, oraz bliższego fizycznego ulokowania przy procesorze.

1.1.1.21. **Wyjaśnienie.** Prognoza rozgałęzień. Instrukcje skoków powodują wystąpienie konfliktów sterowania co owocuje wstrzymaniem przetwarzania i spowolnieniem wykonania programu. Sposobem na złagodzenie tego efektu jest zastosowanie prognozowania rozgałęzień umożliwiającej spekulacyjne wykonanie kodu za instrukcją skoku. Niestety, w przypadku błędnej prognozy, koszt opróżnienia potoku i wznowienia przetwarzania w odpowiednim punkcie może być znaczny (kilka do kilkunastu cykli procesora).

1.1.1.22. **Wyjaśnienie.** Potok super-skalarny. Wykorzystanie możliwości równoległego wykonania kilku instrukcji prowadzi do koncepcji potoku super-skalarnego, w którym zdublowane lub zwielokrotnione są jednostki wykonawcze. Niekolejne wykonanie instrukcji (*Out-of-Order Execution*) pozwala w jeszcze większym stopniu wykorzystać możliwość równoległego przetwarzania instrukcji w danym bloku podstawowym (patrz podrozdział 1.1.8).

1.1.1.23. **Wyjaśnienie.** Wielowątkowość i zastosowanie wielu procesorów. Budowa procesorów wielordzeniowych (*Chip Multiprocessing - CMP*) oraz wielowątkowych pozwala na zwiększenie ogólnej wydajności systemu poprzez umożliwienie jednoczesnego przetwarzania więcej niż jednego programu lub wątku.

Wymienionych w wyjaśnieniu 1.1.1.01 - siedem modułów funkcjonalnych komputera, umożliwia wieloprogramowe działanie komputera super-skalarnego wraz z podziałem czasu, co łącznie umożliwia zarówno przetwarzanie konwersacyjne użytkownik – komputer, jak również równoległe do konwersacji przetwarzanie wsadowe, w tle prowadzonej konwersacji. Przykładami komputerów super-skalarnych, są wszelkie dostępne na rynku laptopy, notebooki i tablety. Dalej postaramy się przybliżyć rozumienie wymienionych wyżej funkcjonalności.



Rysunek 1.1.1.18. Poglądowy schemat architektury procesora Intel Core 2.

Przez cały okres od zbudowania pierwszych komputerów, obserwujemy gwałtowny wzrost mocy obliczeniowej komputerów.

1.1.1.31. **Wyjaśnienie**. „*Prawo Moore'a* – prawo empiryczne, wynikające z obserwacji, mówiące że ekonomicznie optymalna liczba tranzystorów w układzie scalonym zwiększa się w kolejnych latach zgodnie z trendem wykładniczym (podwaja się w niemal równych odcinkach czasu). Autorstwo tego prawa przypisuje się Gordonowi Moore'owi, jednemu z założycieli firmy *Intel*, który w 1965 r. zaobserwował podwajanie się liczby tranzystorów, co ok. 12 miesięcy[10]. Liczba ta była następnie korygowana i obecnie przyjmuje się, że liczba tranzystorów w mikroprocesorach od wielu lat podwaja się, co ok. 24 miesiące. Na zasadzie analogii, prawo Moore'a stosuje się też do wielu innych parametrów sprzętu komputerowego, np. pojemności dysków twardych czy wielkości pamięci operacyjnej.”…

„Wg dokumentów *International Technology Roadmap for Semi-conductors*, uwzględniających potencjalne problemy z rozwojem i miniaturyzacją, należy oczekiwać kolejnych procesorów dostępnych (na rynku) w latach, 32nm – 2009, 22nm – 2012, 16nm – 2018, 11nm – 2022, a dalszy rozwój w ramach elektroniki stoi pod znakiem zapytania. Ostatnie 10 lat będą miały mniejszą dynamikę wzrostu niż wskazuje na to Prawo Moore'a. Wielu producentów zadeklarowało jednak, że będą w stanie wyprodukować procesory o ścieżkach ~16nm już w roku 2014.”

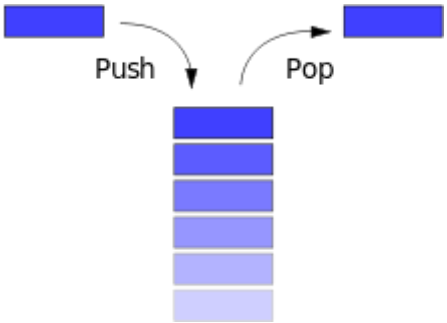
Piśmiennictwo: *Wikipedia* W.2.3., W.2.14., *Stallings W*. S.11.1., *Wojtuszkiewicz K*. W.6.1.

1.1.2. STOS I UKŁAD OBSŁUGI PRZERWAŃ

1.1.2.01. **Historia.** Stos został wymyślony i opracowany (patrz rys. 1.1.2.00.) przez niemieckiego naukowca informatyka *Friedricha L. Baura* w 1955 roku, a opatentowany w 1957. Za ten wynalazek *Friedrich L. Bauer* dostał w 1988 roku od IEEE nagrodę *Computer Society Pioneer Award*.

1.1.2.02. **Podstawowe operacje.** W powyższym opisie pojawiły się pewne operacje, jakie można wykonywać na stosie. Oto ich formalny zapis:

- PUSH (obiekt) – czyli *odłożenie obiektu na stos*;
- POP() – *ściągnięcie obiektu ze stosu* i zwrócenie jego wartości;
- ISEMPTY() - sprawdzenie czy na stosie znajdują się już jakieś obiekty.



Rysunek 1.1.2.00. Schemat ideowy stosu.

Stos jest naturalnym zapisem wyrażeń arytmetycznych i logicznych stosowanym przy obliczaniu wyrażeń zapisanych za pomocą tzw. *odwrotnej notacji polskiej* (RPN). Należy podkreślić, że *notacja polska* (PN) została wprowadzona przez wielkiego polskiego logika Jana Łukasiewicza,

[10] Gordon E. Moore: *Cramming more components onto integrated circuits*, Electronics Magazine **38** (8), 19 kwietnia 1965.

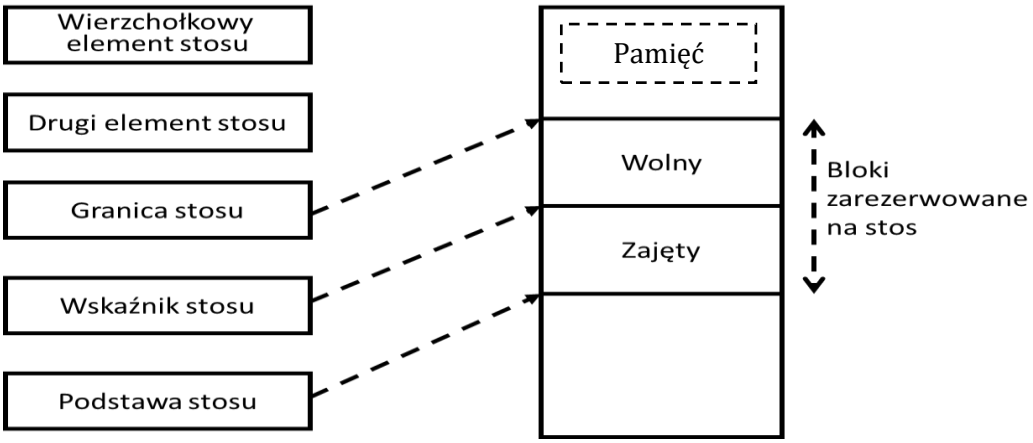
między innymi twórcy trójwartościowego rachunku zdań, co obszernie będziemy omawiać w podrozdziale 2.4.1.

1.1.2.10. **Wyjaśnienie.** Jak widać z powyższego, stos jest uporządkowanym zestawem elementów, z których tylko do jednego można mieć dostęp w określonej chwili. Punkt dostępu nazywany jest wierzchołkiem stosu. Liczba elementów w stosie, czyli długość stosu, jest zmienna. Elementy mogą być dodawane lub odejmowane tylko do wierzchołka stosu. Stos działa według zasady „*Last In – First O*ut (w skrócie LIFO)", czyli ostatnio wpisany do stosu element, jest pierwszym do odczytania ze stosu. Operacja PUSH dodaje nowy element do stosu, zaś operacja POP usuwa element z wierzchołka stosu.

1.1.2.11. **Wyjaśnienie.** Operacje jednoargumentowe (np. logiczne NOT), są wykonywane na elemencie znajdującym się na wierzchołku stosu i zmieniają element wierzchołkowy stosu na wynik. Operacje dwuargumentowe wykonywane są na dwu wierzchołkowych elementach stosu i powodują usunięcie dwu kolejnych elementów z wierzchołka stosu wpisując na wierzchołku stosu wynik operacji.

1.1.2.12. **Wyjaśnienie.** Dla poprawnej implementacji funkcjonalności stosu koniecznym jest wyposażenie procesora w trzy dodatkowe rejestry (patrz rys. 1.1.2.20.) zawierające:

- *Wskaźnik stosu*, który zawiera adres wierzchołka stosu. Jeśli element jest dodawany lub usuwany ze stosu, wskaźnik jest odpowiednio inkrementowany lub dekrementowany, aby w dalszym ciągu pokazywał wierzchołek stosu;
- *Podstawa stosu*, która zawiera adres najniższej lokacji w zarezerwowanym bloku pamięci. Jeśli dokonywana jest próba operacji POP, gdy stos jest pusty, zgłaszany jest błąd;
- *Granica stosu*, która zawiera adres drugiego końca zarezerwowanego bloku pamięci. Jeśli dokonywana jest próba operacji PUSH, gdy stos jest pełny, zgłaszany jest błąd.



Rysunek 1.1.2.20. Schemat ideowy stosu zastosowanego w komputerach.

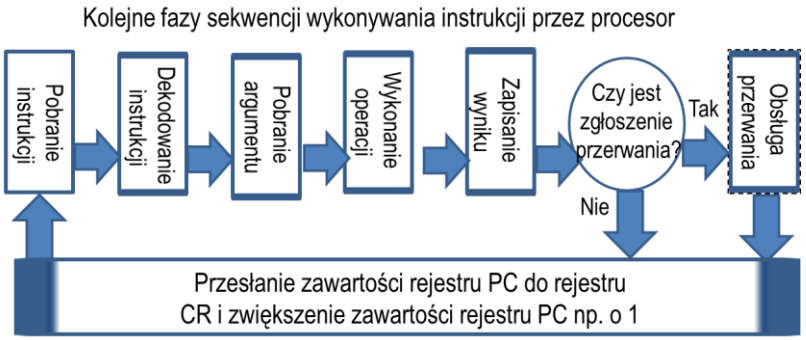
Kolejne innowacje lat pięćdziesiątych, ubiegłego stulecia, dotyczyły zwiększenia współbieżności pracy komponentów komputera - takich jak jednostka arytmetyczno – logiczna oraz wejście i wyjście. W wyniku opracowano dwie metody:

- *Odpytywania stanu urządzeń wejścia i wyjścia* przez programową jednostkę sterowania;
- *System zgłaszania i obsługi zgłoszonych przerwań* przez programową jednostkę sterowania wraz ze wspomaganiem tej obsługi odpowiednim programem.

Mechanizm przerwań został wprowadzony w latach pięćdziesiątych XX wieku, dla poprawienia efektywności przetwarzania. Szybkość procesora była już w pierwszych komputerach wielokrotnie wyższa niż szybkość urządzeń wejścia wyjścia. Przykładowo komputer bez mechanizmu przerwań, po zainicjowania operacji drukowania, musiał czekać na zakończenie operacji drukowania, żeby rozpocząć wykonywanie następnej instrukcji. W wyniku procesor komputera głównie oczekiwał na wykonanie instrukcji wejścia/wyjścia. Przy wykorzystaniu przerwań, procesor jedynie zainicjuje operację wejścia lub wyjścia, po czym przechodzi do wykonania następnej instrukcji programu. W tej sytuacji, procesor nie może zainicjować wykonywania następnej operacji wejścia lub wyjścia, zanim nie otrzyma zwrotnej informacji o zakończeniu wcześniej instrukcji wejścia lub wyjścia.

1.1.2.21. **Wyjaśnienie**. Wyróżniamy następujące klasy przerwań:

- *Programowe* – przerwanie generowane przez jeden z kodów instrukcji, które jedyną wykonaną czynnością, jest spowodowanie przerwania z podaniem określonego numeru przerwania.
- *Warunkowe* – będące wynikiem wykonania np. instrukcji arytmetycznej, dającej wynik ujemny i w takim przypadku spowodowanie przerwania z podaniem określonego numeru przerwania. Do tej klasy zaliczymy, takie przypadki jak: przekroczenie zakresu arytmetycznego, próbę dzielenia przez zero, próby wykonania w programie aplikacyjnym instrukcji, która może być wykonywana tylko w trybie systemu operacyjnego, itp.
- *Zegarowe* – generowane przez wewnętrzne czasomierze procesora, umożliwiając tym samym przejęcia sterowania przez system operacyjny, oraz *Time-out* – generowane wówczas, gdy przekroczony został maksymalny czas oczekiwania przez procesor na sygnał zwrotny ze strony wywołanego przez procesor modułu, np. któregoś z urządzeń wejścia/wyjścia.
- *Wejścia/wyjścia* – generowany przez sterownik urządzenia wejścia lub wyjścia w celu zasygnalizowania prawidłowego zakończenia instrukcji.
- *Defekt sprzętu* – generowane przez uszkodzenia, takie jak nieprawidłowe działanie urządzenia wejścia lub wyjścia, defekt zasilania, błąd pamięci operacyjnej.



Rysunek 1.1.2.30. Schemat ideowy pętli czynności składających się na wykonanie instrukcji przez procesor.

1.1.2.22. **Wyjaśnienie**. Rejestr zwany *rejestrem przerwań* służy do rejestracji zgłaszanych *przerwań*. Przy pewnym uproszczeniem można przyjąć, że liczba bitów w rejestrze przerwań jest równa łącznej liczbie możliwych przerwań i numer bitu w tym rejestrze odpowiada danemu numerowi przypisanemu danemu przerwaniu. Numery bitów odpowiadają priorytetowi przerwania. Im niższy numer, tym wyższy priorytet obsługi przerwania przez procesor. W momencie zgłoszenia przerwania, bit rejestru przypisany danemu przerwaniu jest ustawiany

na wartość 1. W momencie rozpoczęcia obsługi przerwania przez procesor, bit rejestru przypisany danemu przerwaniu jest ustawiany na wartość 0.

1.1.2.23. **Wyjaśnienie**. Sekwencje czynności procesora w czasie wykonywania instrukcji składa się z następujących kroków (patrz rys. 0.1.2.30):

1. *Faza pobrania instrukcji* (rozkazu) do wykonania;
2. *Faza dekodowania części operacyjnej* (kodu rozkazu) instrukcji;
3. *Faza pobrania argumentu* (przypadek instrukcji jednoadresowej) lub argumentów (przypadek instrukcji dwu lub więcej adresowej);
4. *Faza wykonania instrukcji* (np. arytmetycznej dodawania);
5. *Faza zapisania wyniku* wykonania instrukcji (np. arytmetycznej dodawania w akumulatorze).

1.1.2.24. **Wyjaśnienie**. Po zakończeniu fazy 5 sekwencji wykonywania instrukcji, następuje sprawdzenie czy rejestr przerwań zawiera aktualnie co najmniej jeden bit równy 1 (patrz rys. 1.1.2.30), jeśli:

- *Tak* - to następuje identyfikacja przerwania o najwyższym priorytecie, a następnie wywołanie podprogramu obsługi przerwania;
- *Nie* – to następuje przejście do wykonania następnej według wskazań rejestru PC (czyli umieszczeniem zawartości rejestru PC w rejestrze IR), z równoczesnym zwiększeniem zawartości rejestru PC np. o 1.

1.1.2.25. **Wyjaśnienie**. Obsługa przerwania odbywa się w sześciu krokach i wygląda następująco:

1. Procesor wstrzymuje przesłanie do rejestru IR zawartości rejestru PC;
2. Procesor zapisuje zawartość rejestrów (w tym PC, AC i MQ) na stosie;
3. Na podstawie numeru przerwania, ładuje do rejestru PC adres pierwszego rozkazu wybranego programu obsługi przerwania i jednocześnie blokuje badanie zgłoszenia pobrania w pętli czynności wykonania instrukcji (patrz rys. 1.1.3.20);
4. Procesor wykonuje program obsługi przerwania;
5. Po zakończeniu wykonywania programu obsługi przerwania, procesor pobiera ze stosu zapisane zawartości rejestrów i umieszcza je w rejestrach (w tym PC, AC i MQ) i jednocześnie odblokuje badanie zgłoszenia pobrania w pętli czynności wykonania instrukcji (patrz rys. 1.1.3.20);
6. Rozpoczyna wykonywanie przerwanego programu, rozpoczynając od instrukcji o adresie zapisanym w rejestrze PC.

1.1.2.31. **Wyjaśnienie.** System operacyjny zezwalając procesorowi na wykonywanie danego programu aplikacyjnego, ustawia w wewnętrznym czasomierzu procesora - czas przyznany na działanie programu. W toku wykonywania programu, procesor co jednostkę czasu odejmuje po jedynce od licznika czasomierza, aż do momentu wyzerowania stanu licznika czasomierza. Po wyzerowaniu licznika czasomierza procesor generuje przerwanie zegarowe, które powoduje przekazanie procesora do dyspozycji systemu operacyjnego.

Piśmiennictwo: *Mano M.* M.1.1., *Stallings W.* S.11.1., *Wikipedia* W.2.24., W.2.25., *Wojtuszkiewicz K.* W.6.1.

1.1.3. ADRESACJA PAMIĘCI, Z UŻYCIEM REJESTRÓW INDEKSACJI I SEGMENTACJI

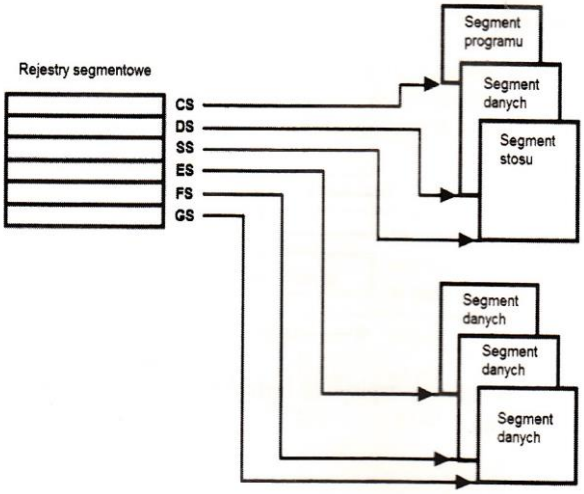
Architektura von Neumanna, nie przewidywała żadnych mechanizmów dynamicznego modyfikowania adresów wykonywanych instrukcji komputera. Innowacja dodania rejestru indeksowego została zrealizowana w komputerze EDSAC w roku 1953, jak to już wcześniej zostało wzmiankowane. Obecnie procesory komputerów są wyposażone w bardzo efektywne mechanizmy wspomagania adresacji pamięci i indeksacji, co łącznie wchodzi w tzw. zarządzanie pamięcią komputera (patrz podrozdział 1.4.6.).

Segmentowa organizacja adresacji pamięci operacyjnej komputera z procesorem należącym do rodziny Intel 80x86 można przedstawić jak na rys. 1.1.3.00., gdzie rejestr SS - określa adres segmentu stosu wykonywanego programu (procesu), rejestr DS – określa adres segmentu sterty (obszaru danych lokalnych) wykonywanego programu (procesu), rejestr CS – określa adres segmentu służącego do przechowywania bloku kontrolnego i przemieszczalnego kodu programu (procesu), rejestr ES – określa segment służący do przechowywania danych globalnych udostępnianych przez system aplikacji wykonywanemu programowi (procesowi). Rejestry FS i GS – są dostępne jedynie w nowszych wersjach procesorów rodziny Intel 80x86.

1.1.3.10. **Wyjaśnienie**. Adres słowa pamięci, w przypadku stosowania mechanizmu rejestrów segmentowych, składa się z pary liczb. A mianowicie:

1. Adresu bezwzględnego słowa (bajtu) pamięci przypisanego segmentowi, dokładniej mówiąc adresu pierwszego słowa (bajtu) segmentu – w skrócie segment;
2. Adresu słowa (bajtu) segmentu, wyznaczonego względem adresu pierwszego słowa (bajtu) segmentu – zwanego potocznie przesunięcie;

adres fizyczny słowa (bajtu) = segment : przesunięcie[11].



Rysunek 1.1.3.00. Segmentowa organizacja pamięci w procesorach rodziny Intel 80x86.

Uwaga: stosowanie mechanizmu segmentowej adresacji pamięci, nie wyklucza stosowania mechanizmów rejestrów indeksowych. Więcej można powiedzieć, że z reguły mechanizmy rejestrów segmentowy i rejestrów indeksowych – są stosowane łącznie. W przypadku stosowania tych dwóch mechanizmów nazywając zawartość wybranego rejestru indeksowego – indeksem, adres fizyczny słowa (bajtu) będzie wynosi:

adres fizyczny słowa (bajtu) = segment + przesunięcie + indeks.

[11] Przesunięcia w stosunku do adresu początku segmentu, czyli numer kolejny wskazanego slowa segmentu.

Dalej ograniczymy się do krótkiej charakterystyki, trybów adresacji dla komputera o rozkazach jednoadresowych, podkreślając jednocześnie, że współczesne komputery skalarne – są wyposażone zarówno w instrukcje jednoadresowe, jak i dwu lub trzyadresowe. Adresacja służy między innymi do dostępu do wskazanych danych.

1.1.3.11. **Wyjaśnienie**. Współcześnie, przy klasycznej architekturze komputera o rozkazach jednoadresowych, wyróżnia się siedem podstawowych trybów adresowania, wymienionych poniżej (dodatkowo zestawionych w tabeli 1.1.3.10):

1. Tryb natychmiastowy adresacji - argument zapisany jest w części adresowej instrukcji (np. instrukcja porównania, sprawdzająca – czy liczba zapisana w akumulatorze jest większa niż 25. Liczba 25 jest zapisana w części adresowej instrukcji).
2. Tryb bezpośredni adresacji - adres argumentu przechowywanego w pamięci operacyjnej - zapisany w części adresowej instrukcji.
3. Tryb pośredni adresacji - adres słowa w pamięci, w którym zapisany jest adres argumentu, zapisany w części adresowej instrukcji. Np. pozwala na wykonywanie programu generowanego w toku jego powstawania.
4. Tryb rejestrowy adresacji - adres rejestru argumentu zapisany w części adresowej instrukcji.
5. Tryb rejestrowy pośredni adresacji - adres rejestru, w którym zapisany jest adres argumentu, zapisany w części adresowej instrukcji.
6. Tryb adresowy z przesunięciem (indeksowany) - adres logiczny argumentu składa się z: zawartości rejestru i adresu argumentu, zapisanych w części adresowej instrukcji (jest to rozwiązanie, które po raz pierwszy zostało zastosowane w komputerze EDSAC w 1953 roku).
7. Tryb stosowy - argument jest pobierany z wierzchołka stosu.

Tabela 1.1.3.10. Zalety i wady omawianych trybów adresacji.

Lp	Tryb	Algorytm	Główna zaleta	Główna wada
1	Natychmiastowy	Argument = A	Brak odwołania do pamięci	Ograniczona wielkość argumentu
2	Bezpośredni	EA = A	Prostota	Ograniczona przestrzeń adresów
3	Pośredni	EA = (A)	Duża przestrzeń adresów	Kilka odwołań do pamięci
4	Rejestrowy	EA = R	Brak odniesienia do pamięci	Ograniczona przestrzeń adresów
5	Rejestrowy pośredni	EA = (R)	Duża przestrzeń adresów	Dodatkowe odwołanie do pamięci
6	Z przesunięciem	EA = A + (R)	Adresacja względem segmentów	Złożoność struktury adresów
7	Stosowy	EA = wierzchołek stosu	Brak odwołań do pamięci	Ograniczona stosowalność

1.1.3.30. **Wyjaśnienie**. Współczesne komputery skalarne, korzystają dodatkowo z mechanizmu adresacji względnej pamięci RAM, gdzie obszar adresacji zaczyna się od adresu zapisanego w tzw. rejestrze bazowym, zwanym rejestrem segmentu. W procesorach rodziny Intel 80x86 mamy sześć rejestrów segmentowych, patrz rys. 1.1.3.00: (1) Rejestr CS (*Code Segment*) – adres bazowy wykonywanego programu; (2) Rejestr DS (*Data Segment*) – adres bazowy obszaru danych, na których działa wykonywany program; (3) Rejestr SS (*Stake Segment*) – adres bazowy stosu, na którym działa wykonywany program; (4) Rejestry ES, FS i GS – adresy bazowe dodatkowych obszarów danych ewentualnie wykorzystywanych np. przez wykonywany wielowątkowy program. W toku wykonywania programu, przypisane danemu programowi zawartości rejestrów segmentowych są automatycznie sumowane przez jednostkę sterującą w czasie pobierania instrukcji oraz pobierania czy też zapisywania danych w pamięci RAM lub na stosie, z zawartością części adresowej instrukcji.

Piśmiennictwo: *Mano M.* M.1.1., *Stallings W*. S.11.1., *Wojtuszkiewicz K*. W.6.1.

1.1.4. MAGISTRALE I UKŁAD DMA

1.1.4.10. **Wyjaśnienie**. Magistrala jest urządzeniem zapewniającym komunikację pomiędzy poszczególnymi modułami komputera, tym samym jest ona wspólnym nośnikiem transmisji pomiędzy modułami (*shared transmission medium*). Typowymi modułami w architekturze von Neumanna były: procesor, pamięć i urządzenia wejścia/wyjścia – natomiast nośnikiem transmisji pomiędzy tymi modułami był rejestr zwany MBR (*Memory Buffer Register*). Można przyjąć, że współczesne magistrale, są bezpośrednim spadkobiercą funkcjonalność MBR. Współczesna magistrala – jest magistralą wielopoziomową, gdzie poszczególne poziomy są przeznaczone do transmisji z różnymi szybkościami, dopasowanymi do szybkości działania obsługiwanych modułów. Z kolei poszczególne poziomy magistrali połączone są tzw. mostami – urządzeniami umożliwiającymi transmisję pomiędzy połączonymi poziomami magistrali.

1.1.4.20. **Wyjaśnienie**. Każdy poziom magistrali, jest samodzielną magistralą. Magistrala składa się z pewnej liczby linii (zwykle z 50 do 100). Każdej linii magistrali przypisana jest określona funkcjonalność. Składające się na magistralę linie, można podzielić na trzy podstawowe grupy, zwane odpowiednio:

1. *Magistrala danych* przekazuje informacje pomiędzy wskazanym miejscem pamięci albo urządzeniem wejścia/wyjścia a procesorem. Większość używanych obecnie procesorów ogólnego przeznaczenia wykorzystuje magistrale danych szerokości 32 albo 64 bity. Często mówi się o 8, 16, 32, 64 – bitowych procesorach. O wielkości procesora decyduje liczba linii danych w procesorze oraz całkowito-liczbowy rejestr ogólnego przeznaczenia. Np. Intel 8086 są 16-bitowe, Intel 80386 są 32-bitowe, zaś Intel Core 2 Duo są 64-bitowe.

- *Magistrala adresowa*. Miejsce pamięci albo numer portu urządzenia wejścia-wyjścia określa informacja przekazywana magistralą adresową.
- *Magistrala sterowania.* Umożliwia przekazywanie zestawu sygnałów decydujących o tym, jak poszczególne moduły (w tym procesor) komunikuje się z resztą systemu, takich sygnałów jak: zapis w pamięci, odczyt z pamięci, zapis na urządzeniu wyjścia, odczyt z urządzenia wejścia, potwierdzenie przesłania, zapotrzebowanie na magistralę, rezygnacja z magistrali, żądanie przerwania, potwierdzenie przerwania, sygnał synchronizacyjny, przywracanie stanu początkowego magistrali.

1.1.4.30. **Wyjaśnienie**. Działanie magistrali jest następujące:

- Jeśli jeden z modułów zamierza przesłać dane do drugiego, to musi wykonać dwie czynności – (1) uzyskać dostęp do magistrali i (2) przekazać dane za pośrednictwem magistrali.
- Jeżeli natomiast zamierza uzyskać dane z innego modułu, to musi – (1) uzyskać dostęp do magistrali, (2) przekazać zapotrzebowanie do tego modułu i (3) czekać, aż wskazany moduł dokona transmisji.

1.1.4.40. **Wyjaśnienie**. *Sterowanie magistrali*, może być scentralizowane lub zdecentralizowane. W przypadku sterowania scentralizowanego istnieje jedno urządzenie sterujące, zwane *arbitrem*, które podejmuje decyzję, na podstawie przekazanych sygnałów od poszczególnych modułów i ustalonych priorytetów. W układzie rozproszonego sterowania centralne urządzenie sterujące nie występuje, każdy moduł komputera zawiera układ logiczny sterujący dostępem – współpracujące pomiędzy sobą za pośrednictwem magistrali.

Dotychczas rozważaliśmy transmisje pomiędzy modułami, np. pomiędzy pamięcią RAM oraz procesorem, w zasadzie jako sterowane bezpośrednio przez procesor. Podobnie procesor może czytać dane z modułu wejścia i przekazywać dane do modułu wyjścia. W pewnych przypadkach,

potrzebne jest sterowanie transmisją minimalnie angażujące czasowo procesor. W takich przypadkach korzystamy z układu DMA.

1.1.4.50. **Wyjaśnienie**. *Układ DMA* (*Direct Memory Access*), umożliwia transmisję pomiędzy modułami z ograniczeniem udziału procesora do zainicjowania transmisji, a następnie przejęcie sterowania transmisją. Dotyczy to transmisji bloków danych pomiędzy modułem pamięci RAM a modułami wejścia/wyjścia, transmisji pomiędzy poszczególnymi modułami pamięci RAM, transmisji pomiędzy pamięcią RAM a pamięciami masowymi.

Piśmiennictwo: *Wikipedia* W.2.17., *Mano M.* M.1.1., *Stallings W.* S.11.1., *Wojtuszkiewicz K.* W.6.1.

1.1.5. STEROWANIE MIKROPROGRAMOWE

Jeszcze w 1951 roku, *Maurice V. Wilkes* – opublikował pierwszą istotna innowację dotyczącą architektury Von Neumanna, a mianowicie mikroprogramowaną jednostkę sterującą komputera[12]. Wilkes zaproponował pewien rodzaj systematyki w projektowaniu jednostek sterujących, który był pewną „ciekawostką", ale nie mógł być - bezpośrednio realizowany, ze względu na możliwości ówczesnej techniki cyfrowej. W 1964 roku firma IBM zastosowała sterowanie mikroprogramowe, czyli sterowanie otwieraniem i zamykanie bramek logicznych – przez bity poszczególnych mikrooperacji, w nowo wprowadzonej na rynek serii komputerów System/360 IBM. Zamiast więc zaszycia w układach logicznych podstawowej pętli wykonywania poszczególnych instrukcji procesora - sterowania otwierania i zamykania poszczególnych bramek logicznych, zastosowano jak gdyby uproszczony wewnętrzy komputer, z kolei sterujący za pomocą wykonywania swojego wewnętrznego zapisanego w specjalnej pamięci stałej mikroprogramu. Ten wewnętrzny komputer realizuje pętle steriowania poszczególnych instrukcji komputera.

Notacja służąca do opisu mikroprogramów wygląda jak język programowania, którym jest w rzeczywistości. Każdy wiersz w tej notacji opisuje zbiór jednocześnie wykonywanych mikrooperacji i jest nazywany mikrorozkazem. Sekwencja mikrorozkazów jest nazywana mikroprogramem lub oprogramowaniem układowym (*firmware*). Oznacza to, że mikroprogram znajduje się pomiędzy sprzętem komputera – a jego oprogramowaniem.

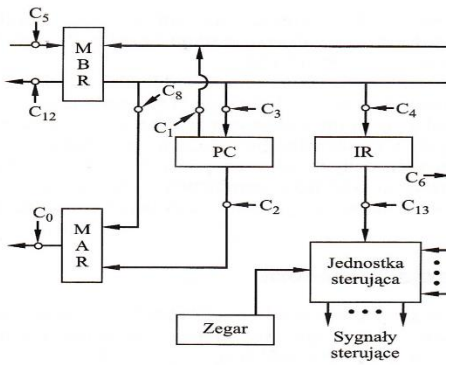
1.1.5.10. **Wyjaśnienie**. Cykl pobierania instrukcji do wykonania, jest cyklem poprzedzającym wykonanie pobranej instrukcji. Na rys. 1.1.5.00. pokazany jest fragment ścieżek danych z bramkami sterującymi przepływem danych. Bramki te, oznaczone są symbolami C_0, C_1, …, C_{13}. Aby wykonać cykl pobierania instrukcji: *w takcie 1* - należy otworzyć bramkę C_2, żeby przesłać zawartość rejestru PC do rejestru MAR; *w takcie 2* – należy otworzyć bramkę C_0, żeby wybrać z pamięci RAM zawartość słowa o adresie pobranym z rejestru PC, jednocześnie powiększając zawartość rejestru PC o jeden; *w takcie 3* – należy wysłać sygnał *czytaj* do pamięci RAM i otworzyć bramkę C_5, celem przeczytania z pamięci zawartości słowa o adresie ustawionym w rejestrze MAR i umieszczenie odczytanego słowa w rejestrze MBR; *w takcie 4* – należy otworzyć bramki C_4, celem przesłania zawartości rejestru MBR do rejestru IR. Przedstawiony powyżej cykl pobierania instrukcji jest realizowany przez jedną mikroinstrukcję w rozumieniu Wilkesa. Wykonanie każdego z taktów trwa jeden lub więcej cykli zegara taktującego procesora.

[12] Rozwiązanie to, jest dzisiaj powszechnie stosowane.

Podobnie do cyklu pobierania instrukcji, można opisać każdy z cykli wykonywania instrukcji, jak np. cykl adresowania pośredniego, cykl przerwania, itd. Przejdźmy z kolei do omówienia zasad działania jednostki sterującej procesora.

1.1.5.20. **Wyjaśnienie.** Budowa mikrorozkazu może być dwojaka:

- *Mikrorozkaz wykonawczy*, którego rola polega na otwarciu wskazanych bramek, przez określona liczbę impulsów zegarowych. Mikrorozkaz wykonawczy składa się z trzech części: (1) bitu określającego czy jest to mikrorozkaz wykonawczy - 0, czy też sterujący; (2) bitów na których zapisana jest liczba impulsów zegarowych, czasu otwarcia wskazanych w trzeciej części mikrorozkazu bramek; (3) bity sterujące otwieraniem bramek, po jednym bicie na każdą bramkę (numer bitu koresponduje z numerem sterowanej bramki, 0 – mikrorozkaz nie otwiera danej bramki, 1 – mikrorozkaz otwiera daną bramkę).
- *Mikrorozkaz sterujący*, którego rola polega na przejście układu sterowania do nowej sekwencji mikrorozkazów w pamięci sterującej. Mikrorozkaz sterujący składa się z trzech części: (1) bitu określającego czy jest to mikrorozkaz wykonawczy, czy też sterujący - 1; (2) bitów badania flag (których numery przypisane są poszczególnym bitom, wartość bitu 0 - stwierdza, że dana flaga nie warunkuje wykonania mikrorozkazu, wartość bitu 1 – stwierdza, że wykonanie mikrorozkazu zależy od stanu flagi) uzależniających wykonanie mikrorozkazu od stanu flag (w przypadku gdy wszystkie bity, tej części mikrorozkazu, są równe 0, mikrorozkaz sterujący jest mikrorozkazem bezwarunkowym i jest zawsze wykonywany; podobnie jest w przypadku, gdy wszystkie bity, tej części mikrorozkazu są równe 1); (3) bity zawierające adres, który w przypadku wykonania danego mikrorozkazu zostanie umieszczony w rejestrze adresu sterowania mikroprogramowanej jednostki sterującej.



Rysunek 1.1.5.00.Fragment ścieżek danych z bramkami sterującymi przepływem danych.

1.1.5.30. **Wyjaśnienie.** Działanie jednostki sterującej procesora – można podzielić na kroki[13], z których część jest wykonywana wielokrotnie.

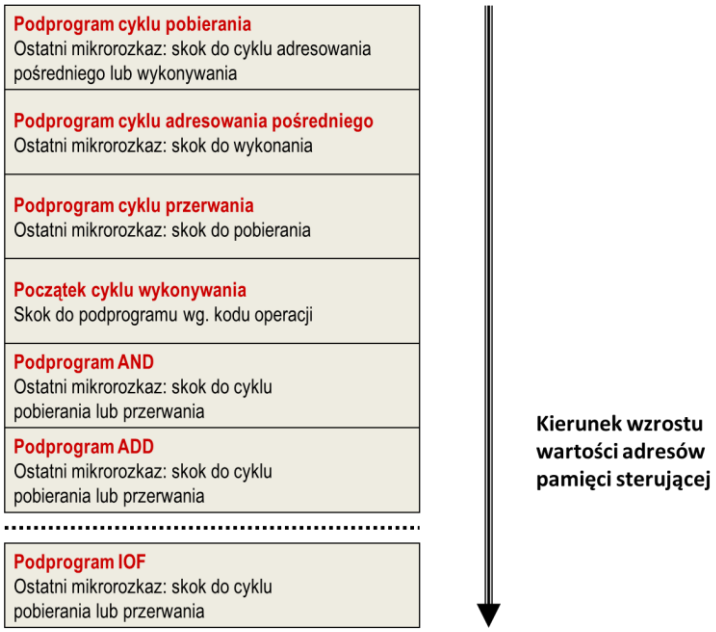
1. W celu wykonania rozkazu szeregująca jednostka logiczna wydaje rozkaz pamięci sterującej odczyt.
2. Słowo (mikrorozkaz), którego adres został umieszczony w rejestrze sterowania, jest wczytane do buforowego rejestru sterowania.
3. Zawartość rejestru buforowego sterowania generuje sygnały sterujące oraz przesyła część adresową słowa do logicznej jednostki szeregowania.
4. Logiczna jednostka szeregowania ładuje nowy adres do rejestru adresu mikro-sterowania na podstawie informacji o następnym adresie przesłanej z buforowego rejestru sterowania

[13] Każdy z kroków z kolei składa się z pewnej liczby taktów (patrz 1.1.5.10).

wraz z uwzględnieniem stanu flag. Zależnie od stanu flag i rodzaju mikrorozkazu (wykonawczy albo sterujący) – podejmowana jest jedna z czterech decyzji:

a) Jeśli mikrorozkaz wykonawczy - i wykonanie go, poprzez otwarcie wskazanych bramek na wskazaną liczbę cykli zegarowych, a następnie zamknięcie bramek i dodanie 1 do rejestru adresu mikro-sterowania oraz przejście do następnego mikrorozkazu w danej sekwencji sterowania.
b) Jeśli mikrorozkaz skokowy, w którym wszystkie bity flag, są równe 0 - skok bezwarunkowy do następnego mikroprogramu standardowego w wyniku mikrorozkazu skokowego. Ładowanie pola adresu mikrorozkazu do rejestru adresu mikro-sterowania.
c) Jeśli mikrorozkaz skokowy, w którym wybrane bity flag – odpowiadające warunkom spełnionym, czyli równym 1 (a pozostałe równe 0) - skok warunkowy wykonywany do następnego mikroprogramu; w przypadku nie spełniania warunku – nie wykonanie skoku warunkowego, przejście do następnego mikrorozkazu po dodaniu 1 do rejestru adresu mikro-sterowania.

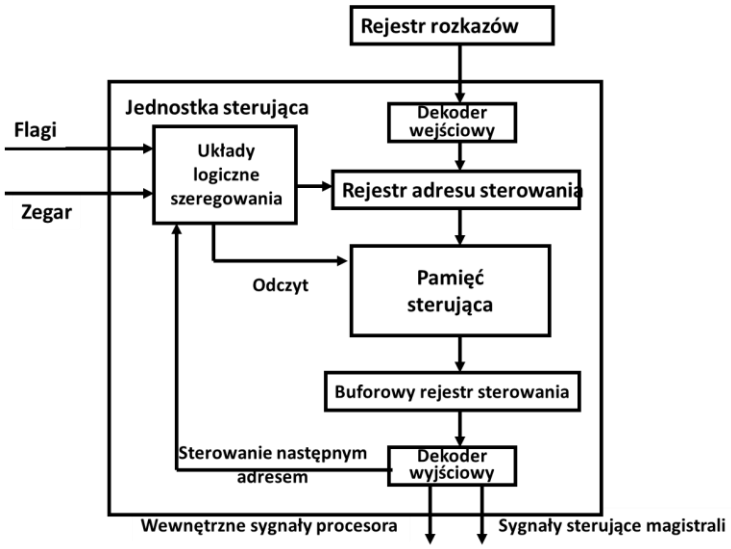
5. Jeśli mikrorozkaz skokowy, w którym wszystkie bity flag, są równe 1 - skok bezwarunkowy do programu komputera, w tym celu – pobranie następnej instrukcji wg zawartości rejestru (PC).



Rysunek 1.1.5.02. Organizacja pamięci sterującej

1.1.5.40. **Wyjaśnienie.** Zalety i wady sterowania mikroprogramowego:

1) Główną zaletą mikroprogramowanej jednostki sterującej jest istotne uproszczenie projektowania logicznego.
2) Rozwiązanie mikroprogramowe jest zarówno tańsze, jak i bardziej odporne na błędy.
3) Układowa jednostka sterująca musi zawierać skomplikowane układy logiczne w celu zapewnienia szeregowania wielu mikrooperacji w cyklu rozkazu.
4) Dekodery i logiczna jednostka szeregowania mikroprogramowej jednostki sterującej – są bardzo prostymi układami kombinacyjnymi.
5) Główną wadą mikroprogramowej jednostki sterującej jest to, że jest ona nieco wolniejsza od jednostki układowej wykonanej w równorzędnej technologii.
6) Mimo powyższej wady, we współczesnych procesorach dominują mikroprogramowe jednostki sterujące.



Rysunek 1.1.5.02. Schemat ideowy mikroprogramowanej jednostki sterującej.

Piśmiennictwo: *Mano M.* M.1.1., *Stallings W.* S.1`.1.

1.1.6. SYSTEM WIELOPOZIOMOWY PAMIĘCI

1.1.6.10. **Wyjaśnienie.** System wielopoziomowy pamięci komputerowa to różnego rodzaju urządzenia i bloki funkcjonalne komputera, służące do przechowywania danych i programów (systemu operacyjnego oraz aplikacji). Potocznie przez „pamięć komputerową" rozumie się samą pamięć operacyjną czyli pamięć RAM. System wielopoziomowy pamięci składa się z kilku poziomów pamięci tworzących strukturę hierarchiczną (patrz rys. 1.1.6.00). Każdy poziom charakteryzuje się inną szybkością działania i inną pojemnością.

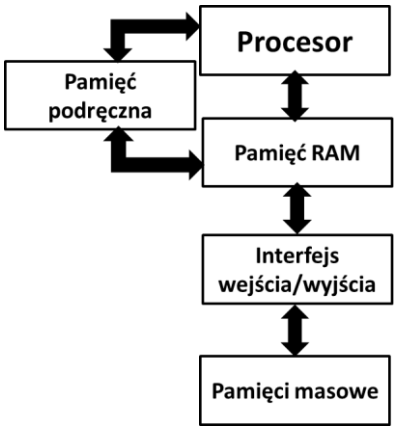
1.1.6.20. **Wyjaśnienie.** RAM (*Random Access Memory* – pamięć operacyjna, czyli pamięć o dostępie swobodnym) – podstawowy rodzaj pamięci komputera. Ze względów historycznych określa ona tylko te rodzaje pamięci o bezpośrednim dostępie, w których możliwy jest wielokrotny i łatwy zapis. W pamięci RAM przechowywane są aktualnie wykonywane programy i dane dla tych programów oraz wyniki ich działania. Pamięci RAM dzieli się na pamięci statyczne (*Static RAM*- w skrócie SRAM) oraz pamięci dynamiczne (*Dynamic RAM* - w skrócie DRAM). Pamięci statyczne są szybsze od pamięci dynamicznych, które wymagają ponadto częstego odświeżania, bez którego szybko tracą swoją zawartość. Pomimo swoich zalet są one jednak dużo droższe; używane są w układach, gdzie wymagana jest duża szybkość (np. pamięć podręczna procesora). W komputerach super-skalarnych jako pamięć operacyjną używa się pamięci DRAM.

1.1.6.30. **Wyjaśnienie.** Pamięć masowa (*mass storage memory*), jest to pamięć trwała, przeznaczona do długotrwałego przechowywania dużych ilości danych. Pamięć masowa, w odróżnieniu od pamięci RAM, nie adresuje pojedynczych bajtów oraz ma wydłużony czas dostępu. Wyróżniamy dwie podstawowe klasy pamięci masowych, ze względu na sposób dostępu:

- *Sekwencyjny* - pamięć masowa zorganizowana jest za pomocą jednostek danych zwanych rekordami, a dostęp (zapis i odczyt) do wymaganej komórki pamięci wg kolejności rekordów;
- *Bezpośredni* – pamięć masowa z dostępem do poszczególnych rekordów.

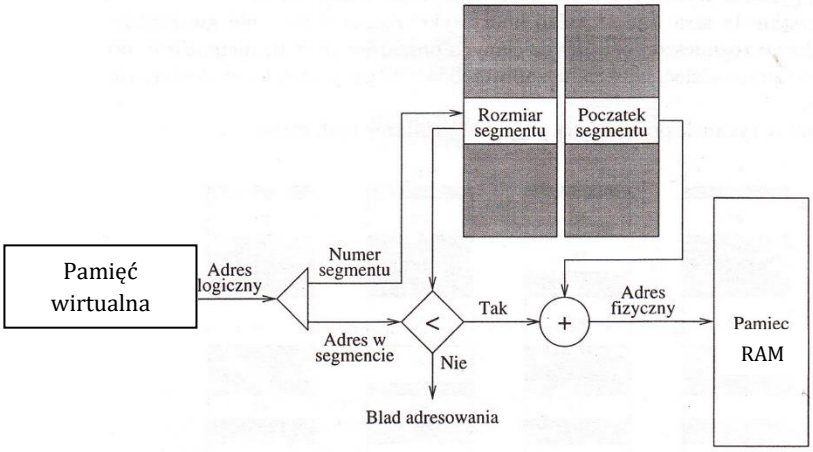
Ponadto można wyróżnić następujące rodzaje pamięci masowej, biorąc pod uwagę rodzaj zastosowanego nośnika danych:

- Nośnik magnetyczny: (a) dysk twardy – pamięć magnetyczna dyskowa; (b) dyskietka – pamięć magnetyczna dyskietkowa; (c) pamięć taśmowa – pamięć magnetyczna taśmowa.
- Nośnik optyczny: płyta (CD-R, CD-RW, CD-ROM, DVD, Blu-ray, HD DVD) – pamięć zapisywana i odczytywana w napędzie optycznym zgodnego z tym typem płyty.
- Pamięć półprzewodnikowa: (a) karty pamięci (wszelkie rodzaje wymiennych kart pamięci); (b) pamięć SSD (tzw. dysk SSD, następca dysku twardego); (c) pamięci USB (tzw. *pendrive*).



Rysunek 1.1.6.00. Położenie pamięci RAM względem procesora i interfejsów wejścia/wyjścia.

1.1.6.40. **Wyjaśnienie.** Pamięć wirtualna, mechanizm zarządzania pamięcią komputera zapewniający procesowi wrażenie pracy w jednym dużym, ciągłym obszarze pamięci operacyjnej podczas, gdy fizycznie może być ona pofragmentowana, nieciągła i częściowo przechowywana na urządzeniach pamięci masowej. Systemy korzystające z tej techniki poprawiają wykorzystanie fizycznej pamięci RAM w systemach wielozadaniowych. Rozszerzenie pamięci na dyski twarde w rzeczywistości jest tylko naturalną konsekwencją zastosowania techniki pamięci wirtualnej. Pamięć wirtualna działa na zasadzie przedefiniowania adresów pamięci (fizycznych) na adresy używane przez procesy (logiczne) tak, aby „oszukać" procesy i dać im wrażenie pracy w ciągłej przestrzeni adresowej.



Rysunek 1.1.6.01. Tłumaczenie adresu logicznego na adres fizyczny przy segmentacji przez MMU.

1.1.6.41. **Wyjaśnienie.** TLB (*Translation Lookaside Buffer*) jest sprzętowym podzespołem mechanizmu zarządzania pamięcią wirtualną. Jest to bufor pamięci - typu asocjacyjna pamięć

podręczna (*cache*), który zawiera fragmenty tablicy stron pamięci wirtualnej komputera. TLB posiada stałą liczbę wpisów i służy do szybkiego odwzorowywania adresów logicznych pamięci wirtualnej na adresy RAM (pamięci fizycznej).

1.1.6.50. **Wyjaśnienie.** Jednostka zarządzania pamięcią, inaczej MMU (*Memory Management Unit*) – zestaw układów realizujących dostęp do pamięci fizycznej żądanej przez procesor. Wśród zadań tych układów znajdują się funkcje translacji adresów logicznych pamięci wirtualnej na adresy fizyczne pamięci RAM, ochronę pamięci, obsługę pamięci podręcznej (*cache*), zarządzanie magistralami danych. Mechanizm dynamicznego odwzorowania adresów MMU (*Memory Management Unit*) – umożliwił również użycie wektorowej grafiki komputerowej, w postaci z jaką mamy współcześnie do czynienia.

Piśmiennictwo: *Mano M.* M.1.1., *Stallings W.* S.11.1., *Wikipedia* W.2.15., W.2.20., W.2.21., W.2.22.

1.1.7. Arytmetyka FP

Klasyczna logika pierwszej połowy XX wieku poświęcała wiele miejsca na badanie podstaw liczb rzeczywistych. Arytmetyka zmiennopozycyjna (*Floating-Point*) jest przybliżeniem arytmetyki liczb rzeczywistych[14] i trzeba to wyraźnie powiedzieć, że powstała pod wpływem ograniczonych możliwości, wynikającej z narzuconej długości słowa, jakim operują komputery. Liczb rzeczywistych jest nieskończenie wiele, natomiast zapis zmiennopozycyjny wykorzystuje skończoną liczbę bitów, a za tym, może zapisać tylko skończoną liczbę wartości. Korzystając z arytmetyki zmiennopozycyjnej wykonujemy działania na przybliżeniach liczb rzeczywistych. Działania na przybliżeniach liczb rzeczywistych charakteryzują się pewnymi własnościami – innymi niż dla liczb rzeczywistych. Na przykład: liczbie zero w arytmetyce zmiennopozycyjnej odpowiada przedział wartości liczb rzeczywistych z otoczenia liczby rzeczywistego zera; kolejność wykonywania działań na liczbach zmiennopozycyjnych może mieć wpływ na dokładność wyniku, a tym samym na wartość wyniku. Pod wpływem informatyki, współczesna logika zainteresowała się nieciągłymi zbiorami liczb, np. określonych na tzw. kratownicach. Nie mniej jednak, trudno byłoby zakwestionować fakt, że źródłem powstania arytmetyki liczb zmiennopozycyjnych są potrzeby obliczeniowe, a nie rozważania teoretyczne dotyczące podstaw matematyki.

1.1.7.10. **Wyjaśnienie**. Liczba zmiennopozycyjna jest reprezentowana przez parę liczb opatrzonych znakami – zwanych odpowiednio mantysą i cechą. Wartość liczby zmiennopozycyjnej wyznaczana jest według wzoru:

$$liczba\ zmiennopozycyjna = mantysa \times e^{\text{cecha}};$$

Zapis zmiennopozycyjny z podstawą $e = 10$, jest przecież powszechnie stosowany przez fizyków i astronomów; np. $7{,}234 \times 10^{-12}$.

IEEE[15] przyjął trzy standardy formatów liczb zmiennopozycyjnych, przy podstawie $e = 2$. Są to mianowicie:

1. *Format zmiennopozycyjny pojedynczej precyzji* [24 – bitowa mantysa (bity $0 \div 22$, znak mantysy bit 31) i 8 – bitowy cecha (bity $23 \div 30$, z przesunięciem 127)];
2. *Format zmiennopozycyjny o podwójnej precyzji* [53 – bitowa mantysa (bity $0 \div 53$, znak mantysy bit 63) i 11 – bitowy cecha (bity $53 \div 62$, z przesunięciem 1023)];

[14] Operacje zmiennopozycyjne, są wykonywane przez dodatkową jednostkę arytmetyczną procesora, zwaną FPA Unit (*Floating-Point Arithmetic Unit*).
[15] The Institute of Electrical and Electronics Engineering (Instytut Inżynierii Elektrycznej i Elektronicznej).

3. *Format zmiennopozycyjny rozszerzony* (65 – bitów mantysa i 15 – bitowy cecha) – używany np. tylko w rejestrach procesora. Dwanaście dodatkowych (w stosunku do formatu podwójnej precyzji) bitów mantysy nazywamy bitami kontrolnymi.

1.1.7.11. **Wyjaśnienie** *format zmiennopozycyjny pojedynczej precyzji*. W przypadku 24 bitowej mantysy otrzymujemy dokładność 6½ cyfry dziesiętnej. Zaś 8 bitowa cecha – wykładnik z przedziału $10^{-38} \div 10^{+38}$.

1.1.7.12. **Wyjaśnienie** *format zmiennopozycyjny o podwójnej precyzji*. W przypadku 53 bitowej mantysy otrzymujemy dokładność 14½ cyfry dziesiętnej. Zaś 11 bitowa cecha – wykładnik z przedziału $10^{-308} \div 10^{+308}$.

Standard działań na liczbach zmiennopozycyjnych IEEE obejmuje ponadto: normalizację i wartości nienormalizowane; zaokrąglanie; specjalne wartości zmiennopozycyjne; wyjątki obliczeń zmiennopozycyjnych i działania na liczbach zmiennopozycyjnych.

1.1.7.13. **Wyjaśnienie** *przybliżenie zera*. Szerokość przedziału liczb rzeczywistych – odpowiadającego wartości zera zmiennopozycyjnego zależy od formatu. Najszerszy przedział odpowiada formatowi pojedynczej precyzji ($-10^{-38} \div +10^{-38}$). Odpowiednio węższe przedziały odpowiadają formatom: podwójnej precyzji i rozszerzonemu.

1.1.7.14. **Wyjaśnienie** *normalizacja i wartości nieznormalizowane*. Znormalizowana wartość zmiennopozycyjna to taka, w której najstarszy bit mantysy zawiera jedynkę. Przechowywanie liczb zmiennopozycyjnych w postaci znormalizowanej zapewnia maksymalną liczbę bitów dokładności, a wyniki obliczeń są dokładniejsze, jeśli wykonujemy działania tylko na liczbach znormalizowanych. Niemal wszystkie liczby nieznormalizowane można znormalizować, są jedynie dwa przypadki, kiedy normalizacja nie jest możliwa:
1. Kiedy liczba jest równa zero, ponieważ wszystkie bity mantysy są równe zero.
2. W sytuacji, kiedy w mantysie najstarsze bity są równe zero, a jednocześnie cecha osiągnęła minimalną wartość.

1.1.7.15. **Wyjaśnienie** *zaokrąglenie*. Jeśli procesor działa na formacie rozszerzonym, to przy zapisywaniu wyników z rejestrów procesora do komórek pamięci zachodzi konieczność zaokrąglania wyniku. Zaokrąglanie powoduje pozostawienie wartości bez zmian, jeśli wszystkie bity kontrolne są zerami. Jeśli jednak bity kontrolne, zawierają – chociaż jedną jedynkę, to dodanie jedynki najwyższego bitu kontrolnego, jeśli jest on równy jeden, spowoduje przeniesienie jedynki na ostatnią, najniższą pozycję mantysy.

1.1.7.16. **Wyjaśnienie** *specjalne wartości zmiennopozycyjne*. Jeśli cecha liczby zmiennopozycyjnej składa się z samych jedynek, to mamy do czynienia z tak zwaną nie-liczbą, czyli NaN, która oznacza, że wynik wykonanej operacji jest nieokreślony lub że operacja została wykonana nieprawidłowo. W szczególności NaN powstaje w wyniku: próby dzielenia przez zero, próba dzielenia przez tzw. nieskończoność (liczbę z poza przedziału liczb zmiennopozycyjnych), odejmowania lub dodawania do liczby zmiennopozycyjnej tzw. nieskończoności.

1.1.7.17. **Uwaga**. *Tzw. nieskończoność*, nie zawsze jest nieskończonością w rozumieniu liczb rzeczywistych. Każda liczba z poza przedziału wartości liczb zmiennopozycyjnych, jest tzw. nieskończonością.

1.1.7.18. **Wyjaśnienie.** Kolejność wykonywania działań wpływa na dokładność wyniku. Odejmowanie od siebie liczb zmiennopozycyjnych o bliskich wartościach powoduje stratę dokładności. Działania mnożenia i dzielenia powodują stosunkowo szybką kumulację błędów zaokrągleń. Dlatego też kolejność wykonywania działań na liczbach zmiennopozycyjnych może mieć istotny wpływ na dokładność wyniku.

Piśmiennictwo: *Hyde R.* H.2.1.

1.1.8. Potok

Rozwój technologii elektronowej (przejście od lamp elektronowych do dyskretnych półprzewodników, a następnie do coraz wyższego stopnia integracji w obwodach scalonych) - spowodował wbudowywanie w komputery coraz bardziej złożonej funkcjonalności. W latach siedemdziesiątych ubiegłego wieku, powstały dwa kierunki rozwoju architektury komputerów. A mianowicie:

1. *Architektura CISC* (*Complex Instruction Set Computer*) – zakładająca tworzenie komputerów o coraz bardziej rozbudowanej liście rozkazów oraz zmiennej długości tych ostatnich tak, aby maksymalnie przybliżyć listę rozkazów komputera do złożonych wyrażeń języków programowania wysokiego rzędu. Architektura CISC stała się typową w tym okresie, dla tzw. komputerów *mainframe*. Potokowość, o której mowa dalej – była wykorzystywana w architekturze niektórych bardzo silnych komputerów mainframe.
2. *Architektura RISC* (*Reduced Instruction Set Computer*) – zakładająca tworzenie komputerów o uproszczonej liście rozkazów i rozkazach o stałej długości oraz znacznie większej liczbie rejestrów - dostępnych dla programisty. Architektura RISC stałą się dominującą w tym okresie, wśród minikomputerów. Potokowość, o której mowa dalej – pojawiła się w architekturze niektórych komputerów RISC.

W latach osiemdziesiątych ubiegłego wieku, połączono te dwa historycznie różne trendy architektoniczne komputerów i pojawiły się pierwsze procesory wykonane w technologii wielkiej, a później bardzo wielkiej skali integracji, które posiadały zalety zarówno architektury CISC, jak i RISC.

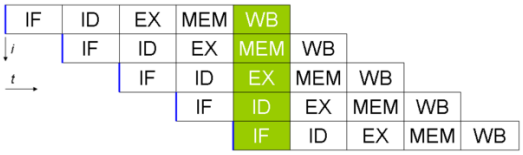
1.1.8.10. **Wyjaśnienie.** Potokowość (*pipelining*) – technika budowy procesorów polegająca na podziale logiki procesora odpowiedzialnej za proces wykonywania sekwencji instrukcji procesora (programu) na specjalizowane grupy kroków w taki sposób, aby każda z grup wykonywała część pracy związanej z wykonaniem kolejnego kroku każdej instrukcji. Grupy kroków, są połączone w potok (*pipe*) – i są realizowane równocześnie, pobierając dane od poprzedniego elementu sekwencji. W każdej z tych grup instrukcja jest na innym stadium (kroku) wykonania. Można to porównać do taśmy produkcyjnej.

1.1.8.20. **Wyjaśnienie.** W uproszczeniu, potok wykonania instrukcji procesora (patrz rys. 1.1.8.01) może wyglądać następująco[16]:

1. Pobranie instrukcji z pamięci – *instruction fetch* (IF)
2. Zdekodowanie instrukcji – *instruction decode* (ID)
3. Wykonanie instrukcji – *execute* (EX)
4. Dostęp do pamięci – *memory access* (MEM)
5. Zapisanie wyników działania instrukcji – *write back* (WB).

[16] Porównaj – podrozdział 0.1.2. Stos i układ obsługi przerwań.

W powyższym pięcio-stopniowym potoku, przejście przez wszystkie stopnie potoku (wykonanie jednej instrukcji) zabiera co najmniej 5 cykli zegarowych. Jednak ze względu na jednoczesną pracę wszystkich stopni potoku, jednocześnie wykonywanych jest 5 rozkazów procesora, każdy w innym stadium wykonania. Oznacza to, że taki procesor w każdym cyklu zegara rozpoczyna i kończy wykonanie jednej instrukcji i statystycznie wykonuje rozkaz w jednym cyklu zegara. Tak więc dodatkowe zwiększenie liczby stopni umożliwia osiągnięcie coraz wyższych częstotliwości pracy. Realizacja sprzętowa potoku, wymaga istotnego zwiększenia liczby rejestrów procesora, dla przechowywania danych i wyników pośrednich - kroków, każdej wykonywanej instrukcji.



Rysunek 1.1.8.01. Uproszczony schemat potokowości (zielone wykonywane równocześnie)

1.1.8.30. **Wyjaśnienie.** Podstawowym mankamentem techniki potoku są instrukcje skoku, powodujące w najgorszym wypadku potrzebę wycofania rozkazów, które następowały zaraz po instrukcji skoku i rozpoczęcie zapełniania potoku od początku (czyli od adresu, do którego następował skok) Wykonanie instrukcji skoku może powodować ogromne opóźnienia w wykonywaniu programu – tym większe, im większa jest długość potoku. Dodatkowo szacuje się, że taki skok występuje co kilkanaście rozkazów. Z tego powodu, niekiedy zmienia się kolejność wykonania instrukcji (jeśli to nie spowoduje zmiany wyniku działania programu) oraz wstawia się w wybranych miejscach dodatkową instrukcję *do nothing*, czyli nic nie rób.[17]

Piśmiennictwo: *Stallings* W. S.11.1., *Wikipedia* W.2.26.

1.1.9. Procesor wielordzeniowy

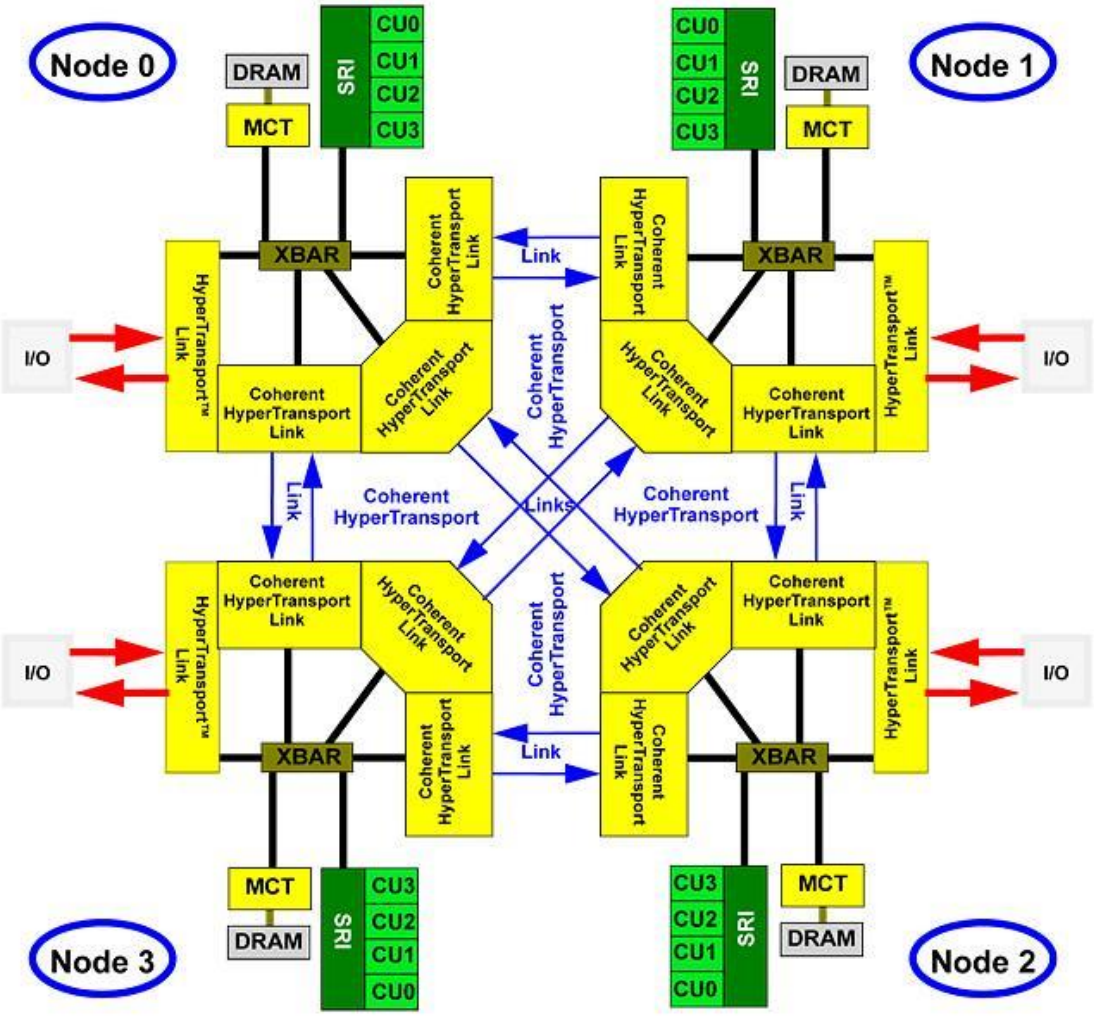
1.1.9.10. **Wyjaśnienie.** Rdzeń procesora (*core*) jest to układ w procesorze odpowiedzialny za obliczenia. Oprócz rdzenia w skład procesora wchodzą jeszcze inne urządzenia jak np. pamięć cache, czy też magistrala (złożona z części: danych, adresowej i sterowania) do komunikacji procesora z pozostałymi modułami komputera oraz zestaw rejestrów umożliwiający wykonywanie instrukcji w reżimie dwu lub więcej potokowym. W roku 2005 producenci procesorów zakończyli wojnę o wzrost mocy obliczeniowej poprzez zwiększanie szybkości taktowania, zamiast tego zaczęli pakować do jednego procesora kilka rdzeni. Teoretycznie powinno to zapewniać podwojenie wydajności, w praktyce doszło jednak do tzw. wielordzeniowego kryzysu.

1.1.9.20. **Wyjaśnienie.** Procesor wielordzeniowy (*multi-core processor*) – rodzaj procesora posiadający więcej niż jeden rdzeń. Technologia ta ma na celu zwiększenie wydajności procesora, zmniejszenie zużycia energii i bardziej efektywnego jednoczesnego przetwarzania wielu zadań.

1.1.9.30. **Wyjaśnienie.** Najpopularniejszymi tego typu procesorami są jednostki dwurdzeniowe. Firma Intel opracowała *technologię wielowątkowości*, nazwaną *Hyper-Threading*, tzn. jeden rdzeń emulujący działanie dwóch rdzeni. Procesory wielordzeniowe mają zwielokrotnioną, w zależności od liczby rdzeni, pamięć podręczną.

[17] Stosuje się także skomplikowane metody predykcji skoku.

Aby zrozumieć problem efektywnego wykorzystania możliwości procesora wielordzeniowego, musimy przybliżyć pojęcie jakim jest wielowątkowość w systemach operacyjnych. Każda aplikacja czy gra - może posiadać kilka, równolegle wykonujących się instrukcji zlokalizowanych w różnych wątkach. Oczywiście każdy rdzeń procesora wykonuje tylko jedną instrukcję na raz, jednakże z punktu widzenia samego programu, jak długo poszczególne wątki mogą być wykonywane niezależnie, procesor wykorzystuje kilka rdzeni równocześnie.



Rysunek 1.1.9.10. Schemat ideowy 16 rdzeniowego 64 bitowego procesora firmy AMD.

Każdy program posiada co najmniej jeden wątek, tzw. wątek główny. Jeśli mamy do czynienia z aplikacją okienkową, z reguł wątek ten odpowiada za interakcję z użytkownikiem. Dzięki temu, po kliknięciu przycisku nasz program reaguje, coś wykonuje, innymi słowy działa. Na pewno znana jest każdemu użytkownikowi sytuacja - w której obciążona aplikacja przestaje odpowiadać na nasze działania, okna nie daje się przeciągnąć, jego zawartość się nie odświeża. Dzieję się tak gdy program zaczyna wykonywać obliczenia w wątku głównym. W tym stanie program nie może przetwarzać komunikatów i dlatego nie odpowiada na nasze działania. Idealną sytuacją jest, gdy skomplikowane obliczenia aplikacja wykonuje w wątku pobocznym. Wtedy to, interfejs pozostaje „przy życiu" a my mamy kontrolę nad programem. Zastosowanie wątków pobocznych wiąże się z problemami synchronizacji danych programu, dlatego też nie jest to powszechna praktyka. Skoro mamy poprawnie napisany program, z wątkiem pobocznym,

to dlaczego nie działa on 2x szybciej na maszynie dwurdzeniowej? Z tego prostego powodu, iż wątek główny zajmuje minimalny czas procesora (gdyż nie ma nic do roboty poza reagowaniem na akcje użytkownika), a poboczny w którym aplikacja wykonuje obliczenia maksymalny dostępny. Procesor nie może sobie tego jednego wątku rozbić na dwa rdzenie gdyż wynik działania danego rozkazu, zależny jest od wyniku rozkazu poprzedniego. W praktyce wygląda to tak że rozkazy są wykonywane naprzemiennie na każdym z rdzeni obciążając je po ok 50%. Nie mniej, jest to wykorzystywanie jedynie jednego rdzenia w całości.

Rozwiązaniem problemu jest zastosowanie dwóch lub większej liczby wątków wykonujących obliczenia, ale co ważne, tak aby rozkład pracy był rozłożony równomiernie pomiędzy nie. W takiej sytuacji, gdy 2 wątki są obciążone, 2 rdzenie procesora pracują na pełnych obrotach. Procesory 4 rdzeniowe nie są już dziś czymś nadzwyczajnym, ostatnio doczekaliśmy się już 16 rdzeniowych jednostek. W zależności od tego na ile wątków problem został rozbity, na tyle program będzie szybciej działał na coraz to większej ilości rdzeni. Oczywiście nie można popadać w skrajności, nie ma sensu dzielić operacji która trwa 1 sekundę na 16 wątków tylko po to by na odpowiednio wydajnym komputerze trwało to 16x krócej. Sama inicjacja tylu wątków w systemie pochłonęła by w takiej sytuacji więcej czasu. Zadania które trwają dostatecznie krótko, wykonuje się w wątku głównym, gdyż użytkownik i tak nie zdążył by w tym czasie zrobić niczego innego z oknem programu. Gorzej jeśli niespodziewanie operacja się wydłuży czy wpadnie w nieskończoną pętle.

Procesory składające się z wielu rdzeni to nie nowość ostatnich lat. Już dawno temu twórcy komputerów zauważyli, że wielordzeniowość zwiększa wydajność obliczeniową układu, pozwalając na jednoczesne przeprowadzanie wielu obliczeń. Jednak ze względu na możliwości konstrukcyjne i koszty systemy wielordzeniowe realizowane były głównie w postaci zestawu komputerów z procesorami jednordzeniowymi, połączonych ze sobą szybką siecią wymiany danych. Ich zastosowaniem były głównie serwery w centrach naukowych i obliczeniowych, w których niezbędna jest duża moc obliczeniowa.

Piśmiennictwo: *Wikipedia* W.2.3., W.2.14., W.2.25.

1.2. SIECI KOMPUTEROWE

1.2.1. PRZEŁĄCZANIE PAKIETÓW

Obok rozwoju komputerów rozwinięto począwszy od 1950 roku szereg aplikacji opartych o systemy komputerowe. Jedną z najważniejszych, jak to dzisiaj widać, są sieci komputerowe, a w szczególności globalna sieć komputerowa zwana Internetem, który posiada niemały wpływ na współczesne życie ludzkości.

Wszystko zaczęło się na początku lat 60 XX wieku. Po udanych próbach z bronią termojądrową w ZSRR zaczęto w USA oraz Wielkiej Brytanii szukać rozwiązań alternatywnych dla tradycyjnej koncepcji łączności telefonicznej opartej o komutację łączy. W 1962 roku w USA (dwa niezależne zespoły) i Wielkiej Brytanii podjęto prace nad koncepcję przełączania pakietów. W 1964 roku opracowano zastosowanie technologii przełączania pakietów w łączności głosowej dla potrzeb wojskowych. W roku 1969 US-DARPA (*Defense Advance Research Projects Agency*) sfinansowała program badawczy ARPAnet - prac nad tworzeniem sieci z komutacją pakietów. W ramach projektu opracowano pierwsze przełączniki pakietów – zwane procesorami IMP (*Interface Message Processor*), wykonane przez firmę BBN. 1 maja 1969 roku na UCLA

zainstalowano pierwszy procesor IMP. Wkrótce dalsze procesory zainstalowano w instytucie SRI, w UC Santa Barbara i Uniwersytecie Utah (sieć ARPAnet w 1969 roku liczyła cztery węzły). W 1972 roku sieć ARPAnet liczyła 15 węzłów. W 1972 roku opracowano pierwszy protokół komunikacyjny NCP oraz pierwszy program poczty elektronicznej bazujący na tym protokole komunikacyjnym. Pierwotnie sieć ARPAnet była pojedynczą zamkniętą siecią.

Sieć o nazwie ARPAnet została zbudowana z myślą o badaniach nad sieciami o wysokiej wydajności, niezawodności oraz niezależności od rozwiązań oferowanych przez poszczególnych producentów sprzętu komputerowego i oprogramowania. Stosowane obecnie rozwiązania w sieciach rozległych - w większości wywodzą się od rozwiązań opracowanych w ramach programu badawczego ARPAnet. Eksperyment z siecią ARPAnet – przerodził się w wykorzystywanie sieci do normalnego eksploatacyjnego przekazywania danych przez organizacje, które początkowo dla potrzeb eksperymentu dołączyły się do tej sieci. W tej sytuacji w roku 1975 ARPAnet została przekwalifikowana na sieć eksploatacyjną, a administracja siecią została przekazana do DCA (*Defense Communication Agency*). Prace nad rozwojem sieci były dalej prowadzone. Wcześniej opracowane wersje podstawowych protokołów TCP/IP – były dalej rozwijane i doskonalone. Lata 1970 – 1980 to okres powstania szeregu zamkniętych sieci komputerowych dedykowanych różnym zastosowaniom: ALOHAnet, Telenet (BBN), Cyklades (Francja), Tymnet Services, GE Information Services, SNA (IBM)]. W roku 1983 protokóły TCP/IP zostały przyjęte jako *Military Standards* (w skrócie MIL STD), przy czym wszystkie komputery dołączone do sieci musiały już stosować protokoły TCP/IP. Dla ułatwienia tej operacji DARPA ufundowała grand dla zespołu kierowanego przez Bolt'a, Beranka i Neman'a (tzw. zespół BBN) na Uniwersytecie Berkeley, celem dołączenia protokołów TCP/IP do systemu operacyjnego UNIX. Powstała w ten sposób wersja UNIX'a BSD, która została masowo wykorzystana przez amerykańskie uniwersytety. W 1983 roku ARPAnet został podzielony na dwie części. Mniejsza o nazwie MILNET podporządkowana została bezpośrednio Departamentowi Obrony USA oraz przeznaczona dla potrzeb sił zbrojnych. Pozostała część zachowała starą nazwę ARPAnet i została przeznaczona dla szerokiego użytkowania. Całość obu sieci, objęto wspólną nazwą Internet. Dalszy rozwój sieci, pozwolił na dołączenie wielu dalszych organizacji wraz z ich własnymi sieciami. Obecnie nazwą Internet - obejmuje się całość połączonych sieci działającej w oparciu o protokoły TCP/IP.

W roku 1989 zespół fizyków CERN kierowany przez *Tima Bernes-Lee* opracował koncepcję strony WWW i przeglądarki internetowej, wykorzystując koncepcję hipertekstu wywodzącą się jeszcze z 1945 roku i rozwijaną w latach 60 XX wieku. Prezentowanie wyników w postaci stron WWW rozszerzonych o informacje graficzne rozpowszechniło się po 1994 roku, początkowo na środowiska akademickie, a w kolejnych latach po 1995 na środowisko finansowe i szeroko rozumiane środowisko biznesowe.

1.2.1.01. **Wyjaśnienie**, czym jest sieć komputerowa. Sieć komputerowa składa się, z co najmniej dwóch komputerów – wzajemnie połączonych. Komputery sieci komunikują się pomiędzy sobą – wymieniając dane (informacje). Trzy podstawowe metody współpracy komputerów w sieci to: (1) Klient serwer; (2) Peer to Peer (P2P); (3) Rozgłaszanie.

1.2.1.02. **Wyjaśnienie**, czym jest Internet. Internet jest siecią sieci komputerowych, czyli siecią globalną złożoną z wielu lokalnych sieci komputerowych lub sieci obsługujących wydzielone organizacje (np. korporacje ponadnarodowe lub służby państwowe).

1.2.1.03. **Wyjaśnienie**, z czego składa się Internet. Internet składa się z sieci komputerowych stanowiących jego obrzeże, z sieci komputerowych dostawców usług internetowych ISP (*Internet Service Providers*) oraz tak zwanego rdzenia sieci. Ten ostatni będąc również siecią komputerową, zapewnia połączenia pomiędzy poszczególnymi ISP, którzy z kolei zapewniają łączność sieci lokalnych.

1.2.1.04. **Wyjaśnienie**. W skład sieci komputerowych składających się na Internet wchodzą dwa rodzaje urządzeń zwane odpowiednio: *hostami* (dzielonymi na klienty oraz serwery) i *routerami*. Urządzenia te, nazywamy węzłami sieci komputerowej.

1.2.1.05. **Wyjaśnienie**. Adresacja węzłów sieci Internet. Podstawowym adresem w sieci Internet jest adres IP, który w sposób jednoznaczny określa położenie węzła (routera lub hosta) w sieci Internet. Adres IP jest 32-bitowym słowem, opisanym w dokumencie - RFC 791. Zwykle jest reprezentowany w postaci czterech grup liczb dziesiętnych rozdzielonych kropkami. Np.: IP 127.0.0.1 lub IP 222.222.222.220. Stara konwencja adresacji zakładała, że część pierwsza adresu IP identyfikuje sieć. Wyróżniano wówczas trzy klasy sieci wchodzących w skład Internetu:

1. Klasa A, adresem sieci jest pierwszy bajt adresu IP;
2. Klasa B, adresem sieci są dwa pierwsze bajty adresu IP;
3. Klasa C, adresem sieci są trzy pierwsze bajty adresu IP.

Piśmiennictwo: *Kurose J.* K.8.1., *Sosińsky* B. S.9.1.

1.2.2. Piramida protokołów TCP/IP

1.2.2.01. **Wyjaśnienie**. Protokoły komunikacyjne są zbiorami reguł i procedur określających sposób:

- nawiązywania komunikacji pomiędzy węzłami sieci,
- przygotowania treści wiadomości do przesłania,
- sterowania przesłaniem wiadomości,
- interpretacji odebranej wiadomości,
- reagowania w sytuacjach awaryjnych.

1.2.2.02. **Wyjaśnienie**. *Piramida protokołów TCP/IP*. W ramach projektu ARPAnet, opracowano zestaw protokołów sieci również dla transmisji pakietowej – zwany TPC/IP. Piramida protokołów TPC/IP zawiera obecnie następujące warstwy:

1. *Warstwa aplikacji*, która zawiera protokoły, takie jak: wirtualny terminal TELNET, transfer plików FTP, poczta elektroniczna SMTP, mapowania nazw hostów na adresy sieciowe DNS, artykułów oraz grup dyskusyjnych NNTP, pobierania stron WWW – HTTP i wiele innych.
2. *Warstwa transportowa*, która zawiera dwa podstawowe protokoły TCP i UDP.
3. *Warstwa sieci* zawierająca podstawowy protokół IP, która umożliwia przekazywanie pakietów (w formie tzw. *datagramów*) pomiędzy routerami i hostami końcowymi.
4. *Warstwa dostępu do łącza danych*, która przekazuje pakiety (tzw. ramek) pomiędzy węzłami sieci.
5. *Warstwa fizyczna*, która jest odpowiedzialna za przekazywanie pojedynczych bitów ramek pomiędzy dwoma węzłami sieci.

1.2.2.10. **Wyjaśnienie**. Warstwa aplikacji. W warstwie aplikacji znajdują się aplikacje sieciowe i ich protokoły. Internetowa warstwa aplikacji korzysta z wielu protokołów:

- *HTTP* (obsługa pobierania stron internetowych);

- *SMTP* (obsługa wiadomości poczty elektronicznej);
- *FTP* (obsługa transferu plików);
- *Telnet* (obsługa zdalnej pracy za pośrednictwem sieci);
- *DNS* (obsługa translacji nazw internetowych użytkowników końcowych na 32-bitowe adresy sieciowe;
- *NFS* (Network File System – sieciowy system plików).

Protokoły warstwy aplikacji działają na systemach końcowych. Aplikacja z jednego systemu końcowego może korzystać z protokołu wymiany pakietów z innymi systemami końcowymi. Pakiety informacji w warstwie aplikacji nazywamy *komunikatami***.**

1.2.2.20. **Wyjaśnienie**. *Warstwa transportowa* przesyła komunikaty warstwy aplikacji pomiędzy klientem a serwerem. Dwa podstawowe protokoły warstwy to *TCP* i *UDP*. Oba są w stanie transportować komunikaty warstwy aplikacji. Protokół *TCP* zapewnia aplikacjom usługę transportu zorientowaną na połączenia. Usługa ta gwarantuje niezawodne dostarczenie komunikatu warstwy aplikacyjnej do miejsca jej przeznaczenia, a także oferuje kontrolę przepływu. Protokół *TCP* dzieli długie komunikaty na krótsze segmenty i zapewnia kontrolę przeciążenia sieci. Protokół *UDP* świadczy swoim aplikacjom usługę bezpołączeniową. Jest to usługa uproszczona, nie zapewnia niezawodności ani kontroli przepływu. Pakiet warstwy transportowej nazywamy *segmentem*.

1.2.2.30. **Wyjaśnienie**. Internetowa *warstwa sieci* jest odpowiedzialna za przesyłanie pakietów zwanych *datagramami* od jednego hosta do drugiego. Protokół internetowej warstwy transportowej (*TCP* lub *UDP*) źródłowego hosta przekazuje segment i adres docelowy warstwie sieci, tak jak nadawca zostawiający na poczcie list z adresem odbiorcy. Warstwa sieci oferuje następnie segmentowi usługę polegającą na dostarczeniu go do warstwy transportowej docelowego hosta. Internetowa warstwa sieci obejmuje protokół *IP*. Definiuje on pola datagramu, a także jak mogą być przetwarzane przez system końcowy i routery. Warstwa sieciowa obejmuje obok *IP* kilka protokołów routingu.

1.2.2.40. **Wyjaśnienie**. *Warstwa dostępu do łącza*. Internetowa warstwa sieci przesyła datagram za pośrednictwem serii routerów znajdujących się między źródłowym i docelowym węzłem. Aby przemieścić pakiet z jednego węzła (hosta lub routera) do kolejnego zlokalizowanego na trasie, warstwa sieci korzysta z usług warstwy łącza danych. Warstwa sieci każdego węzła przekazuje *datagram* niżej położonej warstwie łącza danych, która dostarcza go do kolejnego węzła znajdującego się na trasie, a następnie przemieszcza *datagram* do warstwy sieci. Pakiet warstwy łącza danych nazywany jest *ramką*.

1.2.2.50. **Wyjaśnienie**. *Warstwa fizyczna*. Zadaniem warstwy łącza danych jest przemieszczaniem całych ramek od jednego elementu sieci do kolejnego z nim sąsiadującego, natomiast rolą warstwy fizycznej jest przesyłanie poszczególnych bitów ramki pomiędzy dwoma węzłami sieci. Protokoły warstwy fizycznej są zależne od łącza, a także od rzeczywistej szybkości transmisji oferowanej przez nośnik łącza (na przykład skrętka lub światłowód jedno-modowy). Przykładowo, standard *Ethernet* korzysta z wielu protokołów warstwy fizycznej (poszczególne protokoły dotyczą: skrętki, kabla koncentrycznego cienkiego, kabla koncentrycznego grubego, światłowodów itd.). W każdym przypadku *bit* jest w inny sposób przesyłany łączem.

1.2.2.60. **Wyjaśnienie**. *Dlaczego TCP/IP?* Popularność protokołów *TCP/IP* wynika z utrafienia, we właściwym czasie w globalne zapotrzebowanie wymiany danych. Protokoły *TCP/IP* są stosem protokołów i mają pięciowarstwową strukturę. Każda z warstw stosu protokołów TCP/IP przy przekazywaniu danych w kierunku rosnącego numeru warstwy – dodaje informacje sterujące, między innymi w celu identyfikacji wybranego protokołu i numeru warstwy. Przy odwrotnym przekazywaniu danych w kierunku malejącego numeru warstwy – usuwane są informacje sterujące dotyczące numeru warstwy przekazującej dane. Informacje sterujące noszą nazwę nagłówków, gdyż są umieszczane na początku przekazywanych danych.

1.2.2.70. **Wyjaśnienie**. *Istotne cechy protokołów TCP/IP*. Standard otwartych protokołów, swobodnie dostępnych i opracowanych niezależnie od specyfiki sprzętu komputerowego lub systemu operacyjnego (protokoły *TCP/IP* są sprawnym narzędziem do łączenia różnorodnego sprzętu). Niezależność od fizycznych właściwości sieci, a tym samym protokoły *TCP/IP* umożliwiają łączenie wielu różnych typów sieci (np. sieci *Ethernet*, sieci *IBM Token Ring*, sieci wykorzystującej np. łącza telefoniczne, sieci *X.25*, itd.). Wspólny schemat adresacji, pozwalający dowolnemu urządzeniu korzystającemu z protokołów TCP/IP na jednoznaczne zaadresowanie innego urządzenia, w sieci tak rozległej jak światowy Internet. Standaryzowane protokoły wyższych poziomów zapewniających zgodność szeroko dostępnych usług.

Piśmiennictwo: *Kurose J.* K.8.1., *Sosińsky B*. S.9.1.

1.2.3. TRANSMISJA POŁĄCZENIOWA I BEZPOŁĄCZENIOWA

Dwie podstawowe metody transmisji w sieci działającej zgodnie z zasadami protokołów TCP/IP, to odpowiednio:

1. *Transmisja połączeniowa*, przy której działa sprzężenie zwrotne pomiędzy odbiorcą oraz dostawcą.
2. *Transmisja bezpołączeniowa*, przy której brak jest informacji zwrotnej od odbiorcy.

1.2.3.10. **Wyjaśnienie**. Transmisja połączeniowa zakłada: (1) Nawiązanie połączenia pomiędzy nadawcą i odbiorcą, poprzedzające rozpoczęcie transmisji; (2) Kolejne pakiety (porcje) danych wysyłanych przez nadawcę, po dokonaniu transmisji i poprawnym otrzymania ich przez odbiorcę, są podstawą do zwrotnego przesłania sygnału - potwierdzenia do dostawcy – o poprawnym odbiorze pakietu przez odbiorcę, jeśli jednak po czasie wystarczającym dla przesłania pakietu i sygnału zwrotnego, nadawca nie otrzyma sygnału zwrotnego, to zakłada, że pakiet nie dotarł do odbiorcy – nadawca ponownie wysyła do odbiorcy dany pakiet i oczekuje na sygnał potwierdzenia; (3) Proces wysyłania kolejnych pakietów i oczekiwania na sygnał zwrotny potwierdzenia przez odbiorcę jest powtarzany wielokrotnie, aż do przekazania całego komunikatu odbiorcy, w rozbiciu na serię pakietów danych; (4) Rozwiązanie ustanowionego połączenia następuje po zakończeniu odebrania komunikatu, na drodze wymiany określonych sygnałów pomiędzy odbiorcą i nadawcą.

Transmisja połączeniowa jest wykorzystywana np. przy przekazywaniu poczty elektronicznej, pobieraniu stron www, składania oświadczenia PIT, prowadzenia rozmów lub wideokonferencji za pomocą Skype, itp.

1.2.3.20. **Wyjaśnienie**. *Transmisja bezpołączeniowa* zakłada, że nadawca wysyła kolejne pakiety (porcje) danych adresowane do odbiorcy, jedynie z nadzieją, że wysłane dane dotrą do odbiorcy. W odróżnieniu od transmisji połączeniowej, nie ma miejsca na nawiązanie połączenia pomiędzy nadawcą i odbiorcą.

Transmisja bezpołączeniowa jest wykorzystywana np. przy przekazywaniu internetowych transmisji z imprez sportowych, czy wiadomości przeznaczonych dla wielu anonimowych odbiorców, jak program telewizyjny rozpowszechniany za pomocą Internet.

Można powiedzieć krótko, transmisje połączeniowe przeznaczone są do kontaktów pomiędzy dwoma lub więcej osobami, w zasadzie niedostępnych dla osób postronnych, natomiast transmisje bezpołączeniowe przeznaczone są do rozgłaszania informacji skierowanych do anonimowego odbiorcy w całej rozległej sieci Internet.

Piśmiennictwo: *Kurose J*. K.8.1., *Sosińsky B.* S.11.1.

1.3. SYSTEMY PLIKÓW

1.3.1. STRUKTURA KATALOGÓW

Katalogi służą do grupowania plików. Struktury katalogów ewoluowały w miarę rozwoju systemów operacyjnych, odzwierciedlając pojawiające się potrzeby użytkowników.

1.3.1.10. **Wyjaśnienie**. *Katalog jednopoziomowy* - to najprostsza struktura katalogów. W systemie jest tylko jeden katalog, w którym zgromadzone są wszystkie pliki. Każdy plik jest jedno znacznie identyfikowany przez swoją nazwę. Taka prymitywna struktura realizuje jedynie podstawową potrzebę gromadzenia wielu plików w systemie.

1.3.1.20. **Wyjaśnienie**. *Katalog dwupoziomowy*. Jeżeli z komputera korzysta wielu użytkowników, to wszyscy oni muszą korzystać ze wspólnego katalogu. Powoduje to konflikty w nazywaniu plików między użytkownikami. Problem ten rozwiązano wprowadzając dwupoziomową strukturę katalogów. Katalog główny zawiera szereg podkatalogów, po jednym dla każdego użytkownika. W podkatalogach znajdują się pliki - bez możliwości tworzenia dalszych podkatalogów. W takiej strukturze plik jest jednoznacznie identyfikowany przez podanie nazwy podkatalogu i nazwy pliku. Rozwiązanie likwiduje konflikty między użytkownikami, jednak nie umożliwia grupowania użytkowników o podobnym profilu działania.

1.3.1.30. **Wyjaśnienie**. *Drzewiasta struktura katalogów*. Pomysł grupowania plików użytkowników w podkatalogi łatwo uogólnić. Katalog może zawierać pliki i podkatalogi, a cała struktura katalogów może mieć kształt drzewa, którego korzeń stanowi główny katalog. Każdy użytkownik może mieć swój katalog oraz dodatkowo może dowolnie grupować swoje pliki w podkatalogi. Pliki są identyfikowane poprzez podanie ścieżki w drzewie (złożonej z nazw podkatalogów) oraz nazwy pliku.

1.3.1.40. **Wyjaśnienie**. *Katalog o strukturze grafu acyklicznego*. W strukturze tej dopuszczamy, aby katalogi i pliki tworzyły graf acykliczny. Oznacza to, że każdy plik i podkatalog (ale nie katalog główny) występuje w jednym lub więcej katalogach. Warunek acykliczności gwarantuje nam, że z katalogu głównego możemy dotrzeć do każdego katalogu i pliku. Przy takiej strukturze katalogów mamy nową operację: wstawienie istniejącego pliku lub podkatalogu do określonego katalogu. W wyniku wykonania takiej operacji mamy więcej niż jedno dowiązanie do danego pliku/pod-katalogu. Ponadto plik/katalog ten może w różnych katalogach pojawiać się pod różnymi nazwami. Tak więc nazwa przestaje być atrybutem pliku/podkatalogu, lecz jest związana z katalogiem, w którym występuje. Natomiast z każdym plikiem i podkatalogiem

wiążemy nowy atrybut: licznik dowiązań. W momencie utworzenia pliku lub podkatalogu licznik ten jest ustawiany na 1. Każde wpisanie nazwy w pliku/katalogu do kolejnego katalogu powoduje zwiększenie licznika dowiązań, a każde usunięcie pliku/podkatalogu z określonego katalogu powoduje jego zmniejszenie o 1. W momencie, gdy wartość licznika spada do zera, plik/podkatalog jest faktycznie usuwany.

Piśmiennictwo: *Silberschatz A.* S.6.1., *Stencel K.* S.12.1.

1.3.2. System plików

1.3.1.10. **Wyjaśnienie**. *System plików* pełni w systemie operacyjnym rolę pamięci trwałej. Informacje zapisane w pamięci trwałej nie są gubione między kolejnymi uruchomieniami systemu. Zwykle pamięć trwała to dyski magnetyczne (tak też przyjmiemy na potrzeby tych rozważań). Informacje przechowywane w systemie plików są pogrupowane w pliki i katalogi
Niemniej warto zapoznać się z tym, co system plików może nam udostępniać - być może kryje nie znane nam jeszcze możliwości.

1.3.1.20. **Wyjaśnienie**. W systemie plików występują dwa podstawowe rodzaje obiektów: *pliki* i *katalogi*. Pliki służą bezpośrednio do przechowywania informacji, a katalogi służą do grupowania plików i innych (pod)katalogów.

1.3.1.30. **Wyjaśnienie**. Pojęcie pliku jest zapewne doskonale znane czytelnikowi, niemniej przedstawmy pokrótce typowe cechy pliku:

- *Nazwa* - symboliczna nazwa identyfikująca plik w obrębie katalogu, w którym się on znajduje (w niektórych systemach plików, ten sam plik może występować pod wieloma nazwami i w wielu katalogach. Nie zawsze więc plik ma jednoznacznie określoną nazwę, natomiast zawsze nazwa i katalog jednoznacznie określają plik.).
- *Typ* - określa rodzaj informacji przechowywanej w pliku (W zależności od rodzaju systemu operacyjnego, typ pliku może być mniej lub bardziej kontrolowany przez system operacyjny. Przykładowe typy plików to: program wykonywalny, plik tekstowy, grafika w określonym formacie itd.).
- *Treść* - informacje przechowywane w pliku (W zależności od rodzaju systemu operacyjnego, treści plików mogą posiadać zróżnicowaną strukturę lub nie. W systemach Unix, Linux i Windows pliki to ciągi bajtów określonej długości. Ich interpretacja zależy od typu pliku).
- *Wskaźnik pliku* - w przypadku plików o dostępie sekwencyjnym, wskaźnik pliku wskazuje miejsce w pliku, którego będzie dotyczyć kolejna operacja czytania / dopisania do zawartości pliku.
- *Prawa dostępu* - w systemach zapewniających ochronę danych, z każdym plikiem (jak również z katalogiem) są związane prawa dostępu, określające, jakie operacje mają prawo wykonywać dani użytkownicy (dokładny zestaw praw dostępu i ich znaczenie zależy od systemu operacyjnego).
- *Czas* utworzenia / ostatniej modyfikacji / ostatniego dostępu.

Wszystkie z tych atrybutów mają również odniesienie do katalogów, pod warunkiem, że za treść katalogu przyjmiemy listę plików i podkatalogów znajdujących się w danym katalogu. Pliki i katalogi możemy traktować jak trwałe obiekty pewnego abstrakcyjnego typu danych.

1.3.1.40. **Wyjaśnienie**. Z punktu widzenia użytkownika istotne jest, jakie operacje można wykonywać na plikach i katalogach. Operacje te możemy podzielić na dwie grupy:

1. operacje na całych plikach/katalogach

2. oraz operacje modyfikujące i odczytujące informacje z pojedynczego składnika pliku/katalogu.

Typowe operacje na całych plikach / katalogach to:

(a) zmiana nazwy pliku/katalogu,
(b) przeniesienie pliku/katalogu do innego katalogu,
(c) usunięcie pliku/katalogu,
(d) skopiowanie pliku,
(e) utworzenie nowego (pustego) katalogu,
(f) wypisanie treści pliku / zawartości katalogu.

1.3.2.50. **Wyjaśnienie**. Implementacja systemu plików na nośniku magnetycznym lub optycznym wygląda następująco. Pliki przechowywane są (zwykle) na dyskach magnetycznych. Struktura logiczna dysków jest o wiele prostsza niż systemu plików. Dysk jest podzielony na sektory - jednostki tej samej wielkości. Wielkość sektorów jest (zwykle) potęgą 2 i przeważnie jest to 512 bajtów (ze względów historycznych). Z punktu widzenia systemu operacyjnego, pojedynczy dysk można traktować jak liniową tablicę sektorów. Implementacja systemu plików ma architekturę warstwową, zbliżając stopniowo proste urządzenia, jakimi są dyski, do złożonego systemu plików. Wyróżniamy następujące warstwy: (1) sterownik dysku; (2) podstawowy system plików; (3) moduł organizacji plików; (4) logiczny system plików.

Piśmiennictwo: *Silberschatz A*. S.6.1., *Stencel K.* S.12.1.

1.3.3. ROZPROSZONY SYSTEM PLIKÓW I REPLIKACJA

1.3.3.10. **Wyjaśnienie**. *Sieciowe systemy plików* często nazywa się również rozproszonymi systemami plików, ponieważ udostępniane przez nie pliki i katalogi mogą fizycznie znajdować się na rożnych komputerach w sieci domowej, akademickiej lub korporacyjnej. Przykładem sieciowego systemu plików jest *NFS* (Network File System), który został opracowany przez firmę Sun we wczesnych latach 80, ubiegłego wieku, od tamtego czasu był wielokrotnie udoskonalany i obecnie jest dostępny dla wszystkich, systemów Unix/Linux, a nawet istnieje oprogramowanie dla systemu Windows (wykonane przez dostawców niezależnych od firmy Microsoft).

1.3.3.20. **Wyjaśnienie**. *Specyfikacje systemu NFS* były dostępne niedługo po opublikowaniu informacji o pierwszej wersji, dzięki czemu NFS stał się faktycznym standardem rozproszonych systemów plików. Protokół NFS umożliwia serwerom plików eksportowanie centralnych zbiorów plików i katalogów do wielu systemów klienckich. Dobrym przykładem plików i katalogów, które warto udostępnić w formie zasobu centralnego z jednoczesnym dostępem wielu użytkowników, są katalogi domowe użytkowników, zestawy narzędzi programistycznych i scentralizowane zasoby danych, takie jak kolejki poczty i katalogi wykorzystywane do przechowywania internetowych forów dyskusyjnych.

1.3.3.30. **Wyjaśnienie**. *Replikacja plików* to proces dotyczący rozproszonego systemu plików, gdzie poszczególne pliki, są prowadzone w postaci równolegle aktualizowanych kopii danego pliku i polega na bieżącego powielaniu wprowadzanych do jednej kopii pliku modyfikacji na pozostałe kopie. Wikipedia rozróżnia trzy rodzaje replikacji:

1. Replikacja migawkowa (*snapshot replication*) — dane rozprowadzane są w stanie z pewnego określonego momentu. Taki rodzaj replikacji znajduje głównie zastosowanie przy danych,

które nie są często modyfikowane. Jednak takie modyfikacje mogą być znaczne. Wszelkie zmiany pomiędzy migawkami nie są monitorowane.

2. Replikacja transakcyjna lub przyrostowa (*transaction replication*) — dane rozprowadzane są na podstawie logów transakcji. Umożliwia zachowanie zasady ACID[18], ponieważ dane zmieniane są tylko na głównym serwerze.
3. Replikacja dwukierunkowa lub łącząca (*merge replication*) — dwukierunkowe rozprowadzanie danych, zarówno od serwera, jak i od klientów, które mogły być również przeprowadzone bez połączenia pomiędzy serwerami. W czasie synchronizacji może dojść do konfliktu, który musi być rozwiązany przez osobę przeprowadzającą aktualizację.

Piśmiennictwo: *Silberschatz A.* S.6.1., *Stencel K.* S.12.1.

1.4. SYSTEM OPERACYJNY

1.4.1. CZYM JEST SYSTEM OPERACYJNY

System operacyjny jest programem, który steruje wykonaniem programów użytkowych i działa jako interfejs między użytkownikiem a sprzętem komputerowym. Można przyjąć, że system operacyjny ma dwa cele lub realizuje dwa rodzaje funkcjonalności:

1. *Wygoda*. System operacyjny czyni komputer wygodniejszym dla użytkownika.
2. *Sprawność*. System operacyjny umożliwia sprawne eksploatowanie zasobów komputera.

Kolejną innowacją, było zwiększenie sprawności komputera na drodze wprowadzenie tzw. wielo-programowości, opartej o podział czasu jednostki arytmetyczno-logicznej pomiędzy kilka rezydujących w pamięci głównej komputera – programów aplikacyjnych, a następnie obsługi bieżącego dostępu z wielu urządzeń końcowych (realizujących funkcjonalności urządzeń wejścia /wyjścia). W tym celu, jak już zostało powiedziane wcześniej, wbudowano szereg dodatkowych mechanizmów, takich jak: zegar czasu procesora, rejestrów ograniczania obszarów adresów dla poszczególnych programów aplikacyjnych, mechanizmu stosu, dwóch trybów pracy jednostki sterującej – system operacyjny oraz program aplikacyjny.

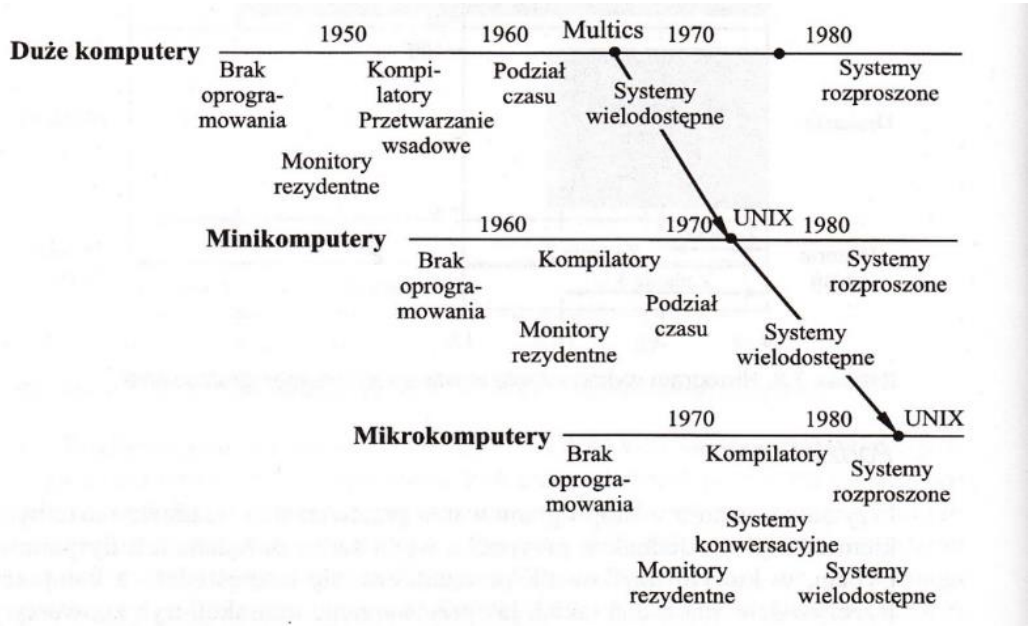
System operacyjny rozszerza funkcjonalność hardware'u udostępniając użytkownikom:

- Funkcjonalność znacznie bliższą sposobowi opisywania algorytmów przetwarzania danych i algorytmom obliczeniowym.
- Możliwość korzystania z urządzeń wejścia – wyjścia bez konieczności posługiwania się szczegółowymi parametrami technicznymi tych urządzeń.
- Możliwości zarządzania zasobami systemu komputerowego.
- Korzystania z bibliotek programów.
- Organizację korzystania z plików.

Podstawowe funkcjonalności sytemu operacyjnego to: (1) Szeregowanie zadań (programów) wysokiego poziomu; (2) Szeregowanie zadań krótkookresowe: (3) Bieżące zarządzanie procesami oraz stanami procesów. Zmiany koncepcji i funkcjonalności systemów operacyjnych w dwudziestym wieku - pokazuje zestawienie (patrz rys. 1.1.9.10).

[18] ACID – zbiór właściwości, które gwarantują poprawne przetwarzanie transakcji w bazach danych. ACID jest skrótem od angielskich słów: atomicity – *atomowość*, consistency – *spójność*, isolation – *izolacja*, durability – *trwałość.*

W procesie zarządzania zadaniami przez system operacyjny potrzebne jest dzielenia czasu poszczególnych rdzeni procesora pomiędzy aktywne wątki. Jądro realizuje nowatorski algorytm szeregowania, który działa w czasie stałym, niezależnie od liczby procesów lub wątków walczących o czas poszczególnych rdzeni procesora. Planista długoterminowy (*Task Scheduler*) wybiera – czyli szereguje zadania-programy, które mają być aktywizowane, inaczej mówiąc przekształcane w procesy i wątki. Planista krótkoterminowy (*Process Scheduler*) obsługuje wiele procesów i wątków – działając w trybie zwanym *Symmetric Multiprocessing* w skrócie SMP.



Rysunek 1.1.9.10. Zmiany koncepcji i funkcjonalności systemów operacyjnych w dwudziestym wieku.

Piśmiennictwo: *Shaw A.* S.3.1., *Silberschatz A.* S.6.1., *Stencel K.* S.12.1.

1.4.2. Interpreter poleceń

1.4.2.10. **Wyjaśnienie**. *Interpreter poleceń*, narzędzie pośredniczące w kontaktach pomiędzy użytkownikiem a systemem operacyjnym. Interpreter poleceń, z jednej strony oczekuje na polecenia użytkownika – przekazywane następnie (po odpowiednim przekształceniu) do systemu operacyjnego, z drugiej zaś strony wyświetla użytkownikowi informacje przekazywane przez system operacyjny.

1.4.2.20. **Wyjaśnienie**. Typowy interpreter poleceń został rozwinięty w ramach doskonalenie systemów operacyjnych rodziny Unix/Linux, umożliwiając obsługę szeregu funkcjonalności, takich jak:

- Rejestracja użytkownika w systemie operacyjnym, umożliwiająca korzystanie z komputera;
- Uruchamianie pojedynczych - interakcyjnych poleceń dla systemu operacyjnego;
- Zapisywanie złożonych poleceń (tzw. skryptów) i następnie (jedno lub wielokrotne) ich wykonywanie;
- Czytanie danych ze standardowego wejścia komputera;
- Przekierowanie wejścia/wyjścia na wskazane urządzenia lub pliki;
- Uruchamianie zadań wsadowych w tzw. tle wykonywania zadań wsadowych.

Piśmiennictwo: *Silberschatz A.* S.6.1., *Stencel K.* S.12.1.

1.4.3. STRUKTURA SYSTEMU OPERACYJNEGO

1.4.3.10. **Wyjaśnienie**. Przyjęto podział struktury systemu operacyjnego na trzy główne elementy:

- Jądro systemu wykonujące i kontrolujące zadania wymienione w 1.4.2.20;
- Interpreter poleceń (zwany również powłoką) – specjalny program komunikujący użytkownika z systemem operacyjnym (patrz podrozdział 1.4.2);
- Systemy plików (patrz podrozdział 1.3.2) – określający sposób zapisu struktury danych na nośnikach i umożliwiający korzystanie z tych danych (np. dysku lub w sieci).

1.4.3.20. **Wyjaśnienie**. Jądro systemu operacyjnego składa się z następujących elementów funkcjonalnych: (1) trzy poziomowego *planisty*, w tym planisty czasu procesora, ustalającego które zadanie i jak długo będzie wykonywane; (2) przełącznika zadań, odpowiedzialnego za przełączanie pomiędzy uruchomionymi zadaniami; (3) modułów zapewniających - synchronizacje i komunikację pomiędzy zadaniami, - obsługi przerwań i zarządzania urządzeniami, - zapewniającego przydział i ochronę pamięci.

1.4.3.30. **Wyjaśnienie**. W ramach jądra systemu operacyjnego można wyróżnić:

- Podsystem zarządzania procesami.
- Podsystem zarządzania pamięcią.
- Podsystem zarządzania systemem plików.
- Podsystem wejścia/wyjścia.
- Podsystem pamięci pomocniczej.
- Podsystem usług sieciowych.
- Podsystem ochrony.

Piśmiennictwo: *Shaw A*. S.4.1., *Silberschatz A.* S.6.1., *Stencel K*. S.12.1.

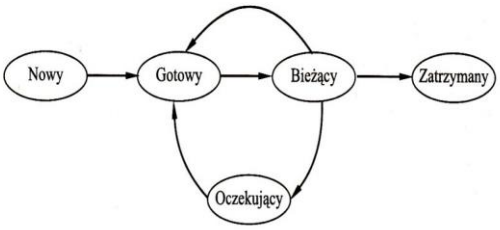
1.4.4. PROCESY I WĄTKI

Pojęcie procesu – odpowiadającego danemu programowi, jako realizacji tegoż programu zadania komputerowego, we współczesnej informatyce wiąże się ściśle z pojęciem systemu operacyjnego, a szczególności wielozadaniowego i wielodostępnego systemu operacyjnego. Przez proces, w rozumieniu systemu operacyjnego, czyli programu zarządzającego pracą systemu komputerowego, rozumiemy, więc zwykle – program komputerowy w toku wykonywania przez procesor komputera.

1.4.4.10. **Definicja**. Proces programu, to automatyczna realizacja programu w środowisku komputerowym, czyli sztucznym (stworzonym przez człowieka) środowisku. Wyróżniamy pięć podstawowych stanów procesu (patrz rys.1.4.4.01):

1. *Proces nowy* – utworzony przez system operacyjny, który po uzyskaniu dostępu do zasobów – przechodzi w stan proces gotowy;
2. *Proces gotowy* do realizacji i czekający na przydział czasu procesora, który po otrzymaniu czasu procesora - przechodzi w stan proces realizowany;
3. *Proces realizowany*, czyli działający – wykorzystujący czas procesora do momentu:
    a. Wyczerpania się przyznanego czasu procesora dla tegoż procesu - przechodzi w stan *proces gotowy*;
    b. Wyczerpania się któregoś z udostępnionych zasobów, w związku, z czym brak jest możliwości kontynuacji działania - przechodzi w stan *proces oczekujący*;

4. *Proces oczekujący*, czyli proces, którego działanie jest zawieszone w związku z oczekiwaniem na udostępnienie zasobu (zasobów), po uzyskaniem ponownym dostępu do zasobu (zasobów) - przechodzi w stan proces gotowy;
5. *Proces zatrzymany*, czyli program wykonany, oczekający na usunięcie z pamięci.



Rysunek 1.4.4.01. Zależności stanów procesu.

1.4.4.20. **Wyjaśnienie**. Związki poszczególnych stanów procesu ilustruje rys. 1.4.4.01. Należy zauważyć, że pojęcie procesu działającego w sztucznym środowisku, nie jest pomysłem mającym swoje źródło w logice został wprowadzonym przez konstruktorów pierwszych automatów tkackich.

1.4.4.30. **Definicja**. Wątek procesu (*thread*) – część programu wykonywana współbieżnie w obrębie jednego procesu. Inaczej mówiąc, w jednym procesie może istnieć wiele wątków. Różnica między procesem a wątkiem polega na współdzieleniu przez wszystkie wątki działające w danym procesie w jego przestrzeni adresowej oraz wszystkich innych struktur systemowych, natomiast każdy proces posiada przydzielone - niezależne zasoby. Ta ostatnia cecha ma dwie ważne konsekwencje:
1. Wątki wymagają mniej zasobów do działania i też krótszy jest czas ich tworzenia.
2. Dzięki współdzieleniu przestrzeni adresowej pamięci, wątki jednego procesu mogą się między sobą komunikować w sposób, niewymagający pomocy ze strony systemu operacyjnego.

Piśmiennictwo: *Shaw A.* S.4.1., *Silberschatz A.* S.6.1., *Stencel K.* S.12.1.

1.4.5. SYNCHRONIZACJA PROCESÓW I ZAGROŻENIE ZAKLESZCZENIAMI

1.4.5.10. **Wyjaśnienie**. Synchronizacja programów wykonywanych czyli procesów (*process synchronization, process coordination*), to działania mające na celu ustalenie właściwej kolejności wykonywania procesów współpracujących, w szczególności procesów korzystających z tzw. zmiennych dzielonych, czyli zmiennych równocześnie dostępnych – np. dla dwóch równolegle wykonywanych procesów. Do synchronizacji służą rozmaite mechanizmy programowe - implementowane przez jądro systemu operacyjnego, takie jak semafory, obszary krytyczne, znaczniki czasu.

1.4.5.20. **Wyjaśnienie**. Zakleszczenie czyli blokada wzajemna (*deadlock*) przez parę lub więcej procesów, jest pojęciem opisującym sytuację, w której co najmniej dwa różne rozpoczęte procesy (wykonywane programy) czekają na siebie nawzajem - na wykonanie następnych kroków, więc żaden z procesów nie może się zakończyć. Problem zakleszczenia występuje w wielozadaniowych systemach operacyjnych, gdzie wiele zadań w tym samym czasie konkuruje o wyłączny dostęp do zasobów. Zjawisko jest również ważne w systemach zarządzania na przykład bazami danych.

Piśmiennictwo: Shaw A. S.4.1., *Silberschatz A.* S.6.1., *Stencel K.* S.12.1.

1.4.6. ZARZĄDZANIE PAMIĘCIĄ RAM

Ważnym zasobem, który jest zarządzany przez system operacyjny - jest pamięć operacyjna *RAM*. Dla zwiększenia efektywności, biorąc pod uwagę sposób, że sprzęt zarządza pamięcią wirtualną, pamięć jest segmentowana w tak zwane strony (np. w rozmiarze 4KB dla większości architektur 64 bitowych). Np. system operacyjny Linux zawiera środki do zarządzania: dostępną pamięcią, a także mechanizmami sprzętowymi dla mapowań fizycznych i wirtualnych. Zarządzanie pamięcią to coś więcej niż zarządzanie buforami o pojemności 4KB. System operacyjny Linux oferuje abstrakcje łączenia 4KB bufory, takie jak tz. płyty podzielnika. Ten schemat zarządzania pamięcią korzysta z 4KB buforów jako jego podstawą, a potem tworzy strukturę śledzenie (od wewnątrz), które strony są pełne, które częściowo zapełnione, a które puste. Dzięki temu system pamięci może dynamicznie rosnąć i kurczyć się w oparciu o zapotrzebowania wykonywanych programów (procesów). Obsługa wielu użytkowników pamięci - wykonywanych programów aplikacyjnych powoduje, że są chwile, gdy dostępna fizyczna pamięć może być zbyt mała w stosunku do potrzeb. Z tego powodu fragmenty zawartości pamięci operacyjnej – tzw. strony mogą być przeniesione z pamięci operacyjnej na dysk, a w ich miejsce kopiowane są z dysku potrzebne strony. Proces ten, realizowany przez system zarządzania pamięcią *MMU* (patrz 1.1.6.50.) i nazywany jest *zamianą* ponieważ strony są zamienione z pamięci RAM na dysk twardy oraz odwrotnie.

1.4.6.10. **Wyjaśnienie**. Cele zarządzanie pamięcią przez *MMU* można sformułować następująco:

- *Large Address Spaces*. System operacyjny umożliwia operowanie znacznie większą pojemnością pamięci niż fizycznie istniejącą pojemnością pamięci *RAM*.
- *Protection*. Każdy proces dysponuje własną pamięcią odizolowaną od pamięci pozostałych procesów i pamięci dedykowanej dla systemu operacyjnego.
- *Memory Mapping*. Mapowanie umożliwia odwzorowywanie pamięci wirtualnej procesu w pamięci fizycznej (operacyjnej).
- *Fair Physical Memory Allocation*. Przypisanie poprawne obszarów pamięci *RAM* (fizycznej) poszczególnym procesom.
- *Shared Virtual Memory*. Mimo, że każdemu procesowi przydzielana jest pewna część pamięci operacyjnej, to zarządzanie pamięcią ma charakter dynamicznego udostępniania (przypisywania) poszczególnych fragmentów pamięci operacyjnej poszczególnym procesom.

1.4.6.11. **Wyjaśnienie**. Każdy wykonywany program, czyli tzw. proces, ma przydzielonych w pamięci operacyjnej - pięć fragmentów pamięci programu wykonywanego:

1. Blok kontrolny procesu realizującego program (tzw. kontekst procesu).
2. Przemieszczalny kod programu procesu.
3. Dane globalne udostępniane procesowi.
4. Stos procesu.
5. Sterta procesu.

Uwaga*:* Jak wynika z powyższego – obszar pamięci przydzielany procesowi może się składać z pięciu fragmentów, każdy z nich o określonej wielkości.

1.4.6.12. **Wyjaśnienie**. Przypominamy, że wyróżniamy następujące stany procesu (programu wykonywanego)[19]:

[19] bardziej szczegółowe omówienie tematu procesów – patrz podrozdział 1.4.4.

a. Proces *nowy*, który może przejść do stanu *gotowy*. Dzieje się tak po decyzji uruchamiania wykonywania programu, przez system operacyjny komputera.
b. Proces *gotowy* – może przejść do stanu *aktywny*, wtedy gdy przydzielony mu zostanie czas procesora. Proces *aktywny*, po wykorzystaniu przydzielonego mu czasu procesora lub po pojawieniu się procesu w stanie *gotowy* i wyższym priorytecie od procesu aktualnie wykonywanego przez procesor, albo wreszcie w przypadku przepełnienia danymi wynikowymi bufora lub braku danych wejściowych w buforze – proces *aktywny*, może przejść do jednego z trzech stanów: *gotowy* (), o*czekający* (), *zakończony* ().
c. Proces o*czekający*
d. Proces *zakończony*.

1.4.6.13. **Wyjaśnienie**. Blok kontrolny procesu (programu wykonywanego) zawiera szereg pól. Są to mianowicie:
a. Identyfikator procesu (programu wykonywanego).
b. Stan wykonywanego procesu (patrz rys 1.4.4.01.).
c. Kopia zawartości licznika instrukcji, zapisywana w momencie przerwania wykonywania procesu przez procesor komputera.
d. Kopie zawartości pozostałych rejestrów procesora wykorzystywanych przez dany proces, zapisywane w momencie przerwania wykonywania procesu przez procesor komputera (tzw. dane kontekstowe).
e. Informacje o stanie urządzeń wejścia/wyjścia danego procesu, zapisywane w momencie przerwania wykonywania procesu przez procesor komputera.

1.4.6.14. **Wyjaśnienie**. Przemieszczalny kod programu procesu to:
- kod wykonywalny programu, który może być umieszczany w dowolnym obszarze pamięci RAM;
- wszelkie adresy zawarte w kodzie są odpowiednio przesuwane w trakcie ładowania kodu programu do pamięci operacyjnej (poprzez odpowiednie ustawienie zawartości rejestrów segmentowych, patrz podrozdział 1.1.3).

1.4.6.15. **Wyjaśnienie**. Dane globalne udostępniane przez system aplikacji wykonywanemu programowi (procesowi), są wspólnymi danymi systemu aplikacji w skład którego wchodzi również wykonywany program. Dane globalne są przechowywane w obszarze pamięci zwanym – w przypadku procesorów z rodziny Intel 80x80 w jednym z *segmentów danych ES, FS, GS* (patrz podrozdział 1.1.3).

1.4.6.16. **Wyjaśnienie**. *Stos procesu* (patrz 1.1.2.), to mechanizm obsługi stosu przypisany danemu procesowi (programowi wykonywanemu). Stos procesu jest przechowywany w jednym obszarze pamięci zwanym – w przypadku procesorów z rodziny Intel 80x80 *segmentem stosu SE* (patrz podrozdział 1.1.3).

1.4.6.17. **Wyjaśnienie**. *Sterta procesu* (*heap*) to obszar pamięci lokalnej udostępniony na wyłączność wykonywanemu programowi (procesowi). Przechowuje się tam zmienne lokalne procesu. Sterta procesu jest przechowywany w jednym obszarze pamięci zwanym – w przypadku procesorów z rodziny Intel 80x80 *segmentem danych CS* (patrz podrozdział 1.1.3).

1.4.6.21. **Wyjaśnienie**. *Alokacja pamięci* oraz *de-alokacja pamięci* - to odpowiednio przydział i zwolnienie *ciągłego* obszaru pamięci określanego jako sterta (patrz 1.4.6.17). Po uruchomieniu,

nowy proces (program) otrzymuje od systemu operacyjnego obszar dostępnej pamięci, możliwej do wykorzystania przez proces na swoje dane lokalne - nazywany stertą. W trakcie działania proces może zażądać od systemu operacyjnego większej ilości pamięci (*alokacja*) lub też zwolnić niepotrzebny obszar (*de-alokacja*). Systemy operacyjne automatycznie zwalniają pamięć przydzieloną procesom, gdy te zakończą działanie bez uprzedniej de-alokacji otrzymanej pamięci.

1.4.6.22. **Wyjaśnienie**. *Odśmiecanie* sterty (*garbage collection*) – metoda automatycznego zarządzania dynamicznie przydzielonym obszarem pamięci (stertą), w której za czynności zwalniania oraz przemieszczania zawartości celem tworzenia spójnego podobszaru sterty - odpowiedzialny jest nie programista, lecz programowy zarządca nazwany *garbage collector.*

1.4.6.30. **Wyjaśnienie**. Blok kontrolny i przemieszczalny kod programu (procesu), są przechowywane w jednym wspólnym obszarze pamięci zwanym – w przypadku procesorów z rodziny Intel 80x80 *segmentem programu CS* (patrz podrozdział 1.1.3).

1.4.6.40. **Wyjaśnienie**. *Przełączanie kontekstu, przełączanie procesów* – czynność zapamiętania albo odtwarzania stanu procesu (*kontekstu procesu*), po przejściach procesu ze stanu *bieżący* (czyli wykonywany przez procesor) do jednego z trzech stanów: *zatrzymany*, *oczekujący* lub *gotowy*; oraz ze stanu *gotowy* do stanu *bieżący* (patrz 1.4.4). Przełączanie kontekstu umożliwia wielu procesom - dzielić zasoby pojedynczego procesora. Przełączanie kontekstu jest ważną cechą wielozadaniowego systemu operacyjnego. Z reguły przełączanie kontekstu jest zadaniem intensywnym obliczeniowo i wiele czasu przy projektowaniu systemów operacyjnych poświęca się na optymalizację tego zadania.

Piśmiennictwo: *Shaw A.* S.4.1., *Silberschatz A.* S.6.1., *Stencel K.* S.13.1.

1.4.7. SEKCJA KRYTYCZNA SYSTEMU OPERACYJNEGO

Sekcja krytyczna to fragment(y) kodu lub operacje, które nie mogą być wykonywane współbieżnie. Sekcja krytyczna jest otoczona dodatkowym kodem synchronizacji oczekiwania procesów - przed wejściem do sekcji krytycznej, czyli każdy proces musi przejść przez sekcję wejściową, zanim wejdzie do sekcji krytycznej, a wychodząc przechodzi przez sekcję wyjściową. Dopóki jakiś proces przebywający w sekcji krytycznej nie opuści jej, inne procesy chcące wejść do sekcji krytycznej są wstrzymywane w sekcji wejściowej. przechodząc przez sekcję wyjściową, proces opuszczający sekcję krytyczną wpuszcza jeden z procesów oczekujących (jeśli takowe są). Zakładamy tutaj, że każdy proces, który wejdzie do sekcji krytycznej, kiedyś ją opuści.

1.4.7.10. **Wyjaśnienie**. Podstawowa własność, jaką powinna spełniać sekcja krytyczna, to wzajemne wykluczanie: tylko jeden proces naraz może przebywać w sekcji krytycznej. Gdyby było to jedyne wymaganie wobec sekcji krytycznej, to można by ją bardzo prosto zaimplementować: nie wpuszczać żadnych procesów do sekcji krytycznej. Oczywiście nie o to chodzi! Stąd też druga wymagana własność, to wykorzystanie sekcji krytycznej: jeśli jakiś proces chce wejść do sekcji krytycznej, to nie może ona pozostawać pusta.

1.4.7.20. **Wyjaśnienie**. Jeśli wiele procesów oczekuje na wejście do sekcji krytycznej, to nie jest określone, w jakiej kolejności wejdą do niej. Nie ma tu jednak zupełnej dowolności. Przyjęto najmniejsze wymagania zapewniające sensowne działanie sekcji krytycznej, brak zagłodzenia oczekujących procesów: każdy proces, który chce wejść do sekcji krytycznej, ma gwarancję, że

kiedyś do niej wejdzie. Rozwiązanie problemu sekcji krytycznej polega zastosowaniu tzw. algorytmu *Mutual Exclusion*.

1.4.7.30. **Wyjaśnienie**. Algorytm *Mutual Exclusion* dostępu do sekcji krytycznej obsługi danego zasobu, z którego równocześnie korzystać może tylko jeden proces nosi nazwę *Algorithm of Mutual Exclusion*. Jedną z pierwszych wersji algorytmu był opracowany w 1962 roku przez holenderskiego matematyka T.J. Dekkera. Był to poprawny algorytm, stosowany do dzisiaj. Jest to algorytm typu producent-konsument. Ciekawostką jest, że uważano ten algorytm za zbyt złożony i wielokrotnie próbowano go bezskutecznie uprościć. Między innymi jeden z poważnych producentów oprogramowania wprowadził w nowoopracowanym SO - algorytm zawierający poważny błąd. Na błąd ten zwrócili uwagę w 1980 roku *Doran i Thomas*. Uproszczony poprawny algorytm *Mutual Exclusion* opublikował w 1981 roku *G.I. Peterson*.

Piśmiennictwo: *Shaw A*. S.4.1., *Silberschatz A.* S.6.1., *Stencel K.* S.12.1.

1.4.8. ZARZĄDZANIE WEJŚCIEM/WYJŚCIEM

Jak wiemy, komputer składa się z: procesora, urządzeń zewnętrznych, sterowników urządzeń zewnętrznych, pamięci operacyjnej, pamięci podręcznej, sterownika pamięci i magistrali systemowej. Za obsługę tych części składowych konfiguracji komputera odpowiada system operacyjny. Sekcja systemu operacyjnego - obsługi każdego urządzenia wejścia/wyjścia jest sekcją krytyczną systemu operacyjnego (patrz podrozdział 1.4.7).

1.4.8.10. **Wyjaśnienie.** Urządzenia zewnętrzne komunikują się z systemem komputerowym poprzez przesyłanie sygnałów. Łączem między urządzeniem a komputerem jest tzw. port. Porty są podłączane do magistrali, tzn. mediów, którymi przesyłane są sygnały (takim medium może być wiązka przewodów; przesyła się przez nią sygnały w postaci impulsów elektrycznych). Do jednej magistrali może być podłączonych wiele portów. Do każdego portu może być podłączony jeden lub więcej sterowników urządzań, do których z kolei są podłączne urządzenia wejścia/wyjścia. W komputerze mogą również występować układy szeregowe, tzn. że urządzenie A jest podłączone do urządzenia B, które z kolei jest połączone z urządzeniem C, które jest już bezpośrednio połączone do portu komputera.

1.4.8.20. **Wyjaśnienie.** Sterownik jest układem elektronicznym, który nadzoruje pracę portu, szyny lub urządzenia albo większej ich grupy. Sterownik pośredniczy w przesyłaniu informacji z i do urządzeń. Sterownik ma własną pamięć, w której może buforować dane przesyłane do i z urządzeń. Pozwala to na szybkie przesłanie informacji magistralą systemową, niezależnie od prędkości urządzenia. Do magistrali systemowej mogą też być podłączone sterowniki, które łączą główną magistralę z magistralami niższego poziomu (np. *PCI*, *ISA*), do których też mogą być podłączone sterowniki itd. Często też urządzenia zewnętrzne są podłączone do sterownika za pomocą odpowiedniej specjalizowanej magistrali (np. *SCSI*).

1.4.8.30. **Wyjaśnienie.** Procesor przekazuje sterownikom polecenia i dane niezbędne do wykonania tych poleceń. Sterownik ma specjalny rejestr, który służy do przekazywania komunikatów z i do procesora. Procesor porozumiewa się ze sterownikiem pisząc i czytając bity w tym rejestrze. Przesłanie tych bitów do rejestru urządzenia może odbywać się poprzez specjalne instrukcje wejścia/wyjścia będące zleceniami przesłania bitów na adres portu wejścia/wyjścia. Rozkazy te powodują przekazanie bitów z i do rejestrów sterujących urządzenia poprzez magistralę.

1.4.8.40. **Wyjaśnienie**. Urządzenia wejścia/wyjścia dzielimy na dwie podstawowe klasy:

1. Urządzenia znakowe, takie jak np. klawiatura, które transmitują dane asynchronicznie znak po znaku, a odczyt/zapis każdego znaku jest obsługiwany bezpośrednio przez procesor.
2. Urządzenia blokowe, takie jak np. dysk magnetyczny, które transmitują dane blokami synchronicznie, a odczyt/zapis każdego bloku danych jest jedynie inicjowany przez procesor, następnie zaś wykonywany pod nadzorem układu *DMA*.

1.4.8.50. **Wyjaśnienie**. Wyróżnimy ponadto dwa szczególne rodzaje urządzeń, a mianowicie:

1. Urządzenia sieciowe. Urządzenia sieciowe stanowią dość szczególną grupę ze względu na wydajność i adresowanie. Zwykle mają więc szczególny interfejs, inny niż czytaj/pisz/szukaj charakterystyczny dla urządzeń blokowych. Ważnym aspektem urządzeń sieciowych jest nawiązywanie połączenia, nasłuchiwanie z jednego połączenia oraz nasłuchiwanie z wielu połączeń równocześnie i obsługa jednego połączenia, którym akurat przysłano dane.
2. Zegary i czasomierze. Ich zadaniem jest podawanie bieżącego czasu, czasu trwania czegoś i uruchamianie operacji w ściśle określonej chwili. Szczególność tych urządzeń wynika z tego, że nie chodzi tu o transfer danych, ale raczej o synchronizację transferów. Podczas gdy w przypadku np. dysku przerwanie oznacza zakończenie operacji, to w przypadku zegara przerwanie zegarowe jest jedynym oczekiwanym wynikiem jego działania!

Wiemy już, czym jest urządzenie wejścia/wyjścia oraz w jaki sposób korzysta z niego komputer. Kontakt procesora z urządzeniem może być zaimplementowany za pomocą odpytywania, przerwań i DMA. Żądanie procesu użytkownika trafia do jądra systemu operacyjnego, w którym odpowiedni fragment oprogramowania tzw. podsystem wejścia/wyjścia zleca odpowiednie operacje modułowi sterującemu. Moduł sterujący to fragment oprogramowania komunikujący się bezpośrednio ze sprzętowym sterownikiem urządzenia. Moduł sterujący obudowuje charakterystykę urządzenia prezentując jądru systemu operacyjnego standardowy interfejs. Wydajność wejścia/wyjścia ma istotny udział w wydajność całego systemu.

Piśmiennictwo: *Shaw A.* S.4.1., *Silberschatz A*. S.6.1., *Stencel K.* S.12.1.

1.5. ZARYS PROGRAMOWANIA W JĘZYKU C

1.5.0. KILKA SŁÓW O POWSTANIU I DONIOSŁOŚCI JĘZYKA C

Jak podaje Wikipedia: **„C** – *imperatywny, strukturalny język programowania wysokiego poziomu* stworzony na początku lat siedemdziesiątych XX w. przez *Dennisa Ritchiego* do programowania systemów operacyjnych i innych zadań niskiego poziomu”. Poprzednikiem języka C był interpretowany język B, który *Ritchie* rozwinął w język C. Pierwszy okres rozwoju języka to lata 1969 - 1973. W roku 1973 w języku C udało się zaimplementować *jądro systemu operacyjnego Unix*. W 1978 roku *Brian Kernighan* i *Dennis Ritchie* opublikowali dokumentację języka pt. *C Programming Language* (wydanie polskie: *Język ANSI C*). Język C stał się popularny poza Laboratoriami Bella (gdzie powstał) po 1980 roku i stał się dominującym językiem do programowania systemów operacyjnych i aplikacji. Na bazie języka C w latach osiemdziesiątych *Bjarne Stroustrup* stworzył język C++, który ułatwia znacząco programowanie obiektowe. W 1983 roku ANSI powołało komitet X3J11 w celu ustanowienia standardu języka C. Standard został zatwierdzony w 1989 roku jako ANSI X3.159-1989 "Programming Language C". Ta wersja języka jest określana nieformalnie jako ANSI C, standardowe C lub C89. W 1990 roku standard

ANSI C został zaadoptowany przez ISO jako norma ISO/IEC 9899:1990. Ta wersja jest potocznie nazywana C90. Ponieważ normy wydane przez oba ciała standaryzacyjne są identyczne, wobec tego potoczne określenia C89 oraz C90 dotyczą tej samej wersji języka C. W 1999 roku ISO opublikowało normę ISO/IEC 9899:1999, język zgodny z tą normą jest nieformalnie nazywany C99. Ostatnia norma została opublikowana w 2011 roku pod nazwą ISO/IEC 9899:2011. Ta wersja języka jest potocznie nazywana C11 (C1X przed opublikowaniem normy)."

Pojawienie się języka C w 1972 roku, stanowiło przełom w dotychczasowym podejściu do budowy oprogramowania dla komputera o nowoopracowanej architekturze. Złożyło się na to kilka istotnych przyczyn. A mianowicie:

1. Opracowany został język wysokiego poziomu abstrakcji (o notacji zbliżonej do takiego języka jak FORTRAN, czy ALGOL 60), a jednocześnie język bardzo bliski rozwiązań sprzętowych architektury komputera.
2. Napisano kompilator języka C, w tymże języku C. Wystarczyło, więc zdefiniować w notacji BNF poszczególne instrukcje języka C w krótkich sekwencjach instrukcji wewnętrznego kodu danego komputera, a następnie z pomocą prostego programu wczytać kompilator kodu języka C oraz sekwencje instrukcji kodu wewnętrznego realizujące instrukcje języka C; po czym dokonać kompilacji - kompilatora języka C, w ten sposób otrzymujemy instalację języka C – np. na nowo zaprojektowanym komputerze.
3. Kolejnym krokiem, jest możliwość poprawienie sprawności zainstalowanego języka C, poprzez użycie narzędzia optymalizacji kodu wynikowego kompilacji.
4. Dysponując biblioteką podstawowego oprogramowania łącznie z systemem operacyjnym (np. z rodziny systemów LINUX) – napisanym w języku C, możemy po zainstalowaniu na danym komputerze z kompilatorem języka C, zainstalować system operacyjny i bibliotekę oprogramowania, w tym kompilatory języków wysokiego rzędu oraz systemy aplikacyjne.

Tym samym, język C stał się podstawowym narzędziem przenaszalności oprogramowania. Pierwszym chyba, zastosowaniem tej nowej technologii – było opracowanie oprogramowania dla komputera firmy DEC PDP-11, wykorzystując do tego celu komputer PDP-8. Dziś język C, oraz jego rozszerzenie język C++, są powszechnie używanymi językami programowania, przez profesjonalnych programistów systemowych. W wielu przypadkach, język C - zastąpił wcześniej powszechnie stosowane języki symboliczne typu assembler.

Język C jest jednak krytycznie oceniany przez część programistów. Jak podaje Wikipedia: **„...**pozwala na wykonywanie niskopoziomowych operacji, przez co wiele prostych błędów programistycznych nie jest wykrywanych przez kompilator, a przy wykonywaniu programu ujawniają się dopiero po jakimś czasie i w przypadkowych miejscach. Twórcy języka chcieli uniknąć sprawdzeń w czasie kompilacji i wykonywania programu, bo były one zbyt kosztowne czasowo, gdy C był implementowany po raz pierwszy. Z czasem powstały zewnętrzne narzędzia do wykonywania części z tych sprawdzeń. Nic nie przeszkadza implementacji języka w dostarczaniu takich sprawdzeń, ale też nie są one wymagane przez oficjalne standaryzacje. Używanie języka C wymaga od programisty dokładnego zrozumienia pisanego kodu źródłowego, łącznie z mechanizmami kompilacyjnymi, dodatkowo komplikowanymi nieprzenośnością między platformami i kompilatorami, jak również rygorystycznego przestrzegania dobrych praktyk, szczególnie w odniesieniu do funkcji obsługujących wszelkiego rodzaju buforowania. Podobnie brak standaryzacji bibliotek wyższego poziomu jest powodem do uznania C za język niezalecany dla początkujących. Jednakże wiele z tych niedogodności można zniwelować tworząc własne elastyczniejsze rozwiązania. Pod względem zastosowań

praktycznych C nie ustępuje innym językom, traci jednak w stosunku do nich, gdy wziąć pod uwagę czas i inne środki niezbędne do implementacji porównywalnych systemów."

Piśmiennictwo: *Wikipedia* W.2.6.

1.5.1. Tworzenie programu w języku C

1.5.1.10. **Wyjaśnienie.** Komentarz blokowy umieszcza się między sekwencją znaków `"/*"` a `"*/"`, a komentarz liniowy rozpoczyna się sekwencją `"//"` a kończy znakiem końca linii.

1.5.1.11. **Wyjaśnienie.** Lista słów kluczowych języka C określona jest przez normy ISO/IEC 9899-1999 (C99) – patrz tab. 1.5.1.12. Uwaga: Istnieją zależne od implementacji rozszerzenia języka o inne słowa kluczowe jak np. `asm` – dla oznaczenia wstawki kodu asemblera.
Przykład programu:

```
#include <stdio.h>
int main(void)
{
    printf("hello, world\n");
    return 0;
}
```

W powyższym kodzie:

- Dyrektywa `#include` włącza do pliku zawartość odpowiednich plików nagłówkowych – w tym przypadku pliku `stdio.h`, zawierającego m.in. prototyp funkcji `printf`.
- Główna funkcja nazywa się zawsze `main`. Zwraca ona wartość typu całkowitoliczbowego – `int`, w tym przypadku 0.
- Za wyprowadzenie wyniku na standardowe wyjście (zwykle na ekran) odpowiedzialna jest funkcja `printf`.
- Łańcuch tekstowy zamyka się w cudzysłowach: `"łańcuch"`.
- Znak nowej linii zapisuje się jako `"\n"`.

Tabela 1.5.1.12. Słowa kluczowe języka C

auto	default	float	long	sizeof	unsigned	_Imaginary
break	do	for	register	static	void	
case	double	goto	restrict	struct	volatile	
char	else	if	return	switch	while	
const	enum	inline	short	typedef	_Bool	
continue	extern	int	signed	union	_Complex	

Piśmiennictwo: *Tłuczek M.* T.5.1.

1.5.2. Zmienne, stałe i wyrażenia w języku C

1.5.2. 10. **Wyjaśnienie.** Każda zmienna odpowiada przypisanemu miejscu w pamięci operacyjnej komputera, któremu to miejscu pamięci można przypisać różne wartości – danej zmiennej. Przed użyciem zmiennej w programie należy ją zadeklarować.

1.5.2.11. **Wyjaśnienie.** Zmienne deklaruje się za pomocą prostej konstrukcji: typ nazwa; (patrz tab. 1.5.2.12. Należy pamiętać, że podane powyżej rozmiary zmiennych są jedynie orientacyjne i mogą się różnić w zależności od środowiska (w systemach 64-bitowych zmienna `long` posiada zazwyczaj 64-bity). Inną ważną sprawą jest szerokość bajtu. Język C wymaga tylko, by bajt składał się z co najmniej 8 bitów. Jest to zwykle najmniejsza porcja danych, która może być

adresowana. Wielu programistów nie zdaje sobie sprawy z powyższych problemów, co może być (i jest) przyczyną wielu błędów oprogramowania, a w rezultacie powstają różne luki w bezpieczeństwie oprogramowania.

Tabela 1.5.2.12 Lista typów zmiennych w języku C

Typ zmiennej	Symbole konwersji	Typowe wielkości pamięci	Uwagi
`bool`	%b	1 bajt	Tylko w wersji C99 (po włączeniu nagłówka <stdbool.h>)
`char`	%c	1 bajt	Pojedynczy znak ASCII oraz liczby <-128, +127>
`unsigned char`	%c	1 bajt	Pojedynczy znak ASCII oraz liczby <0, 255>
`signed char`	`%c`	1 bajt	Pojedynczy znak ASCII oraz liczby <-128, +127>
`int`	`%d`	4 bajty	Liczba stałoprzecinkowa z zakresu <-2 147 483 648, +2 147 483 647>
`unsigned int`	`%u`	4 bajty	Liczba stałoprzecinkowa z zakresu <0, +4 294 967 295>
`short int`	`%d`	2 bajty	Liczba stałoprzecinkowa z zakresu <-32 768, +32 767>
`unsigned short int`	`%u`	2 bajty	Liczba stałoprzecinkowa z zakresu <-32 768, +32 767>
`long int`	`%ld`	4 bajtów	Liczba stałoprzecinkowa z zakresu <-2 147 483 648, +2 147 483 647>
`unsigned long int`	`%lu`	4 bajtów	Liczba stałoprzecinkowa z zakresu <0, +4 294 967 295>
`float`	`%f`	4 bajty	Liczba zmiennoprzecinkowa z zakresu <-1.2E-38, 3.4E38>, pamiętanych 7 cyfr mantysy
`double`	`%f`	8 bajtów	Liczba zmiennoprzecinkowa z zakresu <-2.22E-308, 1.8E308>, pamiętanych 16 cyfr mantysy

1.5.2.13. **Wyjaśnienie.** Typy pochodne danych, to:

- Wyliczeniowe typy danych: `enum nazwa { jeden, dwa };`
- Struktury: `struct nazwa { typ1 nazwa1; typ2 nazwa2; };`
- Unie: `union nazwa { typ1 nazwa1; typ2 nazwa2; };`
- Pola bitowe: `typ [identyfikator] : długość;`
- Tablice: `typ nazwa[liczba];`

Wskaźniki: `typ *nazwa; typ **nazwa; typ_zwracany (*nazwa_wsk_do_funkcji)(typ parametru1,typ parametru2,...);`

1.5.2.14. **Wyjaśnienie.** Stałe deklarujemy, na jeden z dwu sposobów:

1. Stała symboliczna deklarowana na początku kodu w następujący sposób: `#define NaszaLiczba 13`, (`NaszaLiczba` to symbol stałej, a 13 – przypisana jej wartość).
2. Innym sposobem deklaracji stałej jest użycie słowa kluczowego `const`: np. `const NaszaLiczba 13`.

Piśmiennictwo: *Tłuczek M.* T.5.1.

1.5.3. INSTRUKCJE ARYTMETYCZNE, LOGICZNE, WYJŚCIA I WEJŚCIA, ORAZ SKOKU I WARUNKOWE

1.5.3.01. **Wyjaśnienie.** Symbol „=” jest używany w języku C, jako symbol operacji podstawienia (podobnie jak w pierwszym języku programowania wysoko - poziomowego FORTRAN), natomiast jako znak równości jest wykorzystywana para symboli „==”, zaś jako znak nierówności para symboli „!=”.

1.5.3.02. **Wyjaśnienie.** Instrukcje arytmetyczne (dodawania, odejmowania, mnożenia i dzielenie) mają postać: `x = y + 10; x = y – 6; z = x *y / 2;`

1.5.3.03. **Wyjaśnienie.** Skrócony zapis operacji dodawania jedynki i odejmowania jedynki (przedrostkowo i przyrostkowo): `x++;  ++x;  x--;  --x;`

1.5.3.11. **Wyjaśnienie.** Instrukcje logiczne wykonywane na parach odpowiadających pozycji bitów wchodzących w skład argumentów operacji logicznej (koniunkcja i alternatywa) mają postać: `z = x && y;  z = x || y;`

1.5.3.12. **Wyjaśnienie.** Instrukcja logiczna negacji (jednoargumentowa – wykonywana na poszczególnych bitach argumentu) ma postać: `y = ! x;`

1.5.3.21. **Wyjaśnienie.** Instrukcja wyjścia `printf( )`. Instrukcja ta służy do wyświetlania tekstu na monitorze.

1. Sekwencja wyjściowa `\n` wywołuje przeskok do następnej linii.
2. Sekwencja wyjściowa `\t` wstawia odstęp wielkości jednego tabulatora.
3. Sekwencja wyjściowa `\”` wstawia znak cudzysłowu.
4. Sekwencja wyjściowa `\a` wywołuje sygnał dźwiękowy.
5. Sekwencja wyjściowa `\b` wywołuje backspce.
6. Sekwencja wyjściowa `\\` wstawia znak \.
7. Sekwencja wyjściowa `\?` wstawia znak zapytania.
8. Sekwencja wyjściowa `\’` wstawia znak ’.

1.5.3.22. **Wyjaśnienie.** Instrukcja wejścia `scanf( )`. Funkcja wejścia – pobiera informację z klawiatury, a następnie przydziela jej odpowiednią zmienną. Przykład:

```
int x;
scanf(”%d”, &x);
```

Jak widać, należy podać typ zmiennej (wcześniej zadeklarowanej) w cudzysłowie oraz jej nazwę poprzedzoną znakiem `&`. Przypominamy skróty używane w instrukcji `scanf( ): char %c, int, short %d, long %ld, float, double %f, unsigned int, unsigned short %u, unsigned long %lg.`

1.5.3.23. **Wyjaśnienie.** Strumienie wejścia-wyjścia. Strumień – to ciąg bajtów danych przesyłanych do programu (strumień wejścia) albo wysyłanych na zewnątrz programu (strumień wyjścia). Rozróżniamy dwa rodzaje strumieni: tekstowe (ciągi znaków tekstowych) oraz binarne (ciągi bitów).

1. `Stdin` – strumień standardowego wejścia. Korzysta z danych tekstowych wpisywanych na klawiaturze.
2. `Stdout` - strumień standardowego wyjścia. Służy do przesyłania oraz wyświetlania danych tekstowych na ekranie.
3. `Stderr` - strumień „standardowego błędu”. Służy do przesyłania oraz wyświetlania danych tekstowych na ekranie.
4. `Stdprn` – strumień drukarki. Służy do przesyłania danych tekstowych na drukarkę.
5. `Stdaux` – strumień pomocniczy. Służy do przesyłania danych do portu COM1.

1.5.3.32. **Wyjaśnienie.** Funkcje definiowane. Funkcja musi posiadać *nazwę*: np. `MojaFunkcja`. Dla każdej funkcji musi być stworzony *prototyp* – model, pod którym będzie ona rozpoznawalna w programie:

np. `long | void MojaFunkcja (short jakas_zmienna, int inna-zmienna);`

Funkcja musi być zdefiniowana; *definicja* funkcji musi posiadać nagłówek, szkielet (zawierający wszystkie instrukcje wykonywane w obrębie danej funkcji) oraz opcjonalnie instrukcję powrotu, jeśli funkcja zwraca jakąś wartość (część kończącą każdą funkcję): np.

```
long MojaFunkcja (short zmienna1, int zmienna2)
{
      definicja realizowanej funkcjonalności
}
```

Uwaga. Opcjonalnie funkcja może pobierać pewne argumenty z programu głównego, wykorzystywane przez nią w celu wykonania określonych zadań. Wywołanie funkcji

```
long x;
Int main()
{
  …
    x = MojaFunkcja (zmienna1, zmienna2);
  …
}
```

1.5.3.40. **Wyjaśnienie.** *Etykieta* – w językach programowania jednostka leksykalna służąca oznaczeniu instrukcji w celu wskazania celu *instrukcji skoku* (patrz wyjaśnienie 1.5.3.41). W językach z numerowanymi wierszami kodu funkcje etykiety pełnią numery wierszy. Najczęściej etykiety są pierwszymi wyrażeniami w danym wierszu i mają postać ciągu znaków (liter lub cyfr) oddzielonych od instrukcji znakiem dwukropka, jak niżej:

```
etykieta: instrukcja
```

1.5.3.41. **Wyjaśnienie.** Instrukcja skokowa (zmiany sekwencji wykonywanych instrukcji), zwana w języku C `goto`, ma na celu przekazanie sterowania do sekwencji instrukcji zaczynającą się za wskazaną etykietą, w części adresowej instrukcji `goto <nazwa etykiety>`, jak niżej:

```
goto etykieta;
```

1.5.3.50. **Wyjaśnienie.** Instrukcja `if` (jeśli) to podstawowa instrukcja warunkowa w C – gdy `warunek1` jest spełniony (zwraca wartość niezerową), wykonany zostanie kod zawarty w bloku ograniczonym klamrami. Fragment `else` jest opcjonalny. Problem wiszącego `else` jest rozwiązany przez przyporządkowanie `else` do na najbardziej zagnieżdżonego `if`.

```
if (warunek1)
    instrukcja1
else
    instrukcja2
```

1.5.3.51. **Wyjaśnienie.** Instrukcja `if` z warunkiem równe, ma postać:

```
if (x == y) { … };
```

1.5.3.52. **Wyjaśnienie.** Instrukcja `if` z warunkiem nierówne, ma postać:

```
if (x != y) { … };
```

1.5.3.53. **Wyjaśnienie.** Instrukcja `if` z warunkiem numerycznie mniejsze, m postać:

```
if ( x < y ) { … };
```

1.5.3.54. **Wyjaśnienie.** Instrukcja `if` z warunkiem numerycznie mniejsze lub równe, ma postać:

```
if ( x <= y ) { … };[20]
```

1.5.3.55. **Wyjaśnienie.** Instrukcja `if` z warunkiem numerycznie wiksze, ma postać:

```
if ( x > y ) { … };
```

1.5.3.56. **Wyjaśnienie.** Instrukcja `if` z warunkiem numerycznie większe lub równe, ma postać:

```
if ( x =< y ) { … };
```

1.5.3.57. **Wyjaśnienie.** Instrukcja `if` z alternatywą pary warunków, ma postać:

```
if ( x == y | z == w) { … };
```

1.5.3.58. **Wyjaśnienie.** Instrukcja `if` z koniunkcja pary warunków, ma postać:

```
if ( x == y & z == w) { … };
```

Piśmiennictwo: *Tłuczek M.* T.5.1.

1.5.4. INSTRUKCJE PĘTLI I TABLICE W JĘZYKU C

1.5.4.01. **Wyjaśnienie.** Pętla `while` sprawdza spełnienie warunku przed każdym wykonaniem pętli, jeśli więc warunek nie jest spełniony, instrukcja(e) pętli nie jest (są) wykonywana (ne).

```
while (warunek)
{
      instrukcja
}
```

1.5.4.02. **Wyjaśnienie.** Pętla `do...while` (wykonuj ... dopóki) jest podobna do pętli `while` z tą różnicą, że warunek sprawdzany jest po każdym wykonaniu pętli, a więc instrukcje w pętli zawsze wykonają się co najmniej raz.

```
do
{
  instrukcja
}
while(warunek);
```

1.5.4.03. **Wyjaśnienie.** Pętla `for` (dla) jest rozwinięciem pętli `while` o instrukcję wykonywaną przed pierwszym obiegiem oraz dodatkową instrukcję wykonywaną po każdym przebiegu – najczęściej służącą jako licznik obiegów. Często zmienną liczącą kolejne wykonania ciała pętli nazywa się iteratorem.

```
for (wyrażenie1; wyrażenie2; wyrażenie3)
{
       instrukcja
}
```

1.5.3.31. **Wyjaśnienie.** Operacje na plikach. Aby utworzyć plik w języku C, należy zdefiniować wskaźnik do pliku. W pliku nagłówkowym `<studio.h>` jest zdefiniowany specjalny wzkaźnik do tego celu `FILE *`.

1. Deklaracja wzkaźnika do pliku wygląda tak:
   `FILE *wskaźnik_do_pliku;`
2. Otwieranie i tworzenie pliku – funkcja `fopen( );` „`r`” otwarcie pliku do odczytu, „`w`” otwarcie do zapisu, „`a`” otwarcie w trybie dopisywania.
3. Zamykanie pliku `fclose();`

Uwaga: odnośnie plików tekstowych można używać funkcji do odczytu `fscanf, fgetc, fgets`; do zapisu natomiast `fprintf, fscanf, fputc, fputs`.

Piśmiennictwo: *Tłuczek M.* T.5.1.

[20] Symbol złożony z pary znaków <= odpowiada symbolowi ≤ , podobnie jak symbol => ofpowiada ≥ .

1.5.5. Wskaźniki w języku C

Podstawowe zastosowanie wskaźników, to:

1.5.5.01. **Wyjaśnienie.** Przekazywanie przez wskaźnik zmiennej jako argumentu funkcji.

```
int funkcja(int *x)
y = funkcja(&x)
```

1.5.5.02. **Wyjaśnienie.** Dynamiczny przydział pamięci, w tym celu należy użyć funkcji `malloc`, zwalnianie przydzielonej pamięci dokonujemy z pomocą funkcji `free`.

1.5.5.03. **Wyjaśnienie.** Wykonywanie operacji arytmetycznych na wskaźnikach.

```
int x = 0;
int *ptr1 = &x;
int *ptr2 = ptr1;
```

rezultatem jest po prostu przypisanie adresu, na który wskazuje `ptr1`, wskaźnikowi `ptr2`.

1.5.5.04. **Wyjaśnienie.** Istnieje możliwość użycia łańcucha znaków bez konieczności korzystania z tablicy, korzystając ze wskaźników:

```
char *zdanie = ”Jakiś tekst”;
```

Nazwa zadeklarowanej tablicy użyta bez nawiasów jest wskaźnikiem tej tablicy:

```
int tablica[20];
tablica == &tablica[0];
```

Piśmiennictwo: *Tłuczek M.* T.5.1.

1.5.6. Instrukcja switch

1.5.6.10. **Wyjaśnienie.** Instrukcją decyzyjną `switch` (przełącznik) zastąpić można wielokrotne wywoływanie instrukcji warunkowej `if` np. dla różnych wartości tej samej zmiennej – przykładowo, gdy zmienna może przyjąć 10 różnych wartości, a dla każdej z nich należy podjąć inne działanie.

```
switch (wyrażenie) {
     case wartość1 :
        [instrukcje;]
        break;
     case wartość2 :
        [instrukcje;]
        break;
     default :
        [instrukcje;]
        break;
}
```

Piśmiennictwo: *Tłuczek M.* T.5.1.

1.5.7. Łańcuch znaków i działania na znakach

1.5.7.01. **Wyjaśnienie.** Zmienne typu `char` mogą przechowywać tylko jeden znak.

1.5.7.02. **Wyjaśnienie.** Dla przechowywania łańcucha znaków (pole bitowe) używamy tablicy,

```
char zdanie[14];
```

a następnie przypisać kolejnym elementom tablicy poszczególne znaki:

```
zdanie[0]; zdanie[1]; … ; zdanie[13]= ’\0’,
```

lub prościej

```
char zdanie[] = ”programowanie”;
```

1.5.7.03. **Wyjaśnienie.** Kopiowanie łańcucha znaków umożliwiają funkcje:

`strcpy(), strncpy().`

1.5.7.04. **Wyjaśnienie.** Na łańcuchach znaków można wykonywać operacje logiczne, np. konkatenacje (z pomocą funkcja `strcat()`), itp.

Piśmiennictwo: *Tłuczek M.* T.6.1.

1.5.8. MAKRODEFINICJE

1.5.8.01. **Wyjaśnienie.** Preprocesor języka wysokiego rzędu np. C, to procesor którego działanie poprzedza działanie kompilatora danego języka i zastępuje w tekście programu nazwę funkcji np. `MAX(x,y)` definicją danej funkcji – umieszczonej na początku kodu programu. Program może zawierać szereg definicji funkcji, następnie używanych w programie.

1.5.8.02. **Wyjaśnienie.** Dyrektywa `#define` daje dodatkowe możliwości w stosunku do definiowania prostej stałej symbolicznej. Można z jej pomocą tworzyć sparametryzowane makrodefinicje – dalej zwane krótko makrami. Prostym przykładem jest poniższa makro funkcja wyznaczenia maksimum dwóch liczb:

```
#define MAX(x,y) ((x)>(y)?(x):(y))
```

Ta raczej dziwnie wyglądająca instrukcja ze znakami ? oraz : to nic innego, jak zwykła instrukcja warunkowa zapisana w odmienny sposób. Zapis:

```
wynik_operacji = x > y ? x : y
```

odczytujemy jako:

```
if(x > y)wynik_operacji = x else wynik_operacji = y;
```

Preprocesor języka C, po napotkaniu takiej dyrektywy zamieni wszystkie wystąpienia `MAX(x,y)` w programie ciągiem instrukcji `((x)>(y)?(x):(y))`.

Piśmiennictwo: *Tłuczek M.* T.5.1.

1.5.9. KRYTYKA JĘZYKA C I NIEDOSTĘPNE WŁAŚCIWOŚCI

Język C pozwala na wykonywanie niskopoziomowych operacji, przez co wiele prostych błędów programistycznych nie jest wykrywanych przez kompilator, a przy wykonywaniu programu ujawniają się dopiero po jakimś czasie i w przypadkowych miejscach. Twórcy języka chcieli uniknąć sprawdzeń w czasie kompilacji i wykonywania programu, bo były one zbyt kosztowne czasowo, gdy C był implementowany po raz pierwszy. Z czasem powstały zewnętrzne narzędzia do wykonywania części z tych sprawdzeń. Nic nie przeszkadza implementacji języka w dostarczaniu takich sprawdzeń, ale też nie są one wymagane przez oficjalne standaryzacje.

Używanie języka C wymaga od programisty dokładnego zrozumienia pisanego kodu źródłowego, łącznie z mechanizmami kompilacyjnymi, dodatkowo komplikowanymi nieprzenośnością między platformami i kompilatorami, jak również rygorystycznego przestrzegania dobrych praktyk, szczególnie w odniesieniu do funkcji obsługujących wszelkiego rodzaju buforowania. Podobnie brak standaryzacji bibliotek wyższego poziomu jest powodem do uznania C za język niezalecany dla początkujących. Jednakże wiele z tych niedogodności można zniwelować tworząc własne elastyczniejsze rozwiązania. Pod względem zastosowań praktycznych C nie ustępuje innym językom, traci jednak w stosunku do nich, gdy wziąć pod uwagę czas i inne środki niezbędne do implementacji porównywalnych systemów.

1.5.9.01. **Niedostępna opcja.** Nie można przypisywać tablic (nie mylić ze wskaźnikami traktowanymi jako tablice) lub stringów – kopiowanie może zostać wykonane za pomocą standardowych funkcji; możliwe jest przypisywanie obiektów o typach `struct` lub `union`.

1.5.9.02. **Niedostępna opcja.** Brak odśmieczacza (*garbage collection*).

1.5.9.03. **Niedostępna opcja.** Brak wymagania sprawdzania zakresu tablic.

1.5.9.04. **Niedostępna opcja.** Brak operacji na całych tablicach.

1.5.9.05. **Niedostępna opcja.** Brak funkcji zagnieżdżonych, czyli możliwości wywoływania funkcji z wnętrza danej funkcji.

1.5.9.06. **Niedostępna opcja.** Brak domknięć lub przekazywania funkcji, jako parametru (tylko wskaźniki do funkcji i zmiennych).

1.5.9.07. **Niedostępna opcja.** Brak *generatorów* i *współprogramów*; kontrola przepływu programu w obrębie wątku opiera się tylko na zagnieżdżonych wywołaniach funkcji, nie licząc funkcji bibliotecznych `longjmp` czy `setcontext`.

1.5.9.08. **Niedostępna opcja.** Brak obsługi wyjątków; funkcje standardowe pokazują błędy za pomocą globalnej zmiennej `errno` lub specjalnych zwracanych wartości.

1.5.9.09. **Niedostępna opcja.** Ograniczona obsługa programowania modułowego.

1.5.9.10. **Niedostępna opcja.** Brak polimorfizmu w czasie kompilacji w formie przeciążania funkcji i operatorów.

1.5.9.11. **Niedostępna opcja.** Brak obsługi programowania obiektowego, a w szczególności polimorfizmu, dziedziczenia i ograniczona (tylko w obrębie modułu) obsługa enkapsulacji.

1.5.9.12. **Niedostępna opcja.** Brak bezpośredniej obsługi programowania wielowątkowego i sieci.

1.5.9.13. **Niedostępna opcja.** Brak standardowych bibliotek graficznych i innych.

Piśmiennictwo: *Tłuczek M.* T.5.1., *Wikipedia* W.2.6.

1.6. NASTĘPNY KROK ROZWOJU PRZENASZALNOŚCI OPROGRAMOWANIA

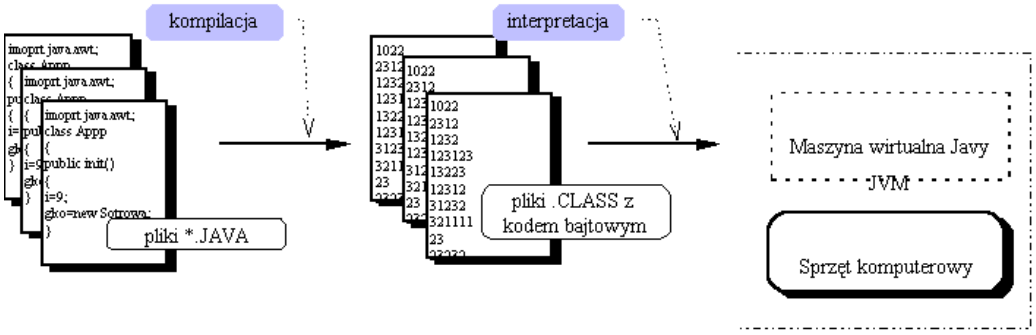
1.6.1. KILKA SŁÓW O POWSTANIU JĘZYKA JAVA, KODU BAJTOWEGO ORAZ JVM

JAVA jest językiem wysokiego poziomu zorientowanym obiektowo (patrz rozdział 4.7.) językiem programowania, zaprojektowany dla ułatwienia przenaszalności oprogramowania pomiędzy komputerami o różnorodnych architekturach i spełniającą idee Randalla Hyde'a „m*yśl nisko - poziomowo, pisz [oprogramowanie] wysoko - poziomowo*", czyli pisz oprogramowanie w języku wysokiego poziomu, ale cały czas uwzględniaj sposób realizacji programu przez sprzęt komputera.

W celu implementacji języka JAVA opracowano tzw. *Maszynę Wirtualną JAVA* (JVM) – specjalizowany program realizujący programy aplikacyjne napisane w języku JAVA i skompilowane (przetłumaczone) do tzw. kodu bajtowego. Zadaniem *Maszyny Wirtualnej JAVA*

jest wykonywanie programu napisanego w kodzie bajtowym – zarówno na drodze interpretacji kolejnych instrukcji kodu (patrz rys. 1.6.1.01), jak i na drodze kompilacji do kodu wewnętrznego procesora danego systemu komputerowego. *Maszyna Wirtualna JAVA* (JVM) może być instalowana – praktycznie na każdym systemie komputerowym z uwzględnieniem różnorodności architektonicznych procesorów.

Kolejnym krokiem rozwoju JVM, było wyposażenie *Maszyny Wirtualnej JAVA* w optymalizatory dla poszczególnych architektur procesorów, zapewniające optymalizację kodu skompilowanego z kodu bajtowego na kod wewnętrzny danego mikroprocesora.



Rysunek 1.6.1.01. Działanie JVM w trybie interpretacji kodu bajtowego.

Jak już zostało powiedziane, JAVA jest językiem wysokiego poziomu, zaprojektowanym na początku lat dziewięćdziesiątych ubiegłego wieku. Składnia języka JAVA jest podobna do składni języka C (patrz rozdział 1.5) oraz rozszerzenia tegoż języka zwanego C++. JAVA była początkowo projektowana jako język dla wielu różnorodnych architektur procesorów, umożliwiający pisanie oprogramowania dla dołączonych do sieci urządzeń, wykorzystywanych dla różnorodnych celów, w tym obsługujących urządzenia gospodarstwa domowego (pralki, lodówki, itd.). Popularyzacja Internetu, jaka miała miejsce w latach dziewięćdziesiątych ubiegłego wieku – spowodowała zainteresowanie językiem JAVA i dalszym rozwojem tegoż.

1.6.1.11. **Wyjaśnienie.** Kod bajtowy Javy to zbiór instrukcji wykonywanych przez *Wirtualną Maszynę JAVA* (JVM). Każdy kod operacji kodu bajtowego ma jeden bajt długości, chociaż niektóre kody operacji wymagają parametrów, co sprawia, że mamy dużo wielobajtowych instrukcji. Nigdy nie użyto wszystkich możliwych 256 kodów operacyjnych.

1.6.1.12. **Wyjaśnienie.** Programista języka JAVA w rzeczywistości nie musi wiedzieć, jak działa kod bajtowy JAVA. Jednak w czasopiśmie *IBM developer Works* zasugerowano: „Znajomość kodu bajtowego JAVA pomaga programiście tak, jak znajomość asemblera pomaga programistom języków C i C++”.

1.6.1.13. **Wyjaśnienie.** Każdy bajt może mieć 256 różnych wartości. Spośród nich w heksadecymalnie „`00`” przez „`ca`”, „`fe`”, i „`ff`” są przypisane wartości. Kod o wartości „`ba`” jest nieużywany z powodów historycznych. Kod „`ca`” jest zarezerwowany jako instrukcja przerwania dla *debuggerów* i nie jest używany w języku. Podobnie kody „`fe`” i „`ff`” nie są używane, gdyż zarezerwowane są do wewnętrznego użytku maszyny wirtualnej - JVM.

1.6.1.14. **Wyjaśnienie.** Instrukcje można podzielić na następujące grupy:

- ładujące i zapisujące (np. „`aload_0`”, „`istore`”),

- arytmetyczne i logiczne (np. „`ladd`”, „`fcmpl`”),
- zmieniające typ (np. „`i2b`”, „`d2i`”),
- tworzące obiekt i manipulujące nim (np. „`new`”, „`putfield`”),
- przenoszące sterowanie (np. „`ife`”, „`goto`”),
- wywołujące metodę i wracające z niej (np. „`invokespecial`”, „`return`”).

1.6.1.15. **Wyjaśnienie.** Jest też kilka instrukcji służących do bardziej wyspecjalizowanych czynności, np. zwracanie wyjątków, synchronizacja itd.

1.6.1.16. **Wyjaśnienie.** Duża część instrukcji ma też *prefiksy* lub *sufiksy* związane z typami zmiennych, na których pracują. Oto ich lista – patrz tabela 1.6.1.17.

Na przykład, instrukcja „`ladd`” dodaje do siebie dwie zmienne typu `long`, a instrukcja „`sadd`” dwie zmienne typu „`short`”. Instrukcje „`const`”, „`load`” i „`store`” mogą ponadto posiadać sufiks wyglądający tak: „`_n`”, z tym, że w miejscu `n` podstawiamy liczbę od `0-3` dla instrukcji „`load`” i „`store`”. Największe możliwe `n` dla instrukcji „`const`” zależne jest od typu zmiennej.

Tabela 1.6.1.17.

Prefiks/Sufiks	Typ zmiennej
i	integer
l	long
s	short
b	byte
c	character
f	float
d	double
a	reference

1.6.1.21. **Wyjaśnienie.** Najczęściej spotykanym językiem tworzący kod bajtowy JAVA jest JAVA. Jest jeden oficjalny kompilator JAVA - `javac` firmy *Sun Microsystems*, który kompiluje kod źródłowy JAVA do kodu bajtowego JAVA. Specyfikacja kodu bajtowego JAVA jest ogólnodostępna, co sprawiło, że powstały kompilatory przystosowane do innych języków programowania.

1.6.1.22. **Wyjaśnienie.** Słabą stroną programów napisanych w JAVA jest mała szybkość ich wykonywania spowodowana procesem interpretowania kodu bajtowego w przeglądarce. Kod bajtowy JAVA nie jest wykonywany tak szybko, jak zwykły, napisany w kodzie maszynowym procesora program. Aby rozwiązać ten problem wprowadzono "kompilację w locie" (*Just in Time compilation JIT*) kodu bajtowego do kodu maszynowego danego procesora. Kiedy przeglądarka ładuje pierwszy raz aplet JAVA, i jeśli opcja JIT jest aktywna, to przekształca ona kod bajtowy w kod maszynowy danego procesora, który zapisuje do pliku dyskowego. Przeglądarka wykonuje wtedy kod maszynowy. W efekcie, kompilator JIT traktuje kod bajtowy jako język źródłowy, który kompiluje do postaci programu w języku maszynowym lokalnej maszyny. Ponieważ w kodzie bajtowym nie było w trakcie kompilacji do kodu maszynowego danego procesora żadnych zmian, więc w rezultacie program ciągle jest bezpieczny i niezależny od sprzętu. Kompilacja JIT powoduje, że program napisany w JAVA pracuje kilka razy szybciej niż wykonywany przez najlepszy interpreter kodu bajtowego.

1.6.1.31. **Wyjaśnienie.** Według szacunków różnych firm i autorów publikacji, liczba urządzeń wyposażonych w JVM wynosi od 3 do 10 miliardów, w tym 1 miliard komputerów, a liczba programistów tworzących oprogramowanie na tę platformę – od 6,5 do 9 milionów.

Piśmiennictwo: *Chappel D.* C.1.1., *Shell S.* S.4.1.

1.6.2. Wirtualna maszyna JAVA

Jak już powiedzieliśmy, kod programu napisanego w języku JAVA - jest kompilowany do postaci kodu bajtowego, który następnie może być wykonany przez maszynę wirtualną na różnych procesorach. *Maszyna wirtualna JAVA* jest odpowiedzialna za ukrycie różnic między poszczególnymi platformami tak, że teoretycznie ten sam program można uruchomić w każdym miejscu.

Wirtualna maszyna JAVA to zestaw aplikacji napisanych na tradycyjne urządzenia i systemy operacyjne. Dostarcza środowiska, w którym może się wykonywać program skompilowany do postaci kodu bajtowego JAVA, zapewniając takie usługi, jak odśmiecanie pamięci czy obsługę wyjątków oraz bibliotekę standardowych podprogramów. W zależności od potrzeb i liczby dostępnych narzędzi, wyróżniane są dwie główne dystrybucje:

- *Java Runtime Environment* (JRE) – zawiera wyłącznie narzędzia niezbędne do uruchomienia aplikacji, tzw. środowisko uruchomieniowe;
- *Java Development Kit* (JDK) – zawiera również narzędzia dla programistów pozwalające na tworzenie aplikacji na platformę JVM.

Wirtualna Maszyna JAVA (JVM) nie jest nazwą konkretnego produktu. Dostępna publicznie specyfikacja pozwala różnym producentom oprogramowania na tworzenie własnych maszyn wirtualnych pracujących pod kontrolą różnych środowisk i urządzeń. Firma Oracle Corporation, twórca i właściciel znaku towarowego JAVA, udostępnia swoją specyfikację maszyny wirtualnej, aby inne firmy także mogły używać go w swoich produktach pod warunkiem, że ściśle przestrzegają oficjalnej specyfikacji i dodatkowych regulacji. Począwszy od wersji JAVA 7, wzorcową implementacją JVM jest OpenJDK będąca otwartym oprogramowaniem.

W skład maszyny wirtualnej JAVA (JVM) wchodzą następujące składowe:

- *Interpreter* – wykonuje krok po kroku instrukcje programu zapisane w postaci kodu bajtowego,
- *Kompilator JIT* – opcjonalny komponent wchodzący w skład części implementacji, który kompiluje najczęściej wykonywane fragmenty kodu do postaci kodu maszynowego, dzięki czemu mogą być one wykonywane bezpośrednio przez procesor komputera. Pozwala na zwiększenie wydajności,
- *Zarządca pamięci* – zarządza stertą, na której przechowywane są wszystkie obiekty wykonywanej aplikacji oraz zapewnia automatyczne zwalnianie nieużywanej pamięci,
- *Weryfikator kodu bajtowego* – kluczowym dla bezpieczeństwa aspektem jest weryfikacja kodu bajtowego przed jego uruchomieniem, której celem jest sprawdzenie poprawności wszystkich odwołań oraz upewnienie się, że wykonanie danego fragmentu nie zaszkodzi stabilności lub bezpieczeństwu systemu, na którym uruchamiana jest maszyna wirtualna. Zajmuje się tym weryfikator kodu bajtowego.
- *JAVA API* – zestaw bibliotek programistycznych udostępniających takie usługi, jak obsługę plików czy GUI, z których korzystają wykonywane aplikacje. Większość biblioteki

standardowej napisana jest w języku JAVA, dlatego maszyny wirtualne nie muszą dostarczać własnej implementacji.

Wirtualna maszyna JAVA (JVM) została pierwotnie stworzona do wykonywania programów napisanych w języku JAVA. Z biegiem czasu pojawiły się jednak kompilatory potrafiące kompilować wiele istniejących języków do postaci kodu bajtowego maszyny wirtualnej. Listę najważniejszych implementacji JVM - można znaleźć w internecie. W ostatniej dekadzie powstało także kilka nowych języków wysokiego rzędu zaprojektowanych z myślą o wykonywaniu na JVM.

Piśmiennictwo: *Chappel D.* C.1.1., *Shell S.* S.4.1.

1.6.3. .NET FRAMEWORK - ZINTEGROWANE ŚRODOWISKO PROGRAMISTYCZNE

Kolejnym krokiem rozwoju jest - .NET Framework, w skrócie .NET (czytamy - *dot net*), to platforma programistyczna opracowana przez firmę Microsoft, obejmująca środowisko uruchomieniowe (*Common Language Runtime – CLR*) oraz biblioteki klas dostarczające standardowej funkcjonalności dla aplikacji (patrz rys.rys. 1.6.3.41, 1.6.3.42 oraz 1.6.3.43). Technologia ta nie jest związana z żadnym konkretnym językiem programowania, oparta o rozwinięcie kodu bajtowego (patrz 1.6.1.), a programy mogą być pisane w jednym z wielu języków – na przykład C++/CLI, C#, F#, J#, Delphi 8 dla .NET, Visual Basic.NET, - kompilowanych do wspólnego języka pośredniczącego CIL (rozszerzenia omawianego wcześniej języka bajtowego maszyny wirtualnej).

1.6.3.10. **Wyjaśnienie.** Zadaniem platformy .NET Framework jest zarządzanie różnymi elementami systemu: kodem aplikacji, pamięcią i zabezpieczeniami. W środowisku tym można tworzyć oprogramowanie działające po stronie serwera internetowego (IIS) oraz pracujące na systemach, na które istnieje działająca implementacja tej platformy. Z racji jej pochodzenia najpełniej obsługiwane są systemy z rodziny Microsoft Windows, jednak ponieważ zasadnicza część platformy została zgłoszona jako standard ECMA, powstają także jego niezależne wdrożenia, np. Mono i dotGNU. W listopadzie 2014 Microsoft zapowiedział udostępnienie .NET na zasadach Open Source na licencji MIT.

1.6.3.11. **Wyjaśnienie.** .NET jest strategiczną platformą rozwoju oprogramowania Microsoft. W roku 2002 - Microsofcie: ogłosił on, że następca systemu Windows XP będzie pracował w środowisku .NET, a aplikacje starszego typu (EXE) będą miały dostęp do zasobów maszyny przez przekształcenie ich wywołania. W 2006 roku okazało się to jednak nieprawdą, gdyż tylko niewielki procent nowej wersji Windows korzysta z .NET.

1.6.3.12. **Wyjaśnienie.** W skład platformy wchodzą:

- kompilatory języków wysokiego poziomu – standardowo C++/CLI, C#, Visual Basic .NET, J#;
- kompilator *just-in-time* kodu zarządzanego wraz z debuggerem

1.6.3.21. **Wyjaśnienie.** Wobec zjawiska nieprzenośność programów dla Windows pomiędzy różnymi procesorami, Microsoft postanowił rozwiązać ten problem Windows - stosując rozwiązania podobne do zastosowanego w Javie: kompilatory kompilują kod źródłowy do postaci uniwersalnego kodu zwanego kodem pośrednim (nazywa się on obecnie CIL, czyli Common Intermediate Language – wcześniej zaś nazywany był MSIL), metoda klasy jest kompilowana do kodu maszynowego w momencie pierwszego wywołania, kolejne wywołania metody prowadzą już bezpośrednio do skompilowanego kodu. Jest to realizowane przez

dołączenie do każdej metody w czasie ładowania modułu tymczasowego fragmentu kodu (*stub*) który przekazuje sterowanie do kompilatora i jest następnie zastępowany przez skompilowany kod. Jest to tzw. kompilacja w locie (*just in time*). Dostępna jest także możliwość skompilowania całego modułu w trakcie instalacji. Przy okazji przebudowano biblioteki klas ułatwiające dostęp do elementów systemu.

1.6.3.22. Wyjaśnienie. Bloki składowe platformy .NET, to:

- CLR (*Common Language Runtime*) odpowiedzialny za lokalizowanie, wczytywanie oraz zarządzanie typami .NET. To trzon całej platformy .NET ponieważ to właśnie do CLR należy zadanie kompilowania i uruchamiania kodu zapisanego językiem kodu pośredniego (CIL).
- CTS (*Common Type System*) jest odpowiedzialny za opis wszystkich danych udostępnianych przez środowisko uruchomieniowe.
- CLS (*Common Language Specification*) to zbiór zasad definiujących podzbiór wspólnych typów precyzujących zgodność kodu binarnego z dostępnymi kompilatorami .NET

1.6.3.23. **Wyjaśnienie.** Platforma .NET jest rozwijana, istnieją więc kolejne jej wersje (patrz tabela 1.6.3.20.).

Tabela 1.6.3.20. Kolejne wersje .NET Framerwork

Wersja	Numer wersji	Data wydania	Visual Studio	Dołączona do Windows
1.0	1.0.3705.0	2002-02-13	Visual Studio .NET	
1.1	1.1.4322.573	2003-04-24	Visual Studio .NET 2003	Windows Server 2003, Windows XP
2.0	2.0.50727.42	2005-11-07	Visual Studio 2005	
3.0	3.0.4506.30	2006-11-06		Windows Vista, Windows Server 2008
3.5	3.5.21022.8	2007-11-19	Visual Studio 2008	Windows 7, Windows Server 2008 R2
3.5 SP1	3.5.30729.4926	2009-06-10		
4.0	4.0.30319.1	2010-04-12	Visual Studio 2010	
4.5	4.5.50709	2012-08-15	Visual Studio 2012	Windows 8
4.5.1	4.5.50938.18408	2013-10-12	Visual Studio 2013	Windows 8.1
4.5.2	4.5.51209.34209	2014-05-06		

1.6.3.30. **Wyjaśnienie.** *Common Language Infrastructure CLI*. Istotną nowością, jaka pojawiła się w platformie .NET, jest *Common Language Infrastructure*. Każdy język programowania, który spełni odpowiednie standardy (chodzi głównie o tzw. *common object model*), będzie miał dostęp do bogatej biblioteki .NET.

1.6.3.31. **Wyjaśnienie.** Podstawowe języki dostarczane przez Microsoft:

- C#
- Visual Basic .NET
- F#
- C++/CLI (wcześniej Managed C++, wariant C++)
- J# (wariant języka Java opracowany przez Microsoft)

1.6.3.32. **Wyjaśnienie.** Standardy środowiska .NET. W sierpniu 2000 Microsoft, Hewlett-Packard i Intel wspólnie złożyły specyfikację infrastruktury języka C# do ECMA jako propozycję standardu. Prace nad nimi odbywały się w ramach komitetu TC39 w podgrupach TG3 i TG2, przy współudziale m.in. IBM i Fujitsu. Zostały one ostatecznie zatwierdzone w grudniu 2001 jako ECMA-334 (CLI) i ECMA-335 (C#), a opis techniczny jako TR/84, a następnie przekazane do akceptacji przez ISO. W kwietniu 2003 ISO uznało nadesłane standardy, nadając im numery

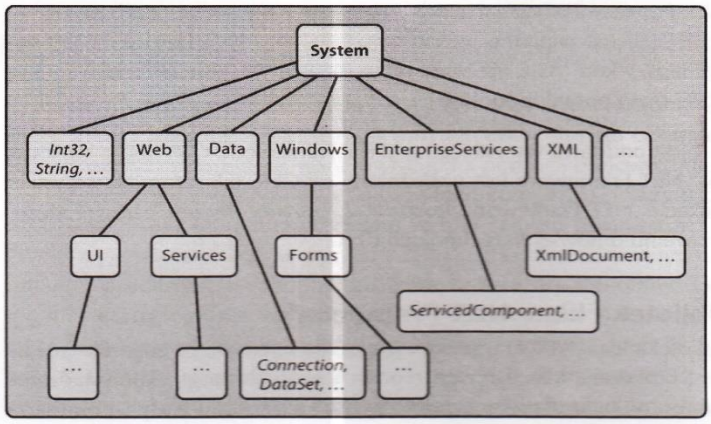
ISO/IEC 23270 (C#), ISO/IEC 23271 (CLI) oraz ISO/IEC 23272 (CLI TR), a ECMA przyjęła je jako drugie wydanie swoich standardów.

1.6.3.33. **Wyjaśnienie.** Technologie .NET:

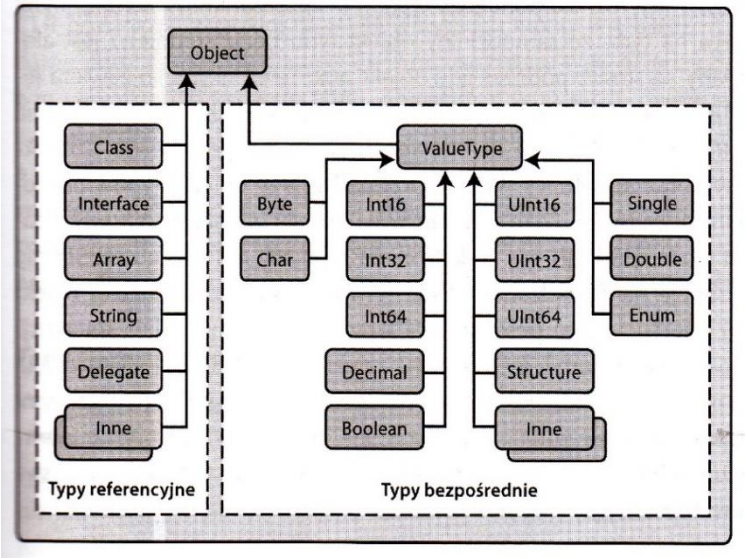
- Platforma .NET niesie ze sobą kilka pochodnych technologii.
- Można tu wymienić ADO.NET, ułatwiający dostęp do baz danych, oraz ASP.NET, służąca do budowania dynamicznych stron WWW.

1.6.3.34. **Wyjaśnienie.** Implementacje .NET. W obecnie najbardziej znane platformy .NET to:

1. Microsoft .NET Framework – darmowe środowisko udostępniane przez Microsoft.
2. Mono – projekt Novella na licencji Open Source .
3. DotGNU Portable.NET – implementacja powstająca w ramach projektu GNU.



Rysunek 1.6.3.41. Biblioteka Klas .NET

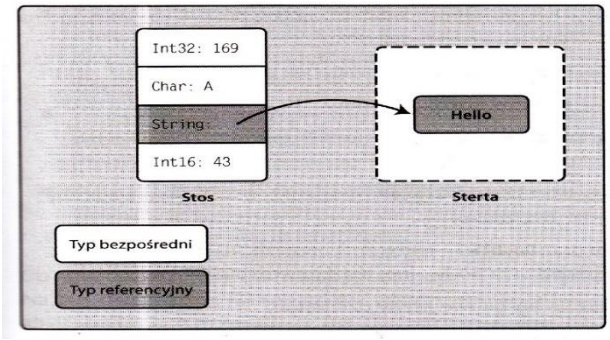


Rysunek 1.6.3.42. Typy danych .NET

1.6.3.35. **Wyjaśnienie.** Środowiska programistyczne:

- Flagowym produktem jest tu Microsoft Visual Studio, umożliwiające pisanie w kilku językach programowania, istnieje jednak wiele innych komercyjnych narzędzi (jak choćby firmy Borland).
- Środowisko open source rozwija środowisko MonoDevelop, jednak jest ono wciąż we wczesnej fazie rozwoju.
- Innym środowiskiem zastępczym dla wspomnianego MS VS jest SharpDevelop.

W roku 2005 została wydana druga wersja środowiska .NET Framework 2.0, wraz z nią udostępniono darmowe zintegrowane środowisko programistyczne Visual Studio 2005 Express, które składa się z kilku osobnych produktów (Visual Basic 2005 Express Edition, Visual C# 2005 Express Edition, Visual C++ 2005 Express Edition, Visual J# 2005 Express Edition, Visual Web Developer 2005 Express Edition, SQL Server 2005 Express Edition).



Rysunek 1.6.3.43. Obiekty typów bezpośrednich są alokowane na stosie, zaś obiekty typów referencyjnych są umieszczane na stercie

1.6.3.36. **Wyjaśnienie.** *Common Intermediate Language:*

- Common Intermediate Language (*Wspólny Język Pośredni*, w skrócie CIL, lub IL) – język najniższego poziomu dla platformy Microsoft .NET odczytywalny przez człowieka. Jest to odpowiednik asemblera jako języka pośredniego dla typowych języków wysokiego poziomu (tu: Common Language Infrastructure (CLI) wyrażający kod w C#, Visual Basic .NET, Managed C++ lub dowolnym języku z wielu (40+) języków kompilowanych do CIL). CIL jest tłumaczony bezpośrednio na kod bajtowy.
- CIL przypomina obiektowy asembler w całości oparty na stosie. Jego wykonanie następuje za pomocą maszyny wirtualnej.
- Początkowo CIL nosił nazwę *Microsoft Intermediate Language* (MSIL), ale uległa ona zmianie wskutek standaryzacji C# i CLI. Czasem jednak można jeszcze spotkać zastosowanie poprzedniej nazwy, szczególnie wśród starszych użytkowników .NET.
- *ilasm.exe* – kompilator języka CIL, dołączony do .NET Framework SDK.

1.6.3.37. **Wyjaśnienie.** Przykładowy kod programu *Hello world* napisany w CIL:

```
.assembly HelloWorld
.class auto ansi HelloWorldApp
{
     .method public hidebysig static void Main() cil managed
     {
          .entrypoint
          .maxstack 1
          ldstr "Hello world."
          call void [mscorlib]System.Console::WriteLine(string)
          ret
     }
}
```

1.6.3.35. **Wyjaśnienie.** Przykłady instrukcji języka CIL:

- `add` – dodaje dwie wartości z wierzchołku stosu i odkłada wynik z powrotem na stos;
- `box` – konwertuje typ bezpośredni na typ referencyjny, czyli pakuje wartość;
- `br` – przekazuje sterowanie (instrukcja skoku) do konkretnej lokacji pamięci;

- `call` – wywołanie określonej metody;
- `ldfld` – ładuje określone pole obieku na stos;
- `ldflj` – kopiuje wartość określonego typu bezpośredniego na stos;
- `newobj` – tworzy nowy obiekt typu bezpośredniego;
- `stfld` – przechowuje wartość ze stosu w określonym polu obiektu;
- `stobj` – przechowuje wartość na stosie w określonym typie bezpośrednim;
- `unbox` – konwertuje zapakowany typ bezpośredni z powrotem do swojej zwykłej formy.

1.6.3.50. **Wyjaśnienie.** Kompilowanie kodu zarządzanego zawsze tworzy kod w języku CIL oraz *metadane*, opisujące ten kod. Metadane - są informacjami o typach zdefiniowanych w powiązanym kodzie zarządzanym i przechowywane są w tym samym pliku co wygenerowany z tych typów CIL. Następnie przedstawimy abstrakcyjny wygląd modułu utworzonego przez kompilator oparty na CLR. Plik zawiera kod CIL, wygenerowany z typów oryginalnego programu, którymi znowu są klasy X, Y oraz Z. Oprócz kodu dla metod każdej z klas plik zawiera również metadane opisujące te klasy, a także wszystkie inne typy zdefiniowane w pliku. Informacje te są ładowane do pamięci w momencie, gdy ładowany jest sam plik, co czyni metadane dostępnymi w chwili uruchomienia. Same metadane mogą być również odczytane bezpośrednio z pliku, w którym są zawarte, co sprawia, że informacje te są dostępne, nawet gdy kod nie jest załadowany do pamięci.

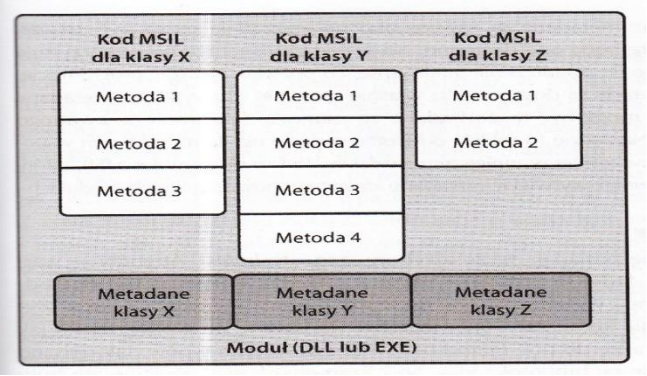
1.6.3.51. **Wyjaśnienie.** *Metadane* dostarczają informacji o każdym z typów. Metadane opisują typy zawarte w module. Pośród informacji przechowywanych dla danego typu znajdują się następujące:

1. nazwa typu
2. widoczność typu, która może być oznaczona jako public lub assembly (własność pakietu),
3. typ, po którym dany typ dziedziczy, o ile taki występuje,
4. interfejsy, które są implementowane przez ten typ,
5. metody, które są implementowane przez ten typ,
6. właściwości, które ten typ udostępnia,
7. zdarzenia, które ten typ może generować.

Dostępne są również bardziej szczegółowe informacje. Np. opis każdej metody zawiera jej parametry oraz ich typ wraz z typem wartości zwracanej przez metodę.

1.6.3.52. **Wyjaśnienie.** Narzędzia mogą wykorzystywać meta dane. Ponieważ metadane są zawsze obecne, narzędzia mogą być pewne, że zawsze będą one dostępne. Na przykład Visual Studio wykorzystuje metadane do dostarczenia *lntelliSense*, co pozwala pokazać programiście, kóre metody są dostępne dla właśnie wpisanej nazwy klasy. Metadane modułu mogą być także badane za pomocą *deasemblera*, zwanego **Ildasm**. Narzędzie to spełnia odwrotne zadanie niż **Ilasm**, o którym wspomniano wcześniej. Jest to *deasembler* CIL, który może również wyświetlić metadane zawarte w poszczególnych modułach.

1.6.3.53. **Wyjaśnienie.** Atrybuty. Metadane posiadają także atrybuty (*attributes*). Atrybuty są wartościami, które są przechowywane w metadanych. Mogą być one odczytywane a następnie wykorzystywane do kontrolowania różnych aspektów wykonywania kodu. Atrybuty mogą być dodawane do typów, takich jak klasy, a także do pól, metod oraz właściwości tych typów.



Rysunek 1.6.3.44. W pliku modułu znajdują się metadane dla każdego typu

1.6.3.54. **Wyjaśnienie.** Biblioteka klas .NET Framework wykorzystuje atrybuty do wielu celów, w tym zwłaszcza do wymagań dotyczących transakcji wskazujących, które metody powinny być udostępnione jako usługi sieciowe wywoływalne za pośrednictwem protokołu SOAP, oraz do opisywania wymagań dotyczących bezpieczeństwa. Atrybuty te mają standardowe nazwy i funkcje zdefiniowane przez różne części biblioteki klas .NET Framework, które je wykorzystują. Programiści mogą także tworzyć własne atrybuty, służące do kontrolowania zachowania w sposób specyficzny dla aplikacji. Aby utworzyć własny atrybut, programista wykorzystujący język programowania oparty na CLR, taki jak C# lub VB, może zdefiniować klasę dziedziczącą po System.Attribute. Obiekt klasy wynikowej będzie automatycznie przechowywał swoją wartość w metadanych, kiedy zostanie skompilowany.

1.6.3.55. **Wyjaśnienie.** Organizowanie kodu zarządzanego – pakiety. Kompletna aplikacja często składa się z wielu różnych plików. Niektóre pliki są modułami - DLL lub EXE, które zawierają kod, podczas gdy inne zawierają różne zasoby, takie np. jak pliki obrazków. W aplikacjach .NET Framework pliki, które tworzą jakąś logiczną jednostkę funkcjonalności, grupowane są w pakiet (*assembly*). Pakiety, opisane w niniejszym podrozdziale, mają istotne znaczenie w przypadku programowania, wdrażania i uruchamiania aplikacji .NET Framework.

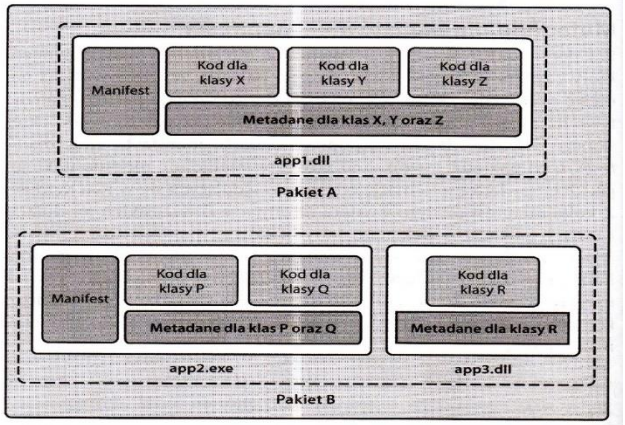
1.6.3.56. **Wyjaśnienie.** Metadane dla pakietów – manifesty (*listy ładunku*). Pakiety są konstrukcjami logicznymi; nie ma jednego pliku głównego, który zawiera wszystkie pozostałe pliki pakietu. W rzeczywistości nie da się powiedzieć, które pliki należą do tego samego pakietu, patrząc tylko na katalog plików. Zamiast tego ustalenie przynależności do określonego pakietu odbywa się poprzez badanie *manifestu* (*manifest*) tego pakietu. Jak opisano powyżej, metadane modułu (takiego jak DLL) zawierają informacje o typach znajdujących się w danym module. Z kolei manifest pakietu zawiera informacje o wszystkich modułach i innych plikach w pakiecie. Innymi słowy, manifest to metadane o pakiecie. Jest on zawarty w jednym z plików pakietu i uwzględnia informacje o pakiecie oraz plikach składających się na niego. Tak jak narzędzia, takie jak Visual Studio, generują metadane dla każdego kompilowanego modułu, tak samo generują one pakiety z odpowiednimi manifestami.

1.6.3.57. **Wyjaśnienie.** Zawartość pakietu - pakiet może być zbudowany z pojedynczego pliku lub grupy plików. W pakiecie z jednym plikiem manifest przechowywany jest w tym pliku, zaś w pakiecie z wieloma plikami - w jednym z plików pakietu. W obu przypadkach manifest opisuje cały pakiet, podczas gdy metadane w poszczególnych modułach opisują typy z tych modułów. Wśród informacji umieszczonych w manifeście pakietu znajdują się następujące:

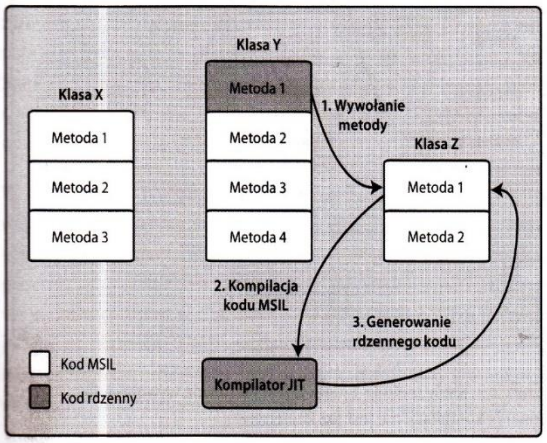
1. Nazwa pakietu

2. Numer wersji pakietu
3. Język pakietu
4. Lista plików zawartych w pakiecie wraz z sumą kontrolną pakietu
5. Informacji, od jakich innych pakietów (wraz z numerem wersji) zależy dany pakiet

1.6.3.58. **Wyjaśnienie.** Ładowanie pakietów - kiedy uruchamiana jest aplikacja zbudowana z wykorzystaniem .NET Framework, pakiety składające się na tę aplikację muszą być odnalezione i załadowane do pamięci. Pakiety nie są ładowane, dopóki nie są potrzebne, zatem jeśli aplikacja nie wywołuje żadnych metod z danego pakietu, nie będzie on załadowany (w rzeczywistości nie musi być nawet obecny na komputerze, na którym uruchamiana jest aplikacja). Zanim jakikolwiek kod z pakietu zostanie załadowany, musi zostać najpierw odnaleziony.



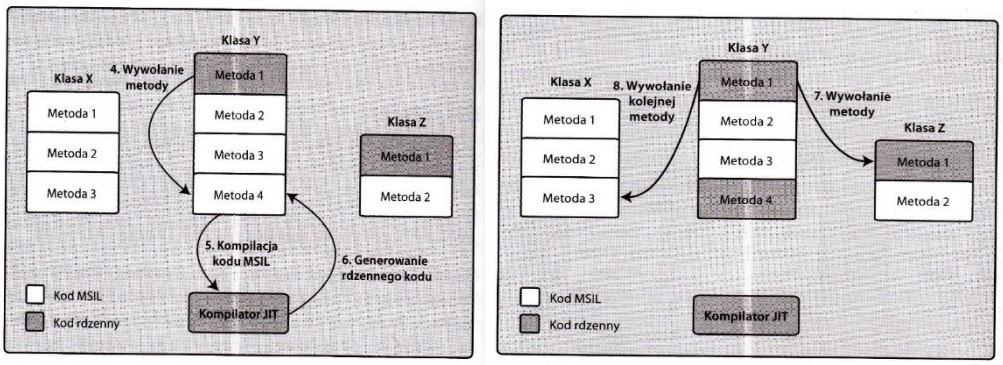
Rysunek 1.6.3.44. Pakiet często jest pojedynczą biblioteką DLL, ale może także zawierać więcej plików



Rysunek 1.6.3.45. Za pierwszym razem, gdy wywoływana jest metoda 1 klasy Z, kompilator JIT musi dokonać translacji kodu metody w CIL na rdzenny kod.

1.6.3.59. **Wyjaśnienie.** Kompilowanie kodu CIL - kompilator, który tworzy kod zarządzany, zawsze generuje ClL. Jednak kod CIL nie może być uruchomiony na żadnym rzeczywistym procesorze. Zanim będzie można go uruchomić, kod CIL musi być jeszcze raz skompilowany do kodu rdzennego dla procesora. na którym zostanie wykonany. Można tego dokonać na dwa sposoby:

1. kod CIL może być skompilowany metoda po metodzie w czasie wykonywania;
2. kod CIL może być skompilowany do kodu rdzennego w całości przed wykonaniem pakietu za pomocą narzędzia NGEN dla danego procesora.



Rysunek 1.6.3.46. Pierwsze i kolejne wywołanie kodu metody

Kiedy wywoływana jest metoda 4 klasy Y - kompilator JIT jest ponownie wykorzystany w celu dokonania translacji metody w CIL na rdzenny procesora kod. Kiedy metoda 1 klasy Z jest ponownie wywoływana, wykonanie kompilacji nie jest już konieczne.

Piśmiennictwo: *Chappel D.* C.1.1.

1.6.4. MONO – ODWZOROWANIE .NET FRAMEWORK NA INNE SYSTEMY OPERACYJNE

Microsoft stanąłby przed sporym wyzwaniem, starając się przekonać klientów, ze długofalowa obsługa .NET Framework na systemach innych od Windows mieści się w jego poważnych planach. Skupienie się na własnym systemie operacyjnym było do tej pory znakiem rozpoznawczym Microsoftu, a także głównym czynnikiem jego sukcesu. W rzeczywistości najbardziej znana obecnie implementacja .NET niezależna od Windows - projekt Mono - nie doczekała się na razie dużego zainteresowania komercyjnego. Niektóre firmy budujące aplikacje w .NET miały nadzieję, że Mono zapewni im bezpieczną, niezależną od Microsoftu alternatywę, kiedy będą takiej potrzebować. Jednak w praktyce taka opcja nie jest zbyt zachęcająca dla wielu organizacji. Czy nam się to podoba, czy nie, większość osób, które tworzą oprogramowanie na bazie.NET Framework, powinna liczyć się z tym, że kod będzie działał na jakiejś wersji Windows.

Mono jest projektem FLOSS prowadzonym przez firmę Xamarin (dawniej przez firmę Novell, a zapoczątkowanym przez firmę Ximian), który ma na celu stworzenie zgodnego ze standardami Ecma zestawu narzędzi umożliwiającego uruchamianie programów stworzonych dla platformy .NET Framework, w skład których wchodzi między innymi kompilator języka C# oraz środowisko CLR (Common Language Runtime). Celem Mono jest nie tylko umożliwienie uruchamiania aplikacji stworzonych dla platformy Microsoft .NET na różnych platformach, ale również umożliwienie twórcom oprogramowania dla systemu Linux znacznie szerszy rozwój. Mono może być uruchamiane na wielu systemach operacyjnych, włączając w to Androida, większość dystrybucji Linuxa, BSD, OS X, Windows, Solaris. Dodatkowo wspierane są najpopularniejsze konsole do gier.

1.6.4.10. **Wyjaśnienie.** Aktualny status prac i plany na przyszłość (patrz również tab. 1.6.4.00). Wersja 3.2.1 (wydana w sierpniu 2013 roku) oferuje podstawowe API .NET Framework, wsparcie dla języków Visual Basic.NET i C# w wersjach 2.0, 3.0 i 4.0, technologię LINQ to Objects, XML i SQL. W chwili obecnej C# 4.0 jest domyślnie wykorzystywany przez kompilator języka C# dostępny w Mono. Dodatkowo wspierany jest graficzny interfejs programowania aplikacji Windows Forms 2.0, ale jego implementacja nie jest jeszcze kompletna.

Tabela 1.6.4.00. Projekt MONO - charakterystyka	
1.	Autor : Xamarin (wcześniej Ximian przejęty później przez Novell) oraz społeczność Mono.
2.	Platforma sprzętowa : x86, x86-64, ARM, MIPS, PowerPC, SPARC, IA-64
3.	System operacyjny : Windows, Linux, OS X, Unix, BSD, Solaris, iOS, Android
4.	Pierwsze wydanie : 30 czerwca 2004
5.	Aktualna wersja stabilna : 3.4.0 (31 marca 2014)
6.	Licencja : GPLv2, LGPLv2 i MIT

Obecnie głównym celem Mono jest zapewnienie pełnego wsparcia dla wszystkich udogodnień .NET 4.0 takich jak WF (Windows Workflow Foundation) oraz w pewnym stopniu WCF (Windows Communication Foundation). Wyjątkiem jest tutaj WPF (Windows Presentation Foundation), który nie jest wspierany ze względu na ogrom pracy, jaki należałoby włożyć, aby funkcjonował on poprawnie. Część z funkcjonalności .NET Framework będących w fazie rozwoju jest częścią subprojektu Mono, któremu nadano nazwę Olive .W projekcie Mono znalazło się również miejsce dla kompilatora i środowiska uruchomieniowego Visual Basic .NET. Pracami nad portem VB.NET zajmuje się Rolf Bjarne Kvinge.

1.6.4.20. **Wyjaśnienie.** W skład projektu Mono wchodzi również otwarta implementacja technologii Silverlight, która nazwana została Moonlight. Jest ona dostępna od wersji Mono 1.9. Moonlight 1.0, które wspiera API Silverlight 1.0 zostało udostępnione w 20 stycznia 2009 roku. Moonlight 2.0 daje możliwości Silverlight 2.0 oraz część z funkcjonalności Silverlight 3.0. Wersja preview Moonlight 3.0, ogłoszona w lutym 2010 roku, dodała wsparcie Silverlight w wersji 3. Jednak rozwój Moonlight został oficjalnie zakończony 29 maja 2012 roku. Oficjalnym powodem była decyzja *Miguela*, który stwierdził, iż istnieją dwa czynniki, które powodują, że dalszy rozwój skazany jest na porażkę – sztuczne ograniczenia Microsoft, które powodują, że Silverlight jest bezużyteczny przy tworzeniu aplikacji na komputery osobiste oraz brak wystarczającej popularności w Internecie, która napędzałaby dalszy rozwój.

1.6.4.30. **Wyjaśnienie.** Mono składa się z trzech grup komponentów:
- Głównych komponentów (Core Components);
- Stosu deweloperskiego Mono/Linux/GNOME;
- Stosu zgodności z systemami Microsoft

1.6.4.31. **Wyjaśnienie.** Główne komponenty:
- Główne komponenty zawierają kompilator C#, wirtualną maszynę dla CLI oraz główne biblioteki klas .NET. Bazują one na standardach Ecma-334 i Ecma-335, przez co Mono jest wolną i otwartą wirtualną maszyną CLI.
- Oba standardy na których bazuje projekt mono, zostały udostępnione przez Microsoft jako OSP (Open Specification Promise), czyli zobowiązanie Microsoft do otwartego rozpowszechniania specyfikacji.

1.6.4.32. **Wyjaśnienie.** Mono/Linux/GNOME - stos rozwojowy Mono/Linux/GNOME dostarcza narzędzi i aplikacji deweloperskich z wykorzystaniem istniejącego środowiska GNOME oraz wolnych i otwartych bibliotek kodu. W jego skład wchodzą:
1. Gtk# do tworzenia GUI (Graphical User Interface);
2. biblioteki Mozilli do pracy z silnikiem renderującym Gecko;
3. biblioteki zgodności z Unix (Mono.Posix);
4. biblioteki do komunikacji z bazami danych;

5. stos zabezpieczeń;
6. RelaxNG (REgular Language for XML Next Generation) – czyli schemat językowy dla XML-a.

Uwaga Gtk# umożliwia integrację aplikacji tworzonych w Mono z pulpitem Gnome jako aplikacji natywnych.

1.6.4.33. **Wyjaśnienie.** Biblioteki baz danych dostarczają możliwość połączenia z najpopularniejszymi bazami danych, takimi jak:

- db4o, Firebird, Microsoft SQL Server (MSSQL), MySQL, ODBC, Oracle, PostgreSQL, SQLite oraz
- z innymi popularnymi silnikami baz danych.

O tym, jakie bazy danych są obsługiwana twórcy projektu informują na bieżąco na swojej stronie internetowej.

1.6.4.40. **Wyjaśnienie.** Zgodność z systemami Microsoft

- Stos zgodności z produktami firmy Microsoft zapewnia możliwość importowania aplikacji MS .NET Framework do systemu GNU/Linux.
- Ta część umożliwia wykorzystanie ADO.NET, ASP.NET i Windows Forms.
- Komponenty te nie są ustandaryzowane przez ECMA, część z nich możliwa była do udostępnienia dzięki umowom miedzy Novellem a Microsoftem.

1.6.4.50. **Wyjaśnienie.** Główne komponenty Mono zawierają:

- Silnik wykonania kodu (*Code Execution Engine*)
- Biblioteki klas
    - Podstawowe biblioteki klas
    - Biblioteki zapewniające zgodność z platformą.NET
    - Charakterystyczne dla Mono biblioteki klas:
        - Międzyplatformowe biblioteki klas dla Mono i .NET Framework (Gtk#, Mono.Cecil, Mono.CSharp, Text.Templating)
        - Charakterystyczne dla Unixa biblioteki klas (POSIX, FUSE (Filesystem in Userspace))
        - Specyficzne dla danej platformy biblioteki klas(umożliwiające połączenie Mono z Mac, iOS, Android i MeeGo)
    - Kompilatory CLI
    - Metadane CLI
    - MonoCLR (Mono Common Language Runtime)
        - Zgodne ze standardami ECMA CLI/.NET CLR
        - Rozszerzenia typowe dla Mono:
            - Wsparcie dla Mono.SIMD
            - Dodatkowe funkcje, metody, kontynuacje i podproframy
    - Charakterystyczne dla Mono rozszerzenia
    - Natywne usługi do współpracy z istniejącymi już komponentami i obiektami COM
    - Wsparcie dla Microsoftowego Security – Transparent Code Framework (.NET 4.5)

1.6.4.60. **Wyjaśnienie.** Code Execution Engine - środowisko uruchomieniowe Mono zawiera silnik, który umożliwia tłumaczenie kodu bitowego ECMA CIL do kodu natywnego, który może zostać wykonany na różnych architekturach procesorów:

- ARM,
- MIPS (jedynie wersji 32-bit),

- SPARC,
- PowerPC,
- S390 (w trybie 64-bit),
- x86, x86-64 I IA-64 w trybie 64-bit.

1.6.4.70. **Wyjaśnienie.** Generator kodu może działać w trzech trybach:

- JIT (Just-in-time) – środowisko uruchomieniowe może w locie zamieniać kod ECMA CIL do natywnego kodu, który może zostać wykonany na urządzeniu.
- AOT (Ahead-of-Time) – w tym trybie kompilator zamienia kod ECMA CIL na kod natywny, który przechowywany jest w formacie charakterystycznym dla danego systemu operacyjnego, architektury systemu komputerowego i CPU) (np. dla plików *.exe, w systemie Linux zostaną wygenerowane pliki *.exe.so). Ten sposób generowania kodu jest najszerzej stosowany, gdyż większość kodu jest kompilowana przed jego użyciem, a nie tak jak ma to miejsce w przypadku kompilatora JIT, tuż przed użyciem. Oczywiście część kodu nadal potrzebuje kompilatora JIT (np. obsługa trampolin czy kodu zarządzającego), także powstałe pliki nie są do końca samowystarczalne.
- Pełna kompilacja statyczna – w tym przypadku kod kompilowany jest całkowicie. Wykorzystane są wszystkie elementy z kompilacji AOT oraz dodatkowo wszelkie trampoliny, funkcje obudowujące (*wrappers*) i dane Proxy, które zostały wykorzystane są wkompilowane do ostatecznego pliku, który może zostać podłączony do programu i kompletnie wyeliminować użycie kompilatora JIT. Możliwość takiej kompilacji istnieje tylko na niektórych z platform, wykorzystana jest w szczególności na systemie iOS i na konsolach PlayStattion 3 i Xbox 360.

1.6.4.80. **Wyjaśnienie.** Code Execution Engine:

- Począwszy od wersji Mono 2.6, istnieje możliwość takiej konfiguracji środowiska, aby używać LLVM (*Low Level Virtual Machine* – maszyna wirtualna niskiego poziomu) jako głównego silnika generowania kodu, zamiast silnika Mono. Ta funkcjonalność jest bardzo przydatna w przypadku programów wykorzystujących duże zasoby komputera oraz w przypadku, gdy szybkość i wydajność wykonania kodu jest ważniejsza niż wydajność uruchomienia aplikacji.
- Od Mono w wersji 2.7 Preview nie ma już potrzeby wyboru silnika generującego kod w trakcie konfiguracji, gdyż może on zostać wybrany jako parametr startowy środowiska (jako jedna z dwóch opcji startowych : – llvm lub – nollvm). Dzięki temu w szybki sposób można wybrać potrzebny w danej chwili generator kodu.

1.6.4.81. **Wyjaśnienie.** Garbage Collector:

- W październiku 2010 roku zaprezentowano *garbage collector* nowej generacji – Sgen-GC (*Simple Generational Garbage Collector*).
- Stał się on integralną częścią Mono 3.1.1 i jest on tam domyślnym silnikiem GC.
- W wersjach Mono 2.8 – 3.1.0 może on zostać wybrany przez użytkownika jako silnik garbage collectora poprzez argument, z kótrym zostanie uruchomione środowisko (–gc=sgen).
- Nowy garbage collector ma wiele udogodnień w stosunku do swoich poprzedników, w szczególności do najbardziej rozpowszechnionego sinika CG Bohema. Wykorzystuje on mechanizm *generational garbage collection.*

1.6.4.82. **Wyjaśnienie.** *Biblioteki Klas:*

- Biblioteki klas dostarczają kompleksowy zestaw narzędzi do tworzenia aplikacji.

- Są one w większości napisane w języku C#, ale zgodnie ze specyfikacją CLS (Common Language Specification) mogą one zostać użyte w którymkolwiek języku dostępnym na platformę .NET.
- Biblioteki klas zebrane są w przestrzenie nazw i opublikowane w postaci współdzielonych bibliotek zebrane w większe zestawy.
- Odwoływanie się do jakichkolwiek klas .NET Framework powoduje właśnie wywołanie wspomnianych bibliotek.

1.6.4.83. **Wyjaśnienie.** *Przestrzenie nazw i klasy*

- Przestrzenie nazw (*namespaces*) są mechanizmem logicznego zebrania podobnych do siebie klas w hierarchiczną strukturę.
- Dzięki temu o wiele łatwiej uniknąć jest konfliktów w nazewnictwie. Przestrzenie nazw zaimplementowane są z wykorzystaniem notacji słów oddzielonych kropkami, dzięki czemu zawsze wiadomo, z której przestrzeni nazw w danej chwili się korzysta.
- Dodatkowo w większości przypadków nadrzędną przestrzenią nazw jest przestrzeń System, więc podrzędne przestrzenie otwierane są jako System.IO (dla operacji wejścia/wyjścia) czy System.Net (dla elementów składowych .NET Framework). Oczywiście istnieje o wiele więcej nadrzędnych przestrzeni nazw, jak na przykład Accessibility czy Windows.

1.6.4.84. **Wyjaśnienie.** Środowisko Mono Develop:

- MonoDevelop jest darmowym, zintegrowanym środowiskiem programistycznym dla środowiska graficznego GNOME, które zostało zaprojektowane do współpracy z językiem C# i innymi językami .NET, takimi jak Nemerle, Boo i Java (poprzez IKVM.NET). Dodatkowo wspiera ono takie języki jak C, C++, Python, Java czy Vala.
- Początkowo MonoDevelop było portem środowiska SharpDevelop do Gtk#, ale później wyewoluowało w narzędzie, które miało sprostać potrzebom bardzo szybko rozrastającej się społeczności użytkowników. Obecnie IDE zawiera system zarządzania klasami, wbudowaną pomoc, uzupełnianie kodu, program Stetic umożliwiający tworzenie interfejsu użytkownika, wsparcie dla projektów i zintegrowany debugger.
- Przeglądarka MonoDoc daje dostęp do dokumentacji API i do przykładowego kodu. Dokumentacja wykorzystuje styl *wiki*, umożliwiając dostęp do edycji i poprawek dla każdego użytkownika.

1.6.4.85. **Wyjaśnienie.** MonoTouch i Mono for Android:

- MonoTouch i Mono for Android są implementacjami platformy programowania Mono dla smartfonów działających pod nadzorem systemów operacyjnych - iOS firmy Apple i Android firmy Google.
- Są one wydane jedynie na licencji komercyjnej, za którą należy zapłacić.

Piśmiennictwo: *Chappell D.* C.1.1.

1.7. NOWE HORYZONTY INFORMATYKI – BIG DATA

1.7.0. NOWE ZJAWISKO SKALI DANYCH

Pojęcie *Big Data* pojawiło się na początku bieżącego stulecia, w związku z wystąpieniem wielkich, tylko częściowo uporządkowanych zbiorów danych, np. danych wszystkich dotychczasowych logowań do przeglądarki Google. Szacuje się, że do 2003 roku wszystkie zarejestrowane dane wynosiły łącznie - około 5 miliardów gigabajtów. Podobna ilość danych była rejestrowana (w układach pamięciowych), co dwa dni w roku 2011, natomiast w roku 2013 – taka ilość danych była już rejestrowana, co dziesięć minut. Te ilości danych, lawinowo rosnące, stworzyły nowe wyzwania dla informatyki.

1.7.0.00. **Wyjaśnienie**. Co rozumiemy pod pojęciem Big Data? Pojęcie *Big Data* obejmuje dane tworzone przez różnorodne aplikacje i urządzenia. Poniżej wymienione zostało kilka grup danych typowo składających się na *Big Data*:

- *Dane z czarnych skrzynek* – samolotów, helikopterów, itd. Zawierające zarówno zapisy rozmów członków załogi jak i najróżniejsze dźwięki z wnętrza statków powietrznych oraz wyniki pomiarowe z urządzeń pokładowych.
- *Dane z mediów społecznościowych* – takich jak Facebook i Twitter zawierające informacje i obrazy wysyłane przez miliony ludzi na świecie.
- *Dane transakcji giełdowych* – obejmujące sprzedaż i zakup papierów wartościowych, zarówno w wyniku przeprowadzanych transakcji przez maklerów, jak również przez systemy automatycznych transakcji.
- *Dane z sieci energetycznych* – dotyczące wytwarzania i konsumpcji energii przez poszczególne węzły sieci energetycznej, jak również dane dotyczące przesyłów między węzłowych energii.
- *Dane z urządzeń transportowych* – różnorodnych modeli pojazdów o stanie technicznym urządzeń, zużyciu paliwa, przejechanych odległości, aktualnym położeniu geograficznym, itp.
- *Dane z wyszukiwarek* – wyszukujące różnorodne informacje z bardzo wielu baz danych, dla różnorodnych baz danych.

Pojęcie *Big Data* obejmuje olbrzymie wolumeny, powstających ze zróżnicowaną szybkością o coraz bardziej zróżnicowanej strukturze – danych. W przybliżeniu, możemy sklasyfikować w trzech typach:

1. *Dane w pełni strukturyzowane* – np. dane zapisane w relacyjnych bazach danych.
2. *Częściowo strukturyzowane dane* – dane XML.
3. *Dane bez standardowej struktury* – dane tworzone przez takie narzędzia jak Word, PDF, Text, Media Logs.

1.7.0.11. **Wyjaśnienie**. Czy zainteresowanie się *Big Data* dostarczy jakichś korzyści? Nie bardzo zdajemy sobie jeszcze sprawę z faktu, jak krytyczne dla naszego życia są *Big Data*. Poniżej wymieniono kilka istotnych korzyści, jakie wynikają z posługiwania się *Big Data*:

a) Wykorzystując dane z mediów społecznościowych, agencje marketingowe dowiadują się, jaki oddźwięk społeczny mają ich kampanie marketingowe, akcje promocyjne oraz inne działania prowadzone na rynkach.
b) Wykorzystując dane z mediów społecznościowych, w rodzaju preferencji oraz percepcji różnorodnych produktów przez ich konsumentów, wspomaga organizacje produkcyjne oraz dystrybucyjne (np. handel detaliczny) w planowaniu działalności.
c) Wykorzystując dane historii chorób dotychczasowych pacjentów, w planowaniu procedur medycznych i leczeniu obecnych i przyszłych pacjentów.

Piśmiennictwo: Mayer-Schonberger Victor M.3.1.

1.7.1. Technologie informatyczne Big Data

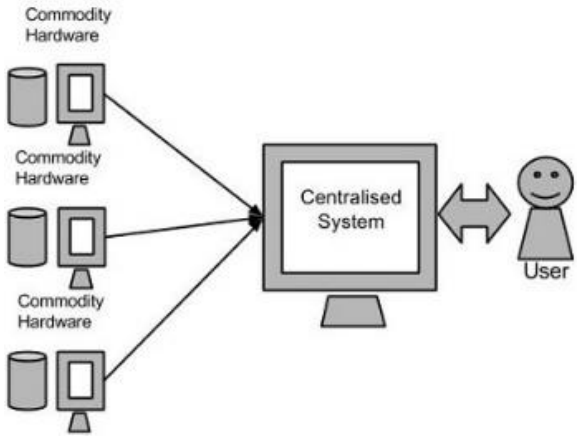
1.7.1.10. **Wyjaśnienie**. Technologie informacyjne dotyczące *Big Data* służą dostarczaniu szczegółowych analiz, które mogą prowadzić do podejmowania decyzji zapewniających wysoką skuteczność, redukcję kosztów oraz redukcję ryzyka w działalności gospodarczej. Korzystanie z *Big Data*, wymaga opracowanie technologii informatycznych opartych o infrastrukturę umożliwiającą operowanie bardzo wielkimi zbiorowiskami danych w czasie rzeczywistym, z jednoczesnym zapewnieniem prywatności oraz bezpieczeństwa tych danych. Różni dostawcy oprogramowania – dostarczyli, co najmniej kilku platform umożliwiających wykonywanie wzmiankowanych zadań. Platformy te umożliwiają realizację następujących funkcjonalności na *Big Data*:

a. Przechwytywanie danych,
b. Oczyszczanie i standaryzację danych,
c. Przechowywanie danych,
d. Selekcja i wyszukiwanie zależności pomiędzy danymi,
e. Dzielenie danych np. według kategorii,
f. Przesyłanie danych,
g. Analiza danych według wskazanych kryteriów,
h. Prezentacja danych.

W dalszym ciągu, omówimy jedną z takich platform opartych o algorytm *MapReduce* - opracowany przez programistów firmy Google, a następnie zrealizowaną, jako Open Source Project przez Apache Software Foundation, pod nazwą HADOOP.

1.7.1.11. **Wyjaśnienie**. Algorytm nazwany `MapReduction` realizuje dwa podstawowe zadania:

1. *The Map Task*: jest pierwszym krokiem algorytmu, który pobiera dane i konwertuje je w zbiór danych (data set), w ramach, którego poszczególne porcje danych są dzielone do postaci mapy n-tek uporządkowanych (klucz, przypisane wartości).
2. *The Reduce Task*: to kolejny krok algorytmu, w którym mapy n-tek uporządkowanych pobierane są, jako dane wejściowe, a następnie łączone są - w mniejszą liczbę n-tek uporządkowanych. Wykonanie wielokrotne tego kroku, jest poprzedzone jednokrotnym wykonaniem pierwszego kroku algorytmu.

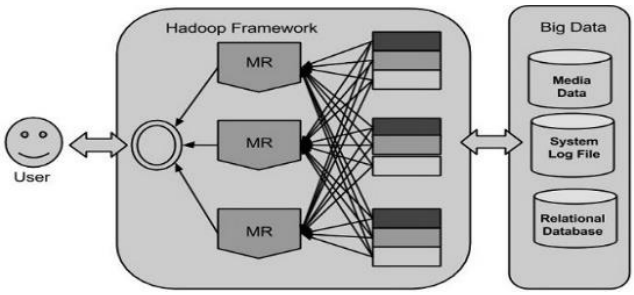


Rysunek 1.7.1.12. Struktura klastra

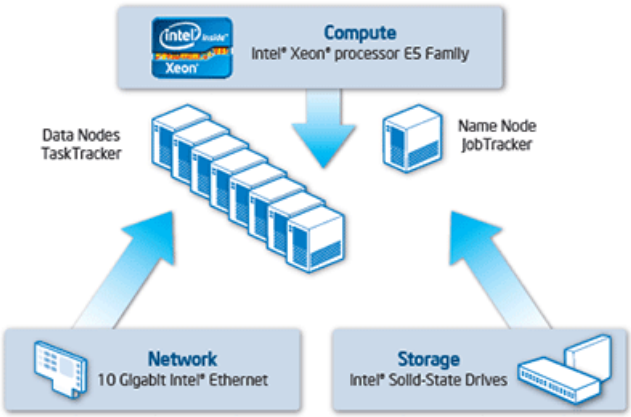
Realizacja algorytmu wymaga zdefiniowania jednego tzw. *JobTracker'a* – mastera klastra platformy oraz po jednym tzw. *TaskTracker* - na każdy podrzędny węzeł (*slave*) klastra platformy (rys. 1.7.1.12). Master klastra jest odpowiedzialny za zarządzanie zasobami, kolejkowanie zasobów, wykorzystywanie (konsumpcja) zasobów i harmonogramowanie

składowych zadań przez poszczególne węzły typu slave, monitorowanie przebiegu realizacji składowych zadań oraz ponowne uruchamiania wykonywania zadania składowego – w przypadku pojawienia się błędu.

Dla potrzeb realizacji powyższego algorytmu opracowano w oparciu o GFS (Google File System), specjalny rozproszony system plików, który nazwano HDFS (Hadoop Distributed File System), porównaj rys. 1.7.1.13. Każdy plik HDFS jest podzielony na szereg bloków, z których każdy przechowywany w jednym węźle klastra platformy.



Rysunek 1.7.1.13. Schemat platformy Hadoop



Rysunek 1.7.1.14. Struktura sprzętu dla programu Hadoop wg. firmy Intel

Jak już zostało powiedziane, *Apache Hadoop* – to otwarta implementacja paradygmatu *MapReduce Google*, która umożliwia tworzenie działających w rozproszeniu aplikacji, przeprowadzające obliczenia na wielkich liczbach danych (przykładowa konfiguracja sprzętowa dla potrzeb platformy *Apache Hadoop* – rys. 1.7.1.14). Jeszcze zanim *Apache Hadoop* osiągnął wydanie stabilne, był już wykorzystywany w poważnych zastosowaniach (Amazon, AOL, Facebook, Yahoo). Autorem i kierownikiem projektu *Apache Hadoop* jest Doug Cutting. Wydanie stabilne 2.7.0, dostępne jest od 21.04.2015.

Piśmiennictwo: *Mayer-Schonberger V*. M.2.1, *Turkinton G*. T.8.1.

1.7.2. Zmiany podejścia – konsekwencja przetwarzania Big Data

W wyniku pojawienia się Big Data pojawiły się trzy nowe podejścia:

1. Pierwsze dotyczy zdolności do analizowania ogromnej liczby danych z określonej dziedziny i brak konieczności ograniczania się do mniejszych zbiorów – zbudowanych z uśrednionych danych, czyli reprezentantów tworzących próbki danych określonej dziedziny.
2. Drugie polega na gotowości do zajmowania się nieuporządkowanymi danymi płynącymi z rzeczywistego świata i nieprzywiązywaniu wielkiej wagi do ich dokładności.

3. Trzecie dotyczy rosnącego znaczenia korelacji i zrezygnowania z pogoni za odkrywaniem nieuchwytnej przyczynowości.

1.7.2.11. **Wyjaśnienie**. *Co zamiast klasyfikacji?* Big Data – spowodowały konieczność odejścia od typowego wcześniej dla nauk eksperymentalnych, operowania małymi zbiorami reprezentantów (próbek), dających się stosunkowo łatwo klasyfikować, np. wprowadzenie:

1. Zamiast próby przyporządkowania każdej danej do ustalonej kategorii, wprowadzenie „*otwartego zbioru tag-ów*", czyli dodawania nowego *tag*-u, tam gdzie istniejące nie pozwalają zaklasyfikować dodawanych danych do ustalonej kategorii.
2. Tym samym również - akceptujemy możliwość uwzględniania błędów w niektórych z wprowadzanych *tag*-ach, co jest nieodłączną cechą – odpowiadającą naturalnemu bezładowi panującego w realnym świecie.
3. Takie podejście, to antidotum na sztuczne systemy, które próbowano narzucić rozgardiaszowi, udając, że wszystko można przedstawić za pomocą jednoznacznych klasyfikacji.

1.7.2.12. **Wyjaśnienie**. *Nowe oblicze Data Mining*. W literaturze polskiej termin *data mining* utożsamia się z takimi pojęciami, jak eksploracja danych, drążenie danych, zgłębianie danych lub odkrywanie wiedzy w zbiorowiskach danych. Ostatnie z wymienionych pojęć nabrało nowego znaczenia w wyniku pojawienia się *Big Data*. Przez odkrywanie wiedzy rozumiemy cały proces wykorzystania *data mining'u* w celu "zidentyfikowania i wydobycia niezliczonej ilości modeli analitycznych wiedzy, stosowanie do sprecyzowanych ograniczeń i celów, z wykorzystania wszelkich możliwych metod, techniki narzędzi do przetwarzania wstępnego, modelowania i przekształcania bazy faktów i do oceny wyników wyszukiwania danych". Podsumowania i zależności będące wynikiem eksploracji zwane są modelami lub wzorcami, a ich przykładami mogą być:

- Równania liniowe lub nieliniowe,
- Reguły,
- Grafy,
- Struktury drzewiaste,
- Wzorce rekurencyjne w szeregach czasowych.

W powyższej definicji dane określono, jako dane obserwacyjne, co sugeruje, że gromadzi się je z innych przyczyn niż cele analiz prowadzących do wydobywania wiedzy. Oznacza to, ze cele eksploracji danych nie odgrywają żadnej roli w strategii gromadzenia danych. Jest to cecha, która odróżnia eksplorację danych od statystyki. Inną taką cechą jest wspomniana w definicji wielkość zbiorowisk danych. W przeciwieństwie do statystyki, eksplorację danych stosuje się do Big Data, a nie do jego próbki. Zastosowanie metod *data mining* obejmuje przede wszystkim automatyczne odkrywanie nieznanych wcześniej wzorców oraz przewidywanie trendów i zachowań. Uogólniając, można wskazać pięć głównych typów zadań (zastosowań) eksploracji danych, które odpowiadają różnym celom osób analizujących dane:

1. Eksploracyjna analiza danych (*exploratory data analysis*);
2. Modelowanie opisowe (*descriptive modeling*),
3. Modelowanie przewidujące - predykcyjne (*predictive modeling*),
4. Odkrywanie wzorców i reguł (*pattern and rules search*),
5. Wyszukiwanie według zawartości wzorca (*pattern similarity search*).

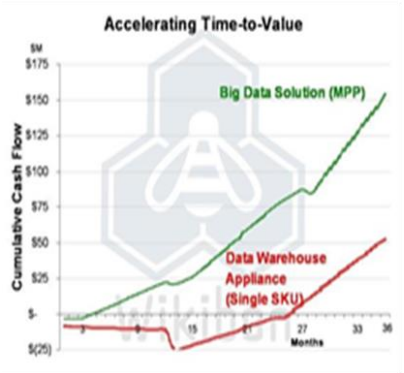
1.7.2.13. **Wyjaśnienie**. *Tradycyjne podejście do opisu rzeczywistości, a Big Data*. Większość instytucji zajmujących się badaniami rzeczywistości (np. urzędy statystyczne, instytucje badania

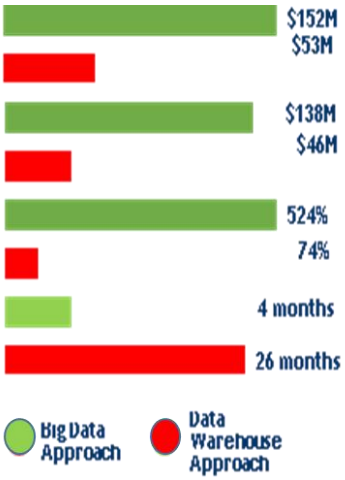
popytu, instytucje badania opinii publicznej, itp.), pracują w oparciu o stosunkowo małe próbki danych, z założenia próbki reprezentatywne (tzw. próbki losowe) dla prowadzonego badania. Od 1934 roku, kiedy Jerzy Neyman[21] opublikował pracę poświęconą między innymi, zasadom budowy reprezentatywnych próbek losowych, metoda badań oparta na próbkach (rzekomo reprezentatywnych), była dominującą. Jak pokazała praktyka, nie zawsze udaje się, zbudowanie losowej próbki reprezentatywnej ze względu na przyjęty cel badania. W pewnych sytuacjach, Big Data stwarza możliwość oparcia badania na niemal 100% reprezentacji, co czasami prowadzi do zaskakujących wniosków.

Piśmiennictwo: Mayer-Schonberger Victor M.3.1, Neyman Jerzy N.1.1.

1.7.3. Porównanie ROI obliczanego klasyczną metodą, z metodą Big Data

SAS Institute - w publikacji internetowej[22], zwócił uwagę na bardzo ciekawy wniosek. Wniosek wynikający z zastosowania podejścia opartego o bezpośrednią analizę Big Data przy wyznaczaniu ROI dla pewnej zrealizowanej inwestycji w porównaniu do klasycznej metody użycia danych z próbki „losowej" za pośrednictwem analitycznej bazy danych z OLAP. Okazało się, co widać na wykresie, że wynik obliczeń oparty o podejście Big Data jest znacznie korzystniejszy od opartych o dane z próbki, wyników OLAP (rys. 1.7.3.10). Oczywiście, jest to prawdopodobnie dość szczególny przypadek. Ale źle dobrane próbki zdarzają się częściej niż myślimy.





Rysunek 1.7.3.10. Porównanie obliczeń próbki i *Big Data*, zwrotu nakładów ROI (według SAS Institute)

Piśmiennictwo: *Mayer-Schonberger* V. M.3.1., *Neyman J*. N.1.1.

1.7.4. Analizy Big Data pokazują istnienie bardzo wielu korelacji

Korelacja oznacza związek pomiędzy zmiennymi. Analiza korelacji służy do „wychwycenia" zachodzących związków pomiędzy dwiema różnymi zmiennymi (właściwościami, cechami). Dotychczas przeprowadzone analizy *Big Data* wskazują na istnienie bardzo wielu, nieraz wręcz zaskakujących korelacji. W dotychczasowej praktyce, przywykliśmy do szukania związków przyczynowo – skutkowych, bo często wydaje się, że każde zdarzenie musi mieć swoją przyczynę. Tymczasem *Big Data* uczy nas, że związki korelacyjne są bardzo częste, natomiast związki przyczynowo – skutkowe, występują w rzeczywistości, rzadziej niż nam się to wydaje.

---

[21] Jerzy Neuman, *On the Two Different Aspects of the Representative Method of Stratified Sampling and the Method of Purposive Selection*, "Journal of the Royal Statistical Society" 97, No 4, p. 558-625.
[22] Thomas H. Davenport, Jill Dyche – *Big Data in Big Companies*, International Institute for Analytics, May 2013.

Na marginesie, warto zauważyć, że starożytni Grecy – twórcy rachunku zdań - uznali, *że logiczna funkcja implikacji jest odpowiednikiem związku korelacyjnego, a nie związku przyczynowo – skutkowego*. Świadczy to o ich głębokiej przenikliwości.

Piśmiennictwo: *Mayer-Schonberger V*. M.3.1.

1.8. KODOWANIE INFORMACJI

1.8.0. DANE A INFORMACJE

Pojęcia *informacji* i *algorytmu*, są podstawowymi pojęciami dziedziny zwanej informatyką. Algorytmami będziemy zajmować się dalej, między innymi w rozdziale 3.8. Obecnie poświęcimy nieco miejsca informacji, a dokładniej mówiąc liczbie bitów niezbędnej dla przechowywania danej informacji w pamięci komputera. Z praktycznego punktu widzenia, bardzo często dążymy do korzystania z minimalnej liczby bitów, wykorzystanych dla zapisania zadanej informacji.

1.8.0.10. **Wyjaśnienie**. *Kompresją bezstratną*: ciągu binarnego A(x) (sekwencji bitów – pliku binarnego), służącego do zapisania zadanej informacji x, nazywamy każdy taki ciąg binarny $A^{\#}(x)$ który jest krótszy co najmniej o jeden bit (lub więcej bitów), od zadanego ciągu binarnego A(x) – umożliwiający przechowywanie tej samej, zadanej informacji x.

1.8.0.20. **Wyjaśnienie**. *Programów komputerowych kompresji bezstratnych:* istnieje wiele, jednym z popularniejszych jest np. Win RAR[23], należy jednak zauważyć, że w wyniku użycia jednego z tych programów kompresji bezstratnych, uzyskamy najkrótszy ciąg binarny niezbędny do zapisania zadanej informacji.

1.8.0.30. **Wyjaśnienie**. Nie należy mylić kompresji bezstratnych z *kompresjami stratnymi*, te ostatnie służą głównie do kompresji grafiki i muzyki, kosztem jednak wprowadzenia pewnych zmian – uproszczeń w kompresowanym pliku, np. na drodze wykorzystania zależności rekursji, czyli samo-podobieństwa.

1.8.0.40. **Wyjaśnienie**. *Złożoność Kołmogorowa* – to długość najkrótszego programu komputerowego, który generuje dany ciąg znaków, np. ciąg binarny. Nazwa pojęcia pochodzi od nazwiska wybitnego matematyka rosyjskiego Andrieja Kołmogorowa. Rozwinięcie dziesiętne liczby $\boldsymbol{\pi}$, choć nieskończone, ma bardzo niską złożoność Kołmogorowa, ponieważ istnieje bardzo prosty program, który generuje dowolną liczbę jej cyfr. Złożoność Kołmogorowa jest różna dla różnych komputerów (ściślej – maszyn Turinga). Ze względu na nierozstrzygalność problemu stopu maszyny Turinga, nie może istnieć uniwersalny algorytm obliczający złożoność Kołmogorowa.

1.8.0.50. **Wyjaśnienie**. Istnieją ciągi znaków, w szczególności ciągi binarne, które nie dają się kompresować bezstratnie. Ciągi te mają szereg ciekawych własności. Np. można pokazać, że nie dający się kompresować ciąg binarny złożony z n znaków, ma w przybliżeniu tą samą liczbę zer i jedynek, a długość najdłuższego spójnego podciągu zer wynosi w przybliżeniu $\log_2 n$.

Piśmiennictwo: *Sipser M*. S.7.1.

[23] RAR umożliwia kompresję, szyfrowanie, odzyskiwanie danych i wiele innych funkcji opisanych w instrukcji programu.

1.8.1. PODSTAWOWE ZESTAWY ZNAKÓW UŻYWANYCH W INFORMATYCE

1.8.1.10. **Wyjaśnienie**. Pojęcie znaku odnosi się do symbolu zrozumiałego dla człowieka lub dla komputera, zwykle niebędącego liczbą. W zasadzie znak to symbol, który można wpisać z klawiatury i zobaczyć na monitorze. Pamiętajmy, że poza literami do danych znakowych zaliczają się też znaki przystankowe, cyfry, spacje, tabulatory, znaki powrotu karetki (klawisz ENTER), inne znaki kontrolne i znaki specjalne.

1.8.1.20. **Wyjaśnienie**. Większość systemów do kodowania różnych znaków wykorzystuje pojedyncze bajty lub pary bajtów. Dotyczy to także systemów operacyjnych, takich jak Windows i Linux wykorzystujących zestawy znaków ASCII lub Unicode, gdzie znaki zapisuje się w pojedynczym bajcie lub w dwubajtowym ciągu. Zestaw znaków EBCDIC używany w komputerach IBM klasy *mainframe* i w minikomputerach, to kolejny przykład jednobajtowego kodu znaków. Omówimy dalej wszystkie trzy podstawowe zestawy znaków i ich zapis wewnętrzny. sposób tego odwzorowania jest w gruncie rzeczy dość przypadkowy i niezbyt istotny, ale istnienie takich standardów pozwala na komunikowanie się rożnych programów i urządzeń peryferyjnych. Np. standardowe kody ASCII są przydatne, ponieważ w krajach anglosaskich niemal wszyscy je używają. Wobec tego, jeśli użyjemy kodu ASCII 65 do zapisania znaku A, to dowolne urządzenie peryferyjne - na przykład drukarka - prawidłowo zinterpretuje tę wartość jako literę A.

1.8.1.30. **Wyjaśnienie**. Podstawowy zestaw znaków ASCII zawiera tylko 128 różnych znaków, więc pojawia się ważne pytanie - co z pozostałymi 128 wartościami ($80..$FF), które można zapisać na jednym bajcie? Odpowiedź brzmi: pomijamy te znaki. Istnieje możliwość rozszerzenie zestawu ASCII o dalsze 128 znaków. O ile jednak wszyscy użytkownicy - nie będą zgodni co do dokładnego sposobu tego rozszerzenia, to podważony zostanie sens używania standardowego zestawu znaków. Mimo poważnych niedostatków, zestaw znaków ASCII jest standardem wymiany danych, między programami i komputerami. Większość programów potrafi odczytać dane ASCII, a także je zapisywać.

Piśmiennictwo: *Hyde R.* H.5.1.

1.8.2. ZESTAW ZNAKÓW ASCII

1.8.2.10. **Wyjaśnienie**. Znaki ASCII dzieli się na cztery grupy zawierające po 32 znaki, Pierwsze 32 znaki o kodach od $0 do $lF (od 0 do 31) to specjalny zbiór znaków niedrukowalnych nazywanych znakami kontrolnymi. Nazwa ta wzięła się stąd, że realizują one różne funkcje sterujące drukarką i monitorem, a nie są pokazywane jako takie. Przykładami znaków kontrolnych mogą być: *powrót karetki*, powodujący umieszczenie kursora na początku bieżącego wiersza; *nowy wiersz*, przenoszący kursor wiersz niżej; *backspace*, który przesuwa kursor o jeden znak w lewo (czyli cofa). Niestety, niektóre znaki kontrolne powodują różne działania poszczególnych rodzajów i typów urządzeń wyjściowych, Standaryzacja w tym zakresie jest niewielka. Aby mieć pewność, że wiemy, co dany znak kontrolny wykonuje w używanym urządzeniu, trzeba sprawdzić, to w dokumentacji,

1.8.2.20. **Wyjaśnienie**. Druga grupa 32 znaków ASCII to różne symbole przestankowe, znaki specjalne i cyfry. Najważniejsze znaki z tej grupy to *spacja* (kod ASCII $20) oraz *cyfry* (kody $30..$39).

1.8.2.30. **Wyjaśnienie**. Trzecia grupa 32 znaków zawiera wielkie litery. Kody liter od A do Z mieszczą się w zakresie $41,… $5A. W alfabecie łacińskim jest tylko 26 liter, więc pozostałych 6

znaków zawiera różne symbole specjalne. Ostatnia grupa 32 znaków zawiera małe litery, pięć dodatkowych znaków specjalnych i jeszcze jeden znak kontrolny, *delete* (usuwający znak spod kursora). Zwróćmy uwagę na to, że małe litery wykorzystują kody ASCII $61…$7A. Jeśli zamieniamy małe litery na wielkie lub odwrotnie, różnica między małą a odpowiadającą jej wielką literą to tylko jeden bit. Oba maki różnią się tylko piątym bitem. Wielkie litery mają piąty bit równy zero, małe - równy jeden.

Tabela 1.8.2.00			
grupa	Bit 6	Bit 5	Opis grupy
1	0	0	znaki kontrolne
2	0	1	cyfry i znaki przestankowe
3	1	0	wielkie litery i znaki specjalne
4	1	1	małe litery i znaki specjalne

1.8.2.40. **Wyjaśnienie**. Można skorzystać z tego, aby szybko zamieniać małe litery na wielkie i odwrotnie; wystarczy przełączyć tylko jeden bit. Bity piąty i szósty decydują o przynależności znaku do grupy (tabela 1.8.2.00). Wobec tego małe litery na wielkie (lub na znaki specjalne), można zamieniać, ustawiając bity piąty i szósty na zero.

1.8.2.50. **Wyjaśnienie**. Przyjrzyjmy się przez chwilę kodom ASCII cyfr, pokazanym w tabeli 1.8.2.00. Dziesiętny zapis tych kodów niewiele nam mówi. Jednak zapis szesnastkowy ujawnia coś ważnego - młodszy półbajt kodu ASCII to binarny odpowiednik zapisywanej liczby. Jeśli odrzucimy starszy półbajt kodu (ustawimy go na zero), otrzymamy binarny zapis odpowiedniej cyfry. Z drugiej strony, łatwo możemy zamienić liczbę binarną z zakresu 0..9 na jej odpowiednik ASCII.

1.8.2.60. **Wyjaśnienie**. Mimo ze ASCII jest standardem, kodowanie danych za jego pomocą nie gwarantuje nam pełnej przenośności między różnymi systemami. Chociaż literze A na jednym komputerze będzie odpowiadało A na innej, zakres standaryzacji znaków kontrolnych jest niewielki. Faktycznie spośród pierwszych 32 kodów kontrolnych ASCII uzupełnionych kodem *delete* z grupy ostatniej tylko cztery są powszechnie obsługiwane przez większość urządzeń i programów. Są to: *backspace* (BS), *tabulator*, *powrót karetki* (CR). Próba przekazania zwykłego pliku tekstowego między dwoma komputerami może być źródłem frustracji. Nawet jeśli używamy standardowych znaków ASCII, i tak musimy zawczasu skonwertować dane. Na szczęście konwersje tę są proste na tyle, że wiele edytorów tekstu automatycznie obsługuje pliki z różnie kończącymi się wierszami.

Piśmiennictwo: *Hyde R.* H.5.1.

1.8.3. Zestaw znaków EBCDIC

1.8.3.10. **Wyjaśnienie**. IBM na wielu swoich komputerach mainframe i minikomputerach używa kodu EBCDIC. Kod ten pojawia się głównie na dużych komputerach, więc poświęcimy mu niewiele uwagi. EBCDIC to skrót od angielskiego *Extended Binary Coded Decimal Interchange Code*, czyli rozszerzony kod wymiany danych dziesiętnych kodowanych binarnie. Czy istniał kiedyś taki kod nierozszerzony? Owszem, kiedyś w maszynach IBM systemu kart dziurkowanych (lata 1890 – 1950), używano zestawu znaków znanego jako BCDIC. Zestaw ten, był oparty na kodzie dziurek w zapisie dziesiętnym. Pierwsze, co trzeba powiedzieć o EBCDIC - nie jest to pojedynczy zestaw znaków, ale cała rodzina takich zestawów. Zestawy znaków EBCDIC mają wspólną część (na przykład litery zwykle są kodowane tak samo), ale różne wersje EBCDIC (nazywane stronami kodowymi) różnie kodują znaki przestankowe i specjalne. Na

pojedynczym bajcie liczba możliwych kodowań jest ograniczona, w różnych stronach kodowych te same kody są używane do zapisu różnych znaków. Jeśli zatem mamy plik ze znakami EBCDIC i mamy go zapisać jako ASCII, szybko okaże się, że to nie jest wcale proste zadanie.

1.8.3.20. **Wyjaśnienie**. Zanim bliżej zainteresujemy się zestawem znaków EBCDIC, musimy sobie zdać sprawę, że przodek EBCDIC, czyli BCDIC, istniał na długo przed pojawieniem się współczesnych komputerów. BCDIC powstał na potrzeby maszyn systemu kart dziurkowanych. Natomiast EBCDIC został pomyślany jako proste rozszerzenie kodowania tak, by umożliwić stosowanie go w pierwszych komputerach IBM. Jednak EBCDIC odziedziczył po BCDIC pewne nietypowe, dzisiaj już anachroniczne cechy. Na przykład kodowanie liter alfabetu łacińskiego, nie jest ciągłe (czyli, uporządkowane alfabetycznie znak po znaku). Początkowo litery zapewne były kodowane na kolejnych znakach, Jednak kiedy IBM rozszerzył zestaw znaków, użyto kombinacji binarnych niewystępujących w formacie BCD (wartości binarne 1010..1111). Takie wartości binarne występują między dwiema dotąd sąsiadującymi wartościami BCD, więc niektóre ciągi znaków (na przykład litery) nie są zapisywane w kodowaniu EBCDIC w sposób ciągły. Niestety, z uwagi na nietypowość zestawu znaków EBCDIC wiele powszechnie stosowanych algorytmów działających na zestawie ASCII w przypadku EBCDIC po prostu nie działa.

Piśmiennictwo: *Hyde R.* H.5.1.

1.8.4. Dwubajtowe zestawy znaków

1.8.4.10. **Wyjaśnienie**. Z uwagi na ograniczenia kodowania 8-bitowego (co oznacza maksymalnie 128 różnych znaków) oraz na konieczność zapisania większej liczby znaków, w części systemów używa się specjalnych kodów potrzebujących do zapisania jednego znaku na dwóch bajtów. Takie dwubajtowe zestawy znaków nie używają 16 bitów do zapisu każdego znaku; w większości przypadków używany jest jeden bajt, a tylko w niektórych - dwa bajty.

1.8.4.20. **Wyjaśnienie**. Typowy dwubajtowy zestaw znaków rozszerza standardowy zestaw ASCII z dodatkowymi znakami z zakresu $80..$FF. Niektóre z nich informują, że pojawi się drugi bajt. Każdy bajt rozszerzenia pozwala zapisać 256 dodatkowych znaków. Mając trzy wartości rozszerzające, można kodować 1021 różnych znaków. Z każdego bajtu rozszerzającego otrzymujemy 256 znaków, poza tym mamy 253 (256 - 3) znaków w standardzie jednobajtowym (minus trzy, gdyż tyle znaków służy jako znacznik rozszerzający, który w związku z tym nie powinien być liczony jako zwykły znak).

Piśmiennictwo: *Hyde R.* H.5.1.

1.8.5. Zestaw znaków Unicode

Jakiś czas temu inżynierowie firm *Apple Computer* i *Xerox* stwierdzili, ze ich nowe systemy z sprzętu komputerowego z wyświetlaczami mozaikowymi i czcionkami wybieranymi przez użytkownika mogą wyświetlić o wiele więcej niż 256 znaków naraz. Choć można było zastosować kodowania dwubajtowe, stwierdzono, że występują poważne problemy związane z faktem, że znaki mają po dwa bajty, i poszukiwano innego rozwiązania, Okazał się nim zestaw znaków UNICODE. Standard ten przyjął się na całym świecie w związku z rozwojem Internetu i jest obsługiwany przez niemalże każdy system komputerowy i system operacyjny (Mac OS, Windows, Linux, Unix i wiele innych).

1.8.5.10. **Wyjaśnienie**. W UNICODE do zapisu każdego znaku używa się 16-bitowych słów, więc można zapisać do 65.536 różnych znaków. Jest to oczywiście o wiele, wiele więcej niż dotychczasowe 256 możliwych znaków w 8-bitowym bajcie. Mało tego, UNICODE jest zgodny z ASCII. Jeśli najstarszych 9 bitów, jest ustawionych na zero, młodszych 7 to kod ze standardowego zestawu ASCII. Jeśli 9 starszych bitów zawiera wartości niezerowe, całe 16 bitów tworzy znak rozszerzony (rozszerzony względem ASCII). Jeśli ktoś się jeszcze zastanawia, po co właściwie mamy aż tyle kodów znaków, wystarczy przypomnieć, że niektóre azjatyckie zestawy zawierają ich 4096 (tyle znaków mają ich podzbiory w UNICODE). Zestaw znaków UNICODE zawiera nawet kody, które mogą być używane jako znaki definiowane na potrzeby poszczególnych aplikacji.

1.8.5.20. **Wyjaśnienie**. Obecnie wiele systemów operacyjnych i bibliotek do różnych języków programowania zawiera obsługę UNICODE. Na przykład system Microsoft Windows używa standardu UNICODE wewnętrznie, co powoduje, że funkcje systemowe działają odpowiednio szybciej. Jednak UNICODE ma też dwie poważne wady. Po pierwsze, dane zajmują dwa razy więcej miejsca niż w przypadku ASCII lub innych kodowań jednobajtowych. Chociaż obecnie komputery mają znacznie więcej pamięci niż kiedyś (dotyczy to tak pamięci RAM, jak i pamięci dyskowych), podwojenie rozmiaru plików tekstowych, baz danych i ciągów znaków umieszczanych w pamięci (jak ciągi znaków przetwarzane przez edytory i procesory tekstu) może znacząco wpłynąć na wydajność systemu. Druga wada UNICODE jest taka, że większość wcześniejszych - istniejących gdziekolwiek na świecie plików z danymi jest zapisana jako ASCII lub EBCDIC, więc w przypadku korzystania w aplikacji z UNICODE pewną ilość czasu trzeba poświęcić na konwersję zestawu znaków.

Piśmiennictwo: *Hyde R*. H.5.1.

1.9. UWAGI O SYSTEMACH PROGRAMOWANIA I ZASTOSOWANIU KOMPUTERÓW

1.9.1. PARADYGMATY PROGRAMOWANIA KOMPUTERÓW

Warto przyjrzeć się znaczeniu słowa „paradygmat", bezlitośnie nadużywanemu przez filozofów, lingwistów i informatyków — by wymienić tylko paru sprawców zamieszania. Otóż, jak podaje *Słownik języka polskiego PWN*, paradygmat to «przyjęty sposób widzenia rzeczywistości w danej dziedzinie, doktrynie np. zespół form fleksyjnych (deklinacyjnych lub koniugacyjnych), właściwy danemu typowi wyrazów; wzorzec, model deklinacyjny lub koniugacyjny». Jak to się ma do programowania? Szereg autorów, w tym Wikipedia, formułują paradygmat programowania jak niżej:

1.9.1.00. **Definicja**. Paradygmat programowania (*programming paradigm*) — wzorzec programowania przedkładany w danym okresie rozwoju informatyki ponad inne lub szczególnie ceniony w pewnych okolicznościach lub zastosowaniach. Paradygmat programowania definiuje sposób patrzenia programisty na przepływ sterowania i wykonywanie programu komputerowego.
Przykład: W programowaniu obiektowym jest on traktowany, jako zbiór współpracujących ze sobą obiektów, podczas gdy w programowaniu funkcyjnym definiujemy, co trzeba wykonać, a nie, w jaki sposób. Zależności między paradygmatami programowania mogą przybierać skomplikowane formy, ponieważ jeden język może wspierać wiele różnych paradygmatów.

Język, C++ posiada elementy programowania proceduralnego, obiektowego oraz uogólnionego, co stanowi o nim, że jest hybrydowym językiem.

Niektórzy autorzy uważają, że mamy do czynienia z bardzo wieloma różnorodnymi paradygmatami programowania, których zakres nie zawsze jest rozłączny:

1. *Imperatywnym*
2. *Obiektowym*
3. *Proceduralnym*
4. *Strukturalnym*
5. *Funkcyjnym*
6. *Zdarzeniowym*
7. *Logicznym (np. przy użyciu języka Prolog)*
8. *Aspektowym*
9. *Deklaratywnym*
10. *Agentowym*
11. *Modularnym.*

W dalszy ciągu zajmiemy się omówieniem dwóch, naszym zdaniem kluczowych paradygmatów. A mianowicie, paradygmatami programowania: imperatywnego i obiektowego.

1.9.1.10. **Wyjaśnienie**. Programowanie imperatywne to najbardziej pierwotny sposób programowania, ściśle związany z architekturą komputerów wywodzącą się od Johna von Neumana, w którym program postrzegany jest, jako ciąg poleceń dla komputera:

- Ściślej, obliczenia rozumiemy tu, jako sekwencję poleceń zmieniających krok po kroku stan maszyny, aż do uzyskania oczekiwanego wyniku.
- Stan maszyny należy z kolei rozumieć, jako zawartość całej pamięci oraz rejestrów i znaczników procesora.
- Ten sposób patrzenia na programy związany jest ściśle z budową sprzętu komputerowego o architekturze von Neumanna, w którym poszczególne instrukcje (w kodzie maszynowym) to właśnie polecenia zmieniające ów globalny stan.
- Języki wysokiego poziomu — takie jak Fortran, Algol, Pascal, Ada lub C — posługują się pewnymi abstrakcjami, ale wciąż odpowiadają paradygmatowi programowania imperatywnego.

Przykładowo, instrukcje podstawienia działają na danych pobranych z pamięci i umieszczają wynik w tejże pamięci, zaś abstrakcją komórek pamięci są zmienne. Programowanie imperatywne jest tak stare jak komputery, a nawet starsze. Przepisy kuchenne czy sformalizowane procedury urzędowe można uznać za przejaw programowania imperatywnego. Oczywiście używane dzisiaj języki imperatywne są znacznie bogatsze niż te z czasów pionierskich. Niesłychanie ważnymi elementami, o które wzbogacił się ten paradygmat, to programowanie strukturalne (używanie prostych, dobrze zdefiniowanych struktur jak np. pętla, oraz unikanie skoków) i programowanie proceduralne.

1.9.1.20. **Wyjaśnienie**. W programowaniu obiektowym program to zbiór porozumiewających się ze sobą obiektów, czyli jednostek zawierających pewne dane i umiejących wykonywać na nich pewne operacje:

- Podejście obiektowe, obejmujące klasy obiektów o wspólnych atrybutach oraz operacjach (zwanych metodami) i interfejsy umożliwiające łączenia pomiędzy sobą obiekty należące do różnych, a wyposażone w ten sam interfejs,

- Ważną cechą jest tu powiązanie atrybutów (czyli stanu), z operacjami na nich (czyli poleceniami) w całość, stanowiącą odrębną jednostkę — obiekt.
- Cechą nie mniej ważną jest mechanizm dziedziczenia, czyli możliwość definiowania nowych, bardziej złożonych obiektów, na bazie obiektów już istniejących i bardziej złożonych interfejsów na bazie już istniejących.
- Obiekty tworzą konstrukcje kapsułowaną (*an capsulation*), czyli konstrukcje wiążące mniejsze jednostki w zamkniętą całość, które można użytkować, znając ich funkcjonalność – bez wchodzenia w szczegóły realizacyjne.

Zwolennicy programowania obiektowego uważają, że ten paradygmat dobrze odzwierciedla sposób, w jaki ludzie myślą o świecie, nawet, jeśli pogląd ten uznamy za nie w pełni uzasadniony, to niewątpliwie programowanie obiektowe zdobyło ogromną popularność i wypada je uznać za paradygmat obecnie dominujący. Przykładami języków programowania obiektowego są: JAVA i C#.

1.9.1.21. **Definicja**. Abstrakcja to — w naszym rozumieniu — reprezentacja pewnego bytu, w której pominięto nieistotne w danym kontekście szczegóły. Pozwala nam to grupować byty według ich wspólnych cech, albo zajmować się całymi grupami (czyli owymi wspólnymi cechami), albo bytami wewnątrz grupy (czyli szczegółami różniącymi bajty w grupie).

1.9.1.22. **Wyjaśnienie**. Dwie podstawowe abstrakcje w językach programowania to:

1. Abstrakcja procesu: Abstrakcjami procesów są podprogramy. Pozwalają wskazać (przez ich wywołanie), że pewna czynność ma być wykonana, bez wskazywania *jak* ma być wykonana. Szczegóły znajdują się w treści podprogramu, której wywołujący nie musi znać.
2. Abstrakcja danych: Zamknięta całość obejmująca reprezentację pewnego typu danych wraz z podprogramami, umożliwiającymi działanie na tych danych.

1.9.1.21. **Definicja**. Abstrakcyjny typ danych, jest to konstrukcja języka programowania, w której definiujemy typ danych oraz operacje na nim w taki sposób, że inne byty w programie nie mogą manipulować tymi danymi inaczej niż za pomocą zdefiniowanych przez nas operacji. Istotą rzeczy jest tu oddzielenie części „prywatnej" typu (czyli szczegółów reprezentacji danych i implementacji poszczególnych operacji) od części „publicznej" (tego, co można wykorzystywać w innych miejscach programu). Rozdzielenie składników abstrakcyjnego typu danych na część prywatną i publiczną jest możliwe za pomocą zawartych w języku programowania mechanizmów sterowania dostępem.

Piśmiennictwo: *Wikipedia* W.2.32.

1.9.2. ZASTOSOWANIA KOMPUTERÓW

Około roku 1950 pojawiła się opinia kilku specjalistów z USA, w zakresie technik obliczeniowych, że na całym świecie potrzebnych będzie łącznie kilkadziesiąt komputerów. Pamiętać przy tym trzeba, iż współczesny tablet lub notebook posiada moc obliczeniową przekraczającą znacznie, moc obliczeniową komputerów EDSAC i EDVAC. Nie licząc specjalizowanych komputerów, przeznaczonych do bardzo różnorodnych celów, jak np. smartphony albo sterowniki urządzeń AGD, czy też sterowniki obrabiarek, park wykorzystywanych komputerów na pewno przekracza miliard systemów. Oczywiście, zastosowania komputerów dotyczą bardzo wielu dziedzin życia współczesnej ludzkości, takich jak: nauka, przemysł, medycyna, edukacja, administracja publiczna, bankowość i ubezpieczenia, kryminalistyka, twórczość artystyczna i rozrywka, itd.

1.9.2.10. **Wyjaśnienie**. Zastosowania w nauce, to przede wszystkim: badania naukowe (np. symulacja zjawisk i procesów prowadzona na modelach informatycznych) oraz rejestracja i analiza wyników badań doświadczalnych.

1.9.2.20. **Wyjaśnienie**. Zastosowania w przemyśle, to - sterowanie procesami technologicznymi, projektowanie produktów i projektowanie technologii, prowadzenie dokumentacji technicznej oraz technologicznej produktów i procesów, zastosowanie systemów informatycznych takich jak: ERP, CRM, zarządzanie przedsięwzięciami, prowadzenie ewidencji, rachunkowości, itp.

1.9.2.30. **Wyjaśnienie**. Zastosowanie w medycynie, to – prowadzenie dokumentacji medycznej wraz z możliwością lokalnego i zdalnego posługiwania się tą dokumentacją, sterowanie aparaturą, graficzne zobrazowanie wyników badania (np. rezonansu magnetycznego), projektowanie protez, wykorzystanie skanerów 3D i drukarek 3D do wypełniania ubytków kostnych, itd.

1.9.2.40. **Wyjaśnienie**. Zastosowania w edukacji, to – wykorzystanie techniki slajdów w procesie dydaktycznym, nauczanie na odległość, udostępnianie na odległość literatury fachowej, itd.

1.9.2.50. **Wyjaśnienie**. Zastosowania w administracji publicznej, czyli jest to – prowadzenie różnorodnych ewidencji, masowa korespondencja np. związana z pobieraniem danin publicznych, zdalne korzystanie z usług i uzyskiwanie decyzji administracyjnych, itd.

1.9.2.60. **Wyjaśnienie**. Zastosowania w bankowości i ubezpieczeniach, to – prowadzenie ewidencji, zdalna obsługa klientów, prowadzenie masowej korespondencji, itd.

1.9.2.70. **Wyjaśnienie**. Zastosowania w kryminalistyce, to – prowadzenie bazy danych (np. odcisków linii papilarnych), badanie i porównanie materiałów dowodowych, obróbka zdjęć, dokumentowanie zdarzeń, tworzenie portretów pamięciowych, itd.

1.9.2.80. **Wyjaśnienie**. Zastosowania w twórczości artystycznej, to – grafika komputerowa i druk 3D, obróbka zdjęć cyfrowych, projektowanie wnętrz i obiektów (np. architektonicznych), komponowanie muzyki, pisanie książek i innych utworów.

1.9.2.90. **Wyjaśnienie**. Zastosowania w rozrywce, to – zdjęcia cyfrowe, pisanie tekstów i rysowanie, gry komputerowe, wideo-łączność, Internet, itd.

Piśmiennictwo: *Wikipedia* W.2.33.

Część 2.
Co wynika ze współczesnej logiki formalnej?

2.0. UWAGI WSTĘPNE

2.0.1. POCHODZENIE LOGIKI

Źródeł powstania logiki należy szukać w starożytnej Grecji, gdzie między innymi powstał zbiór zasad wnioskowania – zwanych *sylogizmami*. Rozwój logiki helleńskiej obejmuje okres około trzystu lat, rozpoczął się około 500 roku p.n.e., a kończy się około 200 roku p.n.e. Dorobek tych trzystu lat jest w zakresie logiki niezwykle bogaty. Grecy stworzyli *logikę zdań*, *logikę nazw* oraz zaczątki tzw. *logiki funktorów*, formułowali i badali *antynomie*, a ponadto w dziedzinie metalogiki stworzyli pojęcia *systemu zaksjomatyzowanego*, mianowicie *stoicki rachunek zdań* i *euklidesowy system geometrii elementarnej*. Grecy zapoczątkowali też logikę indukcji.

Za ojca logiki helleńskiej uznano powszechnie Arystotelesa. Zanim jednak Arystoteles zaczął uprawiać logikę jako dość wyodrębnioną całość, powstały już zaczątki logiki w postaci przyczynków do mającej się dopiero narodzić odrębnej nauki. Warto wymienić najważniejszych twórców tego procesu formowania się logiki:

1. *Pitagoras z Samos* (urodzony w VI wieku p.n.e., zmarł w Metaponcie) i pitagorejczycy (uczestnicy szkoły filozoficznej założonej przez Pitagorasa, która istniała przez blisko dwieście lat) – twórcy dowodów matematycznych.
2. *Eleaci* (uczestnicy szkoły filozoficznej, o której w karykaturalnej nieco formie można by pokrótce powiedzieć: jeśli wynik rozumowania nie zgadza się z faktami, tym gorzej dla faktów; niemniej jednak uzyskali bardzo istotny wynik dla dalszego rozwoju logiki) – odkrywcy antynomii.
3. *Sokrates* (urodzony w roku 469 p.n.e. w Atenach i zmarły tamże w roku 399 p.n.e.) – twórca metody indukcji prostej oraz metody wnioskowania dialektycznego.
4. *Megarejczycy* (założycielem szkoły megaryjskiej był *Euklides z Megary*, zwany też „Euklidesem Sokratykiem", żyjący w latach 450-380 p.n.e., szkoła ta z zamiłowaniem poszukiwała antynomii; jedną z nich jest antynomia *Kłamca*) – twórcy dalszych antynomii.
5. *Hipokrates* (zwany „ojcem medycyny", genialny lekarz grecki, urodził się w roku 460 p.n.e. na wyspie Kos, zginął w roku 377 p.n.e. w Larissie. Był autorem lub współautorem zbioru pism medycznych pod nazwą *Corpus Hippocrateum*) – stworzył zaczątki logiki indukcji przyrodniczej.
6. *Platon* (zwany Ateńczykiem, urodził się w roku 427 p.n.e., zmarł w roku 347 p.n.e., był uczniem Sokratesa) – domniemany twórca metody aksjomatycznej.
7. *Arystoteles* (urodzony w Stagirze w roku 384 p.n.e., zmarł w Chalkis w roku 322 p.n.e. W roku 367 przybył do Aten. W Akademii platońskiej spędził dwadzieścia lat, z początku jako uczeń oraz zwolennik platonizmu, następnie jako samodzielny, oryginalny myslicie.) – twórca logiki nazw, autor pisma między innymi z dziedziny logiki. Dzieła naukowe Arystotelesa przechowały się w redakcji *Andronikosa z Rodos*. Otrzymały one później nazwę *Organon* (narzędzie) i obejmowały sześć traktatów: (1) *Kategorie*; (2) *O wyrażeniu*; (3) *Analityki wcześniejsze* – traktat o dowodzie i wnioskowaniu; (4) *Analityki późniejsze* – również o dowodzie i wnioskowaniu; (5) *Topikas* – dowody prawdopodobne i sztuka prowadzenia sporów; (6) *O sofizmatach* – traktat o *sylogizmach.*

8. *Tweofrast* (urodzony w Eresos około roku 372 p.n.e., zmarł w Atenach około roku 287 p.n.e.) – kontynuator prac Arystotelesa, między innymi: (1) wzbogacił liczbę czterech trybów sylogizmu kategorycznego o pięć dalszych nazywanych pośrednimi; (2) Wprowadził nowe tryby sylogizmu asertorycznego; (3) Sformułował nieznane Arystotelesowi nowe twierdzenia należące do logiki zdań, takie jak przechodniość implikacji, modus *ponendo-ponens* i modus *tollendo-tollens*; (4) udoskonalił logikę wyrażeń modalnych; (5) przeprowadził badania nad antynomią „*Kłamca*".
9. *Epikur* (urodzony w roku 341 p.n.e., zmarł w roku 270 p.n.e.) - założył w Atenach szkołę filozoficzną zwaną „*Ogrodem Epikura*". Tzw. *Epikurejczycy* byli współtwórcami logiki indukcji.
10. *Stoicy* (szkoła stoików, została założona w III wieku p.n.e. przez Zenona z Kition. Usystematyzowanie logiki stoickiej, zawdzięczamy *Chryzyp'owi* urodzonemu około roku 280 p.n.e.) – pierwsi użyli terminu logika i byli twórcami rachunku zdań.
11. *Euklides* (żył na przełomie IV i III wieku p.n.e., studiował w Atenach na Akademii platońskiej. Za czasów *Ptomeleusza I*, wykładał geometrię w Aleksandrii) – twórca pierwszego wielkiego systemu aksjomatycznego geometrii przedstawionego w pracy zatytułowanej *Elementy*.

Piśmiennictwo: *Greniewski H*. G.2.1.

2.0.2. XIX-WIECZNE PODSTAWY LOGIKI WSPÓŁCZESNEJ

Formalizacja logiki rozpoczęła się dopiero w końcu XIX wieku i została poprzedzona szeregiem bardzo istotnych również XIX wiecznych wyników. Wymienimy jedynie kilku twórców naszym zdaniem najważniejszych, podając ich nazwiska i osiągnięcia:

1. *Mikołaj Łobaczewski* (urodzony w 1793 roku – zmarły w 1856 roku, matematyk rosyjski - profesor Uniwersytetu w Kazaniu), *Karol Fryderyk Gauss* (urodzony w 1777 roku – zmarły w 1885 roku, znakomity matematyk i astronom niemiecki, profesor uniwersytetu w Getyndze), *Jan Bolyai* (urodzony w 1802 roku – zmarły w 1860, Węgier z zawodu oficer wojsk inżynieryjnych) i *Bernard Rieman* (urodzony w 1826 roku – zmarły w 1866 roku, matematyk niemiecki, wykładowca uniwersytetu w Getyndze) – niezależnych od siebie nawzajem - twórcy geometrii nieeuklidesowych.
2. *Jozef Diez Geronne* (urodzony w 1771 roku – zmarły w 1859 roku, matematyk i astronom francuski) – odkrywca zasady dwoistości w geometrii rzutowej.
3. *George Boole* (urodzony w 1815 roku – zmarły w 1864 roku, wybitny logik i matematyk brytyjski, genialny samouk - profesor uniwersytetu w Lincoln) – twórca algebry logiki (nazwanej na jego cześć Algebrą Boole'a).
4. *August De Morgan* (urodzony w 1806 roku – zmarły w 1871 roku, matematyk brytyjski, profesor uniwersytetu w Londynie), *Płaton Porecki* (urodzony w 1846 roku – zmarły w 1907 roku, logik rosyjski, profesor uniwersytetu w Kazaniu) i *Erst Schröder* (urodzony w 1841 roku – zmarły 1902 roku, logik niemiecki, profesor uniwersytetu w Karlsruhe) – kontynuatorzy algebry logiki.
5. *Gottlob Frege* (urodzony w 1848 roku – zmarły w 1925 roku, wybitny logik i matematyk niemiecki, profesor uniwersytetu w Jenie) – twórca pierwszego systemu sformalizowanego logiki.

Przełom, który nastąpił w logice w XX wieku, został przygotowany w dużej mierze przez początki formalizacji tradycyjnej logiki, i przez wyżej wymienionych twórców XIX wieku.

Piśmiennictwo: *Greniewski H.* G.2.1.

2.0.3. TEORIE LOGIKI A INFORMATYKA

Zanim zajmiemy się bliżej kilkoma teoriami logiki formalnej, zastanowimy się w jakim zakresie można mówić o przydatności logiki w formułowaniu podstaw teoretycznych informatyki. Zacznijmy od pewnych istotnych różnic pomiędzy podejściami logiki i informatyki. Logika nie zakłada, że zbiory którymi operuje – posiadają skończone liczby elementów, wprost przeciwnie – najciekawsze wnioski, jak np. rozstrzygalność albo jej brak - dotyczy zbiorowisk poza-skończonych. Informatyka natomiast operuje zawsze zbiorami o skończonej liczbie elementów. Wprawdzie mogą być to liczebności zboru bardzo duże, ale niemniej zawsze są skończone. Pojemność największych pamięci jest zawsze skończona, jak również czas działania w rozumieniu informatyki, jest zawsze dyskretny – kwantowy. W teoriach logiki możemy zetknąć się zarówno z czasem ciągłym, jak i czasem dyskretnym.

W informatyce *Czas Kwantowy* będziemy traktować, jako zbiór elementów czasu - zwanych dalej „*chronami*". Rozmyślnie nie przesądzamy, czy *chron* (dowolny) jest chwilą (utożsamianą z impulsem zegarowym = *punktem chronologicznym*, typowa sytuacja dla urządzeń synchronicznych), czy też okresem czasu (utożsamianym z okresem czasu, jaki upływa pomiędzy dwoma kolejnymi impulsami zegarowymi = *przedziałem chronologicznym*, typowa sytuacja dla urządzeń asynchronicznych). Jednak przyjmujemy alternatywę rozłączną: Albo każdy *chron* jest chwilą, albo też każdy *chron* jest okresem. Ponadto przyjmujemy, że jeżeli każdy *chron* jest okresem, to każde dwa *chrony* mogą nie być równo trwałe. Ta pierwsza różnica podejścia logiki oraz informatyki - nie wydaje się być istotną. Warto zauważyć, że niedoskonałość naszego wzroku umożliwia np. obrazowanie na filmie lub wideo - ruchu z pomocą serii statycznych obrazów, wyświetlanych w odpowiednio szybko w kolejnych krótkich *chronach* czasu. Natomiast sprawa liczebności poza-skończonych zbiorów bardzo istotna w niektórych teoriach logiki, nie znajduje swojej odpowiedniości w informatyce. W konsekwencji predykaty logiczne: kwantyfikator istnienia „∃" oraz kwantyfikator generalizacji „∀"- są, czym innym - mimo stosowania identycznej symboliki, w rozumieniu *Logiki Pierwszego Rzędu* (patrz rozdział 2.7) operującej zbiorami poza skończonymi, a *Notacji Z* (patrz rozdział 4.4) – operująca zbiorami w rozumieniu informatyki – zawsze skończonymi. A przecież *Notacja Z* jest skutecznym narzędziem formułowania i analizowania wymagań na aplikacyjne systemy informatyczne, wykorzystującą metody wnioskowania zaczerpnięte z logiki. Trzeba jednak pamiętać, że *zjawisko częściowej nierozstrzygalności* (w rozumieniu *Kurta Gödla*) – dotyczy między innymi *Logiki Pierwszego Rzędu*, a nie dotyczy teorii skończonych.

Odpowiedniość pomiędzy teoriami logiki a współczesną informatyki, w pewnym zakresie przypomina odpowiedniości pomiędzy płaską geometrią *Euklidesa* a geometrią rastrową, jaką posługuje się grafika komputerowa – celem zobrazowania graficznego krzywych i figur geometrycznych na ekranie monitora komputera. Algorytmy *Bresenhama* - odwzorowania odcinków i fragmentów krzywych płaskich na rastrze (patrz rozdział 4.8), prezentują odcinki prostej oraz wycinki krzywej na ekranie - jako zbiory rozłącznych odcinków odpowiednich długości (poziomych, pionowych lub dwusiecznych kąta - pomiędzy osiami współrzędnych) wyświetlanych z odpowiednim doborem kolorystyki i luminescencji. Rozłączność tych odcinków, jest widoczne na odpowiednio dużym powiększeniu, natomiast jeśli zbiór tych odcinków jest na pograniczu rozdzielczości oka ludzkiego, to wytwarza złudzenie linii ciągłej.

Piśmiennictwo: *de Berg M.* B.4.1., *Liscov B.* L.1.1., *Tiuryn J.* T.4.1.,*Wikipedia* W.2.12., W.2.13., *Woodcock J.* W.7.1., *Zabrodzki J.* Z.1.1.

2.0.4. Program komputerowy jako wyrażenie logiczne

Programowanie komputerów jest dzisiaj jednym z ważniejszych zastosowań logiki, ponieważ każdy program komputerowy jest formułą logiczną, podobnie jak każda formuła zdaniowa rozpatrywana w rachunku zdań. Tak jak zdanie, ze względu na kryteria rachunku zdań - może być prawdziwe lub fałszywe, tak program komputerowy, w sensie poprawności działania jego sterowania, może działać poprawnie (*prawda*) lub niepoprawnie (*fałsz*). Walidacja – to postępowanie mające na celu sprawdzenie w przypadku programu jego poprawność działania. Walidacja programu komputerowego, jest więc niczym innym jak dowodzeniem poprawności działania danego programu. Zanim jednak przystąpimy do omawiania problematyki dowodzenia poprawności programów komputerowych zajmiemy się podstawami formalizacji.

Piśmiennictwo: *Ben-Ari M.* B.2.1., *Hoare C.A.R.* H.4.1., *Holzmann G.* H.3.1.

2.1. Systemy zaksjomatyzowane i sformalizowane

2.1.0. Uwagi wstępne

Jak wspominał Henryk Greniewski, nieżyjący już od wielu lat - logik polski, prof. Leśniewski w 1922 roku zadał słuchaczom pierwszego roku matematyki na Uniwersytecie Warszawskim, pytanie: *Czy wyrażenie „Memena zakefeniła Fufrę" jest zdaniem?* Spróbujmy to rozstrzygnąć. Zauważmy przede wszystkim, że nie jest przewidziane, jaki tu język wchodzi w grę, co stanowi przecież okoliczność istotną. Toteż niejeden pedant oświadczy, że pytanie Leśniewskiego nie jest dostatecznie sprecyzowane, aby można było dać na nie odpowiedź. Taki pedant będzie miał niewątpliwie rację. Nam jednak chodzi o coś innego, o to mianowicie, że niekompletne pytanie Leśniewskiego nasuwa następujące skojarzenia:

1) Odnosi się wrażenie, że zwrot „*Memena zakefeniła Fufrę*"- jest zdaniem w jakimś języku, którego gramatyka jest niezmiernie zbliżona do gramatyki języka polskiego.
2) Wydaje się, że wyraz *„Memena"*- jest nazwą jednostkową należącą do tego języka.
3) Wydaje się również, że wyraz „*Fufra*" - jest nazwą jednostkową, należącą do tego języka.
4) Wydaje się wreszcie, że wyraz „*zakefenić*"- jest funktorem zdaniotwórczym od dwu argumentów jednostkowo nazwowych, należącym do tegoż języka.

Dość trudno jest wyjaśnić ogólnie, co to są wyrażenia formalne. Miałoby się ochotę poprzestać na wyjaśnieniu jedynie za pomocą przykładów. Wyrażenia formalne, to na przykład wyrażenia następujące: *Memena*, *Fufra*, *zakefenić*, itd. Trudno jednak uznać takie za wystarczające, należy - więc dać wyjaśnienie ogólne, co też uczynimy w następnych podrozdziałach.

Piśmiennictwo: *Greniewski H.* G.2.1.

2.1.1. Pojęcie teorii

2.1.1.00. **Wyjaśnienie**. *Teoria* (z greckiego *theoría* - oglądanie, rozważanie) - system pojęć, definicji, aksjomatów i twierdzeń, ustalających relacje między tymi pojęciami i aksjomatami, tworzący spójny system pojęciowy, opisujący jakąś wybraną dziedzinę fizyczną lub abstrakcyjną.

2.1.1.01. **Wyjaśnienie**. W logice formalnej *teorią* nazywamy niesprzeczny zbiór zdań. Dokładniej, niech Π będzie zbiorem zdań zapisanych w pewnym języku Λ. Wtedy Π jest teorią, jeśli nie istnieje zdanie napisane w języku Λ takie że Π dowodzi zarówno tego zdania, jak i jego zaprzeczenia. Zbiór zdań Π dowodzi zdania Φ, jeśli można przeprowadzić formalny dowód

zdania Φ przy użyciu zdań ze zbioru Π oraz aksjomatów i reguł dowodzenia klasycznego rachunku logicznego.

2.1.1.10. **Wyjaśnienie**. Teoria zamknięta. Czasami w definicji teorii dodatkowo zakłada się, że jest ona zamknięta ze względu na operację wnioskowania logicznego. Oznacza to, że jeśli teoria Π dowodzi jakiegoś zdania Φ, to zdanie Φ musi należeć do Π.

2.1.1.20. **Twierdzenie** o zwartości mówi, że zbiór zdań jest niesprzeczny, jeśli każdy jego skończony fragment jest niesprzeczny. W świetle powyższej definicji niesprzeczności wydaje się to oczywiste, bo jeśli z danego zbioru zdań możemy udowodnić zarówno jakieś zdanie, jak i jego zaprzeczenie, to możemy też przeprowadzić ten sam dowód korzystając tylko za skończenie wielu zdań z tego zbioru. Jeśli jednak badamy to zagadnienie z punktu widzenia *semantyki*, a nie *syntaktyki*, to potrzebujemy *twierdzenia o istnieniu modelu*, które w 1931 roku udowodnił austriacki logik i matematyk Kurt Gödel. Mówi ono, że każda spójna teoria (tzn. taka dla której nie istnieje dowód sprzeczności) ma model i umożliwia badanie własności dowolnej teorii przy użyciu metod teorii modeli.

2.1.1.30. **Wyjaśnienie**. Teoria Π w języku Λ jest *zupełna*, jeśli dla każdego zdania Φ napisanego w języku Λ w teorii Π można dowieść zdania Φ lub jego zaprzeczenia (tj.: suma domknięcia Π ze wzgl. na wyprowadzanie oraz jego negacji jest równa zbiorowi wszystkich zdań w Λ). Przy użyciu zakładanego zwykle przez matematyków *aksjomatu wyboru* można wykazać, że każdą teorię w jakimś języku Λ można rozszerzyć do teorii zupełnej w tym języku.

2.1.1.40. **Wyjaśnienie**. Teoria Π w języku Λ jest *rozstrzygalna*, jeśli istnieje algorytm, który dla każdego zdania Φ napisanego w języku Λ rozstrzyga, czy Π dowodzi Φ.

2.1.1.50. **Wyjaśnienie**. Teoria Π jest *kategoryczna*, jeśli Π ma dokładnie jeden model z dokładnością do izomorfizmu. Jest to raczej rzadkie zjawisko, bo kategoryczne są tylko te teorie, które są zupełne i mają model skończony.

Piśmiennictwo: *Greniewski H.* G.2.1.

2.1.2. SYSTEMY ZAKSJOMATYZOWANE

2.1.2.00. **Wyjaśnienie**. Przez „*system zaksjomatyzowany*" rozumiemy każdą taką i tylko taką parę (uporządkowaną[24]) zespołów wyrażeń, która spełnia wszystkie warunki następujące:

1) Zespół pierwszy składa się wyłącznie z wyrażeń następujących rodzajów:
    a) Wyrażeń języka naturalnego, uznanych za powszechnie zrozumiałe;
    b) Wyrażeń niezdefiniowanych, umieszczonych na osobnej liście;
    c) Wyrażeń, które można zbudować za pomocą wyrażeń wymienionych w punkcie a) lub b), stosując powszechnie przyjęte sposoby budowania wyrażeń;
    d) Wyrażeń, które zostają poprawnie zdefiniowane za pomocą wyrażeń wymienionych w punkcie a) lub b), lub c);
2) Zespół drugi składa się wyłącznie z tez (sformułowanych jedynie za pomocą wyrażeń należących do zespołu pierwszego), mianowicie z tez dwu tylko rodzajów:
    a) Tez przyjętych bez dowodu;

[24] Przez „parę uporządkowaną" rozumiemy każdą i tylko taką parę przedmiotów, dla której podany jest sposób odróżniania przedmiotu pierwszego od drugiego (np. tym sposobem może być numeracja obu przedmiotów).

b) Tez dowiedzionych w oparciu o tezy wymienione w punkcie a) przy zastosowaniu powszechnie przyjętych sposobów dedukowania;

3) Tezy przyjęte bez dowodu wyjaśniają rozumienie wyrażeń niezdefiniowanych, umieszczonych na osobnej liście[25].

2.1.2.10. **Wyjaśnienie**. Przez „*wyrażenie pierwotne*" systemu zaksjomatyzowanego rozumiemy każde i tylko takie wyrażenie, które jest niezdefiniowane i umieszczone na osobnej liście wyrażeń niezdefiniowanych tego systemu.

2.1.2.11. **Wyjaśnienie**. Przez „*wyrażenie wtórne*" systemu zaksjomatyzowanego rozumiemy każde i tylko takie wyrażenie, które jest zdefiniowane za pomocą wyrażeń uznanych za ogólnie zrozumiałe lub wyrażeń pierwotnych.

2.1.2.12. **Wyjaśnienie**. Przez „*język*" danego systemu zaksjomatyzowanego rozumiemy pierwszy zespół wyrażeń tego systemu.

Jak widać z wyjaśnień: 2.1.2.00, 2.1.2.10, 2.1.2.11 oraz 2.1.2.12, wyrażenia złożone języka systemu zaksjomatyzowanego są zbudowane ostatecznie z dwu rodzaju wyrażeń:

1) Wyrażeń uznanych za powszechnie zrozumiałe;
2) Wyrażeń pierwotnych.

Doświadczenie uczy, że zwykle język systemu zaksjomatyzowanego jest językiem mieszanym.

2.1.2.20. **Wyjaśnienie**. Przez „*tezę pierwotną*", czyli „*postulat*", ewentualnie „*aksjomat*" systemu zaksjomatyzowanego rozumiemy taką i tylko taką tezę należącą do drugiego zespołu tego systemu, która została przyjęta bez dowodu.

2.1.2.21. **Wyjaśnienie**. Przez „*tezę wtórną*", czyli „*twierdzenie*" systemu zaksjomatyzowanego rozumiemy taką i tylko taką tezę należącą do drugiego zespołu tego systemu, która nie jest tezą pierwotną.

2.1.2.22. **Wyjaśnienie**. Tezy wtórne, spełniające raczej pomocniczą rolę w dowodach, noszą nazwę „*lematów*".

Jak widać z wyjaśnień 2.3.2.00, 2.3.2.20 oraz 2.3.2.21, każda teza - należąca do drugiego zespołu danego systemu zaksjomatyzowanego jest albo tezą pierwotną, albo tezą wtórną.

2.1.2.23. **Wyjaśnienie**. Przez „*teorię*" zawartą w systemie zaksjomatyzowanym rozumiemy drugi zespół jego wyrażeń.

Jak widać z wyjaśnień: 2.1.2.00, 2.1.2.12 oraz 2.1.2.23, każdy system zaksjomatyzowany jest parą uporządkowaną, złożoną z języka tego systemu oraz teorii w nim zawartej. Doświadczenie wykazuje, że teoria dowolnego systemu zaksjomatyzowanego jest teorią w rozumieniu ustalonym w paragrafie poprzednim.

Jako przykład systemu zaksjomatyzowanego (i to przykład klasyczny) podać można dobry szkolny wykład geometrii elementarnej. Na liście wyrażeń pierwotnych znajdujemy tu na przykład wyrażenia: punkt, prosta, płaszczyzna. Wśród postulatów znajdujemy zwykle, między

[25] W związku z punktem 3) wyjaśnienia 2.6.2.00, dobrze jest przypomnieć uwagi o metodzie postulatowej (paragraf 2.4.8).

innymi, tego rodzaju zdanie: Dwa (różne) punkty wyznaczają dokładnie jedną prostą, na której oba leżą.

Piśmiennictwo: *Euclides* E.1.1., *Jordan Z.* J.4.1., *Greniewski H*. G.2.1. , *Kotarbiński T.* K.5.1., *Mostowski A*. M.5.1

2.1.3. Słabe strony aksjomatyzacji

Język dowolnego systemu zaksjomatyzowanego jest konglomeratem, w którym dość istotne znaczenie mają tak zwane wyrażenia ogólnie zrozumiałe. Budowanie wyrażeń złożonych odbywa się według reguł tak zwanych „ogólnie przyjętych", nawet niezebranych na jakiejś wyodrębnionej liście. Nie sposób - zatem twierdzić, że budowanie wyrażeń i definiowanie w języku jakiegokolwiek systemu zaksjomatyzowanego jest obiektywnie unormowane i bezsporne.

Wnioskowanie w obrębie teorii systemu zaksjomatyzowanego odbywa się także wedle reguł tak zwanych „ogólnie przyjętych", nawet niezebranych na jakiejś wyodrębnionej liście. W dodatku dedukcje (dowody twierdzeń) w obrębie teorii bywają nieraz obciążone wadami języka systemu, o których mówiliśmy wyżej. Trudno - więc twierdzić, że rozumowanie w jakimkolwiek systemie zaksjomatyzowanym jest obiektywnie unormowane i bezsporne.

Nic - więc dziwnego, że w niejednym systemie zaksjomatyzowanym zdołano zbudować antynomie. Zbudowanie zaś antynomii w obrębie danego systemu zaksjomatyzowanego świadczy o braku wartości tego systemu, o jego niezgodności z rzeczywistością.

Dla usunięcia wzmiankowanych powyżej wad systemów zaksjomatyzowanych, dla nadania cech bezsporności w obrębie teorii systemu (1) procesowi budowania wyrażeń języka oraz (2) procesowi dedukcji (dowodzenia), dla uniknięcia antynomii zbudowano pojęcie doskonalsze od pojęcia *systemu zaksjomatyzowanego*, mianowicie pojęcie *systemu sformalizowanego*. W rozdziale następnym będziemy mogli pozwać również pojęcie systemu sformalizowanego, poznać płynące zeń korzyści, poznać jego zakres zastosowań i wreszcie przeprowadzić jego krytykę. Ponieważ pojęcie systemu sformalizowanego jest dość skomplikowane, wypadnie poznać najpierw pewne pojęcia pomocnicze, mianowicie pojęcie wyrażenia formalnego, pojęcie jednorodnego systemu wyrażeń formalnych oraz pojęcie niejednorodnego systemu wyrażeń formalnych.

Gdy jakiś język (mieszany albo sztuczny) i wypowiedzianą w tym języku teorię przedstawiamy w formie systemu zaksjomatyzowanego, wówczas otrzymujemy tylko pewien „*obraz statyczny*" naszej wiedzy, przedstawiamy pewien etap rozwojowy danego języka i danej teorii, nie otrzymujemy natomiast obrazu „*dynamiki rozwojowej*" ani języka, ani teorii.

Piśmiennictwo: *Euclides* E.1.1., *Greniewski H*. G.2.1.

2.1.4. Czynności formalne, wyrażenia formalne i merytoryczne

Wśród czynności wykonywanych na wyrażeniach wyróżnimy teraz czynności formalne. To wyróżnienie ułatwi nam zredagowanie ogólnego wyjaśnienia - czym są wyrażenia formalne.

2.1.4.00. **Wyjaśnienie**. Mówimy, że dana czynność jest formalna, zamiast mówić, ze spełnione są wszystkie warunki następujące:

1) Czynność ta jest wykonywana wyłącznie na wyrażeniach;
2) Wykonanie czynności tej zawsze wymagają znajomości strony materialnej wyrażenia, na którym czynność tę wykonujemy;

3) Wykonanie tej czynności nigdy nie wymaga znajomości strony znaczeniowej (rozumienia) wyrażenia, na którym czynność tę wykonujemy;
4) Wykonanie tej czynności zawsze wymaga znajomości przynajmniej reguł poprawności językowej (np. kaligraficznych czy gramatycznych) mających zastosowanie do wyrażenia, na których tę czynność wykonujemy;
5) Wynikiem wykonania tej czynności jest przynajmniej w niektórych wypadkach - wyrażenie.

Powszechna praktyka językowa łącznie z powyższym wyjaśnieniem skłaniają nas do przyjęcia następującej reguły:

2.1.1.01. **Reguła**. Prawidłowa odmiana gramatyczna, kopiowanie, zastępowanie, podstawianie, dołączanie, skreślanie i odrywanie są czynnościami formalnymi.
Przykład. Pewien niedbały uczeń nie wie, kogo w języku łacińskim oznacza nazwa jednostkowa „*C. Julius Caesar*", niemniej potrafi nazwę tę prawidłowo deklinować.

2.1.1.10. **Wyjaśnienie**. Mówimy, że dane wyrażenie jest formalne, zamiast mówić, ze spełnione są oba warunki następujące:
1) Istnieją reguły poprawności językowej (np. kaligraficzne czy gramatyczne) umożliwiające wykonywanie przynajmniej jednej czynności formalnej:
    a) Na tym wyrażeniu
    lub
    b) Na tym wyrażeniu łącznie z innymi wyrażeniami;
2) Strona znaczeniowa tego wyrażenia jest pusta (nieokreślona).

Zajmijmy się teraz bliżej wyjaśnieniem 2.1.1.10. Wyjaśnienie to stawia dwa warunki, które dane wyrażenie ma spełniać, aby było uznane za wyrażenie formalne; zauważmy, że pierwszy z tych warunków odnosi się do strony materialnej tego wyrażenia, drugi – do jego strony znaczeniowej. Zauważmy jeszcze, że poznaliśmy, i to już dość dawno, pewne rodzaje wyrażeń formalnych.

2.1.1.11. **Reguła**. Każda zmienna jest wyrażeniem formalnym (wyjaśnienie 2.3.1.10, 2.3.1.21).

2.1.1.12. **Reguła**. Każde wyrażenie okazjonalne wypowiedziane (czy napisane) bez gestu czy też bez sąsiadującego tekstu, nadających wyrażeniu temu stronę znaczeniową, jest wyrażeniem formalnym (wyjaśnienie 2.1.1.10, 2.1.1.00).

Trzeba się, chociaż chwilę zastanowić, nad celowością używania wyrażeń formalnych. Przecież każde wyrażenie formalne - to, że tak powiemy, „*wyrażenie-makieta*" czy też „*wyrażenie-widmo*", języka zaś (obojętnie w danym wypadku, którego) używamy po to, żeby utrzymać łączność w społeczeństwie z innymi ludźmi, żeby opisywać rzeczywistość materialną. Po cóż, więc wprowadzać do języka jakieś „*wyrażenia-makiety*", jakieś wyrażenia, które nic nie wyrażają?

Otóż w niełatwej tej sprawie można na razie poczynić następujące uwagi:
1) Z zakresu naszych wątpliwości należy wyłączyć te wszystkie wyrażenia formalne, które noszą nazwę „*zmienna*". Wszelkiego rodzaju zmienne zdały egzamin przydatności w różnych naukach, zwłaszcza w logice, matematyce, fizyce, technice i informatyce. Był to egzamin w skali historycznej, zmienne bowiem - są stosowane w logice około 2500 lat. Trzeba - więc tylko rozpatrzyć nasze wątpliwości odnośnie do tych wrażeń formalnych, które nie są zmiennymi.

2) Przykład podany w poprzednim paragrafie (patrz 2.1.1.010) wskazuje jasno, ze pewne wyrażenia formalne będące jakoby - zdaniami, jakoby - nazwami, czy wreszcie jakoby - funktorami okazują się przydatne do celów ćwiczebnych; stanowią one dobry materiał do ćwiczeń, z zakresu składni. Skromna to wprawdzie przydatność, ale jednak niewątpliwa.

Okazuje się - więc, że *wyrażenia formalne* (przynajmniej niektóre) *są praktycznie przydatne*. Przekonamy się wkrótce, że są one bardziej przydatne, niż to dotychczas stwierdziliśmy.

2.1.1.20. **Wyjaśnienie**. Mówimy, że dane wyrażenie jest *merytoryczne,* zamiast mówić, że strona znaczeniowa tego wyrażenia *nie jest pusta*.

2.1.1.21. **Reguła**. Żadne wyrażenie merytoryczne nie jest wyrażeniem formalnym (wyjaśnienia 2.1.3.1.10 oraz 2.1.1.20).

Piśmiennictwo: *Greniewski H.* G.2.1.

2.1.5. Problematyka wyrażeń formalnych

Stwierdzenie istnienia wyrażeń formalnych niebędących zmiennymi nastręcza kłopot, a raczej sporo kłopotów. Aby zorientować się, jako tako w wyrażeniach formalnych, trzeba sobie odpowiedzieć, choćby pokrótce, na następujące pytania:

1) Czy i jakie interesujące rodzaje wyrażeń formalnych można wyróżnić?
2) Czy jest możliwe i celowe używanie łączne wyrażeń merytorycznych i formalnych w jednym tekście?
3) Jaki jest stosunek wyrażeń formalnych niebędących zmiennymi do języków; czy przynajmniej niektóre z nich wchodzą w skład jakiegoś języka?
4) Czy możliwe jest definiowanie wyrażeń formalnych?
5) Czy w dziedzinie wyrażeń formalnych występuje wynikanie, ewentualnie jakiś związek zbliżony do wynikania, czy występuje wnioskowanie, ewentualnie jakiś związek zbliżony do wnioskowania?

Postaramy się na te pytania dać odpowiedź, odpowiedź raczej niewyczerpującą, lecz jak się wydaje - wystarczającą do elementarnego wykładu logiki formalnej.

1) Nie ma żadnych trudności w wynajdywaniu czy tworzeniu wyrażeń formalnych mających gramatyczne właściwości zdań, nazw, funktorów, funkcji i operatorów.
2) Nie ma żadnych trudności gramatycznych w budowaniu wyrażeń kombinowanych, złożonych z wyrażeń merytorycznych i wyrażeń formalnych (nie będących zmiennymi). Inaczej natomiast przedstawia się sprawa celowości używania takich wyrażeń kombinowanych. W naszym wykładzie takie wyrażenia kombinowane (poza funkcjami i operatorami) są zbędne, wystarczy nam - bowiem rozważanie pewnych systemów złożonych z samych wyrażeń formalnych.
3) Nie omawiając całości trudnego zagadnienia stosunku wyrażeń formalnych do języków, powtarzamy, że będziemy mieli wkrótce do czynienia z pewnymi systemami złożonymi z samych wyrażeń formalnych; zgodnie z ustaloną już w logice formalnej terminologią, niektóre z tych systemów nazywać będziemy „*językami sformalizowanymi*". Już teraz jednak uprzedzamy czytelnika, że żaden język sformalizowany nie jest, ściśle mówiąc, językiem (podobnie jak żadna spłaszczona kula nie jest kulą, podobnie jak czyjaś rozwiedziona żona nie jest żoną swego byłego męża).
4) Definiowanie wyrażeń formalnych za pomocą wyrażeń formalnych nie napotyka w zasadzie większych trudności, chociaż właściwie nie zawsze wiadomo, co rozumieć przez definicję

poprawną, której definiendum jest wyrażeniem formalnym. Nie dotyczy to jednak informatyki, gdzie gdy np. korzystamy z tzw. makr, czy podprogramów.

5) Możliwe jest niekiedy naśladowanie wynikania i wnioskowania w dziedzinie wyrażeń formalnych. Zdarzają się wynikania sprowadzające się do zastępowania, podstawiania czy odrywania, a więc do czynności formalnych; w tych właśnie wypadkach łatwo naśladować wynikanie i dedukcję w dziedzinie wyrażeń formalnych.

2.1.5.00. **Wyjaśnienie**. Mówimy krótko (1) „*jakoby-zdanie*", ewentualnie (2) „*jakoby-nazwa*", ewentualnie (3) „*jakoby-funktor*", zamiast mówić:

1) Wyrażenie formalne, mające pewne własności gramatyczne zdania;
   ewentualnie
2) Wyrażenie formalne, mające pewne własności gramatyczne nazwy;
   ewentualnie
3) Wyrażenie formalne, mające pewne własności gramatyczne funktora.

2.1.5.01.**Wyjaśnienie**. Mówimy krótko (1) „*jakoby-funkcja*", ewentualnie (2) „*jakoby-operator*", zamiast mówić:

1) Wyrażenie formalne zawierające chociaż jedną zmienną wolną;
   ewentualnie
2) Wyrażenie formalne zawierające - chociaż jedną zmienną związaną i mające pewne własności operatora.

2.1.5.02. **Wyjaśnienie**. Mówimy, że dane wyrażenie formalne jest jakoby-tezą, zamiast mówić, że jest ono jakoby-zdaniem lub jakoby-funkcją zdaniową.

Trzeba przyznać, że żadne z powyższych trzech wyjaśnień nie grzeszy nadmiarem ścisłości, jednak dalsze uściślenie wymagałoby zbyt szerokiej rozbudowy naszych rozważań.

Piśmiennictwo: *Greniewski H.* G.2.1. ,

2.1.6. Formalizacja i interpretacje wyrażeń

Zajmiemy się teraz pewnymi czynnościami wykonywanymi na wyrażeniach. Każda z tych czynności jest wykonywana albo zawszę na wyrażeniu merytorycznym, albo zawszę na wyrażeniu formalnym. Wynik każdej z tych czynności jest albo zawszę wyrażeniem merytorycznym, albo zawszę wyrażeniem formalnym.

2.1.6.00. **Wyjaśnienie**. Przez tłumaczenie danego wyrażenia merytorycznego rozumiemy podstawienie za to wyrażenie wyrażenia równoznacznego.

2.1.6.01. **Wyjaśnienie**. Przez formalizację danego wyrażenia merytorycznego rozumiemy:

1) Pozbawienie tego wyrażenia niepustego rozumienia (wprowadzenie strony znaczeniowej pustej na miejsce niepustej);

lub

2) Podstawienie za to wyrażenie merytoryczne wyrażenia formalnego.

2.1.6.10. **Wyjaśnienie**. Przez „interpretację merytoryczną danego wyrażenia formalnego" rozumiemy:

1) Wprowadzenie do tego wyrażenia strony znaczeniowej niepustej na miejsce pustej
   lub
2) Podstawienie wyrażenia merytorycznego za to wyrażenie formalne.

2.1.6.11. **Wyjaśnienie**. Przez „interpretację formalną danego wyrażenia formalnego” rozumiemy podstawienie za to wyrażenie wyrażenia formalnego.

2.1.6.20. **Reguła**. Tłumaczenie nie jest czynnością formalną. Uzasadnienie: Dokonanie tłumaczenia wymaga znajomości strony znaczeniowej dwu wyrażeń merytorycznych (wyjaśnienia 2.1.6.00 oraz 2.1.6.00).

2.1.6.21. **Reguła**. Formalizacja jest czynnością formalną (wyjaśnienia 2.1.6.01 oraz 2.1.6.00).

2.1.6.22. **Reguła**. Interpretacja merytoryczna nie jest czynnością formalną. Uzasadnienie: Dokonanie interpretacji merytorycznej wymaga znajomości rozumienia nadawanego wyrażeniu formalnemu lub rozumienia wyrażenia merytorycznego podstawionego za wyrażenie formalne (wyjaśnienia 2.1.6.10 oraz 2.1.6.00).

2.1.6.23. **Reguła**. Interpretacja formalna jest czynnością formalną (wyjaśnienia 2.1.6.11 oraz 2.1.6.00).

Przy posługiwaniu się powyższymi wyjaśnieniami i regułami okazuje się pomocna tabela 2.1.6.30.

Tabela 2.1.6.30

Lp.	Wyrażenie, do którego stosujemy czynność, jest	Czynność	Wyrażenie będące wynikiem czynności jest	Czynność jest
	I.	*II.*	*III.*	*IV.*
1	merytoryczne	tłumaczenie	merytoryczne	nieformalna
2	merytoryczne	formalizacja	formalne	formalna
3	formalne	interpretacja merytoryczna	merytoryczne	nieformalna
4	formalne	interpretacja formalna	formalne	formalna

Dla lepszego zrozumienia wyliczonych wyżej czynności podamy jeszcze pewien przykład. Do przykładu tego będą nam potrzebne cztery wyrażenia, mianowicie dwa wyrażenia merytoryczne i dwa wyrażenia formalne. Oba wyrażenia merytoryczne będą przy tym *równoznaczne* lecz *nierównokształtne*; będą to dwie nazwy jednostkowe pewnego wybitnego logika rzymskiego - nazwa spolszczona „*Boecjusz*” i nazwa łacińska „*Boetius*". Oba wyrażenia formalne są jakoby-nazwami jednostkowymi: „*Awiasz*” oraz „*Bewiasz*”. Zastosujmy teraz reguły z tabeli 2.1.6.30 do niniejszego przykładu:

Ad. 1. „*Boetiusz”* <u>tłumaczenie</u> „*Boetius”* lub odwrotnie (czynność nieformalna);
Ad. 2. „*Boeciusz”* <u>formalizacja</u> „*Awiasz”* lub „*Boetius”* <u>formalizacja</u> „*Bewiasz*” (czynność formalna);
Ad. 3. „*Awiasz”* <u>interpretacja merytoryczna</u> „*Boeciusz”*
lub „*Bewiasz”* <u>interpretacja merytoryczna</u> „*Boetius”* (czynność nieformalna)*;*
Ad. 4. „*Awiasz”* <u>interpretacja formalna</u> „*Bewiasz* lub odwrotnie” (czynność formalna).

Poznaliśmy ostatnio cztery czynności wykonywane zawsze równocześnie na jednym wyrażeniu. Każda z tych czynności daje zawsze - jako wynik jedno wyrażenie. Poznamy ponadto analogiczne czynności, z których każda wykonywana jest za każdym razem na jakimś zespole czy systemie wyrażeń i jako wynik daje zawsze zespół czy system wyrażeń.

Te cztery nowe czynności, to:
1) Tłumaczenie tekstu (z języka na język);
2) Formalizacja (systemu zaksjomatyzowanego);

3) Interpretacja merytoryczna (systemu wyrażeń formalnych);
4) Interpretacja formalna (systemu wyrażeń formalnych).

Tłumaczenie tekstu - to czynność dla naszych celów mniej interesująca, pozostałe trzy czynności poznamy wkrótce, przedtem jednak musimy poznać ważniejsze odmiany systemów wyrażeń formalnych.

2.1.6.40. **Wyjaśnienie**. Przez „*system wyrażeń formalnych*" rozumiemy każdy i tylko taki zespół złożony wyłącznie z wyrażeń formalnych, który zbudowany jest w sposób następujący:
1) Podana jest lista wyrażeń pierwotnych, czyli zbudowanych przed rozpoczęciem budowania systemu;
2) Podane są (w jakimś języku naturalnym) dyrektywy budowania nowych wyrażeń za pomocą wyrażeń zaliczonych już do systemu;
3) Dyrektywy te są wykonalne, to znaczy z wyrażeń pierwotnych można zbudować chociaż jedno wyrażenie nie będące wyrażeniem pierwotnym;
4) Każda czynność przewidziana w którejkolwiek z dyrektyw jest czynnością formalną.

2.1.6.41.**Wyjaśnienie**. Przez „*wyrażenie wtórne*" (danego systemu) rozumiemy każde i tylko takie wyrażenie formalne, które można zbudować za pomocą dyrektyw (tegoż systemu) lub przez zdefiniowanie, a które nie jest (w danym systemie) wyrażeniem pierwotnym.

2.1.6.42. **Wyjaśnienie**. Przez „*jednorodny system wyrażeń formalnych*" rozumiemy każdy i tylko taki system wyrażeń formalnych, w którym ani na liście wyrażeń pierwotnych, ani w tekście dyrektyw nie wprowadzamy żadnego podziału wyrażeń należących do systemu na różne rodzaje (poza podziałem na wyrażenia pierwotne i wtórne).

W związku z wyjaśnieniem 2.1.6.40 zauważymy teraz, że do zbudowania jakiegokolwiek systemu wyrażeń formalnych niezbędne jest rozumienie pewnych wyrażeń merytorycznych (tych mianowicie, z których zbudowane są dyrektywy); mimo to rozbudowa takiego systemu odbywa się przez wykonywanie wyłącznie czynności formalnych. W związku natomiast z wyjaśnieniem 2.1.6.41 - zauważymy, że żaden system wyrażeń formalnych, w którym będziemy odróżniać na przykład jakoby-zdania od jakoby-nazw, nie będzie systemem jednorodnym; natomiast jednorodny system wyrażeń formalnych może się składać wyłącznie z jakoby-zdań czy też, na przykład, wyłącznie z jakoby-nazw. Przykład. Każdy język programowania (np. język C – patrz rozdział 1.5.), jest systemem wyrażeń formalnych.

Piśmiennictwo: *Greniewski H.* G.2.1.

2.1.7. POJĘCIE SYSTEMU SFORMALIZOWANEGO

2.1.7.00. **Wyjaśnienie**. „Przez sprzężony system wyrażeń formalnych" rozumiemy każdą i tylko taką parę uporządkowaną zespołów wyrażeń, która spełnia wszystkie warunki następujące:
1) Zespół pierwszy jest systemem wyrażeń formalnych (jednorodnym lub niejednorodnym);
2) Każde wyrażenie należące do zespołu drugiego pierwszego, ale niekoniecznie na odwrót;
3) Zespół drugi - ma chociaż jedną dyrektywę budowania wyrażeń zaliczanych do tego zespołu, każda z tych dyrektyw jest wykonalna i każda opisuje sposób budowania wyrażeń zespołu drugiego w oparciu o wyrażenia już zaliczone do zespołu pierwszego lub drugiego.

Jak widać z powyższego wyjaśnienia, w każdym systemie sprzężonym mamy do czynienia z - dwoma zespołami wyrażeń: pierwszy z nich jest zwykłym systemem wyrażeń formalnych (może

on być jednorodny lub niejednorodny), drugi zaś jest wynikiem selekcjonowania wyrażeń zespołu pierwszego. Dyrektywy budowania wyrażeń zespołu drugiego są swojego rodzaju selektorem, który pewne wyrażenia już zaliczone do pierwszego zespołu (ale zwykle nie wszystkie), „wybiera" z zespołu pierwszego i zalicza do zespołu drugiego. Można by też przyrównać rolę dyrektyw zespołu drugiego do filtru, który tylko niektóre wyrażenia zaliczone do zespołu pierwszego „przepuszcza" do zespołu drugiego.

Zauważmy jeszcze, ze wyjaśnienie 2.1.7.00 – nie stawia systemowi sprzężonemu wymagania, aby drugi z obu zespołów miał swoją odrębną listę wyrażeń pierwotnych. Jednakże trzeba ponadto zauważyć, że wyjaśnienie 2.1.7.00 - nie czyni również zastrzeżeń przeciwko temu, aby drugi z zespołów stanowiących jakiś system sprzężony miał swoją listę wyrażeń pierwotnych. Wśród sprzężonych systemów wyrażeń formalnych wyróżnimy teraz pewien węższy ich rodzaj, mianowicie systemy sformalizowane.

2.1.7.01. **Wyjaśnienie**. Przez „*system sformalizowany*" rozumiemy każdy i tylko taki sprzężony system wyrażeń formalnych, który spełnia wszystkie warunki następujące:

1) Jeżeli zespół pierwszy tego systemu sprzężonego jest systemem jednorodnym, to każde wyrażenie formalne należące do zespołu pierwszego jest jakoby-tezą (a więc jakoby-zdaniem albo jakoby-funkcją zdaniową);
2) Jeżeli zespół pierwszy tego systemu sprzężonego jest niejednorodnym systemem wyrażeń formalnych, to jednym z rodzajów wyrażeń należących do zespołu pierwszego są jakoby-tezy, a ponadto dyrektywy budowania wyrażeń zespołu drugiego działają w ten sposób, że „przepuszczają" z zespołu pierwszego do zespołu drugiego tylko jakoby-tezy;
3) Jeżeli zespół drugi tego systemu sprzężonego jest systemem wyrażeń formalnych i zawiera chociaż jedno wyrażenie pierwotne, to każde wyrażenie pierwotne zespołu drugiego jest jakoby-tezą.

2.1.7.10. **Wyjaśnienie**. Przez „*język sformalizowany*" rozumiemy pierwszy zespół wyrażeń systemu sformalizowanego.

2.1.7.11**. Wyjaśnienie**. Przez „*teorię sformalizowaną*" rozumiemy drugi zespół wyrażeń systemu sformalizowanego.

2.1.7.12. **Reguła**. Każdy język sformalizowany składa się wyłącznie z wyrażeń formalnych (wyjaśnienia 2.1.7.00, 2.1.7.01, 2.1.7.10).

2.1.7.13. **Reguła**. Każda teoria sformalizowana składa się wyłącznie z wyrażeń formalnych, mianowicie - wyłącznie z jakoby-tez (wyjaśnienia 2.1.7.00, 2.1.7.01, 2.1.7.11).

2.1.7.14. **Reguła**. Do żadnego języka sformalizowanego nie należy żadne wyrażenie merytoryczne (reguły 2.1.7.20 oraz 2.1.7.12).

2.1.7.15. **Reguła**. Do żadnej teorii sformalizowanej nie należy żadna teza prawdziwa (reguła 2.1.7.14, wyjaśnienie 2.1.7.01).

2.1.7.20. **Wyjaśnienie**. Przez „*wyrażenie pierwotne systemu sformalizowanego*" rozumiemy wyrażenie pierwotne odnośnego języka sformalizowanego.

2.1.7.21. **Wyjaśnienie**. Przez „*wyrażenie wtórne systemu sformalizowanego*" rozumiemy wyrażenie wtórne odnośnego języka sformalizowanego.

2.1.5.22. **Reguła**. Każdy język sformalizowany zawiera - chociaż jedno wyrażenie pierwotne (wyjaśnienia 2.1.4.00, 2.1.5.00, 2.1.5.01, 2.1.5.20).

2.1.7.23. **Reguła**. Każdy język sformalizowany zawiera - chociaż jedno wyrażenie wtórne (wyjaśnienia 2.1.7.00, 2.1.7.01, 2.1.7.00, 2.1.7.01, 2.1.7.21).

2.1.7.30. **Wyjaśnienie**. Przez „*tezę pierwotną - czyli postulat, ewentualnie aksjomat, systemu sformalizowanego*" rozumiemy wyrażenie pierwotne (jakoby-tezę) odnośnej teorii sformalizowanej.

2.1.7.31. **Wyjaśnienie**. Przez „*tezę wtórną - czyli twierdzenie, systemu sformalizowanego*" rozumiemy wyrażenie wtórne, jakoby-tezę, odnośnej teorii sformalizowanej.

2.1.7.40. **Wyjaśnienie**. Przez „*dyrektywę językową systemu sformalizowanego*" rozumiemy dyrektywę budowania wyrażeń odnośnego języka sformalizowanego.

2.1.7.41. **Wyjaśnienie**. Przez „*dyrektywę wnioskowania systemu sformalizowanego*" rozumiemy dyrektywę budowania wyrażeń odnośnej teorii sformalizowanej.

Piśmiennictwo: *Greniewski H.* G.2.1.

2.1.8. O METAJĘZYKACH, METATEORIACH I META-SYSTEMACH

Poznaliśmy w podrozdziałach poprzednich różne rodzaje systemów wyrażeń formalnych od najprostszych aż po systemy sformalizowane. Obecnie zajmiemy się w kilku słowach metodami badania systemów zaksjomatyzowanych i systemów wyrażeń formalnych, w szczególności systemów sformalizowanych.

2.1.8.00. **Wyjaśnienie**. Mówimy, że dana dyrektywa jest „*znaczeniotwórcza*", zamiast mówić, że dyrektywa ta ma wszystkie własności poniższe:
1) Jest to dyrektywa budowania wyrażeń;
2) Ma ona zastosowanie wyłącznie do wyrażeń merytorycznych (to jest do poszczególnych wyrażeń lub ich zespołów);
3) W wyniku zastosowania tej dyrektywy otrzymujemy zawszę wyrażenie merytoryczne.

2.1.8.01. **Wyjaśnienie**. Mówimy, że dana dyrektywa jest „*prawdotwórcza*", zamiast mówić, że dyrektywa ta ma wszystkie własności poniższe:
1) Jest to dyrektywa budowania tez;
2) Ma ona zastosowanie wyłącznie do tez (poszczególnych tez, ewentualnie zespołów tez);
3) W wyniku zastosowania tej dyrektywy otrzymujemy zawsze tezę - następstwo tego zespołu tez, do którego dyrektywę zastosowano.

2.1.8.02. **Reguła**. Każda dyrektywa *prawdotwórcza* jest dyrektywą *znaczeniotwórczą*, ale nie każda dyrektywa znaczeniotwórcza jest dyrektywą prawdotwórczą.

Uzasadnienie: weźmy pod uwagę jakąkolwiek dyrektywę prawdo twórczą; jest to w myśl wyjaśnienia 2.1.8.01 - dyrektywa budowania tez, ponieważ zaś każda teza jest wyrażeniem, więc nasza dyrektywa prawdo twórcza jest dyrektywą budowania wyrażeń. W myśl wyjaśnienia 2.1.8.01 - nasza dyrektywa prawdo twórcza ma zastosowanie wyłącznie do tez, każda teza jest wyrażeniem merytorycznym, a więc nasza dyrektywa ma zastosowanie wyłącznie do wyrażeń merytorycznych. W wyniku zastosowania dyrektywy prawdo twórczej otrzymujemy zawsze

tezę (będącą następstwem tez, do których dyrektywę zastosowano), a więc otrzymujemy zawsze wyrażenie merytoryczne. Każda zatem - dyrektywa prawdotwórcza spełnia wszystkie trzy warunki wyjaśnienia 2.1.6.00, a więc każda dyrektywa prawdotwórcza jest dyrektywą znaczeniotwórczą.

2.1.8.03. **Reguła**. Żadna dyrektywa językowa jakiegokolwiek systemu sformalizowanego nie jest dyrektywą znaczeniotwórczą.
Uzasadnienie: Dowolna dyrektywa językowa dowolnego systemu sformalizowanego ma zastosowanie do wyrażeń formalnych, gdy dowolna dyrektywa znaczeniotwórcza ma zastosowanie wyłącznie u wyrażeń merytorycznych.

2.1.8.04. **Reguła**. Żadna dyrektywa wnioskowania jakiegokolwiek systemu sformalizowanego nie jest dyrektywą prawdo twórczą.
Uzasadnienie: Dowolna dyrektywa językowa dowolnego systemu sformalizowanego ma zastosowanie do jakoby-tez (a więc pewnych wyrażeń formalnych), gdy dowolna dyrektywa prawdotwórcza ma zastosowanie wyłącznie do tez (a więc pewnych wyrażeń merytorycznych).

2.1.8.05. **Reguła**. Stosując dowolną dyrektywę prawdotwórczą do zespołu tez prawdziwych otrzymujemy w wyniku zawszę tezę prawdziwą.

2.1.8.10. **Wyjaśnienie**. Przez „interpretację merytoryczną" systemu sformalizowanego rozumiemy podanie takiej dyrektywy, która:

1) Nadaje każdemu wyrażeniu pierwotnemu odnośnego języka sformalizowanego interpretację merytoryczną i to taką, że w tej interpretacji każda (ewentualna) teza pierwotna odnośnego systemu sformalizowanego staje się:
    a) Tezą przyjętą na danym etapie rozwoju nauki za prawdziwą; lub
    b) Hipotezą przyjętą na danym etapie rozwoju nauki;
2) Przekształca każdą dyrektywę językową danego systemu sformalizowanego w dyrektywę znaczeniotwórczą;
   oraz
3) Przekształca każdą dyrektywę wnioskowania danego systemu sformalizowanego w dyrektywę prawdotwórczą.

2.1.8.11. **Wyjaśnienie**. Przez „*formalizację*" systemu zaksjomatyzowanego rozumiemy przekształcenie danego systemu zaksjomatyzowanego w system sformalizowany.

Na pierwszy rzut oka - żaden system sformalizowany nie nadaje się do badania rzeczywistości materialnej. Fakt, ze język sformalizowany nie zawiera ani jednego wyrażenia merytorycznego, teoria sformalizowana zaś nie zawiera żadnej tezy prawdziwej, dyskwalifikuje, zdawałoby się – wszelki system sformalizowany, jako narzędzie badania. Dzięki jednak metodzie interpretacyjnej, to jest dzięki stosowaniu interpretacji merytorycznej, można niekiedy użyć niektórych - systemów sformalizowanych do badania rzeczywistości materialnej. Niekiedy jeden i ten sam - system sformalizowany ma więcej niż jedną interpretację merytoryczną (poznamy takie systemy w części 3), wówczas system taki pozwala na jednoczesne badanie, więcej niż jednego fragmentu rzeczywistości.

2.1.8.20. **Wyjaśnienie**. Mówimy, że dany system sformalizowany jest przepełniony, zamiast mówić, że każda jakoby-teza należąca do języka sformalizowanego danego systemu należy też do odnośnej teorii sformalizowanej.

Pojęcie przepełnienia pierwszy sprecyzował logik polski Stanisław Jaśkowski. Jak wiemy, każdy system sformalizowany jest systemem sprzężonym wyrażeń formalnych, wobec czego zespół dyrektyw wnioskowania ma charakter selektora czy filtru, który pewne tylko jakoby-tezy przepuszcza z języka sformalizowanego do teorii sformalizowanej. System przepełniony - to taki i tylko taki system, w którym wzmiankowany filtr nie działa skutecznie i przepuszcza wszelkie jakoby-tezy z języka sformalizowanego do teorii sformalizowanej. Budowanie, więc systemów przepełnionych jest zbędne, są one wszystkie bezwartościowe. Gdy budujemy jakiś system sformalizowany, co do którego zachodzą wątpliwości, czy nie jest on przepełniony, należy zbudować dowód nieprzepełnienia tego systemu. Związek między przepełnieniem z jednej strony a sprzecznością logiczną (i antynomiami) z drugiej – wykraczają poza zakres tematyczny niniejszej książki.

2.1.8.30. **Wyjaśnienie**. Przez „*metajęzyk*" rozumiemy taki i tylko taki język, w którym wypowiadamy się:
1) O pewnym innym języku czy innych językach
lub
2) O pewnym języku sformalizowanym czy też językach sformalizowanych.
Przykład. Weźmy pod uwagę gramatykę języka rosyjskiego napisaną po polsku. W gramatyce tej występuje metajęzyk, na który składa się: (1) fragment graficznego języka polskiego, wystarczający do sformułowania reguł gramatycznych rosyjskiego języka graficznego oraz (2) odpowiednie nazwy przykładowo dobranych wyrażeń rosyjskich (nazwy te powstają przez ujęcie wyrażenia rosyjskiego w cudzysłów lub wprost wydrukowanie go kursywą, zależnie od przyjętych zwyczajów).

Weźmy pod uwagę jakikolwiek system sformalizowany; zawszę mamy tu do czynienia z metajęzykiem: każda - bowiem dyrektywa językowa i każda dyrektywa wnioskowania napisana jest w metajęzyku (rolę metajęzyka spełnia tu zwykle jakiś naturalny język graficzny czy też jego fragment, niekiedy jednak metajęzykiem może być jakiś język mieszany czy nawet sztuczny). W definicjach wielu systemów sformalizowanych występuje znany nam skrót „$=_{Df}$"; tezy pierwotne i tezy wtórne niejednego systemu sformalizowanego poprzedzamy znanym już nam skrótem „$\vdash$". Oba te skróty należą do metajęzyka.

Każda teoria ma swój przedmiot badania, na przykład przedmiotem badania planimetrii jest płaszczyzna (ściślej - każda płaszczyzna jest przedmiotem badania planimetrii). Przedmiotem badania arytmetyki liczb naturalnych są liczby naturalne itd. Nie wynika z tego, żeby każda rzecz czy każdy zespół rzeczy miał swoją teorię, której jest przedmiotem. Liczne doświadczenia z dziejów nauki zdają się wskazywać, że każda teoria może być przedmiotem innej teorii. Doświadczenia te wskazują nam w każdym razie na istnienie takich teorii, których przedmiotem badania są teorie.

2.1.8.31. **Wyjaśnienie**. Mówimy krótko „*metateoria*", zamiast mówić teoria mająca za przedmiot badania:
1) Jakąś teorię czy jakieś teorie;
lub

2) Jakiś system zaksjomatyzowany czy systemy zaksjomatyzowane;
lub
3) Jakiś system sformalizowany czy jakieś systemy sformalizowane.

2.1.8.32.**Wyjaśnienie.** Mówimy krótko „*metasystem zaksjomatyzowany*" zamiast mówić, „system zaksjomatyzowany taki, że zawarta w nim teoria jest metateorią".

2.1.8.33. **Wyjaśnienie**. Mówimy krótko „*metasystem sformalizowany*", zamiast mówić „system sformalizowany mający taką interpretację merytoryczną, w której odnośna teoria sformalizowana staje się metateorią".

Piśmiennictwo: *Greniewski H.* G.2.1.

2.1.9. BLASKI I NĘDZE SYSTEMÓW SFORMALIZOWANYCH

Zachodzi znaczne podobieństwo między pojęciem systemu zaksjomatyzowanego a pojęciem systemu sformalizowanego. Oprócz podobieństwa występują jednak i różnice. Budowanie wyrażeń języka systemu zaksjomatyzowanego nie jest obiektywnie unormowane i wobec tego nie jest bezsporne. Budowanie wyrażeń języka systemu sformalizowanego (tj. języka sformalizowanego) jest natomiast obiektywnie unormowane przez dyrektywy językowe), a więc powinno być (przy dostatecznie starannej redakcji - dyrektyw językowych) bezsporne.

Budowanie (dowodzenie) tez w systemie zaksjomatyzowanym nie jest obiektywnie unormowane, a więc nie jest bezsporne. Poza tym w niejednym już systemie zaksjomatyzowanym zbudowano, jak pisaliśmy już, *antynomie*. Natomiast budowanie (dowodzenie, selekcjonowanie) jakoby-tez w systemie sformalizowanym jest obiektywnie unormowane (przez dyrektywy wnioskowania), a więc powinno być (przy dostatecznie starannej redakcji dyrektyw) bezsporne.

W sprawie doniosłości systemów sformalizowanych przytoczymy pogląd logika i matematyka polskiego Andrzeja Mostowskiego. "Dowodzenie twierdzeń matematycznych uchodziło z dawien dawna za czynność umysłową, i to nawet za jedną z najtrudniejszych i najsubtelniejszych. W teoriach sformalizowanych zastąpiliśmy jednak tę czynność umysłową przez mechaniczne stosowanie reguł wnioskowania. Teoretycznie rzecz biorąc, można by w nauce sformalizowanej dowodzić twierdzeń zupełnie bezmyślnie, próbując tylko stosować (na wszelkie możliwe sposoby) reguły wnioskowania do aksjomatów, twierdzeń już udowodnionych...
... Inaczej mówiąc, dla każdej teorii sformalizowanej można wyobrazić sobie maszynę, która kolejno wypisuje twierdzenia tej teorii i to tak, że po dłuższym lub krótszym funkcjonowaniu wypisze każde z góry dane twierdzenie.
Droga ta praktycznie nie daje się zrealizować przynajmniej w chwili obecnej, gdyż dowody sformalizowane są tak długie, że nie starczyłoby życia ludzkiego, by dojść w ten sposób do twierdzeń ciekawych... rola intuicji w dowodach twierdzeń (w teoriach sformalizowanych) nie jest bynajmniej istotna: sprowadza się ona do tego, żeby wśród chaosu wszelkich możliwych kroków dowodowych znaleźć takie ich następstwo, które względnie szybko doprowadzi do celu"[26].

Pojęcie systemu sformalizowanego jest niewątpliwie wysoce interesujące. Fakt zbudowania (niejednego już) systemu sformalizowanego jest faktem naukowo doniosłym. Stworzenie

[26] Mostowski, M.5.1, s.231 - 232.

systemów sformalizowanych otworzyło przed nauką nowe perspektywy rozwojowe i zarazem było początkowo źródłem... wielkich - a jak się później okazało - nieuzasadnionych złudzeń.

Wydawało się w pewnym (niedawnym zresztą) okresie pierwszego trzydziestolecia XX wieku, że całą wiedzę logiczną i matematyczną da się ująć - jako merytoryczną interpretację pewnej (niewielkiej stosunkowo) liczby systemów sformalizowanych. Wielką zasługą austriackiego logika *Kurta Gödla* jest, że w 1931 roku pogląd ten obalił w sposób ścisły. Wyniku tego (wchodzącego w skład metateorii systemów sformalizowanych) nie możemy w elementarnym wykładzie w sposób wyczerpujący zreferować (w literaturze polskiej wynik Gödla został systematycznie wyłożony w cytowanej już książce *Andrzeja Mostowskiego*). Praca Gödla jest trudna do zrozumienia. Jej ostateczny rezultaty poprzedzone są czterdziestoma sześcioma definicjami i kilkoma ważnymi twierdzeniami, które trzeba sobie przyswoić. W pracy tej pokazano zdumiewający dla ówczesnych logików rezultat badań nad metodą aksjomatyczną i formalizacją, według których metody te związane są nierozłącznie z pewnym ograniczeniem, sprawiającym, iż nie można wykazać wewnętrznej, logicznej niesprzeczności wielu złożonych systemów dedukcyjnych – bez zastosowania tak skomplikowanych środków dowodowych, że ich własna niesprzeczność wewnętrzna jest równie wątpliwa, jak niesprzeczność owych systemów. Konsekwencją tego wyniku, było stwierdzenie, że system dedukcyjny niesprzeczny, to system, w którym nie da się udowodnić jednocześnie pewnego zdania i jego zaprzeczenia. Tak więc - odkrycie Gödla zniweczyło zakorzenione głęboko przeświadczenie i zburzyło nadzieje, towarzyszące wcześniejszym badaniom nad podstawami matematyki. Ale wyniki tej pracy nie okazały się wyłącznie negatywne, otworzyły one drogę do pionierskiej problematyki, a mianowicie teorii obliczalności i teorii algorytmów, problematyki kluczowej dla informatyki. Niesprzeczność systemu dedukcyjnego, pociąga za sobą własność rozstrzygalności w tym systemie. Rozstrzygalność, jak się dalej okazało, jest równoznaczna z istnieniem algorytmu.

Alan Turing, opublikował w 1934 roku swoją prawdopodobnie najważniejszą pracę matematyczną *„On Computable Numbers”*. To właśnie w tej pracy wprowadził Turing abstrakcyjną maszynę, która była w stanie wykonywać zaprogramowaną matematyczną sekwencję operacji, czyli tak zwany *algorytm*. Następnie w roku 1936, Turing opracował tak zwaną *uniwersalną maszynę Turinga* (patrz podrozdział 3.8.4), która w zależności od instrukcji zapisanej na taśmie, miała wykonywać dowolny algorytm. Obok Turinga do rozwoju nowej dziedziny teorii algorytmów przyczynili się *Alonso Church*, wykorzystując notację nazwaną rachunkiem λ do zdefiniowania algorytmu. W późniejszym okresie pokazano, że obie definicje algorytmu, to jest Turinga i Churcha, są równoważne. Powiązanie nieformalnego pojęcia algorytmu z precyzyjną definicją z wykorzystaniem maszyny Turinga nazwano tezą Churcha-Turinga. Mimo przesadnych nadziei i błędnego filozoficznie podejścia do systemów sformalizowanych powstał kierunek zwany formalizmem. Pojęcie systemu sformalizowanego jest - jak już wspomnieliśmy - wysoce interesujące; dodajmy jeszcze, że w pewnym zakresie jest ono więcej niż pożyteczne; na przykład, gdy chodzi o wykład pewnych najprostszych teorii logiki, gdy chodzi o zasady budowy komputerów, systemów operacyjnych, języków programowania, systemów baz danych, itp.

Piśmiennictwo: *Greniewski H.* G.2.1. , *Kisielewicz A.* K.1.1.,, *Mostowski A.* M.5.1.,*Turing A.* T.6.1.

2.2. DWUWARTOŚCIOWY RACHUNEK ZDAŃ

2.2.0. WPROWADZENIE DO KLASYCZNEGO RACHUNKU ZDAŃ

Logika formalna zajmuje się badaniami modelującymi wzajemne zależności w obrębie systemów dających się opisać w sposób formalny. Jednym z dwu systemów logicznych - posiadających swoje początki w starożytnej Grecji jest logika zdań - zwana klasycznym dwuwartościowym rachunkiem zdań. Przedmiotem zainteresowania dwuwartościowego rachunku zdań są możliwe zależności pomiędzy stwierdzeniami (zdaniami orzekającymi), oraz zależność - w jaki sposób prawdziwość zdań złożonych zależy od prawdziwości ich składowych. Sens samych składowych pozostaje tu całkowicie dowolny i nieistotny. Dlatego w dwuwartościowym rachunku zdań odpowiadają im po prostu zmienne zdaniowe. Zdania złożone budujemy ze zmiennych zdaniowych za pomocą spójników logicznych takich jak alternatywa $\vee$, koniunkcja $\wedge$, negacja $\neg$, czy implikacja $\rightarrow$. Wygodne są też stałe logiczne $\perp$ (fałsz) i $\top$ (prawda), które można uważać za zero-argumentowe spójniki logiczne.

Piśmiennictwo: *Ben-Ari M*. B.2.1., *Jaśkowski St.* J.3.1, *Łukasiewicz J.* Ł.3.1., *Pacholski L.* P.1.1., *Tiuryn J.* T.4.1.

2.2.1. SKŁADNIA RACHUNKU ZDAŃ

2.2.1.10. **Definicja**. Ustalamy pewien przeliczalnie nieskończony zbiór $\wp$ symboli, które nazywamy zmiennymi zdaniowymi i zwykle oznaczać literami *p*, *q*, itp. Zmienne zdaniowe są z kolei argumentami formuł zdaniowych, tworzonych z użyciem funkcji logicznych (np. negacji, alternatywy, koniunkcji, implikacji, różnicy symetrycznej itp.). Dalej pojęcie formuły zdaniowej definiujemy przez indukcję:

2.2.1.11. Zmienne zdaniowe oraz $\top$ i $\perp$ są formułami zdaniowymi;

2.2.1.12. Jeśli napis X jest formułą zdaniową, to także napis $\neg$X jest formułą zdaniową;

2.2.1.13. Jeśli napisy X i Y są formułami zdaniowymi to napisy $(X \rightarrow Y)$ czyli implikacja, $(X \wedge Y)$ czyli koniunkcja i $(X \vee Y)$ czyli alternatywa - też są formułami zdaniowymi.

2.2.1.14. **Wyjaśnienie**. Inaczej mówiąc, formuły zdaniowe to elementy najmniejszego zbioru napisów F_p, zawierającego $\wp \cup \{\top, \perp\}$ i takiego, że dla dowolnych X, Y $\in F_p$ także $\neg$X, $(X \rightarrow Y)$, $(X \wedge Y)$, $(X \vee Y)$ należą do F_p.

2.2.1.20. **Reguła**. Dla pełnej jednoznaczności składni przyjmujemy konwencję formuł w pełnym zapisie nawiasowym. W praktyce wiele nawiasów pomijamy, stosując przy tym następujące priorytety:

1) Negacja;
2) Koniunkcja i alternatywa;
3) Implikacja.

Zatem *na* przykład $\neg X \vee Y \rightarrow Z$ oznacza $((\neg X \vee Y) \rightarrow Z)$, ale napis $X \vee Y \wedge Z$ jest niepoprawny.

Tabela 2.2.1.30	
Negacja zdaniowa	
p	$\neg$p
I,	II.
fałsz prawda	prawda fałsz

Tabela 2.2.1.40					
Alternatywa zdaniowa		Koniunkcja zdaniowa		Implikacja zdaniowa	
p q	p ∨ q	p q	p ∧ q	p q	p → q
I	II	I	III.	I	IV.
fałsz, fałsz fałsz, prawda prawda, fałsz prawda, prawda	fałsz prawda prawda prawda	fałsz, fałsz fałsz, prawda prawda, fałsz prawda, prawda	fałsz fałsz fałsz prawda	fałsz, fałsz fałsz, prawda prawda, fałsz prawda, prawda	prawda prawda fałsz prawda

Tabela 2.2.1.30 zawiera wartości prostej – jednoargumentowej formuły zdaniowej X = ¬p, zaś tabela 2.2.1.40 zawiera wartości prostych dwuargumentowych formuł zdaniowych: alternatywy, koniunkcji i implikacji.

Piśmiennictwo: *Ben-Ari M*. B.2.1., *Jaśkowski St.* J.3.1, *Łukasiewicz J.* Ł.3.1., *Pacholski L.* P.1.1., *Tiuryn J.* T.5.1.

2.2.2. WARTOŚCI LOGICZNE I ZNACZENIE FORMUŁ ZDANIOWYCH

2.2.2.10. **Reguła**. W logice klasycznej interpretacją formuły jest wartość logiczna tj. "*prawda*" (1) lub "*fałsz*" (0). Aby określić wartość formuły zdaniowej trzeba jednak najpierw ustalić wartości zmiennych.

2.2.2.20. **Definicja**. Przez wartościowanie zdaniowe rozumiemy dowolną funkcję ρ, która zmiennym zdaniowym przypisuje wartości logiczne 0 lub 1. Wartość formuły zdaniowej - X przy wartościowaniu ρ oznaczamy przez ‖X‖ i określamy przez indukcję:

2.2.2.21. $\|\bot\| = 0$ oraz $\|\top\| = 1$;

2.2.2.22. $\|X\| = \rho(X)$, gdy X jest symbolem zdaniowym;

2.2.2.23. $\|\neg X\| = 1 - \|X\|$;

2.2.2.24. $\|X \vee Y\| = \max\{\|X\|, \|Y\|\}$;

2.2.2.25. $\|X \wedge Y\| = \min\{\|X\|, \|Y\|\}$;

2.2.2.26. $\|X \to Y\| = 0$, gdy $\|X\| = 1$ i $\|Y\| = 0$;

2.2.2.27. $\|X \to Y\| = 1$, w przeciwnym przypadku.

2.2.2.30. **Wyjaśnienie**. Łatwo można zauważyć, że $\|X \to Y\| = \max\{\|X\|, 1-\|Y\|\}$, czyli $\|X \to Y\| = \|\neg X \vee Y\|$, dla dowolnego ρ. A zatem zamiast formuły X→Y moglibyśmy z równym powodzeniem używać wyrażenia ¬X ∨ Y, lub też odwrotnie: zamiast alternatywy X ∨ Y pisać ¬X→Y. Ten wybór spójników nie jest więc „najoszczędniejszy", w istocie w logice klasycznej wystarczy używać np. implikacji i fałszu. Czasem wygodnie korzystać z tego uproszczenia, przyjmując, że „oficjalnymi" spójnikami są tylko implikacja i fałsz, a pozostałe to skróty notacji, tj. że napisy

2.2.2.31. ¬X oznaczają odpowiednio $X \to \bot$;

2.2.2.32. $\top$ oznaczają odpowiednio $\neg\bot$;

2.2.2.33. X ∨ Y oznaczają odpowiednio ¬X→Y;

2.2.2.34. X ∧ Y oznaczają odpowiednio ¬ (X→ ¬Y).

2.2.2.40. **Wyjaśnienie**. Można też pisać X ↔ Y zamiast (X→Y) ∧ (Y→X). Zauważmy, że ‖X↔Y‖ = 1 wtedy i tylko wtedy, gdy ‖X→Y‖ = ‖Y→X‖ .

2.2.2.51. **Wyjaśnienie**. W rzeczywistości, na co już w latach pięćdziesiątych ubiegłego wieku zwrócili uwagę informatycy w związku z projektowaniem podstawowych funktorów logicznych, że rachunek zdań można zbudować na jednym z dwu funktorów: NAND – czyli zanegowanej koniunkcji (patrz 4.2.1.14.), lub funktora NOR zanegowanej alternatywie (patrz 4.2.1.15.).

2.2.2.60. **Wyjaśnienie**. Notacja i terminologia: Jeśli ‖X‖ = 1 to piszemy też $\rho \models X$ lub $\models X[\rho]$ i mówimy, że X jest spełniona przez wartościowanie ρ. Jeśli Γ jest zbiorem formuł zdaniowych, oraz $\rho \models Y$ dla wszystkich formuł Γ, to piszemy $\rho \models \Gamma$. Wreszcie $\Gamma \models X$ oznacza, że każde wartościowanie spełniające wszystkie formuły z Γ spełnia także formułę X. Mówimy wtedy, że X jest semantyczną konsekwencją zbioru Γ. Jeśli Γ = ∅ to zamiast $\Gamma \models X$ piszemy po prostu $\models X$. Oznacza to, że formuła X jest spełniona przez każde wartościowanie.

2.2.2.70. **Wyjaśnienie**. Na koniec powiedzmy jeszcze, że formułami równoważnymi nazywamy takie formuły X i Y których wartości przy każdym wartościowaniu są takie same (tj. takie, że równoważność X↔Y jest tautologią (patrz 2.4.4.).

Piśmiennictwo: *Ben-Ari M*. B.2.1., *Greniewski H*. G.2.1. , *Pacholski L.* P.1.1., *Tiuryn J.* T.4.1.

2.2.3. METODA ZERO-JEDYNKOWA DOWODU

2.2.3.01. **Wyjaśnienie**. Przez *„tabliczkę zero-jedynkową"* rozumiemy każdą z tabliczek z określonymi wartościami formuły zdaniowej wg zasady 2.2.2.20. (np. tabliczka dla formuły zdaniowej negacji 2.2.3.10 lub tabliczka formuły zdaniowej od dwóch argumentów zdaniowych 2.2.3.20) zawierającą pełną listę wartości argumentów formuły zdaniowej oraz odpowiadające tym kombinacjom argumentów wartości przyjmowane przez formułę.

Tabela 2.2.3.10	
Negacja zdaniowa	
p	$\neg p$
I.	II.
0	1
1	0

Tabela 2.2.3.11					
Alternatywa		Koniunkcja		Implikacja	
$p\ q$	$p \vee q$	$p\ q$	$p \wedge q$	$p\ q$	$p \rightarrow q$
I.	II.	I.	III.	I.	IV.
0 0	0	0 0	0	0 0	1
0 1	1	0 1	0	0 1	1
1 0	1	1 0	0	1 0	0
1 1	1	1 1	1	1 1	1

2.2.3.20. **Wyjaśnienie**. Tabliczka zero-jedynkowa jest narzędziem umożliwiającym sprawdzenie zachodzenia spełnienia przez formułę logiczną. Czyli uzyskania odpowiedzi na pytanie: *czy dana formuła jest tautologią, czy też nie*?
Przykład. Weźmy pod uwagę przykładową formułę zdaniową:

2.2.3.21. $$\Phi = \neg p \rightarrow (p \rightarrow q).$$

Zbudujmy tabliczkę zero-jedynkową umożliwiającą sprawdzenie spełnienia formuły 2.2.3.21.

Tabela 2.2.3.22			
$p\ q$	$p \to q$	$\neg p$	III → II
I.	II.	III.	IV.
0 0	1	1	1
0 1	1	1	1
1 0	0	0	1
1 1	1	0	1

Jak widać z zawartości kolumny „IV" tabeli 2.2.3.22, przy każdej z czterech możliwych kombinacji argumentów p oraz q, mamy 1. Co oznacza, że formuła zdaniowa 2.2.3.21 jest tautologią.

2.2.3.30. **Wyjaśnienie**. Wyjaśnijmy, co rozumiemy pod pojęciem walidacji formuły rachunku zdań. Jeśli okaże się, że dana formuła została pozytywnie *zwalidowana*, to mówimy, że jest ona tautologią, czyli że dla każdej możliwej kombinacji zmiennych zdaniowych – argumentów danej formuły jest ona prawdziwa. W tym celu rozważmy np. formułę zdaniową 2.2.3.21. Każda ze zmiennych – argumentów formuły może przyjmować jedną z dwu wartości *prawda* oraz *fałsz* – kodowanych 1 oraz 0 (patrz tabela 2.2.3.22). Obok dowodu z pomocą tabliczki zero-jedynkowej, możemy dokonać dowodu analitycznego, wyprowadzając zachodzenie tautologii 2.2.3.21, ze wzorów 2.2.2.21 – 2.2.2.27. Czyli mamy dwie równoważne metody walidacji (dowodzenia spełniania lub braku spełniania) formuły zdaniowej: (1) tabliczki zero-jedynkowe oraz (2) dowodu analitycznego.

Piśmiennictwo: *Ben-Ari M*. B.2.1., *Greniewski H*. G.2.1. , *Pacholski L.* P.1.1., *Tiuryn J.* T.4.1.

2.2.4. TAUTOLOGIE JEDNEJ ZMIENNEJ KLASYCZNEGO RACHUNKU ZDAŃ

W podrozdziale 2.2.2. wprowadziliśmy pojęcie tautologii. W niniejszym podrozdziale podamy szereg tautologii od jednego argumentu zdaniowego, z których pewne były znane już w Starożytności, a inne w Średniowieczu.

2.2.4.01. *Bezwzględna zwrotność implikacji*
(w terminologii klasycznej „zasada tożsamości"): $\vdash\ (p \to p)$.

2.2.4.02. *Bezwzględna zwrotność przeciw-implikacji*: $\vdash\ (p \leftarrow p)$.

2.2.4.03. *Bezwzględna zwrotność równoważności*: $\vdash\ (p \leftrightarrow p)$.

2.2.4.11. *Przeciw zwrotność różnicy symetrycznej*: $\vdash\ \neg(p \dot{-} p)$.

2.2.4.21 *Zasada wyłączonego środka:* $\vdash [p \vee (\neg p)]$.

2.2.4.22 $\vdash [(\neg p)\ /\ p]$.

2.2.4.23 $\vdash [p \dot{-} (\neg p)]$.

2.2.4.24. *Zasada sprzeczności:* $\vdash \neg[(\neg p)\ \wedge p]$.

Sformułujemy teraz cztery, krótkie twierdzenia; każde z nich zawiera jedną stałą zdaniową.

2.2.4.31 $\vdash\ (1\ \vee p)$.

2.2.4.32 $\vdash\ (0\ \to p)$.

2.2.4.33 $\vdash\ (p \to\ 1)$.

2.2.4.34 $\vdash \neg(0 \wedge p)$.

2.2.4.40. *Prawo podwójnej negacji zdaniowej:*

$$\vdash \{p \leftrightarrow [\neg(\neg p)]\}.$$

Podamy teraz kilkanaście twierdzeń o bardziej niż dotychczasowe skomplikowanej strukturze. Niektóre z tych twierdzeń żywo przypominają znane nam dobrze wzory algebry szkolnej. Inne jednak ostrzegają, że podobieństwo między rachunkiem zdań a algebrą szkolną jest złudne.

2.2.4.51 $\vdash [p \leftrightarrow (p \vee 0)]$.

2.2.4.52 $\vdash [p \leftrightarrow (p \wedge 1)]$.

2.2.4.53 $\vdash [p \leftrightarrow (p \dot{-} 0)]$.

2.2.4.54 $\vdash [p \leftrightarrow (1 \rightarrow p)]$.

2.4.4.55 $\vdash [p \leftrightarrow (p \leftrightarrow 1)]$.

2.2.4.61 $\vdash [1 \leftrightarrow (p \vee 1)]$.

2.4.4.62 $\vdash [0 \leftrightarrow (p \wedge 0)]$.

2.2.4.71 $\vdash [p \leftrightarrow (p \vee p)]$.

2.2.4.72 $\vdash [p \leftrightarrow (p \wedge p)]$.

Postawmy sobie teraz pytanie, czy negacja zdaniowa jest nieodzowna w języku budowanego systemu, czy też można by się jej wyrzec bez istotnego zubożenia języka, to jest bez zwężania zawężania zakresu myśli dających się wypowiedzieć w danym języku?

Odpowiedzi na to zagadnienie językoznawcze udziela nam dość niespodziewanie rachunek zdań, a to w postaci twierdzeń następujących (prawa rugowania negacji):

2.2.4.81 $\vdash [\neg p \leftrightarrow (p \dot{-} p)]$.

2.2.4.82 $\vdash [\neg p \leftrightarrow (p \rightarrow p)]$.

2.2.4.83 $\vdash [\neg p \leftrightarrow (p \leftrightarrow p)]$.

Na zakończenie zwróćmy jeszcze uwagę na następujące dwa twierdzenia:

2.2.4.90 $\vdash \{[(\neg p) \rightarrow p] \leftrightarrow p\}$.

2.2.4.91 $\vdash \{[p \rightarrow (\neg p)] \leftrightarrow \neg p\}$.

Zastosowanie praktyczne (i to doniosłe) ma postać tak zwana „osłabiona: obu powyższych twierdzeń (równoważność została tu zastąpiona implikacją)”

2.2.4.92 $\vdash \{[(\neg p) \rightarrow p] \rightarrow p\}$.

2.2.4.93 $\vdash \{[p \rightarrow (\neg p)] \rightarrow (\neg p)\}$.

Piśmiennictwo: *Ben-Ari M*. B.2.1., *Greniewski H*. G.2.1. , *Pacholski L.* P.1.1., *Tiuryn J.* T.4.1.

2.2.5. TAUTOLOGIE DWU ZMIENNYCH KLASYCZNEGO RACHUNKU ZDAŃ

Dotychczas omawialiśmy najbardziej znane tautologie jednej zmiennej zdaniowej. Z kolei wymienimy pewne tautologie dwu zmiennych zdaniowych.

2.2.5.01. *Symetria alternatywy*: $\vdash [(p \vee q) = (q \vee p)]$.

2.2.5.02. *Symetria różnicy symetrycznej*: $\vdash [(p \dot{-} q) = (q \dot{-} p)]$.

2.2.5.03. *Symetria równoważności*: $\vdash [(p \leftrightarrow q) = (q \leftrightarrow p)]$.

2.2.5.04. *Symetria koniunkcji*: $\vdash [(p \wedge q) = (q \wedge p)]$.

2.2.5.05. *Względna zwrotność koniunkcji*: $\vdash \{[(p \wedge q) \vee (q \wedge p)] \rightarrow (p \wedge p)\}$.

Dotychczasowe twierdzenia zawierające dwie zmienne były wprawdzie bardzo łatwe, ale też mało użyteczne; przejdziemy teraz do twierdzeń może trudniejszych do zrozumienia, ale za to wysoce użytecznych. Rozpoczniemy od praw De Morgana.

2.2.5.20. **Wyjaśnienie**. Niech F i G będą funkcjami zdaniowymi, z których każda zawiera dwie zmienne zdaniowe i zamiast funkcji F pisali schemat „$(p \lhd q)$”, zamiast funkcji G - schemat „$(p \rhd q)$”. Przez prawo De Morgana dla F i G - rozumiemy twierdzenie mające postać:

$$\{[(\neg p) \lhd (\neg q)] \leftrightarrow [\neg(p \rhd q)]\}.$$

2.2.5.21 *Prawo De Morgana dla alternatywy i koniunkcji*:

$$\vdash \{[(\neg p) \vee (\neg q)] \leftrightarrow [\neg(p \wedge q)]\}.$$

Interpretacja: [($Fałszem$ jest, że p) lub ($fałszem$ jest, że q)] wtedy i tylko wtedy, jeżeli $fałszem$ jest, że *(porazq).*

2.2.5.22. *Prawo De Morgana dla koniunkcji i alternatywy*:

$$\vdash \{[(\neg p) \wedge (\neg q)] \leftrightarrow [\neg(p \vee q)]\}.$$

Interpretacja: [($Fałszem$ jest, że p) oraz ($fałszem$ jest, że q)] wtedy i tylko wtedy, jeżeli $fałszem$ jest, że *(p lub q)*. Interpretacja swobodna: $(nie - p\ oraz\ nie - q)$ wtedy i tylko wtedy, jeżeli $nie -\ (p\ lub\ q)$.

2.2.5.23. *Prawo De Morgana dla różnicy symetrycznej i równoważności*:

$$\vdash \{[(\neg p) \dot{-} (\neg q)] \leftrightarrow [\neg(p \leftrightarrow q)]\}.$$

2.2.5.24. *Prawo De Morgana dla równoważności i różnicy symetrycznej*:

$$\vdash \{[(\neg p) \leftrightarrow (\neg q)] \leftrightarrow [\neg(p \dot{-} q)]\}.$$

Prawa De Morgana mają liczne zastosowania:

1) Są one często pomocne w dowodach twierdzeń logiki i ogólnie twierdzeń matematycznych.
2) Mają zastosowanie przy projektowaniu niektórych rodzajów układów kombinacyjnych (patrz rozdział 4.2).
3) Mogą też być przydatne do redagowania tekstów w języku naturalnym, Zauważmy, że w języku naturalnym trudno jest sformułować negację koniunkcji. Odpowiednie prawo De Morgana pozwala natomiast zastąpić negację koniunkcji - łatwą do sformułowaniu w języku naturalnym alternatywę dwóch negacji. Bardzo trudno jest również sformułować w języku naturalnym negację alternatywy; drugie prawo De Morgana pozwala zastąpić to wyrażenie łatwą do sformułowania w języku naturalnym koniunkcją dwóch negacji.

Spośród twierdzeń zawierających dwie zmienne zajmiemy się teraz tymi, które są formami wnioskowania. Celowo rozpoczniemy ten przegląd form wnioskowania od najprostszych.

2.2.5.31 $\vdash [(p \wedge q) \rightarrow p]$.

2.2.5.32 $\vdash [p \rightarrow (q \vee p)]$.

2.2.5.33 $\vdash \{p \rightarrow [q \rightarrow (p \wedge q)]\}$.

2.2.5.34 $\vdash [p \rightarrow (q \rightarrow p)]$.

2.2.5.35 $\vdash [(\neg p) \rightarrow (p \rightarrow q)]$.

2.2.5.36 $\vdash \{p \rightarrow [(\neg p) \rightarrow q]\}$.

Z każdej z powyższych tez można - otrzymać (po dokonaniu interpretacji merytorycznej) wiele pozornie paradoksalnych zdań. Paradoksalność ta jednak znika, gdy przypomnijmy sobie, że postanowiliśmy używać funktora „jeżeli … to" w szerszym niż potocznie rozumieniu.

W praktyce bywają też przydatne następujące nieskomplikowane twierdzenia, stosowane (po dokonaniu interpretacji merytorycznej), jako formy wnioskowania:

2.2.5.41 $\vdash [(p \wedge q) \rightarrow (p \vee q)]$.

2.2.5.42 $\vdash [(p \wedge q) \rightarrow (p \leftrightarrow q)]$.

2.2.5.43 $\vdash [(p \dot{-} q) \rightarrow (p \vee q)]$.

2.2.5.44 $\vdash [(p \leftrightarrow q) \rightarrow (p \rightarrow q)]$.

2.2.5.45 $\vdash [(p \leftrightarrow q) \rightarrow (p \leftarrow q)]$.

Twierdzenia 2.2.5.41 - 2.2.5.45, można wyrazić w następujący - nieścisły, ale treściwy - sposób:
1) Koniunkcja pociąga za sobą stale alternatywę i równoważność.
2) Różnica symetryczna pociąga za sobą stale alternatywę i dyzjunkcję.
3) Równoważność pociąga za sobą stale implikację i przeciw implikację.

Warto zwrócić uwagę na następujące trzy twierdzenia, używane nieraz, przez matematyków w tak zwanych dowodach apagogicznych:

2.2.5.51 $\vdash \{(p \rightarrow q) \rightarrow \{[p \rightarrow (\neg q)] \rightarrow (\neg p)]\}\}$.

2.2.5.52 $\vdash \{\{p \rightarrow [q \wedge (\neg q)]\} \rightarrow (\neg p)\}$.

2.2.5.53 $\vdash \{\{p \rightarrow [q \leftrightarrow (\neg q)]\} \rightarrow (\neg p)\}$.

Zajmiemy się wkrótce kilku twierdzeniami, które noszą (pochodzące ze średniowiecza) nazwy łacińskie: (1) *modus ponendo-ponens*, (2) *modus popendo-tollens*, (3) *modus toliendo-ponens,* (4) *modus tollendo-tollelrs.* Są to twierdzenia proste, lecz nader pożyteczne. Sformułowania tych od dawna znanych twierdzeń poprzedzimy jednak pewnymi rozważaniami wstępnymi, w których przydatne będą poniższe dwa wyjaśnienia.

2.2.5.6.1. **Wyjaśnienie**. Niech F będzie funkcją zdaniową dwu zmiennych zdaniowych, zamiast funkcji tej będziemy pisali schemat „$(p \circ q)$”. Mówimy, że F jest funkcją modusową wtedy i tylko wtedy, jeżeli ani wyrażenie postaci „$[(p \circ q) \to (\neg q)]$, ani wyrażenie postaci „$[(p \circ q) \to q]$” nie jest twierdzeniem dwuwartościowego rachunku zdań.
Przykłady. *Funkcjami modusowymi są*: alternatywa, dyzjunkcja, różnica symetryczna, implikacja, przeciw implikacja i równoważność. *Funkcjami nie modusowymi są*: koniunkcja i obustronna negacja.

Niech F będzie funkcją zdaniową dwu zmiennych zdaniowych, zamiast tej funkcji będziemy pisali schemat „$(p \circ q)$”; T niech będzie twierdzeniem; pod tymi założeniami wprowadzamy następujące wyjaśnienia:

2.2.5.62. **Wyjaśnienie**. Twierdzenie T nazywamy „*modus ponendo-ponens funkcji F*” - wtedy i tylko wtedy, jeżeli:
1) twierdzenie T ma postać: $[(p \circ q) \to (p \to q)]$
i zarazem
2) F jest funkcją modusową.

2.2.5.63. **Wyjaśnienie**. Twierdzenie T nazywamy „*modus ponendo-tollens funkcji F*” wtedy i tylko wtedy, jeżeli:
1) twierdzenie T ma postać: $\{(p \circ q) \to [p \to (\neg q)]\}$
i zarazem
2) F jest funkcją modusową.

2.2.5.64. **Wyjaśnienie**. Twierdzenie T nazywamy „*modus tollendo-ponens funkcji F*" wtedy i tylko wtedy, jeżeli:
1) twierdzenie T ma postać: $\{(p \circ q) \to [(\neg p) \to q]\}$
i zarazem
2) F jest funkcją modusową.

2.2.5.65. **Wyjaśnienie**. Twierdzenie T nazywamy „*modus tollendo-tollens funkcji F*” wtedy i tylko wtedy, jeżeli:
1) twierdzenie T ma postać: $\{(p \circ q) \to [(\neg p) \to (\neg q)]\}$
i zarazem
2) F jest funkcją modusową.

2.2.5.71. Modus ponendo-ponens implikacji: $\vdash [(p \to q) \to (p \to q)]$.

2.2.5.72. *Modus ponendo-ponens równoważności*: $\vdash [(p \leftrightarrow q) \to (p \to q)$.

2.2.5.73. *Modus ponendo-tollens różnicy symetrycznej*: $\vdash \{(p \dot{-} q) \to [p \to (\neg q)]\}$.

2.2.5.74. Modus pollendo-ponensalternatywy: $\vdash \{(p \vee q) \to [(\neg p) \to q]\}$.

2.2.5.75. Modus tollendo-ponens różnicy symetrycznej: $\vdash \{(p \dot{-} q) \to [(\neg p) \to q]\}$.

2.2.5.76. Modus tollendo-tollens przeciw implikacji: $\vdash \{(p \leftarrow q) \to [(\neg p) \to (\neg q)]\}$.

2.2.5.77. Modus tollendo-tollens równoważności: $\vdash \{(p \leftrightarrow q) \to [(\neg p) \to (\neg q)]\}$.

Piśmiennictwo: *Greniewski H*. G.2.1. , *Knaster B.* K.3.1., *Kotarbiński T*. K.5.1., K.5.2., *Mostowski A*. M.5.1.

2.2.6. *TAUTOLOGIE ZAWIERAJĄCE TRZY LUB CZTERY ZMIENNE*

Obok tautologii jednej i dwu zmiennych zdaniowych, występują również tautologie wielu zmiennych zdaniowych. Podamy jeszcze przykłady tautologii trzech i czterech zmiennych zdaniowych.

2.2.6.00. **Wyjaśnienie**. Niech F będzie funkcją zdaniową, dwu zmiennych zdaniowych, zamiast tej funkcji będziemy pisali schemat „$(p \circ q)$". Twierdzenie mające postać:

$$\{[p \circ (q \circ r)] \leftrightarrow [(p \circ q) \circ r]\}$$ nazywamy łącznością funkcji F.

2.2.6.01. *Łączność alternatywy*: $\vdash \{[p \vee (q \vee r)] \leftrightarrow [(p \vee q) \vee r]\}$

2.2.6.02. *Łączność różnicy symetrycznej*: $\vdash \{[p \dot{-} (q \dot{-} r)] \leftrightarrow [(p \dot{-} q) \dot{-} r]\}$

2.2.6.03. *Łączność koniunkcji*: $\vdash \{[p \wedge (q \wedge r)] \leftrightarrow [(p \wedge q) \wedge r]\}$

2.2.6.04. *Łączność równoważności* (twierdzenie Łukasiewicza):

$$\vdash \{[p \leftrightarrow (q \leftrightarrow r)] \leftrightarrow [(p \leftrightarrow q) \leftrightarrow r]\}$$

2.2.6.10. **Wyjaśnienie**. Niech F oraz G będą funkcjami zdaniowymi, każda dwu zmiennych zdaniowych; zamiast funkcji F będziemy pisali schemat „$(p\Delta q)$", zaś zamiast funkcji G – schemat „$(p\nabla q)$". Twierdzenie mające postać:

$$\{[p\Delta(q\nabla r)] \leftrightarrow [(p\Delta q)\nabla(p\Delta r)]\}.$$

będziemy nazywali „*lewostronną rozdzielnością funkcji F względem funkcji G*".

2.2.6.11. **Wyjaśnienie**. Niech F, G będą funkcjami takimi, jak w 2.4.6.10; będziemy nadal używali schematów „$(p\Delta q)$" oraz „$(p\nabla q)$". Twierdzenie mające postać:

$$\{[(p\Delta q)\nabla r)] \leftrightarrow [(p\Delta q)\nabla(q\Delta r)]\}.$$

będziemy nazywali „prawostronną rozdzielnością funkcji F względem funkcji G".

2.2.6.12. *Lewostronna rozdzielność koniunkcji względem alternatywy*:

$$\vdash \{[p \wedge (q \vee r)] \leftrightarrow [(p \wedge q) \vee (p \wedge r)]\}.$$

2.2.6.13. *Lewostronna rozdzielność alternatywy względem koniunkcji*:

$$\vdash \{[p \vee (q \wedge r)] \leftrightarrow [(p \vee q) \wedge (p \vee r)]\}.$$

2.2.6.14. *Lewostronna rozdzielność koniunkcji względem różnicy symetrycznej*:

$$\vdash \{[p \wedge (q \dot{-} r)] \leftrightarrow [(p \wedge q) \dot{-} (p \wedge r)]\}.$$

2.2.6.15. *Lewo stron na rozdzielność alternatywy względem równoważności*:

$$\vdash \{[p \wedge (q \leftrightarrow r)] \leftrightarrow [(p \wedge q) \leftrightarrow (p \wedge r)]\}.$$

2.2.6.16. *Lewostronna rozdzielność implikacji względem koniunkcji* (czyli prawo składania i rozkładania): $\vdash \{[p \rightarrow (q \wedge r)] \leftrightarrow [(p \rightarrow q) \wedge (p \rightarrow r)]\}.$

W dwuwartościowym rachunku zdań nie ma twierdzenia zasługującego na nazwę „prawostronna rozdzielność implikacji względem koniunkcji". Aby się o tym przekonać, wystarczy zbadać następującą funkcję zdaniową trzech zmiennych zdaniowych, należącą do języka dwuwartościowego rachunku zdań:

$$\{[(p \wedge q) \rightarrow r] \leftrightarrow [(p \rightarrow r) \wedge (q \rightarrow r)]\}.$$

Poznaliśmy już łączności i rozdzielności dwuwartościowego rachunku zdań, teraz zaś zajmiemy się innym rodzajem twierdzeń zawierające trzy zmienne, mianowicie zajmiemy się tzw. sylogizmami.

2.2.6.20. **Wyjaśnienie**. Niech F, G, H będą funkcjami zdaniowymi dwu zmiennych zdaniowych; zamiast tych funkcji będziemy odpowiednio pisali schematy „$(p\Delta q)$”, „$(p\nabla q)$”, „$(p \circ q)$”. Przez „sylogizm postaci F– G– H” będziemy rozumieli twierdzenie mające postać:

$$\{(p\Delta q) \to [(q\nabla r) \to (p \circ r)]\}.$$

2.2.6.21. *Sylogizm mający postać - (alternatywa - implikacja – alternatywa)*:

$$\vdash \{(p \vee q) \to [(q \to r) \to (p \vee r)]\}.$$

2.2.6.22. *Sylogizm mający postać - (różnica symetryczna – różnica symetryczna – równoważność)*:

$$\vdash \{(p \dot{-} q) \to [(q \dot{-} r) \to (p \leftrightarrow r)]\}.$$

2.2.6.23. *Sylogizm mający postać - (koniunkcja - implikacja – koniunkcja)*:

$$\vdash \{(p \wedge q) \to [(q \to r) \to (p \wedge r)]\}.$$

Zwrócimy teraz uwagę na trzy rodzaje sylogizmów:

1) przechodniości,
2) ekstensjonalności lewostronne,
3) ekstensjonalności prawostronne.

2.2.6.30. **Wyjaśnienie**. Przechodniością funkcji F nazywamy sylogizm mający postać:

$$F - F - F.$$

2.2.6.31. *Przechodniość implikacji*:

$$\vdash \{(p \to q) \to [(q \to r) \to (p \to r)]\}.$$

2.2.6.32. *Przechodniość równoważności*:

$$\vdash \{(p \leftrightarrow q) \to [(q \leftrightarrow r) \to (p \leftrightarrow r)]\}.$$

2.2.6.33. *Przechodniość koniunkcji:*

$$\vdash \{(p \wedge q) \to [(q \wedge r) \to (p \wedge r)]\}.$$

Dwa spośród powyższych trzech twierdzeń - przechodniość implikacji i przechodniość równoważności, są często stosowane w praktyce, jako formy wnioskowania.

2.2.6.42. *Ekstensjonalność lewostronna alternatywy*:

$$\vdash \{(p \leftrightarrow q) \to [(q \vee r) \to (p \vee r)]\}.$$

2.2.6.50. **Wyjaśnienie**. Przez „ekstensjonalność prawostronną funkcji F” rozumiemy sylogizm mający postać: $F - równoważność - F.$

2.2.6.51. **Twierdzenie** (metateorii rachunku zdań). Jeżeli F jest funkcją prawdziwościową dwu zmiennych zdaniowych, to prawostronna ekstensjonalność funkcji F jest twierdzeniem dwuwartościowego rachunku zdań.

2.2.6.60. *Prawo łączenia i rozłączania*: $\vdash \{[(p \to q) \wedge (q \to r)] \leftrightarrow [(p \vee q) \to r]\}.$

2.2.6.61. *Prawo komutacji*: $\vdash \{[p \to (q \to r)] \leftrightarrow [q \to (p \to r)]\}.$

2.2.6.62. *Prawo importacji i eksportacji*: $\vdash \{[(p \wedge q) \to r] \leftrightarrow [p \to (q \to r)\}.$

2.2.6.63. $\vdash \{[(p \wedge q) \to r] \leftrightarrow [(p \wedge \neg r) \to \neg q\ ]\}.$

2.2.6.70. *Prawo obustronnej alternatywy*: $\vdash \{[(p \to r) \wedge (q \to s)] \to [(p \vee q) \to (r \vee s)]\}.$

Przykład. Jeżeli 1) o ile rzeka przybierze - odwrót będzie odcięty, a także 2) o ile zabraknie amunicji - nie podobna będzie iść naprzód, w takim razie jeżeli bądź rzeka przybierze, bądź zabraknie amunicji - to bądź odwrót będzie odcięty bądź nie podobna będzie iść naprzód.

2.2.6.71. *Prawo obustronnej koniunkcji*: $\vdash \{[(p \to r) \wedge (q \to s)] \to [(p \wedge q) \to (r \wedge s)]\}$.

Piśmiennictwo: *Greniewski H*. G.2.1. , *Knaster B.* K.3.1., *Kotarbiński T*. K.5.1., K.5.2., *Mostowski A*. M.5.1.

2.2.7. LEMAT O PODSTAWIANIU

2.2.7.11. **Definicja**. Podstawieniem formuły ψ w miejsce zmiennej zdaniowej p nazywamy przekształcenie, które formule φ przyporządkowuje formułę powstałą przez wstawienie formuły ψ w miejsce wystąpienia zmiennej p w formule φ. Formalnie:

$$p[\,p/\psi\,] = \psi$$

$$q\,[\,p/\psi\,] = q, \text{ dla } q \neq p$$

$$(\neg\varphi)[\,p/\psi\,] = \neg(\varphi[\,p/\psi\,])$$

$$(\varphi_1 \vee \varphi_2)[\,p/\psi\,] = (\varphi_1\,[\,p/\psi\,]) \vee (\varphi_2\,[\,p/\psi\,])$$

$$(\varphi_1 \wedge \varphi_2)[\,p/\psi\,] = (\varphi_1\,[\,p/\psi\,]) \wedge (\varphi_2[\,p/\psi\,])$$

$$(\varphi_1 \to \varphi_2)[\,p/\psi\,] = (\varphi_1\,[\,p/\psi\,]) \to (\varphi_2\,[\,p/\psi\,])$$

$$(\varphi_1 \leftrightarrow \varphi_2)[\,p/\psi\,] = (\varphi_1\,[\,p/\psi\,]) \leftrightarrow (\varphi_2\,[\,p/\psi\,])$$

2.2.7.12. Przykład. $(p \to p \vee q)[\,p/q \to p] = (q \to p) \to (q \to p) \vee q$.

W podobny sposób definiujemy jednoczesne podstawienie formuł w miejsce kilku zmiennych zdaniowych $[\,p_1\,/\psi_1\,, \ldots, p_n\,/\psi_n\,]$. Jest to odwzorowanie, które formule φ przyporządkowuje formułę powstałą przez wstawienie formuły ψ_i w miejsce wystąpienia każdego zmiennej p_i w formule φ, dla każdego i =1, . . . , n.

2.2.7.21. **Lemat o podstawianiu**. Jeżeli formuła φ jest tautologią to dla dowolnej zmiennej p i dowolnej formuły ψ formuła φ[p/ψ] jest tautologią.

Powyższy lemat ma bardzo duże znaczenie praktyczne, pozwala bowiem dowodzić, że skomplikowane formuły są tautologiami. Dla przykładu formuła

$$(q \to p) \to (q \to p) \vee q \qquad \text{(z przykładu 2.2.7.12.)}$$

jest tautologią, gdyż jest wynikiem podstawienia formuły $q \to p$ w miejsce zmiennej p w tautologii $p \to p \vee q$.

Piśmiennictwo: *Ben-Ari M*. B.2.1., *Mostowski A*. M.5.1., *Pacholski L.* P.1.1., *Tiuryn J.* T.4.1.

2.2.8. TWIERDZENIE O PUNKTACH STAŁYCH

W tym podrozdziale przedstawimy podstawową wersję twierdzenia *Knastra-Tarskiego* o punktach stałych funkcji monotonicznych. Omawiane twierdzenie dotyczy funkcji, których argumentami są zbiory, a ich podzbiory zawierają punkty stałe. Jest to z punktu widzenia algorytmiki bardzo ważne twierdzenie, ponieważ bardzo wiele algorytmów jest rekursywna, a jeśli prowadzi do rozwiązania monotonicznie, to ma zastosowanie twiedzenie twierdzenie *Knastra-Tarskiego*.

2.2.8.01. **Definicja**. Funkcję f odwzorowującą zbiór $P(X) \Longrightarrow P(X)$ nazwiemy monotonicznym ze względu na zawieranie, jeśli $\forall x \subset X \forall y \subset X\ [x \subset y \to f(x) \subset f(y)]$.

2.2.8.02. **Wyjaśnienie**. Funkcje monotoniczne ze względu na zawieranie zachowują relację zawierania pomiędzy przekształcanymi zbiorami. Nie oznacza to jednak wcale, że argument funkcjonału musi być podzbiorem wartości funkcji na tym argumencie.

2.2.8.03. **Definicja**. Element $x \in X$ jest *punktem stałym* funkcji f odwzorowujący zbiór $X \Rightarrow X$, jeśli $f(x) = x$.

2.2.8.04. **Wyjaśnienie**. Należy zauważyć, że istnieją funkcjonały, które nie mają punktów stałych. Prostym przykładem może być funkcjonał odwzorowujący zbiór {(0, 1), (1, 0)} na siebie.

2.2.8.05. **Definicja**. Punkt $x_0 \subset X$ jest najmniejszym punktem stałym funkcji f odwzorowującej zbiór $P(X) \Rightarrow P(X)$, jeśli $f(x_0) = x_0$ oraz $\forall x_1 \subset X\ f(x_1) = x_1 \rightarrow x_0 \subset x_1$. Czyli wtedy, kiedy każdy inny punkt stały jest jego nadzbiorem.

2.2.8.05. **Definicja**. Punkt $x_0 \subset X$ jest największym punktem stałym funkcji f odwzorowującej zbiór $P(X) \Rightarrow P(X)$, jeśli $f(x_0) = x_0$ oraz $\forall\ x_1 \subset X\ \ f(x_1) = x_1 \rightarrow x_0 \supset x_1$. Czyli wtedy, kiedy każdy inny punkt stały jest jego podzbiorem.

2.2.8.10. **Twierdzenie Knastra-Tarskiego**. Każda monotoniczna funkcja $f: P(X) \Rightarrow P(X)$ posiada najmniejszy i największy punkt stały.

Dowód. Niech będzie dany zbiór $L = \{ x \subset X: f(x) \supset x\}$. Pokażemy, że kres górny zbioru L, oznaczony $\cup L$, jest największym punktem stałym funkcjonału f. Zauważmy, że dla każdego $x \in L$, z monotoniczności funkcjonału f, otrzymujemy $f(\cup L) \supset f(x) \supset x$. Wobec tego również $f(\cup L) \supset \cup L$, skąd otrzymujemy, że $\cup L \in L$. Poddając obie strony poprzedniej zależności funkcji f dzięki jego monotoniczności, otrzymamy $f(f(\cup L)) \supset f(\cup L)$. Wobec czego również $f(\cup L) \in L$. Ponieważ każdy element L jest podzbiorem $\cup L$, to również $f(\cup L) \subset \cup L$. Stąd i z 2.2.8.01 otrzymujemy $f(\cup L) = \cup L$, a więc $\cup L$ jest punktem stałym funkcjonału f. Co więcej, wszystkie punkty stałe należą do zbioru L, wobec czego każdy z nich jest podzbiorem $\cup L$, co oznacza dokładnie, że $\cup L$ jest największym punktem stałym.

Analogicznie wykażemy istnienie najmniejszego punktu stałego. Niech będzie dany zbiór $M= \{ x \subset X: f(x) \subset x\}$. Pokażemy, że $\cap U$ jest najmniejszym punktem stałym. Z monotoniczności f mamy dla każdego $x \in M$, $f(\cap M) \subset f(x) \subset x$, skąd otrzymujemy $f(\cap M) \subset \cap M$. Stąd i z 2.2.8.01 otrzymujemy zależność $\cap M \in M$. Poddając obie strony poprzedniej zależności funkcjonału f dzięki jego monotoniczności, otrzymamy $f(f(\cap M)) \subset f(\cap M)$, skąd wynika, że $f(\cap M) \in M$. Ponieważ $\cap M$ jest podzbiorem każdego elementu M, więc również $\cap M \subset f(\cap M)$. Stąd i z 2.2.8.01 otrzymujemy $f(\cap M) = \cap M$. Oznacza to, że $\cap M$ jest punktem stałym funkcjonału f. Ponieważ wszystkie punkty stałe należą do zbioru M, to $\cap M$ jest najmniejszym punktem stałym.

Piśmiennictwo: *Ben-Ari M.* B.2.1., *Tarski A.* T.2.1.

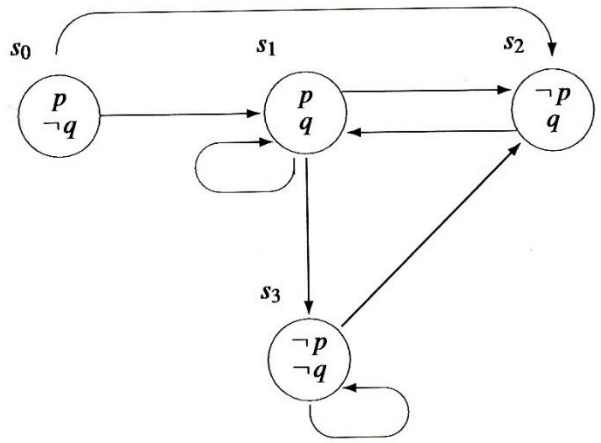
2.3. LOGIKA MODALNA I JEJ ROZSZERZENIA

2.3.0. UWAGI WSTĘPNE

Logika modalna uprawiana była już przez *Arystotelesa* jako sylogistyka zdań modalnych (porównaj 2.0.1). Współczesną postacią logiki modalnej jest modalny rachunek zdań. Cechą charakterystyczną modalnych rachunków zdań jest występowanie w nich: funktora możliwości, oznaczanego <> oraz funktora konieczności, oznaczanego przez □. Twórcą pierwszych systemów modalnego rachunku zdań (nazwanych później S1 i S2) jest *C. I. Lewis*. Następnie powstało jeszcze kilka innych systemów – *Lewisa* (zwanych odpowiednio S3, S4, S5). Intencją *Lewisa* było stworzenie takiej logiki, która lepiej niż implikacja materialna w klasycznym rachunku zdań oddawałaby implikację występującą w języku naturalnym. Lewis nie uświadamiał sobie jeszcze w pełni różnicy między wynikaniem a implikacją ścisłą, współcześnie jednak logiki *Lewisa* interpretuje się powszechnie jako logiki zdań modalnych, na których gruncie właśnie implikację ścisłą zdefiniować można następująco:

2.3.0.10 $$p \rightarrow q =_{\mathrm{Df}} \neg <> (p \wedge \neg q).$$

Dalsze rozszerzenia logiki modalnej prowadzą między innymi do logik temporalnych, w szczególności logik temporalnych dotyczących czasu dyskretnego (nie ciągłego). Wyobraźmy sobie graf stanów dwu zmiennych zdaniowych, którego



Rysunek 2.3.0.00. Graf stanów zmiennych zdaniowych *p* oraz *q*.

wierzchołki służą do przechowywania stanów *prawda* (lub *true*) i *fałsz* (lub *false*). Stany grafu znajdują się w węzłach (patrz rys. 2.3.0.00):

2.3.0.20 $s_0(p) = \top$, $s_0(q) = \bot$, $s_1(p) = \top$, $s_1(q) = \top$, $s_2(p) = \bot$, $s2(q) = \top$, $s_3(p) = \bot$, $s_3(q) = \bot$.

Funkcjonalność przejście (*transition*) ○ - pomiędzy węzłami grafu (patrz rys. 2.3.0.00) opisują zbiory dopuszczalnych przejść:

2.3.0.30 $\circ(s_0) \in \{s_1, s_2\}$, $\circ(s_1) \in \{s_1, s_2, s_3\}$, $\circ(s_2) \in \{s_1\}$, $\circ(s_3) \in \{s_2, s_3\}$.

2.3.0.40 **Wyjaśnienie**. *Funktor możliwości* $<>p = \top$ - w przypadku grafu stanów możemy interpretować, iż wśród stanów węzłów grafu, wystąpi co najmniej jeden raz wartość $p = \top$; natomiast $\neg<>p = \top$ – możemy interpretować, iż wśród stanów węzłów grafu, nie wystąpi co najmniej jeden raz wartość $p = \bot$. Natomiast *funktor konieczności* $\Box p = \top$ - w przypadku grafu stanów możemy interpretować, iż we węzłach grafu mamy stan $p = \top$; natomiast $\neg\Box p = \top$ - możemy interpretować, iż żadnym węźle grafu stanów, niema stanu $p = \top$.

Piśmiennictwo: *Ben-Ari M*. B.2.1., *Chellas B*. C.1.1. *Harel D.* H.1.1., *Tiuryn J.* T.4.1.

2.3.1. System modalny S5

System S5 należy do najszerzej znanych i najprostszych systemów modalnego rachunku zdań. Występujące w nim funktory zdaniotwórcze *możliwości* <> i *konieczności* □, odróżniające go od klasycznego rachunku zdań, można rozumieć intuicyjnie odwołując się do Leibnizowskiej koncepcji światów możliwych: zdanie konieczne jest prawdziwe we wszystkich możliwych światach, zdanie możliwe jest prawdziwe w niektórych możliwych światach. Funktory *możliwości* <> i *konieczności* □ są przy tym wzajemnie definiowalne w następujący sposób:

2.3.1.11 $<>X \leftrightarrow \neg\Box\neg X$;

2.3.1.12 $X \leftrightarrow \neg<>\neg X$;

gdzie X jest formułą zdaniową.

Piśmiennictwo: *Ben-Ari M*. B.2.1., *Chellas B*. C.1.1. *Harel D.* H.1.1., *Tiuryn J.* T.4.1.

2.3.2. Język systemu modalnego S5

2.3.2.10 **Wyjaśnienie**. Słownik języka logiki modalnej systemu S5 jest językiem klasycznego dwuwartościowego rachunku zdań rozszerzonym o *funktory zdaniotwórcze* możliwość <> oraz konieczność □.

2.3.2.20 **Wyjaśnienie**. Na język składają się następujące elementy:

1. zmienne zdaniowe w ilości nieograniczonej, oznaczane przez p, q, r, s...
2. stałe logiczne stanowią funktory zdaniotwórcze: zero-argumentowe – stała *verum* $\bot$ oraz stała *falsum* $\top$; spójniki jednoargumentowe: negacji $\neg$, możliwości <> i konieczności □; spójniki dwuargumentowe koniunkcji $\wedge$, alternatywy $\vee$, implikacji $\rightarrow$ oraz równoważności $\leftrightarrow$.

2.3.2.30 **Wyjaśnienie**. Do zbioru formuł zdaniowych należą natomiast:

1. wszystkie zmienne zdaniowe,
2. wyrażenia $\top$ i $\bot$,
3. wyrażenia $\neg X$, $\Box X$, $<>X$, gdzie X jest formułą zdaniową,
4. wyrażenia $X \wedge Y$, $X \vee Y$, $X \rightarrow Y$, $X \leftrightarrow Y$, gdzie X i Y są formułami zdaniowymi
5. wyrażenia powstające przez zastosowanie reguł 3 i 4 w skończonej liczbie kroków.

Piśmiennictwo: *Ben-Ari M*. B.2.1., *Chellas B*. C.1.1. *Harel D.* H.1.1., *Tiuryn J.* T.4.1.

2.3.3. Aksjomatyka systemu modalnego S5

Na gruncie systemu S5 przyjmuje się następujące aksjomaty i definicje (uzupełniające aksjomatykę klasycznego dwuwartościowego rachunku zdań):

2.3.3.10. Aksjomat $\Box X \rightarrow X$.

2.3.3.20. Aksjomat $<>X \rightarrow \Box<>X$.

2.3.3.30. Aksjomat $\Box(X \rightarrow Y) \rightarrow (X \rightarrow Y)$.

2.3.3.40. Aksjomat możliwości $<>X \leftrightarrow \neg\Box\neg X$.

Na gruncie systemu S5 przyjmuje się następujące dwie reguły dowodzenia:

2.3.3.50. Reguła wnioskowania 1 $\frac{X}{\Box X}$

2.3.3.60. Reguła wnioskowania 2 $\frac{X \to Y, X}{Y}$

Piśmiennictwo: *Ben-Ari M*. B.2.1., *Chellas B*. C.1.1. *Harel D.* H.1.1., *Tiuryn J.* T.4.1.

2.3.4. Wybrane prawa systemu modalnego S5

Poniżej podajemy przykłady praw systemu S5, rozszerzające zestaw praw klasycznego dwuwartościowego rachunku zdań. A mianowicie:

2.3.4.11. **Prawo**. Jeśli X jest konieczne, to X zachodzi rzeczywiście: $\Box X \to X$.

2.3.4.12. **Prawo**. Jeśli X jest możliwe, to jest konieczne, że X jest możliwe: $<>X \to \Box<>X$.

2.3.4.21. **Prawo** rozdzielności konieczności względem implikacji mówi, że jeśli konieczna jest zarazem implikacja i jej poprzednik, konieczny jest też jej następnik: $\Box(X \to Y) \to (\Box X \to \Box Y)$.

2.3.4.22. **Prawo** rozdzielności konieczności względem koniunkcji: $\Box(X \wedge Y) \to (\Box X \wedge \Box Y)$.

2.3.4.31. **Prawo** niniejsza mówi, że co jest prawdziwe, jest zarazem możliwe: $X \to <>X$.

2.3.4.32. **Prawo** niniejsza mówi, że co jest konieczne, jest też możliwe: $X \to <>X$.

2.3.4.33. **Prawo**. Jeśli zachodzi X, to jest konieczne, że X jest możliwe: $X \to \Box<>X$.

2.3.4.41. **Prawo**. Co może być konieczne, zachodzi rzeczywiście: $<>\Box X \to X$.

2.3.4.42. **Prawo**. Jeśli coś jest konieczne, to jest konieczne, że jest to konieczne: $\Box X \to \Box\Box X$.

2.3.4.43. **Prawo**. Jeśli jest możliwe, że coś jest możliwe, to jest to możliwe: $<><>X \to <>X$.

Piśmiennictwo: *Ben-Ari M*. B.2.1., *Chellas B*. C.1.1. *Harel D.* H.1.1., *Tiuryn J.* T.5.1.

2.3.5. Logiki temporalne jako rozwinięcie idei modalnych

2.3.5.10. **Wyjaśnienie**. Dopiero w drugiej połowie XX wieku, pojawiły się rozwiązania pozwalające na wykorzystaniu modalności dla rozszerzenia logiki zadań o funktory czasu i stworzenie narzędzi do dowodzenia twierdzeń, nazwanych logiką temporalną. Obecnie pojęciem „*logika temporalna*" określanych jest wiele różnych logik, jedynym czynnikiem łączącym te różnorodne logiki, jest zależność od czasu.

2.3.5.20. **Wyjaśnienie**. Wzmiankowane logiki różnią się pomiędzy sobą, między innymi, przyjętymi definicjami funktorów czasu. Punktem wyjścia do logiki temporalnej, a ściślej mówiąc do temporalnego rachunku zdań, podobnie jak do *trójwartościowego rachunku zdań Łukasiewicza*, są wyrażenia modalne i ich interpretacja ze względu na czas. Funktory modalne można interpretować względem czasu na trzy sposoby, co prowadzi do trzech różnych logik czasu:

1. Logiki czasu gramatycznego;
2. Logiki czasu liniowego fizycznego i technicznego;
3. Logiki czasu relatywnego i rozgałęzionego.

2.3.5.30. **Wyjaśnienie**.. *Logiki czasu gramatycznego* - określają jedynie czasy, jako przeszłe, teraźniejsze oraz przyszłe, natomiast nie dają możliwości precyzyjniejszego określenia danej chwili czasu.

2.3.5.40. **Wyjaśnienie**. *Logiki czasu liniowego* - dotyczą zarówno czasu ciągłego, jak i dyskretnego. W zasadzie czas liniowy nie ma początku ani końca (podobnie jak prosta w geometrii euklidesowej), natomiast zdarzenia są uporządkowane w czasie, czyli jeśli mamy parę zdarzeń o określonym czasie rozpoczęcia – to potrafimy powiedzieć, które z nich rozpoczęło się wcześniej, a które rozpoczęło się później (chyba, że rozpoczęło się jednocześnie). Z takim czasem mamy do czynienia w wielu eksperymentach fizycznych i w wielu urządzeniach technicznych, jak również w stosunkowo prostych (precyzyjnie mówiąc – jednowątkowych) programach komputerowych.

2.3.5.50. **Wyjaśnienie**. *Logika czasu relatywnego i rozgałęzionego* – dotyczy zdarzeń fizyki relatywistycznej, jak również rozproszonych (w tym również wieloprocesorowych) systemów informatycznych, a wersja rozgałęziania czasu dotyczy złożonych – wielowątkowych programów komputerowych, w szczególności wykonywanych na wieloprocesorowych komputerach lub w sieciach.

Ze względu na tematykę niniejszej książki, interesować nas będzie jedynie czas dyskretny, w wersji liniowej.

Piśmiennictwo: *Ben-Ari M*. B.2.1., *Kazanecka A*. K.6.1., *Pnueli A.* P.3.1., *Prior A.* P.4.1.

2.3.6. LOGIKA CZASU LINEARNEGO

2.3.6.10. **Wyjaśnienie**. Logika czasów gramatycznych wywodzi się z logiki modalnej. Jej twórcą był w 1947 roku – brytyjski logik *Arthur N. Prior* – wprowadzając cztery funktory zależności czasowych:

2.3.6.11. Funktor *H p* - *było w przeszłości zawsze tak, że* p - w skrócie: „*dotąd zawsze*".

2.3.6.12. Funktor *P p* – *było w przeszłości kiedyś tak, że* p - w skrócie: „*nastąpiło*".

2.3.6.13. Funktor *G p* - *będzie w przyszłości zawsze tak, że* p - w skrócie: „*odtąd zawsze*".

2.3.6.14. Funktor *F p* - *będzie w przyszłości kiedyś tak, że* p - w skrócie: „*nastąpi*".

Rachunek zdań rozszerzony przez *Priora*, oznaczono symbolem PL, jest to system czasu linearnego – dyskretnego, w którym relacja poprzedzania "<" jest przechodnia, obustronnie liniowa, bez momentu początkowego i końcowego (tak jak prosta w geometrii euklidesowej).

Funktory *G* i *F* oraz *H* i *P* są wzajemnie zależne. Zależności te mają odpowiednio postać:

2.3.6.21. $G\,\mathrm{p} \equiv [\neg F\,(\neg p)]$.

2.3.6.22. $H\,\mathrm{p} \equiv [\neg P\,(\neg p)]$.

2.3.6.30. **Wyjaśnienie**. Rachunek ten z pewnymi rozszerzeniami pozwala na prowadzenie szeregu formalnych rozważań dotyczących gramatycznych zależności czasowych, np. w językach naturalnych. Natomiast z punktu widzenia nauk ścisłych, w tym informatyki – jest mało przydatny.

2.3.6.40. **Wyjaśnienie**. W odniesieniu do czasu używanego przez fizykę newtonowską i technikę – potrzebne są dodatkowe funkcjonalności:

- Liniowość czasu, który może być dyskretny lub ciągły;
- Warunkowe spełnianie zdania - *p*, w zależności od prawdziwości albo fałszywości drugiego ze zdań - *q*.

W tym celu - *Kamp* w 1968 roku wprowadziła dwa binarne operatory *S* - "*since*" czyli "*od tego momentu*" oraz *U* "*until*" czyli "*aż do momentu*". Oba wzmiankowane operatory są definiowane, jak niżej:

2.3.6.41 Funktor *S*(*p*, *q*) - "*p* ma wartość *prawda*, tak długo jak *q* ma wartość *prawda*".

2.3.6.42 Funktor *U*(p, q) - "*p* będzie miało wartość *prawda*, gdy *q* przyjmie wartość *prawda*".

2.3.6.50. **Wyjaśnienie**. Dodatkowo wprowadzono operator ○ *następstwa czasu* („*the next time*" operator ○), który w przypadku czasu dyskretnego mówi o następnej jednostce (*chronie*) czasu. Dla czasu dyskretnego - operator ○ daje się wyrazić z pomocą operatora *U*.

Piśmiennictwo: *Ben-Ari M*. B.2.1., *Kazanecka A*. K.6.1., *Pnueli A.* P.3.1., *Prior A.* P.4.1., *Trzęsicki K.* T.3.1.

2.3.7. UWAGI O TEMPORALNOŚCI W INFORMATYCE

2.3.7.10. **Wyjaśnienie**. Poprawność działania systemu informatycznego zależy od poprawności programu i od poprawnej konstrukcji urządzeń informatycznych składających się na ten system. Do opisu oraz analizy działania systemów informatycznych stosowane są metody logiczne, w szczególności metody logiki temporalnej. We współczesnej informatyce logiki temporalne mają dwa rodzaje zastosowań:

1. W tzw. układach sekwencyjnych, na razie ograniczymy się do najprostszego przypadku układu sekwencyjnego dwustanowego (tzw. przerzutnika, który przyjmuje jeden z dwóch możliwych stanów „*set*" oraz „*unset*"), którego działanie zależy od czasu dyskretnego (synchronicznego lub asynchronicznego) reprezentowanego przez impulsy zegarowe (przypadek synchroniczny) lub zdarzenia (przypadek asynchroniczny). Opisane w 2.6.2, operatory binarne *S*, *U* oraz *O* – są przydatne do opisania działania układów sekwencyjnych. Obszernej o układach sekwencyjnych będzie mowa w części 4 (patrz podrozdział 4.2.2.).
2. W analizie poprawności i niezawodności działania procesów tworzonych przez system programów komputerowych. Temu przypadkowi przyjrzymy się obecnie bliżej.

2.3.7.20. **Wyjaśnienie**. W ogólnej metodologii nauk termin „*weryfikacja*" lub „*walidacja*" oznacza stwierdzenie poprawności. Termin „*falsyfikacja*" jest używany w znaczeniu: wykrycia błędu. W informatyce termin „*weryfikacja"* jest używany odnośnie zgodności projektu z wymaganiami, zaś „*walidacja*" odnosi do procesu określenia, czy program lub system programów - działa poprawnie czyli jest lub może być *zwalidowany*, czyli zgodnie z podaną specyfikacją; czy też działa błędnie, czyli niezgodnie ze specyfikacją, a tym samym nie może być *zwalidowany*. Oczywiście, że przy założeniu braku sprzeczności w specyfikacji danego programu lub systemu programów.

2.3.7.30. **Wyjaśnienie**. Dość powszechnie uważa się, że poprawność działania programu można zapewnić używając techniki testowania, na zgodność ze specyfikacją oprogramowania. Rzeczywistość wygląda jednak inaczej. Testowanie umożliwia jedynie stwierdzenie, czy uwzględnione w opracowaniu testu kombinacje występujących sytuacji, dają poprawne albo błędne działanie testowanego programu. Innymi słowy, jeśli jakaś kombinacja danych

wejściowych – przy zadanym stanie wewnętrznym programu, nie została sprawdzona w przeprowadzonym teście, to nic nie wiemy o tym, jak program zachowa się w przypadku wystąpienia takiej kombinacji danych. Testowanie dotyczy jedynie tych kombinacji danych oraz stanów, które zostały objęte zakresem testu. Jak z tego widać, na ogół technika testowania nie jest wystarczająca, aby stwierdzić poprawność działania, złożonych systemów programów z procesami współbieżnymi sterowanymi zdarzeniami zewnętrznymi i wewnętrznymi stanami tegoż systemu.

2.3.7.40. **Wyjaśnienie**. Znacznie doskonalszym podejściem od testowania dla zapewnienia poprawnego działania, jest dowiedzenie poprawności działania programu lub systemu programów, czyli *walidacja*. Posługując się terminologią logiki formalnej, dowiedzenie poprawności oznacza, że dla każdego możliwego zdarzenia: reaktywne - działanie systemu na to zdarzenie, jest tautologią. Realizację takiego zadania można powierzyć narzędziom oferowanym przez logikę temporalną. Operatory temporalne okazały się użyteczne do opisu działania systemów informatycznych. Struktura stanów (sekwencja lub drzewo) i zdarzeń zewnętrznych, są kluczowymi pojęciami, dzięki którym logika temporalna nadaje się zarówno do opisywania specyfikacji systemu informatycznego, jak i sprawdzania poprawności działania systemu informatycznego opracowanego na podstawie tej specyfikacji, czyli sprawdzenie czy niema miejsca falsyfikacja systemu.

2.3.7.50. **Wyjaśnienie**. Język logiki temporalnej adaptowany przez izraelskiego informatyka *Amira Pnueli*, dla potrzeb systemów informatycznych, spełnia trzy ważne warunki:

- Zdolność opisania wszystkich rodzajów specyfikacji niezależnie od języka programowania użytego w implementacji (*ekspresywność*);
- Sprawdzenie czy każdy przypadek działania implementowanego systemu - spełnia reguły określone w specyfikacji czyli jest tautologią (*złożoność*);
- Dzięki podobieństwu do języka naturalnego jest łatwy do nauczenia się (*pragmatyka*).

Szczegółowe wyłożenie wariantu logiki temporalne *Pnueli* – wykracza poza ramy niniejszej książki. Język logiki temporalnej może być zastosowany do specyfikacji szerokiego spektrum systemów informatycznych. W wypadku systemów reaktywnych, logika temporalna jest bardziej użyteczna niż logika *Floyd-Hoare'a*, która jest właściwym narzędziem dla programów sekwencyjnych, czyli typu „wejście-wyjście". Język logiki temporalnej tworzy ogólne lingwistyczne ramy dedukcyjne dla systemu stanów w taki sam sposób, jak logika formalna- dla systemów matematycznych.

Piśmiennictwo: *Ben-Ari M*. B.2.1., *Kazanecka A*. K.6.1., *Pnueli A.* P.3.1., *Prior A.* P.4.1., *Trzęsicki K.* T.3.1.

2.4. TRÓJWARTOŚCIOWE RACHUNKI ZDAŃ

2.4.0. UWAGI WSTĘPNE

Poznaliśmy już krótki zarys klasycznego dwuwartościowego rachunku zdań. Obecnie chcielibyśmy dać krótki zarys pewnego nie-dwuwartościowego rachunku zdań, mianowicie trójwartościowego rachunku zdań. Zaczniemy od, przyjęliśmy, że nie ma w żadnym języku zdań, które są zarazem prawdziwe i fałszywe. Ponadto logicy przyjmowali na ogół - choć nie wszyscy - że nie ma w żadnym języku zdań, które nie są ani prawdziwe ani fałszywe. Obie te zasady można łatwo połączyć w jedną, według której każde zdanie jest fałszywe albo prawdziwe (zasada dwu-wartościowości, zwana również prawem wyłączonego środka). Za nim pójdziemy dalej,

postaramy się te zasady sformułować przejrzyście, co można uczynić posługując się merytorycznie zinterpretowanym językiem dwuwartościowego rachunku zdań, wzbogaconym trzema funkcjami zdaniowymi, które tu wprowadzimy jako skróty uzupełniające języka polskiego.

2.4.0.00 $(\overline{Prp}\ x) =_{Df} x\ jest\ zdaniem,$

2.4.0.01 $(\overline{Fls}\ x) =_{Df} x\ jest\ zdaniem\ fałszywym,$

2.4.0.02 $(\overline{Ver}\ x) =_{Df} x\ jest\ zdaniem\ prawdziwym.$

Uwaga: Skrót „$\overline{Prp}$" pochodzi od łacińskiego wyrazu „*propositio*" (zdanie), skrót „$\overline{Fls}$" – od łacińskiego wyrazu „*falsum*" (fałsz), wreszcie „$\overline{Ver}$" – od łacińskiego wyrazu „*verum*" (prawda).

Możemy teraz treściwie i jasno sformułować wszystkie trzy zasady, o których była mowa wyżej.

2.4.0.10. **Zasada** - co najmniej dwuwartościowości:

$$\vdash \{(\overline{Prp}\ x) \rightarrow [(\overline{Fls}\ x) / (\overline{Ver}\ x)]\}.$$

2.4.0.11. **Zasada** - co najwyżej dwuwartościowości:

$$\vdash \{(\overline{Prp}\ x) \rightarrow [(\overline{Fls}\ x) \vee (\overline{Ver}\ x)]\}.$$

2.4.0.12. **Zasad**a dwuwartościowości:

$$\vdash \{(\overline{Prp}x) \rightarrow [(\overline{Flsx}) \dot{-} (\overline{Ver}x)]\}.$$

Przed każdą z trzech powyższych zasad (tj. 2.4.0.10, 2.4.0.11 i 2.4.0.12), postawiliśmy znak asercji, co było zresztą pewnego rodzaju lekkomyślnością, gdyż wprawdzie zasada 2.4.0.10, jak wspomagaliśmy wyżej, jest uznana przez ogół logików, jednak zasada 2.4.010, a co za tym idzie i zasada dwuwartościowości jest uznana jedynie przez większość logików przy istnieniu opozycji sięgającej wstecz aż po *Arystotelesa*.

Dwuwartościowy rachunek zdań jest zbudowany na zasadzie dwuwartościowości, czyli - na zasadzie co najmniej i co najwyżej dwuwartościowości. Ponieważ zaś, jak już wspominaliśmy, zasada co najwyżej dwuwartościowości ma swoich przeciwników, wysunięto postulat zbudowania innego rachunku zdań niż dwuwartościowy. Zadanie to postawił sobie i samodzielnie rozwiązał polski logik *Jan. Łukasiewicz* około roku 1920[27]. U podstaw stworzonego wówczas przez *Łukasiewicza* trójwartościowego rachunku zdań leży zasada trójwartościowości, wedle której każde zdanie jest albo *fałszywe*, albo *prawdziwe*, albo jakoś *obojętne*. W następnych swych pracach Łukasiewicz wykazał możliwość zbudowania n - wartościowego rachunku zdań (dla każdej dowolnej liczby naturalnej n, większej od liczby 1), jak również nieskończenie wielowartościowego rachunku zdań.

Przejdźmy teraz do następnej z interesujących nas spraw – wyjaśnijmy motywy, dla których pewni logicy odstąpili od zasady - co najwyżej dwuwartościowości i wypowiedzieli się przeciwko dwuwartościowemu rachunkowi zdań. Motywy te podamy w porządku chronologicznym. Rozpatrzmy wobec tego kolejno:

1) Zagadnienie niektórych zdań odnoszących się do przyszłości;
2) Zagadnienie wyrażeń modalnych;
3) Zagadnienie wzajemnie niszczących badań.

[27] Podobny wynik osiągnął niemal równocześnie logik amerykański E. Post; jednakże Post (w przeciwieństwie do Łukasiewicza) potraktował całe zagadnie w sposób formalistyczny, oderwany od rzeczywistości.

2.4.0.31. **Wyjaśnienie**. Niektórzy filozofowie i niektórzy logicy twierdzili, że istnieją takie zdania odnoszące się do przyszłości, które nie są ani prawdziwe, ani fałszywe. Inicjatorem tego stanowiska był zresztą Arystoteles. Ażeby obalić ten pogląd wystarczy jednak chyba zauważyć, że co innego wartość logiczna zdania, a co innego znajomość tej wartości. Zdanie odnoszące się do przyszłości, na przykład zdanie „za rok będę w Warszawie", ma swoją wartość logiczną i to tylko jedną z dwu, mianowicie *Prawdy* i *Fałszu*. Dziś wprawdzie nie wiem jeszcze, jaką wartość ma to zdanie, niemniej jednak jest ono fałszywe albo prawdziwe. Przy najlepszej woli trudno więc przyznać, aby istnienie zdań (czy odpowiadających im sądów), odnoszących się do przyszłości i takich zarazem, że aktualnie nie sposób odpowiedzieć na pytanie, czy dane zdanie jest fałszywe, czy prawdziwe - druzgotało zasadę dwuwartościowości!

2.4.0.32. **Wyjaśnienie**. Istnieją (np. w graficznym języku polskim) funktory zdaniotwórcze od jednego argumentu zdaniowego, zwane „*wyrażeniami modalnymi*". Są to funktory następujące:
(1) możliwe jest, że;
(2) niemożliwe jest, że;
(3) konieczne jest, że;
(4) niekonieczne jest, że.

Nie trudno, oczywiście, zbudować w języku polskim odpowiadające tym funktorom funkcje zdaniowe:
1) *możliwe jest, że p*;
2) *niemożliwe jest, ze p*;
3) *konieczne jest, że p*;
4) *niekonieczne jest, że p*.
Powyższe cztery funkcje zdaniowe również nazywa się „*wyrażeniami modalnymi*". Łukasiewicz starał się scharakteryzować wyrażenia modalne za pomocą tabliczek analogicznych do tabliczek zero-jedynkowej dla negacji, w sposób zgodny z potocznym rozumieniem wyrażeń modalny i z tymi tezami zawierającymi wyrażenia modalne, które pozostawili nam w spadku logicy starożytni i średniowieczni od Arystotelesa począwszy.

Wyniki badań Łukasiewicza wykazały, że na gruncie dwuwartościowego rachunku zdań tabliczek takich zbudować nie można, można, natomiast przyjmując istnienie trzech wartości logicznych zamiast dwu, tabliczki takie zbudować bez trudu. Wybierzmy sobie jakieś zdanie fałszywe i nazwijmy je „*Fałszem*", jakieś zdanie obojętne czy też „nijakie" (zakładając na chwilę, że choć jedno takie istnieje) i nazwijmy je „*Neutrum*", wreszcie wybieramy sobie jakieś zdanie prawdziwe i nazwijmy je „*Prawdą*" (oczywiście wszystkie trzy zdania mają być wybrane z tego samego języka, na przykład z polskiego języka graficznego). Po dokonaniu tego wyboru możemy scharakteryzować wyrażenia modalne za pomocą następujących tabel:

Tabela 2.4.0.20 (Możliwość)		Tabela 2.4.0.21 (Konieczność)		Tabela 2.4.0.22 (Niemożliwość)		Tabela 2.4.0.23 (Niekonieczność)	
p	Możliwe jest, że *p*	*p*	Konieczne jest, że *p*	*p*	Niemożliwe jest, że *p*	*p*	Niekonieczne jest, że *p*
I.	II.	I.	II.	I.	II.	I.	II.
Fałsz	Fałsz	Fałsz	Fałsz	Fałsz	Prawda	Fałsz	Prawda
Neutrum	Prawda	Neutrum	Fałsz	Neutrum	Fałsz	Neutrum	Prawda
Prawda	Prawda	Prawda	Prawda	Prawda	Fałsz	Prawda	Fałsz

Trzeba przyznać, że tym razem mamy poważniejsze powody do odrzucenia zasady dwu-wartościowości, niż wówczas, gdy była mowa o zdaniach odnoszących się do przyszłości. Co

prawda i tu można mieć pewne zastrzeżenia. Tadeusz Kotarbiński wykazał, że można zdefiniować wszystkie wyrażenia modalne nadając im rozumienie zgodne z tradycyjnym i nie odstępując od zasady dwuwartościowości, można to uczynić jednak nie w logice zdań, ale dopiero w obrębie logiki nazw. I tym razem trudno - więc orzec, że uzasadniono potrzebę odrzucenia zasady dwuwartościowości.

2.4.0.33. **Wyjaśnienie**. Omówmy wreszcie sprawę badań wzajemnie niszczących, pozornie nader odległą od naszej problematyki a zaczerpniętą z dziedziny praktyki codziennej, mianowicie obrotu towarowego. Gdy kupujący odbiera towar od sprzedawcy (np. gdy przedsiębiorstwo budowlane odbiera partię cegieł od przedsiębiorstwa wytwarzającego cegłę), wówczas sprawdza, czy towar spełnia przepisowe warunki, krótko mówiąc - czy jest „dobry".

Ażeby sprawdzić, czy partia towaru spełnia żądane warunki, kupujący posługuje się metodą próbkowania, to znaczy:
1) wylosowuje z partii pewną stosunkowo nieliczną zbiorowość sztuk,
2) poddaje każdą sztukę należącą do próbki badaniu (próbie).

Taki sposób postępowania uzasadnia się następująco: Badanie całej partii jest zbyt kosztowne (ewentualnie zbyt długotrwałe). Badanie próbki racjonalnie wylosowanej zapewnia praktyczną pewność oceny całej partii.

Dokonanie próby badania - niszczy nieraz badaną sztukę. W dalszym ciągu naszych rozważań, chcąc osiągnąć zwięzłość i jasność, ograniczymy się do sprawy próbkowej oceny partii cegieł oraz zainteresujemy się nieco bliżej tylko, dwojakim badaniem niszczącym, mianowicie:
1) badaniem na zgniecenie,
2) badaniem na wielokrotne zamrażanie.

W wyniku badania na zgniecenie badana sztuka cegły jest zawsze rozkruszona, a więc nie nadaje się - do badania na wielokrotne zamrażanie, którego trzeba dokonać na sztuce nieuszkodzonej. W wyniku badania na wielokrotne zamrażanie - zamarzająca woda zawszę uszkadza wewnętrzną strukturę badanej sztuki, a zatem sztuka cegły, która zastała zbadana na wielokrotne zamrażanie, nie nadaje się już do badania jej na zgniecenie.

Wyobraźmy sobie teraz, że mamy do czynienia z pewną niewielką zbiorowością (np. próbką) cegieł do zbadania. Z tej zbiorowości wyodrębnimy dwie sztuki, każdej z nich nadając nazwę jednostkową; nazwy te będą brzmiały następująco:

$$sztuka\ Nr\ 1,$$
$$sztuka\ Nr\ 2.$$

Poddajemy teraz:
1) $sztuk$ę Nr 1 badaniu na zgniecenie,
2) $sztuk$ę Nr 2 badaniu na wielokrotne zamrażanie.

Załóżmy dalej, że oba te badania przeprowadzono. Jest to może interesujące, że w naszych rozważaniach nie ma żadnego znaczenia, jaki był wynik któregokolwiek z obu badań: negatywny czy pozytywny. Istotne natomiast jest dla nas, że wyżej wspomniane badania zostały przeprowadzone. W konsekwencji tego założenia musimy przyjąć, że:
1) $sztuka\ Nr$ 1 nie nadaje się już do badania na wielokrotne zamrażanie;
2) $sztuka\ Nr$ 2 nie nadaje się już do badania na zgniecenie.

A oto dalsze konstatacje:

1) Nie istnieje żaden sposób stwierdzenia, czy *sztuka Nr* 1 przed poddaniem jej badaniu na zgniecenie miała przepisową odporność na wielokrotne zamrażanie.
2) Nie istnieje żaden sposób stwierdzenia, czy *sztuka Nr* 2 przed podaniem jej badaniu na wielokrotne zamrażanie miała przepisową odporność na zgniecenie.

Weźmy teraz pod uwagę dwa zdania następujące:
(1) *Sztuka Nr* 1 miała (przed poddaniem jej badaniu) przepisową odporność na wielokrotne zamrażanie.
(2) *Sztuka Nr* 2 miała (przed poddaniem jej badaniu) przepisową odporność na zgniecenie.

Postawmy sobie teraz dwa pytania:
Jaka jest wartość logiczna zdania (1)? Innymi słowy, czy (1) jest *fałszywe*, czy *prawdziwe*?
Jaka jest wartość logiczna zdania (2)? Innymi słowy, czy (2) jest *fałszywe*, czy *prawdziwe*?

Można by usłyszeć odpowiedź następującą: „Oprócz zdań fałszywych i prawdziwych istnieją jeszcze inne zdania, powiedzmy - obojętne czy też neutralne. Zdanie (1) oraz zdanie (2) to właśnie zdania neutralne. A więc oprócz dwu wartości logicznych, *Fałszu* i *Prawdy*, istnieje jeszcze trzecia wartość logiczna, nazwijmy ją <<*Neutrum*>>. Zasada dwuwartościowości jest pochodzenia w gruncie rzeczy doświadczalnego. Sformułowano ją i przyjęto w czasach, gdy nie znano pojęcia pary badań wzajemnie niszczących i przyjmowano ją nadal w czasach, gdy pojęcie to wprawdzie już znano, ale nie zwrócono uwagi na związane z tym pojęciem trudności natury logicznej. Obecnie należy odrzucić zasadę dwuwartościowości".

Czy zasada dwuwartościowości została już na tej podstawie definitywnie odrzucona? Raczej nie, ale wstrzymajmy się jeszcze z wyrokiem i udzielimy głosu zwolennikowi dwuwartościowości: „Nie wiemy, jaka jest wartość logiczna zdania (1), nie znamy i nie wyobrażamy sobie żadnej metody, w szczególności metody doświadczalnej, za której pomocą można by dowiedzieć się, jaka jest wartość logiczna zdania (1). Mimo to twierdzę, że albo zdanie (1) jest fałszywe, albo zdanie (1) jest prawdziwe, gdyż zasada dwuwartościowości jest powszechnie obowiązująca. To samo dotyczy zdania (2). Ponadto niemożność sprawdzenia danego zdania nie świadczy o jego wartości logicznej!"[28].

Czy sprawa sądowa, że tak żartobliwie powiemy, przeciwko zasadzie dwuwartościowości została przeprowadzona w sposób wyczerpujący? Stanowczo nie! Świadomie opuściliśmy (a to z braku należytej kompetencji) zreferowanie tych zarzutów przeciwko zasadzie dwu-wartościowości, które dotąd bez większego zresztą powodzenia) formułowano w oparciu o wyniki mikrofizyki.

Po tej dyskusji zajmijmy wreszcie jakieś własne stanowisko. Dałoby ono wyrazić mniej więcej w ten sposób: Istnieje dwuwartościowy rachunek zdań. Wieloletnie doświadczenie wykazuje, że rachunek ten jest wysoce przydatny, że jest dobrym narzędziem uściślania dowodów, jak również redagowania i upraszczania wielu tez. Nic nas nie zmusza do odrzucenia dwuwartościowego rachunku zdań; nauka stoi dziś, praktycznie sprawę ujmując, na gruncie dwuwartościowego rachunku zdań.

[28] Hugon Steinhaus zakomunikował, że zagadnienie prób wzajemnie niszczących ma, jego zdaniem, doniosłość aż filozoficzną. Ta uwaga Steinhausa skłoniła nas do zainteresowania się sprawą od strony logicznej. Wskazanie konkretnej pary prób wzajemnie niszczących (badanie cegły na zgniecenie i badanie na wielokrotne zamrażanie) zawdzięczamy Janowi Oderfeldowi oraz Klemensowi Wiśniewskiemu z Instytutu Matematycznego PAN.

No, dobrze, odpowie czytelnik, ale jeżeli tak sprawa się przedstawia, to po co - było w elementarnym wykładzie logiki dyskutować zasadę dwuwartościowości i jeszcze zapowiadać wykład rachunku trójwartościowego? Nasza odpowiedź będzie krótka i chyba jasna. Nie jest naszą intencją nawracać kogokolwiek na trójwartościową czy wielowartościową „wiarę". Natomiast było i jest naszą intencją „od-dogmatyzowanie" czytelnika, pod ważenie w nim przekonania, że logika to kodeks prawd a priori! Gdy jakaś teoria przez nader długi okres czasu okazuje się przydatnym narzędziem badania rzeczywistości materialnej, gdy przez długie lata teoria ta nie ulega ani razu zbawczym wstrząsom, wówczas przekonanie o prawdziwości każdej z jej tez zaczyna nabierać cech wiary w nienaruszalność, w poza doświadczalne - lub jeszcze gorzej – ponad doświadczalne pochodzenie tej teorii. Tak właśnie przedstawia się sprawa z logiką. I dlatego dobrze jest upowszechniać znajomość trójwartościowego rachunku zdań, chociaż sam przez się jest on narzędziem raczej mało dotychczas przydatnym!

Piśmiennictwo: *Greniewski H.* G.2.1. , *Kotarbiński T.* K.5.1., K.5.2., *Łoś J.* Ł.2.1., *Łukasiewicz J.* Ł.3.2., Ł.3.4., Ł.3.5., Ł.4.1., *Mostowski A.* M.5.1., *Słupecki J.* S.8.2., S.8.3.

2.4.1. ZDANIOWE LOGIKI TRÓJWARTOŚCIOWE

W pracy umieszczonej w Internecie zatytułowanej „*Logika dla informatyków*" - *Jerzy Tiuryn, Jerzy Tyszkiewicz* i *Paweł Urzyczyn* w poniższy sposób scharakteryzowali trójwartościowe logiki zdaniowe[29]:

„Pierwsze *logiki trójwartościowe* skonstruowali niezależnie od siebie polski logik Jan Łukasiewicz i amerykański (ale urodzony w Augustowie) logik i matematyk Emil Post. Motywacje Posta były raczej kombinatoryczne, natomiast Łukasiewicz swoja konstrukcję poparł głębokim wywodem filozoficznym. Argumentował miedzy innymi, że zdania o przyszłości typu „*jutro pójdę do kina*" nie są dzisiaj jeszcze ani prawdziwe ani fałszywe bo przypisanie im którejś z tych wartości zaprzeczałoby istnieniu wolnej woli. Aby logika mogła jakoś zdać sprawę ze statusu zdań o przyszłości, musi im przypisać inną trzecią wartość logiczną.

Zanim przejdziemy do części trochę bardziej formalnej, rozważmy jeszcze dwa przykłady, wzięte z żywej informatyki, gdzie także naturalnie pojawia się trzecia wartość logiczna.

2.4.1.10 Przykład. Rozważmy dwie deklaracje funkcji w Pascalu:

```
function
   f(x,y:boolean):boolean;
   begin
      .
   ..
   end;

function
   g(x,y:boolean):boolean;
   begin
      ...
end;

      a następnie ich użycie
```

[29] W cytowanym tekście zmieniono numerację przykładów i definicji oraz wprowadzono numerację tablic – zgodnie z zasadami zastosowanymi w niniejszej książce (mjg).

```
if f(x,y) and g(x,y) then ... else ...;
```

Wydaje się na pierwszy rzut oka, że to sytuacja rodem z logiki klasycznej, ale nie: przecież zmienne typu `boolean`: *f* oraz *g* - mogą dać w wyniku obliczenia wartości *true*, *false* lub się zapętlić, które to zdarzenie jest forma trzeciej wartości logicznej. Sposób, w jaki się z nią obejdzie funkcja `else` zależy od wyboru programisty: może on zastosować albo krótkie albo długie wyliczenie w swoim programie.

2.4.1.11. Przykład**.** Inna sytuacja to tabela stworzona za pomocą następującej instrukcji SQL, w relacyjnej bazie danych:

```
CREATE TABLE A (
id              INTEGER PRIMARY KEY
                          auto_increment, ...
valid           BOOLEAN, ...

);
```

Przy takiej deklaracji, tabela A będzie mogła w kolumnie `valid` zawierać *trzy* wartości logiczne: `TRUE`, `FALSE` i `NULL`, a logika trójwartościowa objawi swoje działanie przy wykonaniu np. zapytania:

```
SELECT *
FROM A AS A1, A AS A2
WHERE A1.valid and A2.valid
```

2.4.1.20. **Definicja.** *Zbiór formuł zdaniowej logiki trójwartościowej* to zbiór tych formuł zdaniowej logiki klasycznej (patrz rozdział 2.2), w których występują tylko spójniki, $\neg$,$\vee$ i $\wedge$.

Wywołane w ten sposób zawężenie składni zrekompensujemy niezwłocznie po stronie semantyki. Przez *trój-wartościowanie zdaniowe* rozumiemy dowolna funkcję ρ : ZZ $\rightarrow$ {0, ½, 1}, która zmiennym zdaniowym przypisuje wartości logiczne 0, ½ i 1. *Wartość formuły* zdaniowej α przy trój-wartościowaniu ρ oznaczamy przez $[[\alpha]]_{\%}$ i określamy przez indukcje:

2.4.1.21 $[[p]]_\varrho = \varrho(p)$, gdy p jest symbolem zdaniowym;

2.4.1.22 $[[\neg\alpha]]_\varrho = F_\neg([[\alpha]]_\varrho)$;

2.4.1.23 $[[\alpha \vee \beta]]_\varrho = F_\wedge([[\alpha]]_\varrho, [[\beta]]_\varrho)$;

2.4.1.24 $[[\alpha \wedge \beta]]_\varrho = F_\vee([[\alpha]]_\varrho, [[\beta]]_\varrho)$;

2.4.1.25 $[[\neg\alpha]]_\varrho = F_\neg([[\alpha]]_\varrho)$.

2.4.1.30. **Wyjaśnienie**. Różne wybory funkcji: $F_\vee$, $F_\wedge$: {0, ½, 1} × {0, ½, 1} → {0, ½, 1} oraz $F_\neg$: {0, ½, 1} → {0, ½, 1} prowadzą do różnych logik trójwartościowych.

Zaczniemy od logiki najstarszej, zwanej dziś logika *Heytinga-Kleene-Łukasiewicza* (patrz tab. 2.4.1.31):

Tabela 2.3.1.21

$F_\wedge(x, y)$				$F_\vee(x, y)$				$F_\neg$	
x\y	0	½	1	x\y	0	½	1	x	¬x
0	0	0	0	0	0	½	1	0	1
½	0	½	½	½	½	½	1	½	½
1	0	½	1	1	1	1	1	1	0

Jest to logika niewątpliwie nadająca się do rozwiązania zadania, które sobie *Łukasiewicz* postawił. Sposób traktowania wartości logicznej ½ jest taki, że należy ją rozumieć jako, ”jeszcze nie wiadomo”.

Warto zauważyć, że w przypadku tej logiki zachodzą równości:

2.4.1.32 $[[\neg\alpha]]_\varrho = 1 - [[\alpha]]_\varrho$,

2.4.1.33 $[[\alpha \vee \beta]]_\varrho = \max\{[[\alpha]]_\varrho, [[\beta]]_\varrho\}$,

2.4.1.34 $[[\alpha \wedge \beta]]_\varrho = \min\{[[\alpha]]_\varrho, [[\beta]]_\varrho\}$,

znane z 2.2, tak wiec można ją traktować jako literalne uogólnienie logiki klasycznej.

Zachowanie stałych i operacji logicznych w języku SQL rządzi się właśnie prawami logiki *Heytinga-Kleene-Łukasiewicza*.

Zupełnie inną logikę zaproponował *Bochvar* (patrz tab. 2.4.1.41).

Tabela 2.4.1.41

$F_\wedge(x, y)$				$F_\vee(x, y)$				$F_\neg$	
x\y	0	½	1	x\y	0	½	1	x	¬x
0	0	½	0	0	0	½	1	0	1
½	0	½	½	½	½	½	1	½	½
1	0	½	1	1	1	½	1	1	0

Czytelnik bez trudu rozpozna, że jest logika właściwa dla przykładu 2.4.1.10, gdy programista wybierze długie wyliczenie wyrażeń logicznych. W sensie tej logiki stała ½ oznacza awarię lub błąd. Dalej mamy dość egzotycznie wyglądająca logikę *Sobocińskiego* (patrz tab. 2.4.1.42)[30].

Tabela 2.4.1.42

$F_\wedge(x,y)$				$F_\vee(x,y)$				$F_\neg$	
x\y	0	½	1	x\y	0	½	1	x	¬x
0	0	0	0	0	0	0	1	0	1
½	0	½	1	½	0	½	1	½	½
1	0	1	1	1	1	½	1	1	0

Jednak i ona ma swój poważny sens. W niej stała ½ logiczna oznacza „nie dotyczy” lub „nieistotne”. Wszyscy odruchowo wręcz stosujemy tę logikę przy okazji wypełniania różnych, formularzy i kwestionariuszy. Odpowiadając na różne pytania sformułowane „tak lub nie” w niektórych polach na odpowiedzi umieszczamy „nie dotyczy” a potem podpisujemy się pod dokumentem mimo ostrzeżenia „*Świadomy/ma odpowiedzialności karnej za składanie fałszywych zeznań ... oświadczam, że wszystkie odpowiedzi w tym formularzu są zgodne ze, stanem faktycznym.*” Po prostu stosujemy tu logikę Sobocińskiego, w której koniunkcja kilku wyrazów o wartości 1 i kilku wyrazów o wartości ½ daje wynik 1. Na szczęście, organy kontrolne chyba też znają ten rachunek zdań i stosuj, a go do oceny naszych zeznań ...

[30] Logika Sobocińskiego jest powszechnie stosowana przy uproszczaniu tablic decyzyjnych (patrz rozdział 4.3.) oraz pozwala na pokazanie formalnego związku pomiędzy klasycznym dwuwartościowym rachunkiem zdań a rachunkami trój-wartościowymi (uwaga moja mjg).

Przechodząc do logik wyglądających na pierwszy rzut oka jeszcze niezwykłej, natrafiamy na logikę z nieprzemienną koniunkcją i alternatywą, która opisuje spójniki logiczne w Pascalu,, wyliczane w sposób krótki (patrz tab. 2.4.1.43).

Tabela 2.4.1.43

$F_\wedge(x,y)$				$F_\vee(x,y)$				$F_\neg$	
x\y	0	½	1	x\y	0	½	1	x	¬x
0	0	0	0	0	0	0	1	0	1
½	½	½	½	½	½	½	½	½	½
1	0	½	1	1	1	1	1	1	0

Dla każdego z powyższych rachunków logicznych zasadne i interesujące są pytania o to czym jest tautologia, o aksjomatyzacje i systemy dowodzenia. Tak samo jest z innymi logikami wielowartościowymi, bo Czytelnik już zapewne zauważył, że o ile jest jedna sensowna logika dwuwartościowa i kilka, wzajemnie konkurencyjnych sensownych logik trójwartościowych, to przy wzroście liczby wartości logicznych, liczba sensownych logik też musi rosnąć. Tytułem przykładu: można sobie bez trudu wyobrazić logikę w której chcielibyśmy mieć jednocześnie dwie różne stałe odpowiadające „nie wiadomo" i „nie dotyczy". Taka logika miałaby wiec co, najmniej cztery wartości logiczne. Jak łatwo się domyślić, ogromnym obszarem zastosowań logik wielowartościowych jest sztuczna inteligencja i reprezentacja wiedzy…"

Piśmiennictwo: *Tiuryn J.* T.5.1.

2.6.2. PODSTAWY JĘZYKA SFORMALIZOWANEGO WG ŁUKASIEWICZA

Zaczynamy od listy wyrażeń pierwotnych:

2.4.2.00. **Wyrażenia pierwotne** (stałe zdaniowe). Każde z trzech następujących wyrażeń jest zdaniem należącym do języka budowanego:

0, ½, 1[31]

2.4.2.01. **Wyrażenia pierwotne** (zmienne zdaniowe) [32]. Każda zmienna zdaniowa:

$$p, q, r, \ldots$$

należy do języka budowanego.

2.4.2.02. **Wyrażenia pierwotne** (funkcje zdaniowe). Każde z wyrażeń poniższych jest funkcją zdaniową zmiennej zdaniowej lub zmiennych zdaniowych, należącą do języka budowanego:

$$(\neg p),\ p^{-},\ p^{*},\ p^{x},\ p^{+},$$
$$(p \vee q), (p \rightarrow q), (p \leftrightarrow q), (p \wedge q).$$

Lista wyrażeń pierwotnych zostaje zamknięta. Przejdziemy teraz do dyrektyw budowania języka.

[31] Jak pokażemy dalej (rozdział 2.3), używany jest również zestaw trzech innych symboli, na oznaczenie stałych zdaniowych trójwartościowego rachunku zdań. Są to np. zbiór symboli: {F, -, T}.

[32] Dla uproszczenia będziemy pisali „stała zdaniowa'', „zdanie", „funkcja zdaniowa'', zamiast pisać ściśle, lecz rozwlekle „jakoby-stała zdaniowa'', „jakoby-zdanie'', „jakoby-funkcja zdaniowa".

Tablica 2.4.2.20		
Wyrażenie interpretowane	Nazwa wyrażenia interpretowanego	Wyrażenie interpretujące
I.	II.	III.
0	Fałsz	Warszawa nie leży nad Wisłą
½	Neutrum	(Zdanie (1) w paragrafie 2.2.0 ?)
1	Prawda	Warszawa leży nad Wisłą
¬p	Negacja zdaniowa	Fałszem jest, że p
$-p^-$	Niemożliwość	Niemożliwe jest, że p
$*p^*$	Możliwość	Przynajmniej możliwe jest, że p W skrócie: możliwe jest, że p
$\times p^x$	Czysta możliwość	Tylko możliwe jest, że p
$+p^+$	Konieczność	Konieczne jest, że p
(p ∨ q)	Alternatywa	p lub q
(p → q)	Implikacja	Jeżeli p, to q
(p ↔ q)	Równoważność	p wtedy i tylko wtedy, jeżeli q
(p ∧ q)	Koniunkcja	p oraz q

2.4.2.10. **Dyrektywy językowe** dwuwartościowego rachunku zdań (tj. dyrektywy: 2.4.2.10, 2.4.2.11, 2.4.2.12, 2.4.2.13 oraz 2.4.2.14), obowiązują bez wyjątku dla języka budowanego. Przyjmujemy wprawdzie te same dyrektywy budowy języka, co dla dwuwartościowego rachunku zdań, ale w wyniku stosowania tych dyrektyw otrzymamy inny język sformalizowany niż język dwuwartościowego rachunku zdań. Nie ma zresztą w tej okoliczności nic tajemniczego; stosując ten sam zespół dyrektyw do innego zespołu wyrażeń pierwotnych budujemy inny niż poprzednio język sformalizowany.

Zajmiemy się teraz interpretacją merytoryczną naszego systemu (będzie to zresztą nieco „kulawa" interpretacja, gdyż nie sposób dziś dać, jak wiemy, dobrego przykładu na zdanie ani prawdzie, ani fałszywe. Interpretację tę podajemy w tablicy 2.4.2.20.

Piśmiennictwo: *Greniewski H.* G.2.1. , *Łukasiewicz* J., Ł.3.4., *Słupecki J.* S.8.2., S.8.3.

2.4.3. PODSTAWY TEORII SFORMALIZOWANEJ WG ŁUKASIEWICZA

Budowana przez nas teoria sformalizowana nie zawiera ani jednej tezy pierwotnej. Podamy tu dwie dyrektywy budowania twierdzeń, poprzedzając je wyjaśnieniami i tabliczkami pomocniczymi.

2.4.3.00. **Wyjaśnienie**. Przez „zdanie molekularne" rozumiemy każde takie i tylko takie wyrażenie, które powstaje przez podstawienie
- do funkcji zdaniowej, dowolnej byle wymienionej ,w tablicy 2.6.3.20
- za wszystkie zmienne zdaniowe
- jednej, dwu czy też trzech stałych zdaniowych.

Przykłady. Zdania molekularne: ¬0, ¬½, 0^-, $½^-$, 1^-, 0*, ½*, 1*, 0^+, $½^+$, (0 ∨ 0), (0 ∨ ½), (0 ∨ 1)

2.4.3.01 **Wyjaśnienie**. Przez „tabliczkę zero – połówkowo - jedynkową" rozumiemy każdą z niżej podanych tabliczek (od 2.4.3.10 do 2.4.3.18 - włącznie). Żadnych innych tabliczek tak nie nazywamy.

Tabliczka 2.4.3.10		Tabliczka 2.4.3.11		Tabliczka 2.4.3.12		Tabliczka 2.4.3.13		Tabliczka 2.6.4.14	
(Negacja zdaniowa)		(Możliwość)		(Czysta możliwość)		(Niemożliwość)		(Konieczność)	
p	¬p	p	p*	p	P^x	p	p^-	p	p^+
I	II	I	II	I	II	I	II	I	II
0	1	0	0	0	0	0	1	0	0
½	½	½	1	½	1	½	0	½	0
1	0	1	1	1	½	1	0	1	1

Tabliczka 2.4.3.15		Tabliczka 2.4.3.16		Tabliczka 2.4.3.17		Tabliczka 2.4.3.18	
(Alternatywa)		(Implikacja)		(Równoważność)		(Koniunkcja)	
p q	(p∨q)	p q	(p → q)	p q	(p ↔ q)	p q	(p∧q)
I	II	I	II	I	II	I	II
0 0	0	0 0	1	0 0	1	0 0	0
0 ½	½	0 ½	1	0 ½	½	0 ½	0
0 1	1	0 1	1	0 1	0	0 1	0
½ 0	½	½ 0	½	½ 0	½	½ 0	0
½ ½	½	½ ½	1	½ ½	1	½ ½	½
½ 1	1	½ 1	1	½ 1	½	½ 1	½
1 0	1	1 0	0	1 0	0	1 0	0
1 ½	1	1 ½	½	1 ½	½	1 ½	½
1 1	1	1 1	1	1 1	1	1 1	1

2.4.3.20. **Wyjaśnienie**. Przez „wartość zdania molekularnego" rozumiemy *Fałsz* (0), *Neutrum* (½) lub *Prawdę* (1) w zależności od tego, która z tych stałych jest wyznaczona przez odpowiednią tabliczkę zero - połówkowo - jedynkową i odpowiedni wiersz tej tabliczki.
Przykłady. Wartością zdania molekularnego „0^-" jest „1" (tabliczka 2.4.3.13, wiersz pierwszy od góry). Wartością zdania molekularnego „ $½^+$" jest „0" (tabliczka 2.4.3.14, wiersz drugi od góry). Wartością zdania molekularnego „(½ ↔ 1)" jest „½" (tabliczka 2.4.3.17, wiersz szósty od góry).

2.4.3.21 **Wyjaśnienie**. Przez „uproszczenie jednorazowe" wyrażenia należącego do języka budowanego systemu rozumiemy postępowanie niżej opisane:
1) bierzemy pod uwagę wyrażenie należące do języka budowanego systemu;
2) znajdujemy wszystkie zdania molekularne zawarte w wyrażeniu podanym w punkcie 1);
3) każde z tych zdań molekularnych zastępujemy w wyrażeniu wymienionym w punkcie 1) wartością odnośnego zdania.

Przykład. Upraszczamy jednorazowo wyrażenie następujące:

$$\{[(0 \to ½) \leftrightarrow (½ \vee 1)] \leftrightarrow (½ \wedge ½)\}.$$
$$[(1 \leftrightarrow 1) \leftrightarrow (½ \wedge 1)].$$

2.4.3.22. **Wyjaśnienie**. Przez „uproszczenie ostateczne" wyrażenia należącego do języka budowanego systemu rozumiemy postępowanie niżej opisane:
1) bierzemy pod uwagę wyrażenie należące do języka budowanego systemu;
2) stosujemy do tego wyrażenia czynność uproszczenia jednorazowego tyle razy, ile to jest możliwe.

Przykład. Bierzemy pod uwagę wyrażenie:

$$\{[(½ \vee 1) \leftrightarrow (½ \vee 1)] \leftrightarrow (½^x \leftrightarrow ½^*)\}.$$

Uproszczenie przebiega następująco:

$$[(1 \leftrightarrow 1) \leftrightarrow (1 \leftrightarrow 1)].$$
$$(1 \leftrightarrow 1).$$
$$1.$$

Możemy już teraz – po zaznajomieniu się z wyjaśnieniami i przerobieniu przykładów - przystąpić do sformułowania dyrektyw wnioskowania:

2.4.3.30. **Dyrektywa**.
1) Bierzemy pod uwagę funkcję zdaniową należącą do języka budowanego systemu;
2) dokonujemy kolejno wszystkich możliwych podstawień mających postać:
   - do funkcji zdaniowej wymienionej w punkcie 1)
   - za wszystkie zmienne zdaniowe

- podstawiamy stałą zdaniową ,czy też stałe zdaniowe;

3) wynik każdego z podstawień opisanych w punkcie 2) upraszczamy ostatecznie;
4) uznajemy wyrażenie wymienione w punkcie 1) za twierdzenie budowanej teorii, gdy każde z ostatecznych uproszczeń, o których mowa w punkcie 3), jest równokształtne z „0” lub z „½”, wyrażenie wymienione w punkcie 1) nie jest twierdzeniem budowanej teorii.

2.4.3.3.1. **Dyrektywa**.

1) Bierzemy pod uwagę funkcję zdaniową należącą do języka budowanego systemu;
2) w wyrażeniu wymienionym w punkcie 1) zastępujemy każde definiendum definiensem;
3) gdy wyrażenie otrzymane wedle punktu 2) jest twierdzeniem budowanej teorii, wówczas wyrażenie wymienione w punkcie 1) jest twierdzeniem budowanej teorii.

Piśmiennictwo: Greniewski H. G.2.1.

2.4.4. Przyczynek do meta teorii trójwartościowego rachunku zdań

O ile w dwuwartościowym rachunku zdań mieliśmy łącznie cztery różne funkcje jednej zmiennej (patrz podrozdział 2.2.3), to liczba funkcji jednej zmiennej zdaniowej w przypadku rachunku trójwartościowego jest znacznie większa. Dla rozwiązania tego zagadnienia (z resztą łatwego) zbudowaliśmy pewną tablice, które żartobliwie nazywamy „tablicą Mendelejewa". Na początku ograniczymy się do funkcji jednej zmiennej. Pytamy, ile można zbudować (za pomocą metody zero - połówkowo - jedynkowej) nie równoważnych między sobą funkcji zdaniowych jednej zmiennej zdaniowej. Aby na to pytanie dać odpowiedź, budujemy specjalną tablicę 2.4.3.00. Jak wynika z tej tablicy, liczba funkcji zdaniowych jednej zmiennej zdaniowej dających się określić metodą zero - połówkowo - jedynkową wynosi tym razem aż dwadzieścia siedem. Nasza nowa „tablica Mendelejewa” ujawnia tym razem dwadzieścia dwa wolne miejsca przy pięciu już obsadzonych przez wprowadzone uprzednio funkcje.

Tablica 2.4.4.00

	Funkcje zdaniowe jednej zmiennej zdaniowej								
p			p^-			¬p			
I.	II.	III.	IV.	V.	VI.	VII.	VIII.	IX.	X.
0	0	½	1	0	½	1	0	½	1
½	0	0	0	½	½	½	1	1	1
1	0	0	0	0	0	0	0	0	0

	Funkcje zdaniowe jednej zmiennej zdaniowej								
p							p^x		
I.	XI.	XII.	XIII.	XIV.	XV.	XVI.	XVII.	XVIII.	XIX.
0	0	½	1	0	½	1	0	½	1
½	0	0	0	½	½	½	1	1	1
1	½	½	½	½	½	½	½	½	½

	Funkcje zdaniowe jednej zmiennej zdaniowej								
p	p^+						p*		
I.	XX.	XXI.	XXII.	XXIII.	XXIV.	XXV.	XXVI.	XXVII.	XXVIII.
0	1	0	½	1	0	½	1	0	½
½	1	0	0	0	½	½	½	1	1
1	½	1	1	1	1	1	1	1	1

Cztery z tych funkcji są dla nas interesujące, i nimi się będziemy dalej interesować.

Piśmiennictwo: *Greniewski H.* G.2.1. , *Łukasiewicz J.* Ł.3.4.

2.4.5. TAUTOLOGIE JEDNEJ ZMIENNEJ TRÓJWARTOŚCIOWEGO RACHUNKU ZDAŃ

Podobnie jak wówczas, gdy mieliśmy do czynienia z dwuwartościowym rachunkiem zdań, rozpoczniemy od twierdzeń zawierających jedną zmienną.

2.4.5.01. *Bezwzględna zwrotność implikacji* (w terminologii klasycznej „zasada tożsamości"):

$$\vdash (p \rightarrow p).$$

2.4.5.02 *Bezwzględna zwrotność równoważności*:

$$\vdash (p \rightarrow p).$$

Stwierdziliśmy wyżej, że zasada tożsamości jest twierdzeniem nie tylko dwuwartościowego, ale i trójwartościowego rachunku zdań. To samo dotyczy bezwzględnej zwrotności równoważności. Przekonamy się teraz, że istnieją jednak takie (i to bardzo proste) twierdzenia dwuwartościowego rachunku zdań, które nie są twierdzeniami trójwartościowego rachunku zdań. 'Mamy tu na myśli funkcje zdaniowe (1) i (2) (zasada włączonego środka i zasada sprzeczności).

2.4.5.11. *Iteracja negacji* (czyli prawo podwójnej negacji): $\vdash \{p \leftrightarrow [\neg(\neg p)]\}$.

2.4.5.12. *Iteracja możliwości*: $\vdash (p^* \leftrightarrow p^{**})$.

2.4.5.13. *Iteracja możliwości czystej*: $\vdash (p \leftrightarrow p^{xx})$.

2.4.5.14. *Iteracja konieczności*: $\vdash (p^{+} \leftrightarrow p^{++})$.

Zestawiając ze sobą twierdzenia 2.4.5.11, 2.4.5.12, 2.4.5.13 oraz 2.4.5.14, łatwo możemy zauważyć, że:

1) negacja i czysta możliwość mają własność polegającą na tym, że dwukrotne ich zastosowanie daje ten sam wynik, co ich niestosowanie,
2) możliwość i konieczność mają tę własność, że dwukrotne ich zastosowanie daje ten sam wynik, co zastosowanie jednokrotne.
3) Przypomnijmy sobie teraz kilka twierdzeń znanych nam już z dwuwartościowego rachunku zdań i przekonajmy się, że w rachunku trójwartościowym zachowują one nadal „moc obowiązującą".

2.4.5.21. $\vdash [p \leftrightarrow (p \vee 0)]$.

2.4.5.22. $\vdash [p \leftrightarrow (p \wedge 1)]$.

2.4.5.23. $\vdash [p \leftrightarrow (p \leftrightarrow 1)]$.

2.4.5.24. $\vdash [p \leftrightarrow (p \vee p)]$.

2.4.5.25. $\vdash [p \leftrightarrow (p \wedge p)]$.

2.6.5.33. *Twierdzenie Tarskiego*: $\vdash \{p^* \leftrightarrow [(\neg p) \rightarrow p]\}$.

2.4.5.34 $\vdash (p^* \leftrightarrow p^{--})$.

Jak widać z twierdzenia 2.4.4.34, dwukrotne zastosowanie niemożliwości - daje ten sam wynik, co jednokrotne zastosowanie możliwości.

2.4.5.35 $\vdash [p^* \leftrightarrow (\neg p)^{-}]$.

Podamy jeden tylko sposób rugowania niemożliwości:

2.4.5.36 $\vdash [p^{-} \leftrightarrow \neg(p^{*})]$.

Wreszcie podamy dwa sposoby rugowania konieczności z języka rachunku trójwartościowego.

2.4.5.37 $\vdash \{p^{+} \leftrightarrow \{\neg[p \rightarrow (\neg p)]\}\}$.

2.4.5.38 $\vdash [p^{+} \leftrightarrow (\neg p)^{-}]$.

Zajmiemy się teraz formami rozumowania ważnymi w obrębie trójwartościowego rachunku zdań, a zawierającymi jedną zmienną.

2.4.5.41 $\vdash (p^{+} \rightarrow p)$.

Twierdzenie to jest odpowiednikiem formuły zaczerpniętej ze średniowiecznej logiki modalne: *ab oportere ad essevaletcon sequentia*.

2.4.5.42 $\vdash (p \rightarrow p^{*})$.

Twierdzenie to jest, odpowiednikiem formuły: *ab esse ad, possevalet consequentia*.

2.4.5.43. $\vdash (p^{+} \rightarrow p^{*})$.

2.4.5.44. $\vdash [p^{-} \rightarrow (\neg p)]$.

2.6.5.45. $\vdash [p^{-} \rightarrow (\neg p)^{*}]$.

2.4.5.51. $\vdash [p \rightarrow (p \rightarrow p^{+})]$.

Twierdzenie to stanowi, zdaniem *Łukasiewicza*, odpowiednik formuły podanej przez Leibniza, a wywodzącej się od *Arystotelesa*: *unumquodque quando est, oportet esse*.

2.4.5.52. $\vdash \{(\neg p) \rightarrow [(\neg p) \rightarrow p^{-})]\}$.

2.4.5.61. $\vdash \{p \rightarrow \{p \rightarrow [\neg(\neg p)^{x}]\}\}$.

2.4.5.62. $\vdash \{(\neg p) \rightarrow [(\neg p) \rightarrow (\neg p)^{x}]\}$.

2.4.5.63. $\vdash \{p^{x} \rightarrow [p^{x} \rightarrow (\neg p)^{x}]\}$.

2.4.5.64. $\vdash \{(\neg p)^{x} \rightarrow [(\neg p)^{x} \rightarrow p^{x}]\}$.

2.4.5.65. $\vdash \{(\neg p^{x}) \rightarrow [\neg(p^{x}) \rightarrow (\neg p)]\}$.

2.4.5.66. $\vdash \{\neg[(\neg p)x] \rightarrow \neg\{[(\neg p)x] \rightarrow p\}\}$.

Zwróćmy na zakończenie uwagę na dwie klasyczne formy wnioskowania. Każda z nich spełnia wszystkie trzy warunki następujące:

1) zawiera jedną zmienną zdaniową;
2) jest ważna w obrębie dwuwartościowego rachunku zdań;
3) jest nieważna w obrębie trójwartościowego rachunku zdań.

Piśmiennictwo: *Greniewski H.* G.2.1. , *Łukasiewicz J.* Ł.3.4.

2.4.6. Tautologie dwu zmiennych trójwartościowego rachunku zdań

Z kolei przyjrzymy się tautologiom dwu zmiennych trójwartościowych wg Łukasiewicza:

2.4.6.00. *Symetria alternatywy*: $\vdash [(p \vee q) \leftrightarrow (q \vee p)]$.

2.4.6.10. **Wyjaśnienie**. Niech F i G będą funkcjami zdaniowymi dwu zmiennych, niech ponadto H będzie funkcją zdaniową jednej zmiennej zdaniowej; zamiast funkcji F będziemy pisali schemat „$(p\Delta q)$”, zamiast funkcji G będziemy pisali schemat „$(p\nabla q)$”, wreszcie zamiast funkcji H będziemy pisali schemat „$p^{\&}$” albo „$\neg p$” (w przypadku negacji). Będziemy nazywali „prawem De Morgana dla funkcji F i G ze względu na H” - twierdzenie mające postać:

$$[(p^{\&}\Delta q^{\&}) \leftrightarrow (p\nabla q)^{\&}].$$

2.4.6.11. *Prawo De Morgana dla alternatywy i koniunkcji ze względu na negację*:

$$\vdash \{[(\neg p) \vee (\neg q)] \leftrightarrow \neg(p \wedge q)\}$$

2.4.6.12. *Prawo De Morgana dla koniunkcji i alternatywy ze względu na negację*:

$$\vdash \{[(\neg p) \wedge (\neg q)] \leftrightarrow \neg(p \vee q)\}$$ (twierdzenie 2.2.5.22).

2.4.6.13. *Prawo De Morgana dla alternatywy i koniunkcji ze względu na niemożliwość*:

$$\vdash [(p^{-} \vee q^{-}) \leftrightarrow (p \wedge q)^{-}].$$

2.4.6.14. *Prawo De Morgana dla koniunkcji i alternatywy ze względu na niemożliwość*:

$$\vdash [(p^{-} \wedge q^{-}) \leftrightarrow (p \vee q)^{-}].$$

2.4.6.20. **Wyjaśnienie**. Przez „rozdzielność F względem G" rozumiemy prawo De Morgana dla G i G ze względu na F. Należy starannie odróżniać od siebie trzy różne postaci twierdzeń:

1) lewostronną rozdzielność funkcji dwu zmiennych względem funkcji dwu zmiennych (wyjaśnienie 2.4.6.10),
2) prawostronną rozdzielność funkcji dwu zmiennych względem funkcji dwu zmiennych (wyjaśnienie 2.4.6.11),
3) rozdzielność funkcji jednej zmiennej względem funkcji dwu zmiennych (wyjaśnienie 2.2.5.20).

2.4.6.21 *Rozdzielność konieczności względem alternatywy*:

$$\vdash [(p^{+} \vee q^{+}) \leftrightarrow (p \vee q)^{+}].$$

2.4.6.22. *Rozdzielność konieczności względem koniunkcji*:

$$\vdash [(p^{+} \wedge q^{+}) \leftrightarrow (p \wedge q)^{+}].$$

Na tym kończymy nasz krótki zarys trójwartościowego rachunku zdań.

Piśmiennictwo: Greniewski H.G.2.1. , Łukasiewicz, Ł.3.7.

2.5. Algebra Boole'a oraz algebra Kleene'go

2.5.1. Wprowadzenie do Algebry Boole'a

Rozpoczniemy od krótkiego przeglądu algebry Boole'a, stanowiącej matematyczną podstawę cyfrowych układów logicznych, dokładniej mówiąc tzw. układów kombinacyjnych (patrz podrozdział 4.2.2). Tę algebrę - nazwano tak dla uczczenia brytyjskiego matematyka *George'a Boole'a*, który to zaproponował podstawowe zasady algebry w traktacie zatytułowanym *An Investigation of the Laws of Thought on Which to Found the Mathemathical Theories of Logic and Probabilities* (*Badanie praw myśli, które mogą być podstawą matematycznych teorii logiki i*

prawdopodobieństwa). W roku 1938 *Claude Shannon*, asystent na wydziale elektrycznym MIT, zasugerował zastosowanie algebry Boole'a do rozwiązywania problemów projektowania układów przekaźnikowych. Metoda *Shannona* została następnie użyte do analizowania i projektowania elektronicznych układów cyfrowych.

Algebra Boole'a jest wygodnym narzędziem w dwóch obszarach:

1. analizy - jest ekonomicznym sposobem opisywania działania kombinacyjnych układów cyfrowych;
2. projektowania - algebra Boole'a może być stosowana do uproszczonej realizacji pożądanych funkcji kombinacyjnych.

Podobnie jak inne rodzaje algebry, algebra Boole'a używa zmiennych i operacji. W tym przypadku są to logiczne zmienne i operacje. Wobec tego zmienna oznaczane literami lub ciągami liter (nazwami) mogą przyjmować wartość 1 (*Prawda*) lub 0 (*Fałsz*). Podstawowymi operacjami logicznymi są: AND (i), OR (lub) i NOT (nie); reprezentowane symbolicznie przez ∧ , ∨ oraz ¬. A oto tabele: 2.7.1.11 – 2.7.1.13 definiujące wzmiankowane operacje na zmiennych zero – jedynkowych:

Tabela 2.5.1.11.	
a b	a∧b
0 0	0
0 1	0
1 0	0
1 1	1

Tabela 2.5.1.12.	
a b	a∨b
0 0	0
0 1	1
1 0	1
1 1	1

Tabela 2.5.1.13.	
a	¬a
0	1
1	0

Jak widać z powyższych tabel operacja AND daje w wyniku wartość *Prawda* (wartość binarną 1) wtedy i tylko wtedy, gdy jej obydwa argumenty są prawdziwe (mają wartość binarną 1); operacja OR daje wyniku wartość *Prawda* (wartość binarną 1), gdy którykolwiek z jej argumentów lub obydwa mają wartość *Prawda* (wartość binarną 1); operacja jednoargumentowa NOT zwraca wartość swojego argumentu równą *Prawda* (wartość binarną 1) – wtedy i tylko wtedy gdy argument ma wartość *Fałsz* (wartość binarną 0).

Oto uwaga dotyczących notacji, przy braku nawiasów operacja AND ma pierwszeństwo przed operacją OR.

2.5.1.20. Tezy pierwotne - postulaty Algebry Booler'a:

1.	$0 \neq 1$	$1 \neq 0$	
2.	$a \wedge b = b \wedge a$	$a \vee b = b \vee a$	prawa przemienności;
3.	$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$	$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$	prawa rozdzielczości;
4.	$1 \wedge a = a$	$0 \vee a = a$	prawa tożsamości;
5.	$a \wedge \neg a = 0$	$a \vee \neg a = 1$	prawa odwrotności.

Biorąc teraz pod uwagę - symetrię powyższych postulatów łatwo zauważyć, że wszystkie tezy wtórne algebry Boole'a dadzą się ustawić w pary w taki sposób, że zawsze wystarczy zbudować dowód dla jednej tylko z obu tez. Tworzenie par polega na tym, że po zapisaniu danej tezy otrzymujemy drugą, stosując do pierwszej (patrz tabela 2.5.1.30) „słownik".

Oczywiście może się zdarzyć, że obie tezy utworzonej w ten sposób pary będą równokształtne, na przykład:

$$\vdash \{a = [\neg(\neg a)]\}; \qquad \vdash \{a = [\neg(\neg a)]\}.$$

Tabela 2.5.1.30	
I.	II.
0	1
1	0
$(\neg a)$	$(\neg a)$
$(a \wedge b)$	$(a \vee b)$
$(a \vee b)$	$(a \wedge b)$

Wyżej opisaną własność algebry Boole'a nazywa się dwoistością tej algebry.

2.5.1.40. **Wyjaśnienie**. Niech F, G będą funkcjami boole'owskimi, każda dwu zmiennych boole'owskich; zamiast pierwszej z nich będziemy pisali schemat „$(a \nabla b)$", zamiast drugiej - „$(a \Delta b)$". Przez rozdzielność F względem G będziemy rozumieli postulat lub twierdzenie mające postać:

$$\{[a\Delta(b\nabla c)] = [(a\Delta b)\nabla(a\Delta c)]\}.$$

Łatwo zauważyć, że postulat 3 – część lewa, jest rozdzielnością koniunkcji nazwowej względem alternatywy nazwowej, natomiast postulat 3 – część prawa jest rozdzielnością alternatywy nazwowej względem koniunkcji nazwowej.

Piśmiennictwo: *Greniewski H.* G.2.1. , *Kołmogorow A.* K.4.1.

2.5.2. Twierdzenia zasadnicze algebry Boole'a

W algebrze Boole'a można między innymi dowieść twierdzeń:

2.5.2.00 $\vdash [(\neg 0) = 1].$

Interpretacja kolektywna: Dopełnienie niczego jest identyczne (jednostkowo i nieegzystencjalnie) z najszerszym rozpatrywanym przedmiotem (np. wszechświatem). Interpretacja dystrybutywna swobodna: Nie-nic, to tyle - co przedmiot.

Szkic dowodu. Weźmy postulat 2.5.1.20 - 5 (część prawa):

$$\vdash [1 = a \vee (\neg a)].$$

Podstawiając do tego postulatu

nazwę	0
za zmienną nazwową	a

otrzymujemy wynik:

(1) $\vdash [1 = 0 \vee (\neg 0)].$

(2) $\vdash (a \vee b = b \vee a)$ (teza pierwotna 2).

Podstawiając do (2)

nazwy	0	$(\neg 0)$
za zmienne nazwowe	a	b

otrzymujemy wynik:

(3) $\vdash [0 \vee (\neg 0) = (\neg 0) \vee 0].$

(4) $\vdash [1 = (\neg 0) \vee 0]$ (ogniwo (1) i (3).

(5) $\vdash (a = a \vee 0)$ (teza pierwotna 5).

Podstawiając do (5)

nazwę	$(\neg 0)$
za zmienną nazwową	a

otrzymujemy wynik:

(6) $\vdash [(\neg 0) = (\neg 0) \vee 0]$

Korzystając z odnośnego twierdzenia elementarnego rachunku identyczności otrzymujemy z (4) i (6) nasze twierdzenie.

2.5.2.01 $\vdash [(\neg 1) = 0]$.

Interpretacja kolektywna: Dopełnienie najszerszego rozpatrywanego przedmiotu jest identyczne (jednostkowo i nie-egzystencjalnie) z niczym. *Interpretacja dystrybutywna*: Nie-przedmiot jest identyczny (ogólnie i nie-egzystencjalnie) z niczym.

2.5.2.10. Prawo podwójnego przeczenia:

$$\vdash \{a = [\neg(\neg a)]\}.$$

2.5.2.11. Prawo De Morgana:

$$\vdash \{[\neg(a \vee b)] = \{(\neg a) \wedge (\neg b)]\}.$$

2.5.2.12. Prawo De Morgana:

$$\vdash \{[\neg(a \wedge b)] = [(\neg a) \vee (\neg b)]\}.$$

Szkic dowodu. Przeprowadźmy najpierw przekształcenia poniższe:

$$\begin{aligned} (a \wedge b) \wedge (\neg a) &= (b \wedge a) \wedge (\neg a) && \text{(teza 2).} \\ &= b \wedge [a \wedge (\neg a)] && \text{(teza 5).} \\ &= b \wedge 0 && \text{(teza 5).} \\ &= 0 && \text{(teza 5).} \end{aligned}$$

Przekształcenia powyższe i przechodniość identyczności (2.1.2.10) dają nam, jako wynik następujący:

(1) $\vdash [(a \wedge b) \wedge (\neg a) = 0]$.

Przekształcamy dalej:

$$\begin{aligned} (a \wedge b) \wedge (\neg b) &= a \wedge [b \wedge (\neg b)] && \text{(teza 3).} \\ &= a \wedge 0 && \text{(teza 3).} \\ &= 0 && \text{(teza 3).} \end{aligned}$$

Powyższe przekształcenie i przechodniość identyczności dają nam, jako wynik następujący:

(2) $\vdash [(a \wedge b) \wedge (\neg b) = 0]$.

Przeprowadźmy jeszcze krótkie przekształcenie:

$$\begin{aligned} 0 &= 0 \vee 0 && \text{(teza 4).} \\ &= [(a \wedge b) \wedge (\neg a)] \vee [(a \wedge b) \wedge b] && \text{(ogniwo (1), (2)).} \\ &= (a \wedge b) \wedge [(\neg a) \vee (\neg b)] && \text{(teza 4).} \end{aligned}$$

Mamy więc:

(3) $\vdash \{(a \wedge b) \wedge [(\neg a) \vee (\neg b) = 0]\}$

Następne przekształcenie:

$$\begin{aligned} [(\neg a) \vee (\neg b)] \vee a &= [(\neg b) \vee (\neg a)] \vee a && \text{(teza 3).} \\ &= (\neg b) \vee (\neg a) \vee a) && \text{(teza 3).} \\ &= (\neg b) \vee 1) && \text{(teza 3).} \\ &= 1 && \text{(teza 3).} \end{aligned}$$

Mamy więc:

(4) $\vdash \{[(\neg a) \vee (\neg b)] \vee a = 1\}$.

Przeprowadzamy teraz przekształcenie podobne do poprzedniego

$$\begin{aligned} (\neg a) \vee (\neg b) \vee b &= (\neg a) \vee (\neg b) \vee b) && \text{(teza 3).} \\ &= (\neg a) \vee 1 && \text{(teza 3).} \\ &= 1 && \text{(teza 3).} \end{aligned}$$

Skąd:

(5) $\vdash [(\neg a) \vee (\neg b) \vee b = 1]$.

Przekształcając dalej:

$$1 = 1 \wedge 1 \qquad \text{(teza 3).}$$
$$= [(\neg a) \vee (\neg b) \vee a] \wedge [(\neg a) \vee (\neg b) \vee b] \qquad \text{(ogniwo (4), (5)).}$$
$$= (\neg a) \vee (\neg b) \vee (a \wedge b). \qquad \text{(teza 3).}$$

Mamy więc wynik:

(6) $\vdash [1 = (\neg a) \vee (\neg b) \vee (a \wedge b)\ ].$

Z powyższego przekształcenia za pomocą tez przechodniości oraz symetrii otrzymujemy:

$$\vdash \{[\neg(a \wedge b)] = [(\neg a) \vee (\neg b)]\}.$$

co było do okazania.

Piśmiennictwo: *Greniewski H*. G.2.1. , *Kołmogorow A.* K.4.1.

2.5.3. Algebry Kleene'ego

Stephen Cole Kleene (1909 - 1994) - amerykański matematyk, jeden z pionierów informatyki teoretycznej. Zasłynął z prac z teorii rekursji, opracowania koncepcji wyrażeń regularnych i teorii funkcji obliczalnych. Z jego nazwiskiem związane jest nierozłącznie takie pojęcia jak *algebra Kleene'ego*, twierdzenie *Kleene'ego o rekursji*. *Algebra Kleene'ego* (KA – *Kleene algebra*) jest algebrą wyrażeń regularnych i znana jest również pod wieloma różnymi nazwami.

2.5.3.10. **Wyjaśnienie**. *Algebra Kleene'ego* jest strukturą algebraiczną (K, ∪, ∘, *, 0, 1) spełniająca poniższy zbiór aksjomatów (według *Dexter'a Kozena*):

2.5.3.11 $\alpha \cup (\beta \cup \gamma) = (\alpha \cup \beta) \cup \gamma.$

2.5.3.12 $\alpha \cup \beta = \beta \cup \alpha.$

2.5.3.13 $\alpha \cup 0 = \alpha \cup \alpha = \alpha.$

2.5.3.14 $\alpha \circ (\beta \circ \gamma) = (\alpha \circ \beta) \circ \gamma.$

2.5.3.15 $1 \circ \alpha = \alpha \circ 1 = \alpha.$

2.5.3.16 $\alpha \circ (\beta \cup \gamma) = \alpha\beta \cup \alpha\gamma.$

2.5.3.17 $(\alpha \cup \beta) \circ \gamma = \alpha \circ \gamma + \beta \circ \gamma$

2.5.3.18 $0 \circ \alpha = \alpha \circ 0 = 0$

2.5.3.19 $1 \cup \alpha \circ \alpha^* = 1 \cup \alpha^* \circ \alpha = \alpha^*$

2.5.3.20 $\beta \cup \alpha \circ \gamma \leq \gamma \ \rightarrow \alpha^* \circ \beta \ \leq \ \gamma$

2.5.3.21 $\beta \cup \gamma \circ \alpha \leq \gamma \ \rightarrow \beta \circ \alpha^* \leq \ \gamma$

Gdzie operator ≤ definiuje naturalne uporządkowanie częściowe w K:

2.5.3.30 $\alpha \ \leq \ \beta \ =_{\text{Def}} \ \alpha \cup \beta = \beta$

Mówiąc krótko KA jest idempotentnym półpierścieniem względem ∪, ∘, 0, 1 spełnienia ze względu na operację *. Trzy ostatnie aksjomaty mówią, że operator * zachowuje się jak powielacz łańcuch symboli należących do zbioru znaków.

Piśmiennictwo: *Harel* D. H.1.1.

2.5.4. DEFINICJA WYRAŻEŃ REGULARNYCH

Wyrażenia regularne (*regular expressions*, w skrócie *regex* lub *regexp*) – wzorce, które opisują łańcuchy symboli. Teoria wyrażeń regularnych jest związana z teorią języków regularnych. Wyrażenia regularne mogą określać zbiór pasujących łańcuchów, mogą również wyszczególniać istotne części łańcucha.

Wyrażenia regularne to w informatyce teoretycznej ciągi znaków pozwalające opisywać języki regularne. W praktyce znalazły bardzo szerokie zastosowanie, pozwalają bowiem w łatwy sposób opisywać wzorce tekstu, natomiast istniejące algorytmy w efektywny sposób określają, czy podany ciąg znaków pasuje do wzorca lub wyszukują w tekście wystąpienia wzorca. Wyrażenia regularne w praktycznych zastosowaniach są zapisywane za pomocą bogatszej i łatwiejszej w użyciu składni niż ta stosowana w rozważaniach teoretycznych. Co więcej, opisane niżej powszechnie wykorzystywane wsteczne referencje (czyli użycie wcześniej dopasowanego fragmentu tekstu jako części wzorca), powodują, że wyrażenie regularne je zawierające może nie definiować języka regularnego.

Wyrażeniem regularnym nad alfabetem $\sum$ nazywamy ciąg znaków składający się z symboli: $\emptyset, \in, \circ, \cup, *,), ($ oraz symboli a_i z alfabetu $\sum$ następującej postaci:

2.5.4.11 $\emptyset, \in$ (słowo puste) są wyrażeniami regularnymi;

2.5.4.12 Wszystkie symbole $a_i \in \sum$ są wyrażeniami regularnymi;

2.5.4.13 Jeśli e_1, e_2 są wyrażeniami regularnymi, to są nimi również:

e_1*(domknięcie Kleene'ego)

$e_1 \circ e_2$ (konkatenacja)

$e_1 \cup e_2$ (suma)

e_1 (grupowanie)

2.5.4.14 Wszystkie wyrażenia regularne są postaci opisanej w punktach powyższych.

Każde wyrażenie regularne definiuje pewien język formalny. Każdy język definiowany przez wyrażenie regularne jest regularny.

Piśmiennictwo: *Harel* D. H.1.1.

2.5.5. DEFINICJA JĘZYKA OKREŚLANEGO PRZEZ WYRAŻENIE REGULARNE

Język definiowany przez wyrażenie regularne jest definiowany indukcyjnie. Niech L(w) oznacza język definiowany przez w. Wtedy baza indukcji jest następująca:

2.5.5.11 $L(\in) = \{\in\}$ (zbiór zawierający tylko słowo puste).

2.5.5.12 $L(\emptyset) = \emptyset$ (zbiór pusty).

2.5.5.13 $L(a) = \{a\}$ dla dowolnego a z alfabetu $\sum$.

Natomiast do konstrukcji wyrażeń służą 3 symbole:

2.5.5.21 $L(w \cup u) = L(w) \cup L(u)$ (suma języków)

2.5.5.22 $L(w^*) = (L(w))^*$ (domknięcie Kleene'ego)

2.5.5.22 $L(w \circ u) = \{x,y : x \in L(w) \wedge y \in L(v)$ (konkatenacja języków).

2.5.5.30. **Wyjaśnienie**. Gwiazdka „*" wiąże najsilniej, konkatenacja „∘" słabiej, suma „∪" najsłabiej.

Piśmiennictwo: *Harel* D. H.1.1.

2.5.6. Własności wyrażeń regularnych

2.5.6.10. **Twierdzenie**. Wyrażenia są równoważne gdy definiują ten sam język:

$e_1 = e_2 \Leftrightarrow L(e_1) = L(e_2)$

2.5.6.11. Prawo pochłaniania $e \cup e = e$.

2.5.6.12. Prawo przemienności sumy $e_1 \cup e_2 = e_2 \cup e_1$

2.5.6.13. Prawo łańcuch pusty jest elementem neutralnym konkatenacji $\in \circ e = e \circ \in = e$.

2.5.6.14. Prawo łączności sumy $(e_1 \cup e_2) \cup e_3 = e_1 \cup (e_2 \cup e_3)$.

2.5.6.15. Prawo łączności konkatenacji $(e_1 \circ e2) \circ e_3 = e_1 \circ (e_2 \circ e_3)$.

2.5.6.16. Prawo rozdzielności konkatenacji względem sumy $(e_1 \cup e_2) \circ X = e_1 \circ X \cup e_2 \circ X$.

2.5.6.17. Prawo $X(e_1 \cup e_2) = Xe_1 \cup Xe_2$

2.5.6.18. Prawo przemoienności $e^* \circ e = e \circ e^*$

2.5.6.19. Prawo - domknięcie Kleene'ego jest idempotentne $(e^*)^* = e^*$ –

2.5.6.20. Prawo pochałaniania $e^* \circ e^* = e^*$.

Piśmiennictwo: *Harel* D. H.1.1.

2.5.7. Algebra Kleene'ego z testem

Z praktycznego punktu widzenia, wiele prostych czynności wykonywanych przez programy komputerowe, takich jak: sekwencja instrukcji, pętla powtarzania sekwencji instrukcji, nie wymaga narzędzia do opisu działania wykraczającego poza algebrę Kleene'go. Jak wiadomo niezbędnym składnikiem każdego programu jest *test* odpowiedzialny za wybór kolejnej do wykonania sekwencji instrukcji. Takiego rozszerzenia KA dokonał w latach 1966 – 1967 *Dexter Kozen*, nazywając rozszerzoną przez siebie algebrę - *Kleene algebra with test*, w skrócie KAT. KAT jest algebrą Kleene'ego z wbudowaną pod-algebrą Boole'a. Formalnie jest to algebra o parze uporządkowań $(K, B, \wedge, \vee, \neg, \cup, \circ, *, 0, 1)$, taką, że spełnione są następując trzy warunki:

2.5.7.11. **Definicja**. $(K, \cup, \circ, *, 0, 1)$ – jest algebrą Kleene'ego

2.5.7.12. **Definicja**. (B, ∧, ∨, ¬,0, 1) – jest algebrą Boole'a.

2.5.7.13. **Definicja**. B ⊆ K.

Przy czym, jednoargumentowy operator negacji ¬ jest określony jedynie w B, podobnie jak operacje dwuargumentowe ∧ oraz ∨. Funkcje boole'owskie (zbudowane z elementów oznaczonych literami: α, β, … - na które działają operacje boole'owskie) B, które są dalej nazywane testami, są oznaczane literami: φ?, ψ?, … . Operacje boole'owskie, są określone jedynie w ramach budowania wyrażeń będących testami. Elementy K - są oznaczane literami: α, β, … , na których wykonywane są operacje: konkatenacji „∘", sumy Kleene'ego „∪" oraz domknięcia Kleene'ego „*".

Piśmiennictwo: *Harel* D. H.1.1.

2.6. LOGIKA DYNAMICZNA

2.6.1. WPROWADZENIE DO ZDANIOWEJ LOGIKI DYNAMICZNEJ

Zdaniowa logika dynamiczna (PDL, od angielskiej nazwy *Propositional Dynamic Logic*) została zaproponowana przez *V. Pratta* w 1976 r. Niniejszy opis PDL oparty jest o opracowanie *J. Tiuryna*, J. Tyszkiewicza i P. Urzyczyn[33]. PDL jest eleganckim i zwięzłym formalizmem, pozwalającym badać rozumowania dotyczące programów iteracyjnych. Formalizm ten rozszerza logikę modalną poprzez wprowadzenie modalności dla każdego programu z osobna. System o podobnym charakterze, o nazwie *Logika Algorytmiczna*, został zaproponowany w roku 1970 przez *A. Salwickiego.*

Piśmiennictwo: *Harel* D. H.1.1., *Tiuryn J.* T.4.1.

2.6.2. SKŁADNIA PDL

Syntaktycznie PDL jest mieszanina trzech klasycznych składników: logiki zdaniowej, logiki, modalnej oraz algebry wyrażeń regularnych. Język PDL zawiera wyrażenia dwóch rodzajów: *zdania* (lub formuły) ϕ, ψ, ... oraz *programy* α, β, γ, ... Zakładamy, że mamy do dyspozycji przeliczalnie wiele atomowych symboli każdego rodzaju. *Programy atomowe* są oznaczane przez *a, b, c, ...* , a zbiór wszystkich atomowych programów oznaczamy przez Π_0.

Programy są budowane z programów atomowych przy użyciu operacji *złożenia* (;), *niedeterministycznego wyboru* (∪) oraz *iteracji* (*). Intuicyjnie wykonanie programu α ; β oznacza wykonanie α, a następnie wykonanie na danych wynikowych programu α - programu β. Wykonanie programu $\alpha \cup \beta$ oznacza niedeterministyczny wybór wykonania α lub β. Natomiast wykonanie programu α^* oznacza wykonanie α pewna liczbę razy, być może zero. Ponadto mamy operacje, *testowania* tworzącą z każdej formuły warunkowej ϕ - nowy program ϕ?. Wykonanie programu ϕ? jest możliwe tylko wtedy, gdy warunek ϕ zachodzi. Z drugiej strony, formuły mogą odwoływać się do dowolnego programu α poprzez *modalność konieczności* [α]: dla dowolnego zdania ϕ, napis

$$[\alpha]\phi$$

czytamy po (każdym) wykonaniu programu α koniecznie musi zajść ϕ.

[33] Jerzy Tiuryn, Jerzy Tyszkiewicz, Pawe Urzyczyn „Logika dla informatyków", UW - październik 2006

2.6.2.10. **Definicja** formuł i programów jest wzajemnie rekurencyjna. Definiujemy zbiór programów Π oraz zbiór formuł Φ jako najmniejsze zbiory spełniające następujące warunki:

2.6.2.11. **Warunek.** $ZZ \subseteq \Phi$

2.6.2.12. **Warunek.** $\Pi_0 \subseteq \Pi$

2.6.2.13. **Warunek.** jeśli $\phi, \psi \in \Phi$, to $\phi \to \psi \in \Phi$ oraz $\bot \in \Phi$

2.6.2.14. **Warunek.** jeśli $\alpha, \beta \in \Pi$, to $(\alpha ; \beta)$, $(\alpha \cup \beta)$, oraz $\alpha^* \in \Pi$

2.6.2.15. **Warunek.** jeśli $\alpha \in \Pi$ oraz $\phi \in \Phi$, to $[\alpha]\phi \in \Phi$

2.6.2.16. **Warunek.** jeśli $\phi \in \Phi$, to $\phi? \in \Pi$.

Aby uniknąć pisania zbyt wielu nawiasów stosujemy następujące priorytety:
- Jednoargumentowe operatory (wliczając $[\alpha]$) wiążą silniej niż dwuargumentowe.
- Operator ; wiąże silniej niż $\cup$.
- Spójniki logiczne maja takie same priorytety jak zdefiniowano wcześniej.

Tak wiec wyrażenie:

$$[\alpha ; \beta* \cup \gamma*]\phi \vee \psi$$

odpowiada następującemu wyrażeniu z nawiasami:

$$([(\alpha ; \beta*) \cup \gamma*]\phi) \vee \psi.$$

Ponieważ operatory ; oraz $\cup$ okazują się być łączne, więc zwyczajowo będziemy opuszczać nawiasy w wyrażeniach typu $\alpha ; \beta ; \gamma$ lub $\alpha \cup \beta \cup \gamma$.

Przypomnijmy, że negacja $\neg\phi$ jest skrótem formuły $\phi \to \bot$. Dualnie do [] definiujemy *modalność możliwości:*

$$<\alpha> \phi \quad := \quad \neg[\alpha]\neg\phi.$$

Zdanie $<\alpha> \phi$ czytamy istnieje obliczenie programu α, które zatrzymuje sie w stanie spełniającym formułę ϕ. Istotna różnicą pomiędzy modalnościami [] i <> jest to, że $<\alpha> \phi$ implikuje iż program α się zatrzymuje, podczas gdy $[\alpha]\phi$ nie gwarantuje zatrzymania sie programu α. W szczególności formuła $[\alpha]\bot$ wyraża własność mówiącą, że żadne obliczenie programu α nie zatrzymuje się. Natomiast formuła $<\alpha> \bot$ jest zawsze fałszywa.

Piśmiennictwo: *Harel* D. H.1.1., *Tiuryn J.* T.51.

2.6.3. Przykłady tautologii PDL

Pierwsza grupa tautologii to schematy znane z logiki modalnej.

2.6.3.10. **Twierdzenia.** Następujące formuły są tautologiami PDL.

2.6.3.11. **Tautologia** $<\alpha>(\phi \vee \psi) \leftrightarrow <\alpha> \phi \vee <\alpha> \psi.$

2.6.3.12. **Tautologia** $[\alpha](\phi \wedge \psi) \leftrightarrow [\alpha]\phi \wedge [\alpha]\psi$

2.6.3.13. **Tautologia** $[\alpha](\phi \to \psi) \to ([\alpha]\phi \to [\alpha]\psi)$

2.6.3.14. **Tautologia** $<\alpha>(\phi \wedge \psi) \to <\alpha> \phi \wedge <\alpha> \psi$

2.6.3.15. **Tautologia** $[\alpha]\phi \vee [\alpha]\psi \to [\alpha](\phi \vee \psi)$

2.6.3.16. **Tautologia** $<\alpha> \bot \leftrightarrow \bot$

2.6.3.11. **Tautologia** $[\alpha]\phi \leftrightarrow \neg <\alpha> \neg\phi.$

Następna grupa tautologii, specyficzna dla PDL, dotyczy spójników programotwórczych ; i $\cup$ oraz testu ?.

2.6.3.20. **Twierdzenia.** Następujące formuły są tautologiami PDL.

2.6.3.21. **Tautologia** $<\alpha \cup \beta> \phi \leftrightarrow <\alpha> \phi \vee <\beta> \phi$

2.6.3.22. **Tautologia** $[\alpha \cup \beta]\ \phi \leftrightarrow [\alpha]\phi \wedge [\beta]\phi$

2.6.3.23. **Tautologia** $<\alpha\ ;\ \beta> \phi \leftrightarrow <\alpha> <\beta> \phi$

2.6.3.24. **Tautologia** $[\alpha\ ;\ \beta]\phi \leftrightarrow [\alpha][\beta]\phi$

2.6.3.25. **Tautologia** $<\phi?> \psi \leftrightarrow (\phi \wedge \psi)$

2.6.3.26. **Tautologia** $[\phi?]\psi \leftrightarrow (\phi \rightarrow \psi).$

Ostatnia grupa tautologii dotyczy operatora iteracji $*$.

2.6.3.30. **Twierdzenia.** Następujące formuły są tautologiami PDL.

2.6.3.31. **Tautologia** $[\alpha*]\phi \rightarrow \phi$

2.6.3.32. **Tautologia** $\alpha*$ jest semantycznie relacja zwrotną $\Phi \rightarrow <\alpha*> \phi$

2.6.3.33. **Tautologia** $[\alpha*]\ \phi \rightarrow [\alpha]\ \phi$

2.6.3.34. **Tautologia** $<\alpha> \phi \rightarrow <\alpha*> \phi$

2.6.3.35. **Tautologia** $[\alpha*]\phi \leftrightarrow [\alpha*\alpha*]\phi$

2.6.3.36. **Tautologia** Przechodniość relacji $\alpha* <\alpha*> \phi \leftrightarrow <\alpha*\alpha*> \phi$

2.6.3.37. **Tautologia** $[\alpha*]\phi \leftrightarrow [\alpha**]\phi$

2.6.3.38. **Tautologia** $<\alpha*> \phi \leftrightarrow <\alpha**> \phi$

2.6.3.39. **Tautologia** $[\alpha*]\phi \leftrightarrow \phi \wedge [\alpha][\alpha*]\phi$

2.6.3.40. **Tautologia** $<\alpha*> \phi \leftrightarrow \phi \vee <\alpha> <\alpha*>\phi$

2.6.3.41. **Tautologia** zasada indukcji $[\alpha*]\phi \leftrightarrow \phi \wedge [\alpha*](\phi \rightarrow [\alpha]\phi)$

2.6.3.42. **Tautologia** $<\alpha*> \phi \leftrightarrow \phi \vee <\alpha*>(\neg\phi \wedge <\alpha> \phi).$

Podstawą jest założenie, że własność ϕ jest spełniona w pewnym stanie u. Warunek indukcyjny mówi, że w każdym stanie osiągalnym z, u poprzez skończona, liczbę iteracji programu α, kolejne wykonanie α zachowuje własność ϕ. Teza stwierdza, że wówczas ϕ jest spełnione we wszystkich stanach osiągalnych w skończonej liczbie iteracji α.

Piśmiennictwo: *Harel* D. H.1.1., *Tiuryn J.* T.4.1.

2.6.4. Aksjomaty PDL

Aksjomaty logiki zdaniowej PDL dzielą się na dwie grupy. Pierwsza z nich to sześć aksjomatów:

2.6.4.11. **Aksjomat** $[\alpha](\phi \to \psi) \to ([\alpha]\phi \to [\alpha]\psi)$

2.6.4.12. **Aksjomat** $[\alpha](\phi \wedge \psi) \leftrightarrow [\alpha]\phi \wedge [\alpha]\psi$

2.6.4.13. **Aksjomat** $[\alpha \cup \beta]\phi \leftrightarrow [\alpha]\phi \wedge [\beta]\phi$

2.6.4.14. **Aksjomat** $[\alpha\,;\beta]\phi \leftrightarrow [\alpha][\beta]\phi$

2.6.4.15. **Aksjomat** $[\psi?]\phi \leftrightarrow (\psi \to \phi)$

2.6.4.16. **Aksjomat** $\phi \wedge [\alpha][\alpha*]\phi \leftrightarrow [\alpha*]\phi$

Druga grupa zawiera jeden aksjomat indukcji:

2.6.421. **Aksjomat indukcji** $\phi \wedge [\alpha*](\phi \to [\alpha]\phi) \to [\alpha*]\phi$

Zbiór aksjomatów PDL uzupełniają tzw. *Reguły dowodzenia*

2.6.4.31. **Reguła dowodzenia** (MP) $\dfrac{\varphi \quad \varphi \to \psi}{\psi}$

2.6.4.32. **Reguła dowodzenia** (GEN) $\dfrac{\varphi}{[\alpha]\varphi}$

Reguła (GEN) nazywana jest regułą *modalnej generalizacji*. Jeśli ϕ daje sie wyprowadzić w powyższym systemie poszerzonym o dodanie nowych aksjomatów ze zbioru Σ, to będziemy, to zapisywać przez Σ ϕ. Jak zwykle piszemy ϕ, gdy Σ jest zbiorem pustym. Fakt, że wszystkie aksjomaty są tautologiami PDL wynika z 2.6.2.

Wprawdzie obecnie, PDL nie znalazł jeszcze praktycznego znaczenia w walidowaniu reaktywnych współbieżnych programów działających w rozproszonym środowisku, ale należy przypuszczać, że sytuacja ulegnie zmianie. Wydaje się, że poszerzenie własności modalności na poszczególne procesy lub grupy procesów, pozwoli na lepsze modelowanie synchronizacji zegarów pomiędzy poszczególnymi systemami sprzętowymi sieci rozległych.

Piśmiennictwo: *Harel* D. H.1.1., *Tiuryn J.* T.4.1.

2.7. Logika pierwszego rzędu

2.7.1. Krótka charakterystyka logiki pierwszego rzędu

Logika pierwszego rzędu to system logiczny, w którym zmienna, na której oparty jest kwantyfikator, może być elementem pewnej wybranej dziedziny (zbioru), nie może natomiast być zbiorem takich elementów. Tak więc nie mogą występować kwantyfikatory typu „dla każdej funkcji z X na Y ...” (gdyż funkcja jest podzbiorem iloczynu kartezjańskiego zbiorów X × Y), „istnieje własność p, taka że ...” czy „dla każdego podzbioru X zbioru Z ...”. Na przykład w logice pierwszego rzędu można zapisać zdanie „dla dowolnej liczby rzeczywistej istnieje liczba większa”. Logika pierwszego rzędu w ogólnym przypadku nie jest rozstrzygalna (w przeciwieństwie do rachunku zdań), lecz pół-rozstrzygalna (czyli jak mówią matematycy rekurencyjnie przeliczalna). Znaczna część rozważań matematycznych może być

sformalizowana na gruncie logiki pierwszego rzędu, co ma wpływ na preferowanie teorii sformalizowanych na jej gruncie.

2.7.1.10. **Wyjaśnienie**. Termy języka logiki pierwszego rzędu. Niech τ będzie alfabetem języka pierwszego rzędu $\mathcal{L}(\tau)$. Tak więc τ jest zbiorem stałych, symboli funkcyjnych i symboli relacyjnych (predykatów). Każdy z tych symboli ma jednoznacznie określony charakter (tzn. wiadomo czy jest to stała, czy symbol funkcyjny czy też predykat) i każdy z symboli funkcyjnych i predykatów ma określoną *n*-arność (która jest dodatnią liczbą całkowitą). Język $\mathcal{L}(\tau)$ ma też ustaloną nieskończoną listę zmiennych (zwykle x_0, x_1,...). Termy języka $\mathcal{L}(\tau)$ to elementy najmniejszego zbioru **T** takiego, że:

- wszystkie stałe i zmienne należą do **T**,
- jeśli $t_1, \ldots, t_n \in T$ i $\boldsymbol{f} \in \tau$ jest *n*-arnym symbolem funkcyjnym, to $f(t_1, \ldots, t_n) \in \mathbf{T}$.

2.7.1.20. **Wyjaśnienie**. System logiki pierwszego rzędu składa się z:

1) zmiennych nazwowych (litery, za które wolno podstawić nazwy dowolnych przedmiotów)
2) stałych nazwowych (nazwy własne przedmiotów)
3) liter symboli predykatów (predykaty)
4) symboli funkcyjnych (funktory nazwotwórcze od argumentów nazwowych)
5) stałych logicznych (spójniki prawdziwościowe rachunku zdań i kwantyfikatory)
6) znaków pomocniczych (nawiasy)
7) symbolu równości.

Używając symboli wymienionych powyżej i przestrzegając naturalnych reguł możemy budować *poprawnie zbudowane napisy*. Niektóre z tych napisów mogą być interpretowane jako *nazwy* na pewne obiekty, a inne będą mówić o własnościach tych obiektów. Pierwsza grupa napisów poprawnie zbudowanych to *termy*, a druga to *zdania*.

Piśmiennictwo: *Mostowski A.* M.4.1., *Pacholski L.* P.1.1., *Tiuryn J.* T.41.

2.7.2. Język i składnia logiki pierwszego rzędu

2.7.2.10. **Wyjaśnienie**. Język logiki pierwszego rzędu - można traktować jak rozszerzenie rachunku zdań, pozwalające formułować stwierdzenia o zależnościach pomiędzy obiektami indywidualnymi (np. relacjach i funkcjach). Dzięki zastosowaniu *kwantyfikatorów*, odwołujących się do całej zbiorowości rozważanych obiektów, można w logice pierwszego rzędu wyrażać własności struktur relacyjnych oraz modelować rozumowania dotyczące takich struktur. Do zestawu symboli rachunku zdań dodajemy następujące nowe składniki syntaktyczne:

- *Symbole operacji i relacji* (w tym symbol równości =);
- *Zmienne indywidualne*, których wartości maja przebiegać rozważane dziedziny;
- *Kwantyfikatory*, wiążące zmienne indywidualne w formułach.

2.7.2.20. **Wyjaśnienie**. Symbole operacji i relacji są podstawowymi składnikami do budowy najprostszych formuł tzw. *formuł atomowych*. Z tego względu w języku pierwszego rzędu rezygnuje się ze zmiennych zdaniowych.

Formułę bez kwantyfikatorów nazywamy *formułą otwartą*. Natomiast formuła bez zmiennych wolnych nazywa sie *zdaniem*, lub *formuła zamkniętą*. Negację, koniunkcję, alternatywę, symbol prawdy i równoważność formuł definiujemy podobnie jak w przypadku rachunku zdań.

Kwantyfikator egzystencjalny zdefiniujemy jako skrót notacji przy pomocy *uogólnionego prawa De Morgana*:

2.7.2.30 $\exists x\phi$ oznacza $\neg\forall x\neg\phi$.

Zmienne wolne a zmienne związane. Zakładając, że $x \in FV(\phi)$, zauważmy też, że zmienna x może występować w formule ϕ podczas gdy $x \notin FV(\phi)$. Przez *wystąpienie* zmiennej x rozumiemy tu zwykle pojawienie się x w jakimkolwiek termie w ϕ. I tak na przykład w formule[34]

2.7.2.40 $\exists x\forall u(r(x,y) \rightarrow \forall y\exists xs(x,y,z))$

zmienna u nie występuje, podczas gdy x i y występują po dwa razy, a z występuje jeden raz. Bardzo ważną rzeczą jest rozróżnienie wystąpień zmiennych *wolnych* i *związanych* w formułach. Wszystkie wystąpienia zmiennych w formułach atomowych są wolne. Wolne (związane) wystąpienia w formułach ϕ i ψ pozostają wolne (związane) w formule $\phi \rightarrow \psi$. Wszystkie wolne wystąpienia x w ϕ stają się wystąpieniami związanymi w formule $\exists x\phi$ (związanymi przez dopisanie kwantyfikatora $\exists$), a charakter pozostałych wystąpień jest taki sam w ϕ i w $\exists x\phi$. Przykładowo w formule

2.7.2.50 $\exists x\forall u\,(r\,(x, y) \rightarrow \forall\, y\, \exists x\, s\,(x, y, z))$

podkreślone wystąpienie y jest wolne, a nie podkreślone jest związane. Obydwa wystąpienia x są związane, ale przez różne kwantyfikatory.

Piśmiennictwo: *Mostowski A.* M.5.1., *Pacholski L.* P.1.1., *Tiuryn J.* T.4.1.

2.7.3. Logika pierwszego rzędu a informatyka

Logika pierwszego rzędu jest niewątpliwie bardzo interesującą teorią, pozwalająca na wyciąganie wielu ważnych wniosków, ale nie z punktu widzenia informatyki. Jak już zostało powiedziane w 2.0.3., informatyka jest dziedziną zajmującą się zbiorowiskami bardzo licznymi ale zawsze skończonymi. Jedną z interesujących prób wykorzystania dorobku logiki pierwszego rzędu w informatyce jest *Notacja Z.* Jest to język formalnej specyfikacji systemów informatycznych wykorzystujący notację języka logiki pierwszego rzędu, dodatkowo umożliwiający formalne przekształcanie i upraszczanie specyfikacji (czynności tą określono nazwą „*Refinaments*").

J. M. Spivey opublikował w 1989 roku wstępną wersję raportu „*The Z Notation: a Reference Manual*". Ten raport zawierał wyniki prac zespołu nazwanego „*Program Research Group, University of Oxford*". W pracach zespołu wzieli udział: *J. R. Arial, I. J. Hades, C. A. R. Hoar, He Jifeng, C. C. Morgan, J. W. Anders, I. H. Sørensen, J. M. Spivey* and *B. A. Sufrin*. Końcowa wersja raportu została opublikowana w 1998 roku. Notacja Z wprawdzie mogłaby operować, ze względu na swoją formalną strukturę zbiorami poza skończonymi, ale praktyczne jej zastosowania dotyczą jedynie zbiorów skończonych.

Trudno wyobrazić sobie, poważniejsze zastosowania logiki pierwszego rzędu we współczesnej informatyce, ale pewne podejścia typowe dla logiki pierwszego rzędu, mogą znaleźć swoje miejsce we współczesnej informatyce.

Piśmiennictwo: *Ben-Ari M*. B.2.1., *Kisielewicz A*. K.1.1., *Spivey* J. S.10.1.

[34] Zakładamy tu, że *s* oraz *r* są symbolami relacji.

Część 3.
Formalne podstawy informatyki

3.0. Rodzaje języków

3.0.0. Pojęcie języka

Wyrazu „język” używamy w języku polskim, co najmniej w dwojakim rozumieniu. W pewnych wypadkach mówiąc „język” mamy na myśli pewną część ciała ludzkiego czy zwierzęcego. To właśnie jest jedno z rozumień wyrazu „język”. W innych wypadkach mówiąc „język” mamy na myśli pewien system sygnałów, z pomocą - których następuje w obrębie społeczeństwa, porozumiewanie się między ludźmi. To właśnie jest drugie rozumienie wyraz „język”.

Zapamiętajmy, że odtąd będziemy używali wyrazu „język” tylko w tym drugim rozumieniu. Porozumiewanie się między ludźmi zapomogą tego czy innego języka zwykle obejmuje: *informowanie, stawianie pytań, wydawanie rozkazów itd*.

Języki naturalne, ale również niektóre języki sztuczne, składają się z dwóch warstw, a mianowicie z warstwy informacyjnej i warstwy prezentacyjnej. Warstwa informacyjna jest nośnikiem informacji przekazywanych w danym języku. Natomiast warstwa prezentacji służy do odpowiedniego przekazywania (w jakimś sensie interpretowania) informacji – właściwemu odbiorcy. Na fakt ten - zwrócono uwagę stosunkowo niedawno, być może, że dopiero przy opracowywaniu kolejnej wersji języka HTML i wyposażania go w mechanizm nazwany CSS (kaskadowe arkusze styli), który umożliwił oddzielenie warstw informacyjnej i prezentacyjnej.

Piśmiennictwo: *Dobosiewicz S.* D.3.1., *Greniewski H.* G.2.1.

3.0.1. Języki akustyczne, graficzne i gestowo-mimiczne

Odróżniamy trzy rodzaje języków:

1) Języki akustyczne (*język akustyczny - mowa*);
2) Języki graficzne (*język graficzny - pismo*);
3) Języki gestowo-mimiczne.

Podział powyższy jest rozłączny, to znaczy, że żaden język gestowo-mimiczny nie jest językiem akustycznym, żaden język gestowo-mimiczny nie jest językiem graficznym i wreszcie, że żaden język akustyczny nie jest językiem graficznym. Nie twierdzimy natomiast, że podział powyższy jest zupełny, to znaczy, że nie wykluczamy istnienia - chociaż jednego języka, który nie jest ani gestowo-mimiczny, ani akustyczny, ani graficzny. Zgodne z wyżej wprowadzonym podziałem należy odróżniać dwa odrębne języki: polski język akustyczny i polski język graficzny. Takie rozróżnienie nie jest stosowane potocznie, mamy - bowiem zwyczaj utożsamiać w życiu codziennym dowolny język akustyczny z jego odpowiednikiem graficznym. Przeciwko takiemu utożsamianiu przemawiają jednak względy następujące:

1) Istnieją ludzie znający swój ojczysty język akustyczny, lecz nieznający jego odpowiednika graficznego (analfabeci).
2) Każdy język akustyczny jakiejś narodowości (mającej swój język graficzny) powstał znacznie wcześniej niż - odpowiadający mu język graficzny. Innymi słowy, wynalazek (czy adaptacja) pisma był w każdym społeczeństwie, które takiego wynalazku dokonało (lub wynalazek taki adaptowało), dużo późniejszy niż powstanie mowy.
3) Zdarza się niekiedy, ze jeden i ten sam język graficzny jest odpowiednikiem więcej niż - jednego języka akustycznego. Istnieje na przykład jeden graficzny język chiński i

przynajmniej dwa chińskie języki akustyczne. Niejeden Chińczyk z południa, znając południowo-chiński język akustyczny i graficzny język chiński, nie rozumie języka akustycznego północno-chińskiego i odwrotnie.

4) Zdarza się, że znamy język graficzny nie znając jego odpowiednika akustycznego. Tak mianowicie przedstawia się sprawa z niektórymi językami martwymi, na przykład z łaciną i staroegipskim. Znamy, bowiem graficzny język łaciński nie znając jego autentycznej wymowy (teksty łacińskie czytamy głośno w sposób konwencjonalny). To samo - z językiem staroegipskim[35]. Na pierwszy rzut oka język graficzny danej narodowości jest dokładnym odbiciem jego odpowiednika akustycznego.

W rzeczywistości sprawa przedstawia się inaczej. Przekład z języka graficznego na jego odpowiednik akustyczny czy też przekład w kierunku odwrotnym napotyka nieraz znaczne trudności, ze względu na warstwę prezentacyjną. Niech ta prosta anegdota, zastąpi nam tu rozwlekły wywód. Za czasów, gdy na-wsi polskiej rozpowszechniony był jeszcze analfabetyzm i obowiązywała powszechna służba wojskowa. Wzięto pewnego Jasia do wojska. Po dwu tygodniach od Jasia nadszedł na wieś list do jego matki. Matka Jasia była analfabetką i szukała po wsi kogoś, kto by jej przeczytał list od syna. Natknęła się na zawodowego sierżanta, który spędzał tam swój urlop. Ostrym głosem nawykłym do „besztania" i wydawania rozkazów sierżant czytał: *„Kochana Matulu. Jak się miewają świnie, kury i gęsi? Przyślijta mi z dwa pęta kiełbasy, bo mi się strasznie cni". „Oj, takeś zhardział, już po dwu tygodniach* - krzyknęła matula – *nic nie dostaniesz".* Minęły jeszcze dwa tygodnie. Od Jasia nadszedł nowy list. Nowy kłopot, kto list przeczyta. Natknęła się Jasiowa matka na staruszka nauczyciela. Ten drżącym, płaczliwym głosem czytał: *„Kochana Matulu. Jak się miewają świnie, kury i gęsi? Przyślijta mi z dwa pęta kiełbasy, bo mi się strasznie cni". „Oj, mój chrobocku* - powiada matula - *toś ty taki biedak, zaraz kiełbasy ci poślę"*[36].

Widzimy, więc, że przekład z języka graficznego na jego odpowiednik akustyczny bywa niepozbawiony dowolności (w powyższym żartobliwym przykładzie dowolność tę stwarzała mimowolna intonacja). Ale ta wzmiankowana mimowolna intonacja, nie jest niczym innym jak warstwą prezentacyjną języka.

Rozróżniając języki: gestowo-mimiczne, akustyczne i graficzne nie po to bynajmniej, że języków tych używamy nieraz łącznie czy równocześnie. Aktor na scenie zwraca się do publiczności jednocześnie w języku akustycznym i gestowo-mimicznym, w jego wystąpieniu warstwa prezentacyjna języka odgrywa bardzo istotną rolę. Usuńcie z przedstawienia teatralnego gesty i mimikę nie naruszając mówionych tekstów, a z najlepszego przedstawienie niewiele zostanie. Załatwienie poważnej sprawy przez, telefon jest trudniejsze niż w rozmowie bezpośredniej, gdyż w czasie rozmowy telefonicznej słyszymy tylko odpowiedzi drugiej strony dawane w języku akustycznym, pozbawieni jesteśmy widoku mimiki i gestów nieraz istotnie nas informujących o drugiej strony, a tym samym następuje zubożenie warstwy prezentacyjnej języka.

Daliśmy powyżej dwa przykłady posługiwania się równocześnie akustycznym i gestowo-mimicznym. Nie trudno dać przykład trzech języków naraz: gestowo-mimicznego, akustycznego i graficznego. W czasie dobrze prowadzonej lekcji matematyki nauczyciel pisze na tablicy, mówi

[35] Uprzejmości Olgierda Wojtasiewicza zawdzięczamy kilka uwag niniejszego paragrafu.

[36] Tę interesującą anegdotę zawdzięczamy uprzejmości Tadeusza Pasierbińskiego.

do uczniów, a ruchy jego rąk i mimika nie są bynajmniej obojętne, bo w procesie nauczania warstwa prezentacyjna odgrywa nie mniejszą rolę niż w przedstawieniu teatralnym.

Piśmiennictwo: *Dobosiewicz S.,* D.3.1., *Greniewski H.* G.2.1.

3.0.2. Języki naturalne, sztuczne i mieszane

Należy odróżniać trzy rodzaje języków:

1. Języki naturalne;
2. Języki sztuczne (symboliki), a w szczególności języki formalne;
3. Języki mieszane.

Podział powyższy jest doniosły tylko z punktu widzenia logiki formalnej, przy czym jest on niezależny od uprzednio wprowadzonego podziału na języki gestowo-mimiczne, graficzne i akustyczne. Nie wynika stąd, że ma on jakiekolwiek znaczenie dla językoznawstwa.

3.0.2.00. **Wyjaśnienie**. „*Język naturalny*", to tyle, co język obsługujący całe społeczeństwo (narodowość, naród), każdy język taki powstaje na drodze żywiołowej.

Mówiąc, że każdy język naturalny powstał w drodze żywiołowej, nie przeczymy temu, że pewne fragmenty danego języka naturalnego rozwijają się w sposób nie żywiołowy, lecz planowany. Na przykład każdy fragment języka naturalnego będący terminologią naukową czy techniczną rozwija się zwykle w sposób planowany, a nie żywiołowy.

3.0.2.10. **Wyjaśnienie**. „*Język sztuczny*", to tyle, co „język stworzony świadomie przez poszczególnych ludzi i to w sposób umowny"; każdy taki język obsługuje wąski (w porównani z językiem naturalnym) zakres potrzeb społeczeństwa.

Rozwojowi informatyki towarzyszy niemal eksplozja powstawania i rozwoju sztucznych języków. Przykładowo, powstało wiele języków programowania (komputerów), począwszy od języka FORTRAN – jeszcze na początku lat pięćdziesiątych XX wieku; język SQL – wyszukiwania informacji w bazach danych; język UML – zunifikowany język modelowania systemów informatycznych; BPMN – notacja modelowania procesów biznesowych; SGML i jego następca XML – języki znaczników, odgrywające dziś olbrzymią rolę, jako narzędzia dokumentowania i administrowania dokumentami.

3.0.2.20. **Wyjaśnienie**. Niekiedy okazuje się celowe utworzenie z jakiegoś języka naturalnego i jakiegoś języka sztucznego –„konglomeratu", który również jest językiem. Każdy taki i tylko taki konglomerat nazywamy *językiem mieszanym*.
Przykłady. Znamy przykłady wszystkich trzech rodzajów wyżej wymienionych: Znany jest nam wszystkim przykład języka naturalnego - to nasz język ojczysty, polski język akustyczny. Uczyliśmy się w szkole średniej algebry, geometrii, fizyki chemii, informatyki. W podręczniku każdego z tych przedmiotów widzieliśmy wzory lub schematy. Każdy z tych wzorów lub schematów był napisany w jakimś języku sztucznym; w innym – zresztą języku sztucznym były napisane wzory algebraiczne, a w innym jeszcze języku sztucznym (w innej symbolice) - wzory chemiczne, a w jeszcze innym języku sztucznym programy komputerowe (informatyka). W podręczniku geografii mieliśmy niewątpliwie niejedną mapkę: jest to dla nas nowy przykład, języka sztucznego; sporządzając mapy wypowiadamy się, bowiem w pewnym języku sztucznym w języku map geograficznych. Istnieje też język sztuczny - wykresów statystycznych. Podręcznik szkolnej algebry nie składał się jednak z samej symboliki; oprócz wzorów podręcznik ten

zawierał przecież sporo tekstu w języku naturalnym. Krótko mówiąc, podręcznik algebry był napisany w języku mieszanym.

Podane przykłady mogłyby stworzyć błędną sugestię, jakoby tworzenie języków sztucznych było monopolem nauk ścisłych (zwłaszcza matematyki). Otóż wcale tak nie jest. Języki sztuczne powstają i rozwijają się nie tylko dla celów naukowych, ale również dla celów wyłączni praktycznych. Trzy pouczające przykłady możemy zaczerpnąć z transportu (morskiego, kolejowego i drogowego):

- Międzynarodowy system sygnalizacyjny chorągiewek, stosowany powszechnie przez statki morskie,
- System sygnałów kolejowych,
- Międzynarodowy system znaków drogowych (niezbędny dla kierowców samochodowych, ale przydatny też dla pieszych).

W ciągu ostatnich kilkudziesięciu lat (począwszy od 1951 roku) nastąpił, jak wiadomo, gwałtowny rozwój komputerów - maszyn do przetwarzania informacji. Jak już powiedzieliśmy, rozwój ten był nierozłącznie związany z powstaniem nowym języków sztucznych (symbolik) mających zastosowanie w programowaniu komputerów, opisywaniu struktur baz danych, wyszukiwania informacji w bazach danych, projektowaniu systemów informatycznych, opisywaniu procesów biznesowych, itp.

Piśmiennictwo: *Greniewski H.* G.2.1. , *Wilkosz W.*, W.3.1.

3.0.3. Język sformalizowany

3.0.3.10. **Wyjaśnienie**. *Język sformalizowany* (jest z reguły językiem sztucznym) – jest to podzbiór zbioru wszystkich słów nad skończonym alfabetem. Język sformalizowany jest kluczowym pojęciem w informatyce, logice matematycznej i językoznawstwie. Język sformalizowany nie jest uściśleniem pojęcia języka naturalnego i nie powinien być z nim mylony.

Aby zdefiniować język sformalizowany, najpierw definiuje się alfabet, wybierając jakiś niepusty zbiór skończony elementów. Elementy tego zbioru nazywane są symbolami. Ciągi symboli, o skończonej długości, nazywane są słowami. Dowolny zbiór takich ciągów nazywany jest językiem sformalizowanym.
Przykłady języków sformalizowanych:

- Zbiór wszystkich słów złożonych z liter polskiego alfabetu i występujących w pewnym słowniku;
- Zbiór takich słów złożonych z cyfr od 0 do 9, które przedstawiają liczby całkowite;
- Zbiór słów złożonych z zer i jedynek, w których zer jest więcej niż jedynek;
- Zbiór prawidłowo napisanych równań matematycznych;
- Zbiór programów, które po skompilowaniu i uruchomieniu mogą działać na komputere o danej architekturze.

3.0.3.15. **Wyjaśnienie**. *Alfabet.* Alfabetem może być dowolny skończony zbiór. Przykładowe alfabety to:

- Zbiór liczb całkowitych należących do jakiegoś przedziału, np. 0 i 100, albo od 0 do 9;
- Zbiór wszystkich bądź niektórych liter alfabetu łacińskiego;
- Zbiór symboli ASCII;
- Zbiór złożony z dokładnie jednego symbolu;

- Zbiór kart do gry;
- Zbiór złożony z możliwych wartości koloru piksela (trójki liczb całkowitych, od 0 do 255 każda);
- Zbiór wszystkich obrazów możliwych do wyświetlenia przy ustalonej rozdzielczości ekranu i liczbie kolorów.

3.0.3.20. **Wyjaśnienie**. *Alfabetami* nie mogą być:
- Zbiór pusty - nie dało by się z niego ułożyć żadnego słowa. Można jednak rozszerzyć definicję języka formalnego na ten przypadek – wtedy nad takim alfabetem istnieje tylko jedno słowo – słowo puste – i tylko dwa języki – język pusty, oraz język zawierający tylko słowo puste.
- Zbiory nieskończone, np. zbiór wszystkich liczb naturalnych, czy rzeczywistych.

3.0.3.25. **Wyjaśnienie**. *Słowo.* Słowami są dowolne skończone ciągi symboli. Przykłady słowa to:
- Słowa języka naturalnego, np. „*Ala*", „*słoneczko*" (jeśli alfabet obejmuje litery);
- Słowo puste (nad dowolnym alfabetem), oznaczane $\in$;
- Liczba zapisana w systemie dziesiętnym (jeśli alfabet obejmuje cyfry arabskie) lub rzymskim (jeśli alfabet obejmuje znaki I, V, X, L, C, M, D);
- Wyrażenie algebraiczne;
- Film o dowolnej skończonej długości, wyświetlany na ekranie o ustalonej rozdzielczości i liczbie kolorów (jeśli alfabetem jest zbiór obrazów możliwych do wyświetlenia na tym ekranie).

3.0.3.30. **Wyjaśnienie**. Słowami nie są:
- Ciągi o nieskończonej długości, np. reprezentacje liczb niewymiernych w systemie dziesiętnym - choć już reprezentacje symboliczne, np. π - są.
- Nieuporządkowane zbiory symboli, np. zbiór samogłosek.

3.0.3.35. **Wyjaśnienie**. *Języki sformalizowane na zbiorach nieskończonych*. W niektórych zastosowaniach, przydatne jest operowanie na ciągach elementów z nieskończonego zbioru, np. zbioru liczb naturalnych. Zbiór takich ciągów nie jest językiem, ale to ograniczenie można obejść, jeśli tylko zbiór używanych elementów jest przeliczalny. Wtedy te elementy można przedstawić jako słowa nad skończonym alfabetem.
Przykładowo, aby operować na ciągach liczb naturalnych, zapisuje się te liczby w sposób pozycyjny. Np. ciąg "10 200 317 852", zawierający 14 symboli, należy do języka ciągów liczb naturalnych zapisanych w postaci pozycyjnej, za pomocą cyfr arabskich oraz spacji.

3.0.3.40. **Wyjaśnienie**. *Metody definiowania języków.* Dla każdego alfabetu (nawet jednoelementowego), liczba słów nad tym alfabetem jest nieskończona - przeliczalna (oznaczana $\aleph_0$). Liczba zbiorów słów (liczba języków), jest zatem nieprzeliczalna. Ponieważ każda metoda opisania może objąć tylko przeliczalną liczbę elementów, nie istnieje metoda opisania wszystkich języków nad żadnym niepustym alfabetem. Dlatego opisuje się jedynie wybrane klasy języków. Przykładowo *hierarchia Chomsky'ego* precyzuje cztery klasy języków, w zależności od tego jak złożona jest gramatyka formalna opisująca dany język.

3.0.3.45. **Wyjaśnienie**. *Gramatyki formalne.* Gramatyki formalne są najpopularniejszym sposobem opisywania języków sformalizowanych. Opis w postaci gramatyki składa się z:

- Określenia symboli alfabetu, na którym zbudowany jest język. Są to tzw. *symbole terminalne*.
- Określenia dowolnego skończonego zbioru symboli pomocniczych, tzw. *symboli nieterminalnych*
- Określenia jednego *symbolu startowego*, należącego do symboli nieterminalnych.
- Określenia pewnego skończonego zbioru *reguł przepisywania* (zwanych też *produkcjami*). Każda reguła, to para słów (*lewe* → *prawe*), w których mogą występować symbole terminalne i nieterminalne.

Do tak opisanego języka należy każde słowo, dla którego potrafimy zbudować taki ciąg, że:

- Pierwszym elementem ciągu jest słowo złożone z symbolu startowego;
- Każde kolejne słowo w ciągu można uzyskać przez zastąpienie w poprzednim słowie fragmentu równego lewemu słowu jakiejś reguły przez prawe słowo tej reguły;
- Ostatnim elementem ciągu jest dane słowo.

3.0.3.50. **Wyjaśnienie**. *Przynależność słowa do języka.* Nie istnieje ogólna metoda, która dla danej gramatyki formalnej i danego słowa pozwoliłaby stwierdzić, czy dane słowo należy do języka opisywanego przez tę gramatykę. Wynika to z faktu, że gramatyki formalne mogą w szczególności definiować zachowanie *maszyny Turinga* i powyższy problem wymagałby rozwiązania *problemu stopu* (patrz podrozdział 3.7.6). Dlatego w praktycznych zastosowaniach używa się wybranych klas gramatyk, dla których taka weryfikacja jest możliwa do przeprowadzenia. W ogólności, im więcej języków potrafi opisać dana klasa gramatyk, tym problem tej weryfikacji ma większą złożoność.

Przykładowo, *hierarchia Chomsky'ego* wprowadza podział na następujące klasy, które można zdefiniować przez złożoność automatu rozpoznającego należenie do języka:

- *Gramatyki regularne* – rozpoznawalne przez *automaty skończone* (patrz podrozdział 3.8.2);
- *Gramatyki bezkontekstowe* – rozpoznawalne przez *automaty ze stosem* (patrz podrozdział 3.8.3);
- *Gramatyki kontekstowe* – rozpoznawalne przez *automaty liniowo ograniczone*,
- *Dowolne gramatyki* – w ogólności nierozpoznawalne automatycznie. Ich podzbiór, rozpoznawalny przez *maszyny Turinga*, definiuje klasę języków określanych jako rekursywne (patrz podrozdział 3.8.4).

3.0.3.55. **Wyjaśnienie**. *Języki sformalizowane a języki naturalne.* Języki sformalizowane są używane do opisu języków naturalnych, choć nie jest to łatwe. Od 1956 roku stosowane są np. tzw. gramatyki generatywne Chomsky'ego. Problemem jest *kontekstowość* języka naturalnego, tzn. zależność reguł gramatycznych, a szczególnie reguł interpretacji (semantyki) języka naturalnego od kontekstu, czyli sąsiednich zdań, a nawet wypowiedzi bezpośrednio z daną nie sąsiadujących.

3.0.3.60. **Wyjaśnienie**. Aktualnie istnieje wiele systemów komercyjnych przetwarzających język naturalny (np. udostępniony przez firmę Google). Tłumaczenie wolnego tekstu jest bardzo niedokładne, pozwala jednak zrozumieć treść i wspomaga pracę tłumaczy (przyspieszenie nawet 4 razy. Nieco lepsze wyniki zostały osiągnięte w tłumaczeniu tekstów specjalistycznych.

Piśmiennictwo: *Pawlak Z*. P.2.1., *Sipser M*. S.7.1., *Wirth N*. W.5.1.

3.1. O WYRAŻENIACH

3.1.1. RÓWNOKSZTAŁTNOŚĆ WYRAŻEŃ

W każdym języku mamy do czynienia z jakąś równokształtnością. Równokształtność występująca w jednym języku może się różnić od równokształtności występującej w innym języku. Łatwiej jest wyjaśnić, na czym polega równokształtność w językach pozbawionych wszelkich odmian gramatycznych (deklinacji, koniugacji, stopniowania) niż w językach mających jakąś przynajmniej jedną odmianę gramatyczną. Najeży tu jeszcze dodać, że język sztuczny jest z reguły pozbawiony wszelkich odmian gramatycznych.

3.1.1.01.**Wyjaśnienie częściowe**. Jeżeli w danym języku brak wszelkiej odmiany gramatycznej, to dwa wyrażenia tego języka są równokształtne wtedy i tylko wtedy, jeżeli każde z nich może być uważane za wierną kopię pozostałego.

Zgodne z powyższym wyjaśnieniem należy uznać, że w każdym akustycznym języku pozbawionym wszelkiej odmiany gramatycznej każde wyrażenia tak samo brzmiące dla ucha ludzkiego są równokształtne. Zgodnie z powyższym wyjaśnieniem należy również uznać, ze w każdym graficznym języku pozbawionym, wszelkiej odmiany gramatycznej każde dwa wyrażenia będące odbitkami tego samego składu czcionek drukarskich są równokształtne.

3.1.1.10.**Wyjaśnienie częściowe**. Jeżeli w danym języku występuje odmiana gramatyczna, to dwa wyrażenia tego języka są równe wtedy i tylko wtedy, jeżeli spełniony jest - chociaż jeden z warunków następujących:

1) Każde z tych wyrażeń może być uważane za wierną kopię (np. przepisanie czy przedruk - niewprowadzający innych zmian niż kaligraficzne czy typograficzne) drugiego

lub

2) Jedno z tych wyrażeń jest odmianą gramatyczną pozostałego z tych wyrażeń.

Na to, czy dane dwa wyrażenia są równokształtne, czy też nie są równokształtne, wywierają wpływ strony materialne tych wyrażeń, ich odmiany gramatyczne - natomiast wpływu raczej nie wywierają.

Przykłady. Weźmy pod uwagę następujące dwa wyrazy języka polskiego

Adam Adam

Jest to przykład dwóch wyrazów języka polskiego, które są równokształtne między sobą, lecz nie są identyczne, (czyli tożsame), to znaczy nie są jednym i tym samym przedmiotem.
Podamy jeszcze dwa przykłady wyrażeń równokształtnych:

Adam ADAM

Adam Adamowi

Wielokrotne i powszechne doświadczenie skłania nas do przyjęcia następujących reguł równokształtności:

3.1.1.11. **Reguła**. Każde wyrażenie jest równokształtne z sobą.

3.1.1.12. **Reguła**. Jeżeli jakieś wyrażenie (powiedzmy „wyrażenie pierwsze”) jest równokształtne z jakimś wyrażeniem (powiedzmy z „wyrażeniem drugim”), to wyrażenie drugie jest równokształtne z wyrażeniem pierwszym.

3.1.1.13. **Reguła**. Jeżeli wyrażenie pierwsze jest równokształtne z wyrażeniem drugim i wyrażenie drugie jest równokształtne wyrażeniem trzecim, to wyrażenie pierwsze jest równokształtne z wyrażenie trzecim.

Pojęcie równokształtności ma dużą doniosłość praktyczną, o czym świadczą uwagi poniższe: Wraz z postępem technicznym zmieniają się metody produkowania napisów. Po piśmie ręcznym zjawia się pismo drukowane, a następnie maszynowe, a wreszcie komputerowe. Wiadomo powszechnie, że wynalazek druku, a następnie wynalazek maszyny do pisania zwiększył szybkość fabrykowania napisów i umożliwił masowe ich wytwarzanie. Mniej natomiast zwraca się uwagi na to, że wynalazek druku i wynalazek maszyny do pisania zwiększyły prędkość czytania. Dlaczego fakt ten jest dla nas interesujący? Aby czytać, trzeba rozpoznawać napisy równokształtne. Rozpoznawanie równokształtności jest znacznie trudniejsze przy odczytywaniu napisów sporządzanych ręcznie (zwłaszcza w wypadku „niewyraźnego charakteru pisma"), niż w wypadku odczytywania napisów drukowanych czy też pisanych na maszynie lub tworzonych przez komputer.

Piśmiennictwo: *Greniewski H.* G.2.1.

3.1.2. RÓWNOZNACZNOŚĆ WYRAŻEŃ

3.1.2.00. **Wyjaśnienie**. Wyrażenie **A** jest równoznaczne z wyrażenie **B** wtedy i tylko wtedy, jeżeli rozumienie wyrażenia **A** jest identyczne z rozumieniem wyrażenia ***B***. Innymi słowy - dwa wyrażenia są wtedy i tylko wtedy równoznaczne, gdy mają to samo rozumienie.

Na to, czy dane dwa wyrażenia są równoznaczne, czy też nie są równoznaczne, wywierają wpływ tylko strony znaczeniowe, (czyli rozumienia) tych wyrażeń, natomiast ich strony materialne żadnego wpływu nie wywierają. Praktyka językowa upoważnia nas do przyjęcia następujących reguł równoznaczności:

3.1.2.01. **Reguła**. Każde wyrażenie jest równoznaczne z sobą.

3.1.2.02. **Reguła**. Jeżeli jakieś wyrażenie pierwsze jest równoznaczne z wyrażeniem drugim, to wyrażenie drugie jest równoznaczne z wyrażeniem pierwszym.

3.1.2.03. **Reguła**. Jeżeli wyrażenie pierwsze jest równoznaczne z wyrażeniem drugim i wrażenie drugie jest równoznaczne z wyrażeniem trzecim, to wyrażenie pierwsze jest równoznaczne z wyrażeniem trzecim.

Równoważność wyrażeń, jest bardzo istotnym czynnikiem upraszczania algorytmów i programów komputerowych. W części czwartej, zajmiemy się Notacją Z, w której między innymi, oferowana jest metoda przekształcania modułów algorytmów do równoważnej im postaci programów komputerowych.

Piśmiennictwo: *Greniewski H.* G.2.1.

3.1.3. RÓWNOKSZTAŁTNOŚĆ A RÓWNOZNACZNOŚĆ

Nie należy, oczywiście, równokształtności utożsamiać z równoznacznością. Często mamy do czynienia z takimi językami, w których:

1) Istnieją pary wyrażeń takich, które między sobą nie są równokształtne ani równoznaczne;
2) Istnieją pary wyrażeń takich, które między sobą nie są równokształtne są równoznaczne;

3) Istnieją pary wyrażeń takich, które między sobą są równokształtne - lecz nie są równoznaczne;
4) Istnieją pary wyrażeń takich, które między sobą są równokształtne i zarazem równoznaczne.

Powyższe uwagi zilustruje my za pomocą przykładów zaczerpniętych z graficznego języka polskiego, podanych w tablicy 3.1.3.10.

Tablica 3.1.3.10		
Wyrażenie, w którym zawarte jest wyrażenie ***A***	Wyrażenie, w którym zawarte jest wyrażenie ***B***	Uwagi
I.	II.	III.
Adam lub Bolesław opuścili wykład ***A***	Adam oraz Bolesław opuścili wykład ***B***	Wyrażenie ***A*** nie jest równokształtne z wyrażeniem ***B*** i nie jest z nim równoznaczne
Adam i Bolesław opuścili wykład ***A***	Adam oraz Bolesław opuścili wykład ***B***	Wyrażenie ***A*** nie jest równokształtne z wyrażeniem ***B***, lecz oba wrażenia są równoznaczne
Euklides uczył filozofii w Megarze ***A***	Euklides uczył geometrii w Aleksandrii ***B***	Wyrażenie ***A*** jest równokształtne z wyrażeniem ***B***, lecz nie jest równoznacznie (chodzi tu o dwu różnych Euklidesów)
Arystoteles był twórcą logiki ***A***	Arystoteles był Grekiem ***A***	Wyrażenie ***A*** jest równokształtne i równoznacznie z wyrażeniem ***B***

Piśmiennictwo: *Greniewski H.* G.2.1.

3.1.4. Kolejność

W każdym języku oprócz równokształtności i równoznaczności mamy jeszcze do czynienia z kolejnością wyrażeń. W wypadku języka akustycznego - będzie to jedna tylko kolejność, mianowicie kolejność w czasie, inaczej natomiast przedstawia się sprawa w językach graficznych. Nie ma w tym nic dziwnego, gdyż wyrażenia języka akustycznego muszą być układane kolejno (wtedy, gdy się za pomocą takiego języka wypowiadamy) w obrębie jednowymiarowego czasu. Wyrażenia języka graficznego układamy natomiast na dwuwymiarowej powierzchni (zwykle na płaszczyźnie), gdzie mamy do czynienia z dużym wyborem różnych kolejności liniowych. Znane są między innymi następujące rodzaje kolejności stosowanych w różnych językach graficznych:

1) „od lewej do prawej”,
2) „od prawej do lewej”,
3) „z góry na dół”,
4) „z dołu do góry”.

W europejskich językach naturalnych graficznych stosuje się, jak wiadomo, kolejność „od lewej do prawej” (i „z góry na dół”, gdy kończy się wiersz), jednak nie konsekwentnie, gdyż przy zapisywaniu liczb stosujemy kolejność „od prawej do lewej” przyjętą od Arabów wraz z powszechnie i na co dzień u nas używanym pozycyjnym systemem dziesiętnym zapisywania liczb. O tym, że zapisując liczby w dziesiętnym systemie pozycyjnym stosujemy kolejność „od prawej do lewej” zamiast zwykłej europejskiej kolejności „od lewej do prawej”, najłatwiej było się przekonać obserwując pracę maszynistki (w czasie - gdy maszynopisanie nie zostało jeszcze zastąpione programami edycji tekstu działającym na komputerze), która w tekście maszynopisu ma umieścić kolumnę liczb zapisanych „równo pod sobą” (np. kolumnę liczb do zsumowania). W takim stadium pisania tempo pracy bardzo dobrej nawet maszynistki malało gwałtownie, wypisywała ona - bowiem najpierw jednostki, potem - po lewej stronie jednostek! - dziesiątki, następnie setki itd., każda zaś używana dawniej u nas maszyna do pisania była dostosowana do europejskiej kolejności pisania „od lewej do prawej”. Oczywiście, zarówno dziesiętność jak i

pozycyjność arabskiego systemu zapisywania liczb jest zupełnie niezależna od kolejności zapisu. Można by z powodzeniem zapisywać liczby w systemie pozycyjnym i dziesiętnym stosując zarazem kolejność „od lewej do prawej”, ale tego nie rozumiano w Europie w wieku X naszej ery, gdy nauczono się arabskiego systemu zapisywania liczb. W językach sztucznych (na przykład algebry i informatyki), z którymi mieliśmy do czynienia w szkole średniej, stosuje się również kolejność „od lewej do prawej", jednak z powyższym wyjątkiem przy zapisywaniu liczb. Natomiast w języku map geograficznych, który również poznaliśmy w szkole, mamy do czynienia z dwiema kolejnościami na raz (południe-północ, zachód-wschód). Podobnie - w języku wykresów statystycznych występują dwie kolejności. W językach graficznych sztucznych, graficznych mieszanych i językach formalnych, z którymi będziemy mieli do czynienia, stosować będziemy także kolejność „od lewej do prawej”. Bez względu jednak na to, jaka kolejność obowiązuje w danym języku, zawsze mają zastosowanie następujące reguły kolejności:

3.1.4.01. **Reguła**. Żadne wyrażenie nie poprzedza siebie.

3.1.4.02. **Reguła**. Jeżeli wyrażenie pierwsze poprzedza wyrażenie drugie, to nieprawdą jest, że wyrażenie drugie poprzedza wyrażenie pierwsze.

3.1.4.03. **Reguła**. Jeżeli wyrażenie pierwsze poprzedza wyrażenie drugie, zaś wyrażenie drugie poprzedza wyrażenie trzecie, to wyrażenie pierwsze poprzedza wyrażenie trzecie.

Łatwo zauważyć, że reguła 3.1.4.01 - oraz reguła 3.1.4.02, wydadzą się, fałszywe temu, kto nie odróżnia równokształtności wyrażeń od identyczności wyrażeń (podrozdział 3.1.1).

Piśmiennictwo: *Greniewski H.* G.2.1. , *Wilkosz W.* W.3.1, W.3.2.

3.1.5. ZASTĘPOWANIE I PODSTAWIANIE

Zajmiemy się teraz czynnościami, z którymi mamy często do czynienia przy budowaniu wyrażeń należących do danego języka. Są to czynności wyliczone w tytule niniejszego i następnego paragrafu.

3.1.5.00. **Wyjaśnienie**. Mówimy, że w wyrażeniu - ***A*** zastępujemy wyrażenie ***B*** wyrażeniem ***C***, zamiast mówić, że:

1) Wyrażenie ***B*** jest częścią wyrażenia ***A***;
2) Sporządzamy nowe wyrażenie (nazwijmy je „***D***”)w ten sposób, ze w zasadzie kopiujemy wyrażenie ***A*** bez żadnych zmian, oprócz jednej polegającej na tym, że zamiast kopiować wyrażenie ***B*** kopiujemy wyrażenie ***C***.

3.1.5.10. **Wyjaśnienie**. Mówimy, że do wyrażenia ***A*** podstawiamy za wyrażenie ***B*** wyrażenie ***C***, zamiast mówić, że w wyrażeniu ***A*** zastępujemy każde wyrażenie będące częścią wyrażenia ***B*** i zarazem równokształtne z wyrażeniem ***B*** - wyrażeniem równokształtnym z wyrażeniem ***C***.
Przykład. Weźmy pod uwagę poniższe wyrażenie (jakiegoś języka) oznaczone literą „***A***”, a w nim część oznaczoną literą „***B***”:

A											
0	1	x	y	z	0	0	x	y	z	0	1
		B									

oraz weźmy również pod uwagę poniższe wyrażenie (tegoż języka) literą „***C***”:

C		
α	β	γ

W wyrażeniu $\boldsymbol{A}$ zastępujemy wyrażenie $\boldsymbol{B}$ - wyrażeniem $\boldsymbol{C}$ i otrzymujemy wynik następujący:	Do wyrażenia $\boldsymbol{A}$ podstawiamy za wyrażenie $\boldsymbol{B}$- wyrażenie $\boldsymbol{C}$ i otrzymujemy wynik następujący:
$0\ 1\ \alpha\ \beta\ \gamma\ 0\ 0\ x\ y\ z\ 1\ 0$	$0\ 1\ \alpha\ \beta\ \gamma\ 0\ 0\ \alpha\ \beta\ \gamma\ 1\ 0$

W dalszych rozdziałach, gdy będziemy formułowali reguły czy dyrektywy podstawiania, będziemy w dyrektywach tych wyraźnie odróżniali od siebie:

- Wyrażenie, do którego podstawiamy,
- Wyrażenie, za które podstawiamy,
- Wyrażenie, które podstawiamy.

Wyrażenia te będą wymieniane albo wskazywane zwykle w wyżej ustalonej kolejności.

Piśmiennictwo: *Greniewski H*. G.2.1.

3.1.6. Dołączanie, skreślanie i odrywanie

3.1.6.00. **Wyjaśnienie**. Mówimy, że wyrażenie $\boldsymbol{C}$ powstaje przez dołączenie wyrażenie $\boldsymbol{A}$ do wyrażenia $\boldsymbol{B}$, zamiast mówić, że wyrażenie $\boldsymbol{C}$ daje się rozbić na dwie części $\boldsymbol{A_1}$, $\boldsymbol{B_2}$ spełniające wszystkie warunki następujące:

1) Wyrażenie $\boldsymbol{A_1}$ nie ma żadnej wspólnej części z wyrażeniem $\boldsymbol{B_1}$;
2) Między wyrażeniem $\boldsymbol{A}$ a wyrażeniem $\boldsymbol{B}$, nie tkwi żadne wyrażeni domyślne;
3) Jeżeli w języku, do którego należą wyrażenie $\boldsymbol{A_1}$, $\boldsymbol{B_2}$ występuje odmiana gramatyczna, to formy gramatyczne wyrażenia $\boldsymbol{A_1}$ i wyrażenia $\boldsymbol{B_1}$ są wzajemnie do siebie dopasowane;
4) Wyrażenie $\boldsymbol{A_1}$ jest równokształtne z wyrażeniem $\boldsymbol{A}$;
5) Wyrażenie $\boldsymbol{B_1}$ jest równokształtne z wyrażeniem $\boldsymbol{B}$.

Przykłady. Wyrażenie „ojciec Piotra” powstaje przez dołączenie wyrażenia „ojciec” do wyrażenia „Piotr”. Wyrażenie „Jan jest studentem” powstaje przez dołączenie wyrażenia „Jan” do wyrażenia „jest studentem”. Wyrażenie „prawdą jest, że dwa razy dwa równa się cztery” powstaje przez dołączenie wyrażenia „prawdą jest, że” do wyrażenia „dwa razy dwa jest cztery”. W algebrze szkolnej pisze się często zamiast iloczynu „$a \cdot b$”, krócej „ab”. Wobec tego wyrażenie „ab” nie powstaje przez dołączenie wyrażenia „a” do wyrażenia „b”, gdyż między odnośnymi literami tkwi domyślna kropka. Zauważmy jeszcze, że dołączane wyrażenia do wyrażenia - nie zawsze daje, jako wynik wyrażenie. Niekiedy wynikiem dołączenia jest twór do danego języka nienależący. Ściśle biorąc, należałoby odróżniać dołączanie lewostronne od dołączania prawostronnego. Nie będziemy jednak tego czynić, aby nie komplikować naszych rozważań.

3.1.6.01. **Wyjaśnienie**. Mówimy, że wyrażenie $\boldsymbol{C}$ powstaje z wyrażenia $\boldsymbol{A}$ przez skreślenie wyrażenia $\boldsymbol{B}$, zamiast mówić, że wyrażenie $\boldsymbol{A}$ powstaje przez dołączenie wyrażenia $\boldsymbol{B}$ do wyrażenia $\boldsymbol{C}$.

Przykład. Modne w czasach PRL wyrażenie „przodownik pracy” powstaje przez dołączenie wyrażenia „przodownik” do wyrażenia „praca”, natomiast wyrażenie „przodownik” powstaje z wyrażenia „przodownik pracy” przez skreślenie wyrażenia „praca”.

3.1.6.10. **Wyjaśnienie**. Mówimy, że wyrażenie $\boldsymbol{C}$ powstaje przez oderwanie wyrażenia $\boldsymbol{A}$ od wyrażenia $\boldsymbol{B}$ przy pominięciu wyrażenia $\boldsymbol{X}$, zamiast mówić, że:

1) Wyrażenie $\boldsymbol{C}$ powstaje z wyrażenia $\boldsymbol{B}$ przez, skreślenie wyrażenia $\boldsymbol{A}$ i skreślenie wyrażenia $\boldsymbol{X}$ i ponadto

2) Część wyrażenia *B* równokształtna z wyrażeniem ***A*** poprzedza według obowiązującej kolejności w danym języku, odnośną część wyrażenia ***B***, która jest równokształtna z wyrażeniem ***C***.

Przykład. Weźmy pod uwagę wyrażenia następujące:
(1) Jan ma gorączkę,
(2) Jeżeli Jan ma gorączkę, to Jan jest chory.
(3) Jan jest chory,
(4) Jeżeli..., to …
Jak łatwo zauważyć, w tym przykładzie wyrażenia (3) powstaje przez odjęcie wyrażenia (1) od wyrażenia (2) przy pominięciu wyrażenia (4).

Piśmiennictwo: *Greniewski H.* G.2.1.

3.2. RODZAJE WYRAŻEŃ WYSTĘPUJĄCYCH W JĘZYKACH NATURALNYCH

3.2.0. UWAGI WSTĘPNE

Zajmiemy się w tym rozdziale pewnego rodzaju „szufladkowaniem" wyrażeń dowolnego języka naturalnego. Będziemy się stale posługiwali językiem polskim (graficznym) - jako przykładem języka naturalnego. Znane jest ze szkoły średniej, mianowicie z lekcji gramatyki, pewne „poszufladkowanie" wyrażeń; mam tu na myśli tak zwany podział na części mowy. W logice potrzebny nam będzie jednak inny podział niż ten, który stosuje w gramatyce. Wyróżnimy trzy rodzaje wyrażeń w językach naturalnych:

1) Zdania,
2) Nazwy,
3) Funktory

Piśmiennictwo: *Greniewski H. G.2.1.*

3.2.1. ZDANIA

Pewne, ale nie wszystkie, wyrażenia językowe uważamy za zdania. W terminologii logiki uważamy za zdania jedynie te wyrażenia, które w gramatyce nazywają się „zdaniami oznajmującymi". Pytań, nakazów ani w zdaniowej formie wyrażonych życzeń - nie nazywamy „zdaniami" w terminologii logiki. Należy podkreślić, że nie chodzi tu o różnicę poglądów między językoznawcami a logikami, lecz jedynie o różnicę terminologiczną.

Odróżniamy zdania fałszywe od zdań prawdziwych. Zamiast mówić „*zdanie prawdziwe*", będziemy niejednokrotnie mówili krótko „*prawda*". Zamiast mówić „*zdanie fałszywe*", będziemy niejednokrotnie mówili krótko „*fałsz*". Stojąc na stanowisku logicznej teorii prawdy - powiemy jeszcze, że zdanie jest wtedy i tylko wtedy prawdziwe, jeżeli jest ono wiernym odbiciem rzeczywistości materialnej.

Zagadnienie istnienia zdań w teorii nierozstrzygalnej (patrz rozdział 2.7), które nie są ani fałszywe, ani prawdziwe mieliśmy okazję przedyskutować przy okazji omawiania *Logiki Pierwszego Rzędu*. Tutaj natomiast zajmiemy wyraźne stanowisko w innej doniosłej sprawie:

3.2.1.00. **Zasada**. Żadne zdanie nie jest zarazem fałszywe i prawdziwe.
Wprowadziliśmy już następujący podział zdań:

1) Zdania fałszywe,

2) Zdania prawdziwe.

Wprowadzimy teraz niezależnie od tego znanego nam podziału jeszcze drugi:

1) Zdania proste,
2) Zdania złożone.

3.2.1.10. **Wyjaśnienie**. „Zdanie proste", to takie i tylko takie zdanie, którego żadna część właściwa nie jest zdaniem.

Uwaga: Porównajmy powyższe wyjaśnienie z wyjaśnieniem mówiącym, że „*wyraz*" to takie i tylko takie wyrażenie, którego żadna część właściwa nie jest wyrażeniem. Łatwo zauważy, ze mamy tu do czynienia z daleko idącym podobieństwem pojęcia *wyrazu* i pojęcia *zdania prostego*. Żaden wyraz nie zawiera w sobie części właściwej będącej wyrażeniem i żadne zdanie proste nie zawiera w sobie części właściwej będącej zdaniem.

3.2.1.11. **Wyjaśnienie**. „*Zdanie złożone*", to takie i tylko takie zdatne, które zawiera w sobie, chociaż jedną część właściwą będącą zdaniem.

Przykład. Weźmy teraz pod uwagę trzy zdania poniższe należące do polskiego języka graficznego:

(1) Adam Mickiewicz był Polakiem.
(2) Adam Mickiewicz był Polakiem oraz Juliusz Słowacki był Polakiem.
(3) Adam Mickiewicz i Juliusz Słowacki byli Polakami.

W myśl wyjaśnienia 3.2.1.10, zdanie (1), a także zdanie (3) - są zdaniami prostymi, natomiast w myśl 3.2.1.11, zdanie (2) - jest zdaniem złożonym.

W oparciu o wyjaśnienia 3.2.1.10 i 3.2.1.11, otrzymujemy bezpośrednio reguły poniższe:

3.2.1.12. **Reguła**. Żadne zdanie nie jest zarazem proste i złożone.

3.2.1.13. **Reguła**. Każde zdanie jest proste albo złożone.

Kombinując ze sobą oba wyżej wprowadzone podziały ogółu zdań danego języka otrzymamy nowy podział (mianowicie czwórpodział) uwidoczniony w tablicy 3.2.1.20.

Tablica 3.2.1.20		
Rodzaj zdań	Rodzaj zdań	
	Fałszywe	Prawdziwe
I.	*II.*	*III.*
Proste	Zdania fałszywe proste	Zdania prawdziwe proste
Złożone	Zdania fałszywe złożone	Zdanie prawdziwe złożone

Zapytajmy teraz, czy znając fałszywość, ewentualnie prawdziwość zdań prostych możemy orzec - nie odwołując się bezpośrednio do doświadczenia, że zdanie złożone zbudowane w dany sposób z tych zdań prostych jest - fałszywe czy prawdziwe. Niektóre z takich zagadnień dają się rozwiązać pozytywnie, to znaczy ze w niektórych przypadkach wartość logiczna (to jest fałszywość czy też prawdziwość) zdań prostych sama wyznacza wartość logiczną zdania złożonego, zbudowanego z tych zdań prostych. Takim właśnie wyznaczeniem wartości logicznej niektórych zdań złożonych zajmuje się między innymi logika zdań, którą poznaliśmy w części 2.

Piśmiennictwo: *Greniewski H*. G.2.1.

3.2.2. NAZWY

Po zdaniach zainteresujemy się drugim rodzajem wyrażeń - mianowicie nazwami. Można podawać, jako przykłady nazw nie tylko rzeczowniki, ale również przymiotniki (później się przekonamy, że i zaimki). Nazwami (w polskim języku graficznym) będą wyrażenia: Sokrates, człowiek, zwierzę, roślina, biały, niebieski, czerwony.

Odróżnimy najpierw dwa rodzaje nazw:
1) Nazwy puste,
2) Nazwy niepuste.

3.2.2.00. **Wyjaśnienie**. Mówimy, że nazwa jest pusta, zamiast mówić, że nie oznacza ona żadnego przedmiotu.

3.2.2.01. **Wyjaśnienie**. Mówimy, że nazwa jest niepusta, zamiast mówić, że oznacza ona przynajmniej jeden przedmiot.
Przykłady. Weźmy pod uwagę wyrażenie „*kwadratowe koło*". Nazwa „*kwadratowe koło*" nie oznacza żadnego przedmiotu, ponieważ nie istnienie ani jedno - kwadratowe koło, więc jest nazwą pustą. Jako inny przykład nazwy pustej można podać wyraz „*nic*".

Jakąkolwiek nazwę pustą weźmiemy, to zawsze prawdą jest, że nie oznacza ona żadnego przedmiotu. Można by wobec tego zapytać, czy nazwy puste są w ogóle potrzebne? Przecież nazwy są po to w języku, aby coś oznaczały, a żadna nazwa pusta nic nie oznacza. Słuchacz na wykładzie logiki, gdy, mowa o nazwach pustych, odnosi takie wrażenie, jak postacie powieści *Gogola Martwe Dusze* podczas podpisywania umowy sprzedaży nieżyjących już chłopów. Na tego rodzaju wątpliwości odpowiemy krótko: wówczas, gdy mamy do czynienia z działem logiki zwanym rachunkiem nazw, przekonamy się o przydatności nazw pustych, przekonamy się, że mają one w logice podobne znaczenie, jak cyfra zero w algebrze szkolnej. Jako nazwę pustą można również traktować niezainicjowaną wartość atrybutu w bazach danych – wartość tę, oznaczamy zwykle symbolem „*null*". Dalej - zajmiemy się jeszcze nazwami jednostkowymi i nazwami ogólnymi.

3.2.2.20. **Wyjaśnienie**. Mówimy, że nazwa jest jednostkowa, zamiast mówić, że oznacza ona dokładnie jeden przedmiot.
Przykład. Nazwą jednostkową jest nazwa „Arystoteles".

W praktyce dawanie przykładów i budowanie nazw jednostkowych jest częstokroć niełatwe. Trudności tworzenia nazw jednostkowych można zrozumieć, jeżeli obejrzy się jakąś szczegółowo prowadzoną ewidencję osobową. Na karcie personalnej znajdziemy tam kolejno: 1) imię, 2) ewentualnie drugie imię, 3) nazwisko, 4) imię ojca, 5) imię matki, 6) nazwisko matki z domu, 7) miejsce urodzenia i 8) datę urodzenia. Należy zdać sobie sprawę, że ten długi zapis obejmujący tyle danych, razem dopiero, jako całość, tworzy nazwę jednostkową. Wkrótce poznamy pewne inne sposoby budowania nazw jednostkowych (podrozdział 3.7.2).

Niekiedy potocznie używa się nazw jednostkowych, co do których, można mieć wątpliwości, jaki mianowicie przedmiot oznaczają. Oto przykład: Prawdą jest, ze Adam Mickiewicz był mężczyzną, który nie ukończył 25 lat życia, i prawdą jest, ze Adam Mickiewicz był mężczyzną, który ukończył 25 lat życia. Nie zdajemy sobie na ogół sprawy, że w każdym z obu przytoczonych zdań nam, ta równokształtna z „Adam Mickiewicz" oznacza inny przedmiot.

Każde ciało materialne[37], w szczególności każdy człowiek, nie tylko bowiem, jest przestrzennie rozciągły, ale również trwa w czasie i ma pewien rozmiar czasowy. W naszym przykładzie należy odróżnić dwa przedmioty: Adam Mickiewicz od urodzenia do ukończenia 25 lat życia i drugi przedmiot: Adam Mickiewicz od ukończenia 25 lat życia do swej śmierci. Każdy z tych obu przedmiotów jest zresztą częścią trzeciego przedmiotu, mianowicie Adama Mickiewicza, traktowanego ciągłości czasowej (wykres 3.2.2.21).

Wykres 3.2.2.21. Adam Mickiewicz							
Adam Mickiewicz (w całej rozciągłości czasowej)							
Adam Mickiewicz od urodzenia, do ukończenia 25 lat życia				Adam Mickiewicz od ukończeniu 25 lot życia do swojej śmierci			
1800	1810	1820	1823	1830	1840	1850	1855
Data urodzenia 24.XII-1796			Data ukończenia 25 lat życia 24 XII 1823				Data śmierci 26 XI 1855

3.2.2.22. **Wyjaśnienie**. Mówimy, że nazwa jest ogólna, zamiast mówić, że nazwa ta oznacza więcej niż jeden przedmiot.
Przykłady. Nazwy ogólne: człowiek, mężczyzna, kobieta, mebel, zielony, czerwony, rozciągły.

Pozornie łatwe do zrozumienia wyjaśnienie 3.2.2.22 niekiedy może wywołać nieporozumienie[38]. W tablicy 3.1.3.10 wykorzystaliśmy fakt - polegający na tym, że było dwu *Euklidesów* i że w konsekwencji oznacza raz *Euklidesa-filozofa*, innym zaś razem oznacza *Euklidesa-matematyka*. Czyżby, więc nazwa „*Euklides*" była nazwą ogólną? I na pierwszy rzut oka, że tak jest i to w zgodzie (pozornej, jak z wyjaśnieniem 3.2.2.22. Aby sprawę tę należycie wyjaśnić, weźmy napis następujący:

(1) $Euklides.$

Napis ten nie jest wyrażeniem. W ogóle żaden napis (a także żadna seria dźwięków, żaden ruch ręki ludzkiej) nie jest wyrażeniem. Napis (ewentualnie seria dźwięków, ewentualnie ruch ręki ludzkiej) może być jedynie stroną materialną jakiegoś wyrażenia. Kojarząc napis (1), jako stronę materialną z pewnym rozumieniem, czyli z pewną stroną znaczeniową, otrzymujemy w wyniku pewne wyrażenie - będące nazwą jednostkową Euklidesa-filozofa. Kojarząc natomiast napis, (1) jako stronę materialną z innym rozumieniem (stroną znaczeniową) - otrzymujemy w wyniku pewne wyrażenie będące nazwą jednostkową Euklidesa-matematyka. Obie te nazwy są różnymi wrażeniami, chociaż mają tę samą stronę materialną, którą jest napis (1).

Twierdzimy to w myśl zasady mówiącej, że wszystkie i tylko te wyrażenia są identyczne, które mają nie tylko tę samą stronę materialną, ale i tę samą stronę znaczeniową. Tak więc:

1) Napis (1) nie jest wyrażeniem;
2) Tym bardziej nie jest on nazwą (bo każda nazwa jest wyrażeniem);
3) Tym bardziej nie jest on nazwą ogólną (bo każda nazwa ogólna jest nazwą).

Nasze rozróżnienia dotyczące rodzajów nazw można by przedstawić graficznie (rysunek 3.2.2.23).

[37] Niejeden humanista, a także niejeden fizyk czy chemik zapyta, po co piszemy „ciało materialne" zamiast po prostu „ciało"? Czyżby autorzy wierzyli w istnienie ciał niematerialnych? Wyjaśniamy, ze w matematyce używa się napisu „ciało" oraz jego odpowiednika akustycznego w innym zgoła rozumieniu i z tego właśnie powodu (tj. przez wzgląd na czytelników matematyków) piszemy rozwlekle wprawdzie, lecz dobitnie „ciało materialne", zamiast po prostu „ciało" w zwykłym, potocznym rozumieniu.
[38] Zwrócił mi na to uwagę Tadeusz Kotarbiński.

Nazwy
_____|_____
Puste Niepuste
_____|_____
Jednostkowe Ogólne
Rysunek 3.2.2.23

3.2.2.30. **Wyjaśnienie**. Mówimy, że pewien przedmiot jest desygnatem pewnej nazwy, zamiast mówić, że nazwa ta oznacza ten przedmiot.

Opierając się na wyjaśnieniach niniejszego paragrafu przyjmujemy reguły następujące:

3.2.2.31. **Reguła**. Żaden przedmiot nie jest desygnatem jakiejkolwiek nazwy pustej.

3.2.2.32. **Reguła**. Jeżeli dwa (różne) przedmioty są desygnatami tej samej nawy, to nazwa ta jest ogólna.

Omówimy jeszcze pokrótce pewne zagadnienie, mianowicie zagadnienia nazw wyrażeń. Nazywać możemy każdy dobrze wyodrębniony przedmiot, czemu więc nie mielibyśmy nadawać nazw wyrażeniom jednego czy wielu języków? Aby wypowiadać zdania o przedmiotach, potrzeba nam zazwyczaj nazw tych przedmiotów; nazwy te wchodzą, zatem w skład odnośnych zdań. Gdy chcemy mówić, pisać, czy tez stawiając sprawę ogólnie – wypowiadać się na temat wyrażeń, potrzebne nam są zwykle nazwy tych wyrażeń. Zazwyczaj nazwę wyrażenia tworzymy ujmując to wyrażenie w cudzysłów (używanych jednak w językach naturalnych nie tylko w tym celu). Ten nader wśród logików rozpowszechniony sposób tworzenia nazw wyrażeń nasuwa wątpliwość: Czy przez postawienie cudzysłowu otrzymujemy nazwę jednostkową wyrażenia objętego tym cudzysłowem, czy też nazwę ogólną oznaczającą zarówno wyrażenie objęte tym cudzysłowem, jak i każde wyrażenie równokształtne i zarazem równoznaczne? Wydaje nam się, ze jedynie rozsądne jest stanowisko, wedle, którego nazwa wyrażenia utworzona za pomocą cudzysłowu jest zawsze nazwa ogólną. Stanowisko to precyzujemy w następującym wyjaśnieniu:

3.2.2.40. **Wyjaśnienie**. Biorąc wyrażenie w cudzysłów otrzymujemy nazwę ogólną, oznaczającą każde i tylko takie wyrażenie, któż kształtne i zarazem równoznaczne z wyrażeniem objętym cudzysłowem.

Teraz pewna konsekwencja zajętego przez nas stanowiska. Niektórzy mówią tak: Człowiek jest ssakiem. Jasne, że taki sposób mówienia jest niepoprawny, gdyż nie wiadomo, co mówiący miał na myśli: czy chciał przez to powiedzieć, że każdy człowiek jest ssakiem, co jest, oczywiście prawdą, czy też, ze pewien wskazany przez niego człowiek jest ssakiem - co jest też prawdą, ale nader banalną, czy może miał na myśli, że tylko pewien człowiek jest ssakiem - co z kolei jest fałszem. Najprawdopodobniej mówiący miał na myśli, że każdy człowiek jest ssakiem, toteż powinien powiedzieć - każdy człowiek jest ssakiem, zamiast bąknąć: człowiek jest ssakiem. W konsekwencji uwag powyższych i wyjaśnienia 3.2.2.40, nie powinien pisać „*Arystoteles*" (jest to w polskim języku graficznym nazwą *Arystotelesa*), tylko powinniśmy pisać: każdy „*Arystoteles*". Jeżeli to sformułowanie wydaje się komuś rażące czy dziwaczne, może on wówczas równie dobrze pisać: Każde wyrażenie równokształtne i zarazem równoznaczne z wyrażeniem: *Arystoteles* - w polskim języku graficznym jest nazwą *Arystotelesa*.

Piśmiennictwo: *Greniewski H.* G.2.1.

3.2.3. FUNKTORY ZDANIOTWÓRCZE OD ARGUMENTÓW ZDANIOWYCH

Zaznajomiliśmy się już z dwoma rodzajami wyrażeń język naturalnego mianowicie ze zdaniami i nazwami. Zajmiemy się teraz funktorami. Odróżniać zresztą będziemy różne rodzaje funktorów, każdy z tych rodzajów opiszemy osobno nie siląc się na wprowadzenie ogólnego pojęcia funktora[39].

3.2.3.00. **Wyjaśnienie**. Mówimy, że wyrażenie jest funktorem zdaniotwórczym od jednego argumentu zdaniowego, zamiast mówić, że wyrażenie to dołączone do dowolnego zdania daje łącznie nowe zdanie.
Przykłady. Wyrażenie „$prawdą\ jest, że$" - jest funktorem zdaniotwórczym jednego argumentu zdaniowego, gdyż napisane przed zdaniem daje łącznie zdanie. Na przykład: prawdą jest, że Warszawa jest stolicą Polski. Wyrażenie „$fałszem\ jest, że$" jest funktorem zdaniotwórczym od jednego argumentu zdaniowego, gdyż napisane przed zdaniem daje łącznię z tym zdaniem nowe zdanie, na przykład: fałszem jest, że Paryż leży nad Wisłą. Wyrażenie „$wierzę, że$" jest funktorem zdaniotwórczym od jednego argumentu zdaniowego, ponieważ wyrażenie to dołączone do jakiegokolwiek zdania daje w wyniku nowe zdanie, na przykład: wierzę, że przestrzeń, w której żyjemy, jest euklidesowa.

3.2.3.01. **Wyjaśnienie**. Mówimy, że wyrażenie jest funktorem zdaniotwórczym od dwu argumentów zdaniowych, zamiast mówić, że wyrażeń to dołączone do dowolnych dwu zdań daje nowe zdanie.
Przykłady. Jeśli połączymy dwa zdania spójnikiem „lub" czy też spójnikiem „$albo$", czy też spójnikiem „$oraz$", ta oczywiście otrzymamy znowu zdanie (zdanie złożone).Trzeba tu jeszcze dodać, że spójniki nie wyczerpują całego repertuaru funktorów zdaniotwórczych od dwóch argumentów zdaniowych. Również wyrażenie „$jeżeli \ldots to$" jest takim funktorem. Funktorem zdaniotwórczym od dwóch argumentów zdaniowych jest także wyrażenie „$wtedy\ i\ tylko\ wtedy, jeżeli$".

3.2.3.02. **Wyjaśnienie**. Mówimy, że wyrażenie jest funktorem zdaniotwórczym od n argumentów zdaniowych (gdzie n jest dodatnią liczbą naturalną), zamiast mówić, że wyrażenie to dołączone do n dowolnych zdań daje łącznie nowe zdanie.
Przykład. Funktorem zdaniotwórczym od trzech argumentów zdaniowych jest wyrażenie - „$albo \ldots\ albo \ldots\ albo$".

Zauważamy jeszcze, że funktory zdaniotwórcze od dużej liczby argumentów zdaniowych nie mają większego znaczenia w językach naturalnych, natomiast bywają one nader użyteczne w niektórych językach sztucznych. W związku z dołączaniem, o którym mowa we wszystkich trzech wyjaśnieniach paragrafu niniejszego, dobrze będzie zauważyć, że określiliśmy już w wyjaśnieniu 1.1.6.00, na czym dołączanie to polega.

Piśmiennictwo: *Greniewski H.* G.2.1. , *Kotarbiński T.* K.5.2.

[39] Pojęcie funktora wprowadził do logiki zmarły w okresie międzywojennym Stanisława Leśniewski, profesor Uniwersytetu Warszawskiego. Sam wyraz „funktor" został wprowadzony przez logika polskiego Tadeusza Kotarbińskiego. Jak pokażemy, w części czwartej, pojęcie funktora ma bardzo istotne znaczenie dla informatyki, a w szczególności dla tzw. podejścia obiektowego do programowania komputerów.

3.2.4. FUNKTORY ZDANIOTWÓRCZE OD ARGUMENTÓW NAZWOWYCH

Po funktorach zdaniotwórczych od argumentów zdaniowych zainteresujemy się teraz funktorami również zdaniotwórczymi, lecz od nazw argumentów nazwowych.

3.2.4.00. **Wyjaśnienie**. Mówimy, że wyrażenie jest funktorem zdaniotwórczym od jednego argumentu – jednostkowo nazwowego, zamiast mówić, że wyrażenie tu dołączone do nazwy pustej lub jednostkowej tworzy łącznie z nią zdanie.
Przykłady. Funktorem zdaniotwórczym od jednego argumentu jednostkowo nazwowego jest wyrażenie „*pracuje*". Pisząc to wyrażenie obok nazwy „*Sokrates pracuje*". Tak samo dopisując obok tego funktora nazwę jednostkową „*Arystoteles*" otrzymamy, jako wynik dołączenia zdanie „*Arystoteles pracuje*". Zauważmy jeszcze, że jeżeli do naszego przykładowego funktora dołączyć nazwę pustą, na przykład „*kwadratowe koło*"– to wynikiem dołączenia również będzie zdanie, (co prawda fałszywe, ale to nam nic tu nie przeszkadza) „*kwadratowe koło pracuje*". Dołączając zaś do naszego funktora nazwę ogólną, na przykład „*człowiek*", otrzymamy, jako wynik dziwoląg „*człowiek pracuje*" (nie wiadomo tu, czy chodzi o każdego człowiek a, czy o przynajmniej niektórych, czy najwyżej niektórych czy też o pewnego wskazanego człowieka).

Funktorami zdaniotwórczymi od jednego argumentu jednostkowego są też wszystkie następujące wyrażenia polskiego języka graficznego: odpoczywa, żyje, umiera, jest człowiekiem, jest ssakiem jest kręgowcem, jest robotnikiem.

3.2.4.01. **Wyjaśnienie**. Mówimy, że wyrażenie jest funktorem zdaniotwórczym od jednego argumentu ogólno nazwowego, zamiast mówić, że wyrażenie to dołączone do dowolnej nazwy zawsze daje łącznie z nią zdanie.
Przykład. Funktorem zdaniotwórczym od jednego argumentu ogólno zdaniowego jest wyrażenie „*pewien … pracuje*". Istotnie, przez dołączenie do naszego przykładowego wyrażenia nazwy „*człowiek*" otrzymamy zdanie „*pewien człowiek pracuj*". Tak samo dołączenie do naszego wyrażenia przykładowego nazwy ogólnej „*mężczyzna*" daje, jako wynik zdanie „*pewien mężczyzna pracuje*". Ale to nie wszystko, dobraliśmy, bowiem dopiero dwie nazwy, (przy czym obie były ogólne) i okazało się, że przez dołączenie otrzymamy zdanie; nie wiemy jeszcze wcale, czy zawszę otrzymamy wynik dołączenia taki, jakiego wymaga od nas wyjaśnienie 3.2.4.01. Czyńmy, więc dalsze próby! Łatwo dobrać niejedną jeszcze nazwę tak, żeby wszystko poszło gładko. Weźmy na przykład pod uwagę nazwę ogólną „*robotnik*"; jako wynik dołączenia otrzymamy znowu zdanie „*pewien robotnik pracuje*". Spróbujmy teraz dołączyć nazwę jednostkową, na przykład „*Arystoteles*", w wyniku otrzymamy dość dziwaczne wprawdzie wyrażenie, które jednak bez trudu można uznać za zdanie. Mamy tu na myśli wyrażenie „*pewien Arystoteles pracuje*". A teraz spróbujemy dołączyć nazwę pustą, na przykład nazwę „*Archimedes nie będący Archimedesem*". Wynikiem dołączenia będzie znowu zdanie (fałszywe, dziwaczne, ale zdanie).

3.2.4.10. **Wyjaśnienie**. Mówimy, że wyrażenie jest funktorem zdaniotwórczym od dwu argumentów jednostkowo nazwowych, zamiast mówić, że wyrażenie to dołączone do dwu nazw, z których każda jest pusta lub jednostkowa zawsze tworzy łącznie z nimi zdanie.
Przykład. Funktorem zdaniotwórczym od dwóch argumentów jednostkowo nazwowych jest wyrażenie „*jest mniejszy od*". Weźmy teraz pod uwagę dwie nazwy, nazwę „*Piotr*" i nazwę „*Jan*". Wypiszmy te nazwy łącznie ze wspomnianym funktorem, a otrzymamy w wyniku zdanie następujące: *Piotr jest mniejszy od Jana*. (Sprawę deklinacji pomijamy, jako omówioną w wyjaśnieniu 3.1.1.01).

3.2.4.11. **Wyjaśnienie**. Mówimy, że wyrażenie jest funktorem zdaniotwórczym od dwu argumentów ogólno nazwowych, zamiast mówić, że wyrażenie to dołączone do dwu dowolnych nazw daje zawszę zdanie.
Przykład. Funktorem zdaniotwórczym od dwu argumentów ogólno-nazwowych jest wyrażenie „*każdy … jest mniejszy od każdego …*”. Dołączmy dwie nazwy, mianowicie „*lis*” i „*człowiek*”, a otrzymamy, jako wynik zdanie „*każdy lis jest mniejszy od każdego człowieka*”.

3.2.4.20. **Wyjaśnienie**. Mówimy, że wyrażenie jest funktorem zdaniotwórczym od n argumentów jednostkowo nazwowych, zamiast mówić, że wyrażenie to dołączone do n nazw, z których każda jest pusta lub jednostkowa, daje zawszę zdanie.
Przykład. Funktorem zdaniotwórczym od trzech argumentów nazwowych jest wyrażenie „*i … są rodzicami …*”. Z funktora tego i trzech nazw następujących: Adam, Czesława, Bolesław, budujemy zdanie „*Adam i Czesława są rodzicami Bolesława*”. (Znowu pomijamy sprawę deklinacji, jako już wyjaśnioną).

3.2.4.21. **Wyjaśnienie**. Mówimy, że wyrażenie jest funktorem zdaniotwórczym od n, argumentów ogólno nazwowych, zamiast mówić, że wyrażenie to dołączone do n dowolnych nazw - daje zawsze zdanie.

Piśmiennictwo: *Greniewski H*. G.2.1.

3.2.5. Funktory nazwotwórcze od argumentów nazwowych

Od funktorów zdaniotwórczych przejdziemy teraz do funktorów nazwotwórczych.

3.2.5.00. **Wyjaśnienie**. Mówimy, że wyrażenie jest funktorem nazwotwórczym od n argumentów jednostkowo nazwowych, zamiast mówić, że po dołączeniu do tego wyrażenia n nazw pustych lub jednostkowych otrzymujemy zawsze w wyniku nową nazwę pustą lub jednostkową.
Przykład. Funktorem jednostkowo-nazwotwórczym od jednego argumentu jednostkowo nazwowego jest każde z wyrażeń: ojciec, matka. Jeżeli do funktora „*ojciec*” dołączyć nazwę jednostkową, na przykład „*Jan*”, to otrzymamy w wyniku nazwę jednostkową „*ojciec Jana*”.

3.2.5.10. **Wyjaśnienie**. Mówimy, że wyrażenie jest funktorem nazwotwórczym od n argumentów ogólno nazwowych, zamiast mówić, że po dołączeniu do tego wyrażenia n nazw dowolnych zawsze otrzymujemy w wyniku nazwę pustą lub jednostkową.
Przykłady. Funktorami jednostkowo nazwotwórczymi od jednego argumentu ogólno nazwowego są wyrażenia następujące: *najmniejszy*, *największy*, *najniższy*, *najwyższy*, jeżeli jest pewne, że wśród ogółu przedmiotów nie istnieje więcej niż jeden przedmiot *najmniejszy*, nie istnieje więcej niż jeden przedmiot *największy*, np. jeżeli do funktora „*najniższy*” dołączymy nazwę ogólną „*człowiek*”, to otrzymamy w wyniku nazwę jednostkową „*najniższy człowiek*”. Jeżeli do funktora „*największy*” dołączymy nazwę ogólną „*liczba naturalna*”, to otrzymamy, jako wynik nazwę pustą „*największa liczba naturalna*”. (Nazwa ta jest pusta, gdyż nie istnieje wśród liczb naturalnych liczba największa).

3.2.5.20. **Wyjaśnienie**. Mówimy, że wyrażenie jest funktorem ogólno-nazwotwórczym od n argumentów jednostkowo nazwowych, zamiast mówić, że po dołączeniu do tego wyrażenia n nazw pustych lub jednostkowych zawszę otrzymamy w wyniku nazwę.
Przykłady. Funktorami ogólno-nazwotwórczymi od jednego argumentu jednostkowo nazwowego są wyrażenia następujące: *dziecko*, *syn*, *córka*, *wnuk*, *wnuczka*, *prawnuk*. Jeżeli do

funktora, „*dziecko*" dołączyć nazwę jednostkową „*Piotr*", otrzymamy w wyniku nazwę „*dziecko Piotra*". Jeżeli Piotr nie ma dzieci, to nazwa ta jest pusta, jeżeli Piotr ma jedno tylko dziecko, to nazwa ta jest jednostkowa. Jeżeli wreszcie Piotr ma więcej niż jedno dziecko, to nazwa ta jest ogólna.

3.2.5.30. **Wyjaśnienie**. Mówimy, że wyrażenie jest funktorem ogólno nazwotwórczym od n argumentów ogólno nazwowych, zamiast mówić, że po dołączeniu do tego wyrażenia n dowolnych nazw otrzymamy w wyniku nazwę.
Przykłady. Funktorami ogólno-nazwotwórczymi od jednego argumentu ogólno-nazwowego są wyrażenia: *człowiek dorosły tej wagi, co …* ; *człowiek dorosły tego wzrostu, co …* .

Dołączając do funktora „*człowiek dorosły tej wagi, co …*'' nazwę ogólną „*dwuletnie dziecko*" otrzymamy, jako wynik nazwę (zapewne pustą) „*człowiek dorosły tej wagi, co dwuletnie dziecko*". Dołączają do funktora „*człowiek dorosły tego wzrostu, co …*" nazwę jednostkową „*Tadeusz Kościuszko w chwili ukończenia lat dwudziestu*" otrzymamy, jako wynik dołączenia nazwę (zapewne ogólną) „*człowiek dorosły tego wzrostu, co Tadeusz Kościuszko w chwili ukończenia lat dwudziestu*''.

Poznaliśmy - zatem, w podrozdziale niniejszym, cztery rodzaje funktorów nazwo twórczych od argumentów nazwowych, mianowicie:
1) Funktory jednostkowo-nazwotwórcze od argumentów jednostkowo nazwowych.
2) Funktory jednostkowo-nazwotwórcze od argumentów ogólno-nazwowych.
3) Funktory ogólno-nazwotwórcze od argumentów jednostkowo nazwowych.
4) Funktory ogólno-nazwotwórcze od argumentów ogólno-nazwowych.

Trzeba jeszcze dodać, że dalecy jesteśmy od wyczerpania w ten sposób wszystkich funktorów nazwo twórczych od argumentów nazwowych, mimo to jednak dalej tego tematu - rozwijać nie będziemy.

Piśmiennictwo: *Greniewski H.* G.2.1.

3.2.6. INNE FUNKTORY

Omówiliśmy kilka rodzajów funktorów, mianowicie:
1) Funktory zdaniotwórcze od argumentów zdaniowych;
2) Funktory zdaniotwórcze od argumentów nazwowych;
3) Funktory nazwotwórcze od argumentów nazwowych.

Czytelnikowi nasunie się niewątpliwie pytanie, czy istnieją nazwotwórcze od argumentów zdaniowych? Odpowiemy krótko, że istnieją, lecz omawianie ich w naszym krótkim kursie logiki, w zestawieniu z tematyką informatyki, nie jest potrzebne. Gdybyśmy zaś do naszej listy rodzajów funktorów dopisali funktory nazwotwórcze od argumentów zdaniowych, czy wtedy mielibyśmy wyliczone wszystkie rodzaje funktorów? Na to pytanie wypadnie nam odpowiedzieć negatywnie, gdyż istnieją jeszcze funktory zdaniotwórcze od argumentów funktorowych, na przykład funktor „*jeżeli … to*" (użyty międzyzdaniowo) dołączony do funktorów „*śpi*" i „*odpoczywa*" daje w wyniku zdanie, „*jeżeli śpi, to odpoczywa*". Czy to np. wszystko? Otóż nie, w niejednym języku odnaleźć można obok funktorów zdaniotwórczych i nazwotwórczych jeszcze innego rodzaju funktory, mianowicie – funktory funktorotwórcze.

3.2.6.00. **Wyjaśnienie**. Mówimy, że wyrażenie jest funktorotwórczym od argumentów danego rodzaju, zamiast mówić, że wyrażenie to dołączone do swych argumentów daje łącznie nowy funktor.
Przykłady. Funktorem funktor twórczym bywa wyraz: „*nie*"(obok licznych innych ról, jakie spełnia w języku polskim). Weźmy pod uwagę wyrażenie „*jest prawdą, że …*". Wiemy, ze jest to funktor (mianowicie funktor zdaniotwórczy od jednego argumentu zdaniowego). Dołączmy do tego funktora wyraz „*nie*", a otrzymamy w wyniku wyrażenie „*nie jest prawdą, że …*". Otrzymaliśmy, więc w wyniku dołączenia nowy funktor zdaniotwórczy od jednego argumentu zdaniowego.
Każde z wyrażeń: *każdy*, *tylko pewien*, *przynajmniej pewien* – *np*., jest także funktorem funktor twórczym.

Piśmiennictwo: *Greniewski H.* G.2.1.

3.2.7. WYRAŻENIA OKAZJONALNE

W językach naturalnych, zarówno graficznych jak i akustycznych, występują często wyrażenia, które same przez się nie mają ustalonego rozumienia. Nabierają one rozumienia zależnie od okoliczności, w których są użyte; na przykład zależnie od osoby, która dane zdanie wypowiada lub pisze. Niekiedy zaś znaczenie takiego wyrażenia zostaje ustalone przez gest (np. wskazanie ręką osoby, wypowiadającej lub piszącej dane zdanie). Wreszcie rozumienie takiego wyrażenia wyznaczają niekiedy wyrażenia sąsiednie. Takie i tylko takie wyrażenia nazywamy właśnie okazjonalnymi.

3.2.7.00. **Wyjaśnienie**. Mówimy, że dane wyrażenie jest okazjonalne, zamiast mówić, że wyrażenie to jest tak czy inaczej rozumiane w danym języku zależnie od okoliczności, w których zostaje ono użyte.
Uwaga: W językach naturalnych spotykamy wyrażenia okazjonalne będące zdaniami, nazwami lub funktorami.
Przykłady. Zdaniem okazjonalnym jest wyraz „*tak*", ma on takie czy inne rozumienie w zależności od pytania, po którym wyraz ten następuje. Wśród rożnych nader ról spełnianych w języku polskim przez wyraz „*nie*" bywa on również zdaniem okazjonalnym; rozumienie jego jest uzależnione od poprzedzającego bezpośrednio pytania. Nazwami okazjonalnymi są wyrażenia: *ja*, *ty*, *on*, *tu*, *tam*, *tutejszy*, *tamtejszy*, *teraźniejszy*, *przeszły*, *przyszły*. Okazjonalnymi funktorami zdaniotwórczymi od jednego argumentu jednostkowo zdaniowego są wyrażenia*: czyni to*, *czyni tamto*, *czyni tak*. Podamy jeszcze jeden przykład (tym razem nieco bardziej skomplikowany) funktora okazjonalnego. Funktorem tym jest wyrażenie „… *porusza się* …". Jak wiadomo ruch jest względny, żadne ciało nie porusza się w jakimś absolutnym sensie, lecz względem jakiegoś układu odniesienia?

W konsekwencji mamy do czynienia z funktorem zdaniotwórczym od dwu argumentów nazwowych „… *porusza się względem* …", występującym na przykład w zdaniu „*Słońce porusza się względem danego układu inercyjnego*". Oprócz funktora zdaniotwórczego od dwóch argumentów nazwowych „… *porusza się względem* …" używamy też okazjonalnego funktora zdaniotwórczego od jednego argumentu nazwowego „ … *porusza się*", występującego na przykład w zdaniu „*Słońce porusza się*". Funktor ten ma takie lub inne rozumienie w zależności od tego, jaki układ odniesienia jest ustalony przez sąsiadujące zdania. Okazjonalnym funktorem jednostkowo-nazwotwórczym od jednego argumentu ogólno-nazwowego jest wyraz „*ten*", choćby, dlatego, że

wyraz „*człowiek*" jest nazwą ogólną, a wyrażenie „*ten człowiek*" jest jednostkową nazwą okazjonalną.

Piśmiennictwo: *Greniewski H.* G.2.1. , *Kotarbiński T*. K.5.1, K.5.2.

3.3. RODZAJE WYRAŻEŃ WYSTĘPUJĄCYCH W JĘZYKACH SZTUCZNYCH I MIESZANYCH

3.3.0. UWAGI WSTĘPNE

W językach sztucznych i mieszanych znajdujemy obok wyrażeń spotykanych w językach naturalnych, mianowicie zdań, nazw i funktorów, także innego rodzaju wyrażenia, niespotykane w ogóle w językach naturalnych: zmienne (w przypadku języka XML, są to *elementy* i *atrybuty* – patrz 3.3.9.), funkcje i operatory (w przypadku języka XML, są to np. – odwołania do zewnętrznych *deklaracji DTD* – patrz 3.3.9.). Omówimy kolejno te nowe dla nas rodzaje wyrażeń.

Trzeba będzie w rozdziale tym dawać przykłady na różne rodzaje wyrażenia, w związku, z czym powstaje pewna trudność. Polega ona na wyborze języka, w którym będziemy formułowali nasze przykłady. W poprzednim rozdziale trudności takiej nie było i zrozumiałe, dlaczego. Rozpatrywaliśmy tam, bowiem jedynie rodzaje wyrażeń występujących w językach, toteż wybór języka, w którym podawaliśmy przykłady, nie nasunął wątpliwości; przykłady nasze był zaczerpnięte po prostu z graficznego języka polskiego. W rozdziale niniejszym mówimy o rodzajach wyrażeń występujących w językach sztucznych (tj. w takiej czy innej symbolice) lub w językach mieszanych. Przykłady muszą, więc być dobrane w ten sposób, aby były zaczerpnięte z jakiegoś języka sztucznego czy mieszanego. Druga możliwość jest łatwiejsza do zrealizowania. Dokonamy wyboru pewnego języka mieszanego i w nim będziemy głównie formułowali przykłady różnego rodzaju funkcji.

Piśmiennictwo: *Greniewski H.* G.2.1.

3.3.1. STAŁE I ZMIENNE

3.3.1.00. **Wyjaśnienie**. Mówimy krótko „*wolno podstawiać*" (np. do wyrażenia $\boldsymbol{A}$ wyrażenie $\boldsymbol{B}$ za wyrażenie $\boldsymbol{C}$), zamiast mówić, że obowiązuje w danym języku dyrektywa (czy reguła) gramatyczna zapewniająca, że jeżeli wyrażenie, do którego podstawiamy, jest wyrażeniem danego języka, to wynik podstawienia też będzie wyrażeniem tegoż języka (wyjaśnienie 3.1.5.10).

3.3.1.01. **Wyjaśnienie**. Mówimy, że wyrażenie $\boldsymbol{A}$ jest zawarte w wyrażeniu $\boldsymbol{B}$, zamiast mówić, że wrażenie $\boldsymbol{A}$ jest częścią wyrażenia $\boldsymbol{B}$ (wyjaśnienie 3.1.0.11).

3.3.1.11. **Wyjaśnienie**. Mówimy, że wyrażenie $\boldsymbol{A}$ jest jak gdyby zawarte w wyrażeniu $\boldsymbol{B}$, zamiast mówić, że istnieje takie wyrażenie $\boldsymbol{A_1}$ równokształtne z wyrażeniem $\boldsymbol{A}$, że $\boldsymbol{A_1}$ jest zawarte w wyrażeniu $\boldsymbol{B}$.
Przykład (do wyjaśnień 3.3.1.10 oraz 3.3.1.11). Weźmy pod uwagę trzy wyrażenia poniższe:
(1) Warszawa,
(3) <u>Warszawa</u> jest stolicą Polski.
←←(2)→→
Łatwo zauważyć, że wyrażenie (1) jest równokształtne z wyrażeniem (2); dalej należy zauważyć, że wyrażenie (2) jest zawarte w wyrażeniu (3), zaś wyrażenie (1) nie jest zawarte w wyrażeniu (3); wreszcie, że wyrażenie (1) jest jak gdyby zawarte w wyrażeniu (3).

3.3.1.20. **Wyjaśnienie**. Mówimy krótko „*stała*” zamiast mówić „*zdanie lub nazwa, lub funktor*”.

Jak wynika z powyższego:
1) Każde zdanie jest *stałą*;
2) Każda nazwa jest *stałą*;
3) Każdy funktor jest *stałą*;
4) Każda *stała* jest albo zdaniem, albo nazwą, albo funktorem.

3.3.1.21. **Wyjaśnienie**. Bierzemy pod uwagę wyrażenie ***A***, następnie bierzemy pod uwagę wyrażenie ***B*** jak gdyby zawarte w wyrażeniu ***A***. Mówimy, że wyrażenie ***B*** jest zmienną, zamiast mówić, że spełnione są oba warunki następujące:
1) Strona znaczeniowa wyrażenia ***B***, czyli rozumienie wyrażenia ***B*** jest nieokreślone, czyli - puste;
2) Do wyrażenia ***A*** za wyrażenie ***B*** wolno podstawiać pewne wyrażenia z ***B*** nierównokształtne.

Ze względu na pierwszy warunek wyjaśnienia 3.3.1.21 niektórzy dawniejsi logicy zamiast mówić „*zmienna*”, mówili „*indeterminata*”, czyli „*wyrażenie niewyznaczone*”; ze względu na tenże warunek inni znowu logicy mawiali, że „*zmienna*” to tyle, co „*puste miejsce*”.

Trzeba tu szczerze powiedzieć, że wyjaśnienie 3.3.1.21 – nie może być traktowane, jako szczyt naukowej ścisłości. Szczegółowa analiza tego wyjaśnienia daje w wyniku pewne wątpliwości dotyczące możliwości ostrego odgraniczenia zmiennych od stałych. Nie będziemy tu jednak zajmowali się tymi dla elementarnego kursu nadmiernie subtelnymi rozróżnieniami. Pojęcie zmiennej znane jest zresztą z kursu algebry szkolnej, którą nazywamy niekiedy rachunkiem literowym. Każda litera występująca we wzorach algebry szkolnej jest właśnie zmienną, ale zmienną nader szczególnego rodzaju. Litery te, to zmienne nazwowe, na których miejsce wolno podstawiać tylko nazwy jednostkowe liczb. Liter - jako zmiennych używamy zazwyczaj i w logice. Nie jest to zresztą konieczność, logicy greccy ze szkoły filozoficznej stoików używali, jako zmiennych liczebników porządkowych. Jest poważnym błędem mylić zmienne z nazwami pustymi. Strona znaczeniowa dowolnej zmiennej jest pusta, natomiast strona znaczeniowa dowolnej nazwy pustej (np. nazwy „*kwadratowe koło*” czy też nazwy „*syn bezdzietnej matki*”) - nie jest pusta, pomimo że żadna nazwa pusta nie oznacza przedmiotu.

Odróżniamy dwa rodzaje zmiennych:
(1) Zmienne wolne;
(2) Zmienne związane.

Niezależnie od tego odróżniać będziemy trzy typy zmienny:
1) Zmienne zdaniowe;
2) Zmienne nazwowe;
3) Zmienne funktorowe.

3.3.1.22. **Wyjaśnienie**. Przez „*zmienną wolną*” rozumiemy taką i tylko taką zmienną, za którą wolno podstawiać nie tylko zmienne, ale i jakieś wyrażenia niebędące zmiennymi.

3.3.1.23. **Wyjaśnienie**. Przez „*zmienną związaną*” rozumiemy taką i tylko taką zmienną, za którą wolno podstawiać tylko zmienne.

Rozróżnienie zmiennych wolnych i zmiennych związanych wydaje się na pierwszy rzut oka dość dziwne, a co gorsza - zbędne. Wkrótce przekonamy się, że jest to jednak rozróżnienie celowe i że ma ono doniosłość praktyczną. Przekonamy się dalej, że podczas lekcji matematyki w szkole średniej mieliśmy już do czynienia zarówno ze zmiennymi wolnymi, jak i związanymi. Ponadto, gdy nieco później będziemy omawiali *operatory*, czytelnicy znający wyższą matematykę będą mogli się przekonać, że rozróżnienie zmiennych wolnych i związanych pomaga lepiej zrozumieć symbolikę, z którą mamy do czynienia w analizie matematycznej i algebrze wyższej.

3.3.1.30. **Wyjaśnienie**. Przez „*wolną zmienną zdaniową*" rozumiemy taką i tylko taką zmienną wolną, za którą wolno podstawiać przynajmniej niektóre zdania.

3.3.1.40. **Wyjaśnienie**. Przez „*wolną zmienną nazwową*" rozumiemy taką i tylko taką zmienną wolną, za którą wolno podstawiać przynajmniej niektóre nazwy.

Wyjaśnienie 3.3.1.40, wymaga jeszcze dodatkowego omówienia. Trzeba powiedzieć, że w praktyce mamy do czynienia z nader różnymi rodzajami wolnych zmiennych nazwowych: Niekiedy stosujemy wolne zmienne nazwowe, za które wolno podstawiać nazwy puste, jednostkowe i ogólne. *Arystoteles* w swojej sylogistyce, używał wolnych zmiennych nazwowych, za które wolno podstawiać nazwy niepuste, nie wolno zaś podstawiać nazw pustych. Matematycy współcześni z reguły używają takich zmiennych nazwowych wolnych, za które wolno podstawiać tylko nazwy jednostkowe. W matematyce używa się często zmiennych nazwowych wolnych, za które wolno podstawiać tylko nazwy jednostkowe liczb naturalnych (w takiej roli występują litery: k, l, m, n).

3.3.1.41. **Wyjaśnienie**. Przez „*wolną zmienną jednostkowo nazwową*" rozumiemy taką i tylko taką zmienną wolną, za którą wolno podstawiać przynajmniej pewne nazwy puste i jednostkowe, nie wolno zaś podstawiać żadnych nazw ogólnych.

3.3.1.42. **Wyjaśnienie**. Przez „*wolną zmienną ogólno-nazwową*" rozumiemy taką i tylko taką zmienną wolną, za którą wolno podstawiać przynajmniej pewne nazwy puste, jednostkowe i ogólne.

3.3.1.50. **Wyjaśnienie**. Przez „*wolną zmienną funktorową*" rozumiemy taką i tylko taką zmienną wolną, za którą wolno podstawiać przynajmniej pewne funktory.

W szkole średniej nie mamy zazwyczaj do czynienia ze zmiennymi funktorowymi. Inaczej przedstawia się sprawa na uczelni wyższej, przynajmniej - gdy chodzi o słuchaczy matematyki, informatyki, fizyki i biologii. Na wykładach analizy matematycznej czy matematyki dla biologów słuchacze mają często do czynienia z wyrażeniami takimi, jak:

$$y = f(x)$$
$$z = \varphi(x, y)$$
$$u = \phi(x, y, z).$$

W wyrażeniach tych każda z liter f, φ, ϕ jest wolną zmienną funktorową, za którą wolno podstawiać pewne funktory jednostkowo-nazwotwórcze od argumentów jednostkowo nazwowych.

Uwaga: W tym miejscu należy zrobić uwagę terminologiczną. A mianowicie, znak „=" w matematyce, jest używany zarówno do oznaczenia operatora porównania (wyrażeń lewej i prawej strony), jak również, jako operatora podstawienia. Autorzy pierwszego języka programowania FORTRAN, przyjęli znak „=" jako symbol operacji podstawiania (na zmienną

umieszczoną po lewej stronie – podstawiana jest wartość umieszczona po prawej stronie znaku równości). Autorzy języka ALGOL 60, próbowali przywrócić tradycyjne znaczenie symbolu „=", wprowadzając do ALGOL'u 60, jako symbol operatora podstawienia „:=". Propozycja ta, jednak nie przyjęła się. W większości współczesnych języków programowania, przyjęto znak „=" jako symbol operacji podstawiania, zaś jako symbol operacji porównania przyjęto „==".

3.3.1.60. **Wyjaśnienie**. Przez „związaną zmienną zdaniową" rozumiemy taką i tylko taką zmienną związaną, która jest równokształtna z jakąś wolną zmienną zdaniową.

3.3.1.61. **Wyjaśnienie**. Przez „związaną zmienną nazwową" rozumiemy taką i tylko taką zmienną związaną, która jest równokształtna z jakąś wolną zmienną nazwową.

3.3.1.62. **Wyjaśnienie**. Przez „związaną zmienną funktorową" taką i tylko taką zmienną związaną, która jest równokształtna z jakąś wolną zmienną funktorową.

Zapamiętajmy sobie teraz, że w niniejszym wykładzie będziemy stale używać:

- Jako zmiennych zdaniowych - liter: p, q, r, s oraz $p_1, p_2, p_3, \ldots$
- Jako zmiennych nazwowych - liter: a, b, c, x, y, z oraz $a_1, a_2, a_3, \ldots$
- Jako zmiennych funktorowych – liter alfabetu greckiego: $\varphi, \phi, \ldots, \varphi_1, \varphi_2, \varphi_3, \ldots$

Jak widać z powyższego, jako zmiennych wolnych i związany używamy odpowiednio równokształtnych liter. O tym, czy dana zmienna jest w danym wyrażeniu wolna, czy związaną decyduje jej rola w danym wyrażeniu, co zaraz wyjaśnimy za pomocą przykładu.

Przykład. Napiszmy w języku algebry szkolnej następujące wyrażenie

(1) $$a + b = c$$

Łatwo zauważyć, że zmienne jednostkowo nazwowe występujące w wyrażeniu (1) są zmiennymi wolnymi, można bowiem do wyrażenia (1) podstawić za zmienne w nim występujące nie tylko inne zmienne jednostkowo nazwowe, ale również nazwy jednostkowe liczb. W obu wypadkach, jako wynik podstawienia otrzymamy znowu wyrażenie należące do języka algebry szkolnej. Podstawmy do wyrażenia (1) za każdą ze zmiennych jednostkowo nazwowych: a, b, c, odpowiednio zmienne jednostkowo nazwowe: x, y, z; jako wynik podstawienia otrzymamy wyrażenie:

(2) $$x + y = z$$

należące do języka algebry szkolnej. Podstawmy teraz do wyrażenia (1) za każdą ze zmiennych nazwowych: a, b, c, odpowiednio nazwy jednostkowe liczb: 1,2,3. Jako wynik podstawienia otrzymamy (w języku algebry szkolnej) zdanie prawdziwe:

(3) $$1 + 2 = 3$$

a więc wyrażenie należące do języka algebry szkolnej. Podstawmy wreszcie do wyrażenia (1) za każdą ze zmiennych nazwowych: a, b, c, odpowiednio nazwy jednostkowe liczb: $3, 4, 5$; jako wynik tego podstawienia otrzymamy zdanie w języku algebry szkolnej:

(4) $$3 + 4 = 5.$$

Powyższe wyrażenie jest oczywiście zdaniem fałszywym, ale to okoliczność w tym przypadku obojętna. Znaczenie ma jedynie ten fakt, że jako wynik podstawienia otrzymaliśmy zdanie (4), a więc wyrażenie należące nadal do naszego języka.

W podrozdziale 3.3.0 zapowiedzieliśmy, że zbudujemy pewien język mieszany nadający się specjalnie do formułowania przykładów. Postawione zadanie jest już obecnie łatwe do zrealizowania.

3.3.1.70. **Wyjaśnienie**. Przez „JPM" (skrót od „język polski mieszany") będziemy rozumieli tu język polski (graficzny) z niżej opisanymi zmianami i uzupełnieniami, w tym możliwość stosowania znaczników:

1) Obok interpunkcji używanej zwykle w języku polskim będziemy używali nawiasów;
2) Wprowadzamy do naszego języka wolne zmienne zdaniowe występujące tylko w takich wyrażeniach, w których składnia polska pozwala wstawiać zdania;
3) Wprowadzamy do naszego języka wolne zmienne nazwowe, występujące tylko w takich wyrażeniach, w których składnia polska pozwala używać nazw (np. odpowiednich rzeczowników czy przymiotników);
4) Wprowadzamy do naszego języka wolne zmienne funktorowe, występujące tylko w takich wyrażeniach, w których składnia polska pozwala wstawiać funktory.
5) Wprowadzamy do naszego języka, dla zwiększenia jednoznaczności, możliwość stosowania znaczników – wzorowaną na języku XML (patrz podrozdział 3.3.8).

Dokładne przeanalizowanie powyższego wyjaśnienia wykazuje, że nie grzeszy ono nadmiarem ścisłości, ma jednak dla nas zasadniczą wartość, ułatwia - bowiem w praktyce budowanie wyrażeń języka mieszanego (JPN).

Piśmiennictwo: *Greniewski H.* G.2.1.

3.3.2. Pojęcie funkcji

3.3.2.00. **Wyjaśnienie**. Przez *„funkcję"* rozumiemy takie i tylko takie wyrażenie, które zawiera - chociaż jedną zmienną wolną.

Zapamiętajmy, że wyrazu *„funkcja"* używa się w życiu potocznym w rozmaitych naukach w nader rożnych rozumieniach. Na przykład: - gdy biolog mówi *„funkcja"*, ma na myśli czynność; - gdy matematyk mówi *„funkcja"* to na myśli przyporządkowanie jednoznaczne; - gdy logik mówi *„funkcja"*, ma na myśli - jak stwierdziliśmy to przed chwilą - wyrażenie należące do języka i zawierające przynajmniej jedną zmienną wolną.

Można się dziwić, że w nauce panuje taki zamęt terminologiczny. Wydaje się niezrozumiałe, że tak naukowy, zdawałoby się, termin jak *„funkcja"* jest używany aż w trzech tak różnych rozumieniach. Nie trudno jednak ten stan rzeczy wyjaśnić historycznie. Wyraz *„funkcja"* pochodzi od łacińskiego wyrazu *„functio"*. Pierwotne rozumienie łacińskiego wyrazu *„functio"* jest takie, jak polskiego wyrazu *„czynność"*. Nic więc dziwnego, że biolog mówi *„funkcja trawienia"* zamiast *„czynność trawienia"*. Następnie należy zauważyć, że pojęcie *funkcji* (w rozumieniu przyjętym przez matematyków) jest wynikiem trwającego wieki procesu abstrahowania. Punkt wyjściowy tego procesu – to konkretne czynności rachunkowe wykonywane przez człowieka, punkt końcowy to współczesne pojęcie matematyczne funkcji, czyli przyporządkowania jednoznacznego. Zilustrujemy ten długotrwały proces historyczny za pomocą jednego tylko przykładu.

Przez *dodawanie* (np. liczb naturalnych) rozumieli niegdyś matematycy pewną czynność prowadzącą od zapisu dwu liczb (np. naturalnych zwanych *dodajną* i *dodajnikiem*) do uzyskania zbudowanej według wymagań danego systemu liczbowego nazwy trzeciej liczby tak zwanej *sumy*. Dzisiaj matematyk mówiąc *„dodawanie liczb naturalnych"* ma na myśli pewne przyporządkowanie jedno znaczne (dane na przykład przez przepis), mianowicie przyporządkowanie, które każdej parze liczb naturalnych (obojętne czy kiedykolwiek przez nas zapisanych) przepisuje jedną i tylko jedną liczbę naturalną, zwaną sumą tamtych. Takie

przyporządkowanie nazywa się również *funkcją* (czynnością), chociaż tylko wywodzi się z czynności. Wreszcie, wyrażenia zwane przez logików *funkcjami* zawdzięczają historycznie swe powstanie zapisom, które czynili matematycy badając *funkcje* we wspomnianym poprzednio rozumieniu tego słowa.

3.3.2.01. **Wyjaśnienie**. Mówiąc „*liczba zmiennych występujących w danej funkcji*" mamy na myśli liczbę otrzymaną w sposób następujący: numerujemy, poczynając od liczby 1, w ustalonej dla danego języka kolejności zmienne wolne zawarte w tej funkcji; jeżeli jednak napotykamy zmienną wolną równokształtną z jakąś zmienną, która numer już otrzymała - nie numerujemy jej. Najwyższy numer otrzymany w wyniku takiego, postępowania jest poszukiwaną liczbą zmiennych wolnych.

W następnych paragrafach omówimy różne rodzaje *funkcji*, podając za każdym razem przykłady danego rodzaju funkcji, zaczerpnięte zwykle z JPM.

Piśmiennictwo: Greniewski H. G.2.1.

3.3.3. FUNKCJE ZDANIOWE

3.3.3.00. **Wyjaśnienie**. „*Funkcja zdaniowa jednej zmiennej zdaniowej*", to taka i tylko taka funkcja, która zawiera jedną zmienną zdaniową wolną i staje się zdaniem, gdy za tę zmienną zdaniową podstawimy zdanie.

Zauważmy, że łatwo jest budować przykłady na funkcje zdaniowe jednej zmiennej zdaniowej, wystarczy, bowiem w tym celu napisać jakikolwiek funktor zdaniotwórczy od jednego argumentu zdaniowego i dołączyć do tego funktora zmienną zdaniową. Utworzona w ten sposób całość będzie funkcją zdaniową jednej zmiennej zdaniowej.
Przykład. Wyrażenie „*jest prawdą, że*" jest funktorem zdaniotwórczym od jednego argumentu zdaniowego; dołączając do tego funktora: zmienną zdaniową otrzymamy funkcję zdaniową jednej zmiennej zdaniowej, mianowicie otrzymamy wyrażenie: *jest prawdą, że p*.

Trzeba jeszcze zaznaczyć, że podany wyżej sposób budowania funkcji zdaniowej jednej zmiennej zdaniowej nie jest sposobem jedynym. Aby nie być gołosłownym, podamy jeszcze sposób drugi: napiszmy funktor zdaniotwórczy od dwóch argumentów zdaniowych i dołączmy do niego dwie równokształtne zmienne zdaniowe; otrzymamy w wyniku funkcję zdaniową jednej zmiennej zdaniowej.

Przykład. Wyrażenie „*lub*" jest, jak wiemy, funktorem zdaniotwórczym od dwu argumentów zdaniowych; dołączamy do tego funktora dwie równokształtne zmienne zdaniowe, wynik tego dołączenia, to wyrażenie „*p lub p*". Zauważmy, że ostatnie wyrażenie w cudzysłowie zawiera wprawdzie zmienne, ale ponieważ są one równokształtne, wyrażenie to jest funkcją jednej zmiennej.

3.3.3.01. **Wyjaśnienie**. „Funkcja zdaniowa dwu zmiennych zdaniowych to taka i tylko taka funkcja, która zawiera dwie (nie równokształtne) zmienne zdaniowe i staje się zdaniem, gdy za obie te zmienne podstawimy zdania.
Przykłady. Funkcją zdaniową dwóch zmiennych zdaniowych jest: *p lub q, jeżeli p, to q*.

Jednym ze sposobów budowania funkcji zdaniowych dwóch zmiennych zdaniowych jest też sposób następujący: napisać funktor zdaniotwórczy od dwóch argumentów zdaniowych i dołączyć do niego dwie nie równokształtne zmienne zdaniowe.

Podamy teraz wyjaśnienie bardziej ogólne, które obejmie 3.3.3.00 oraz 3.3.3.01 jako nader szczególne ewentualności.

3.3.3.02. **Wyjaśnienie**. „*Funkcja zdaniowa n zmiennych zdaniowych*", to taka i tylko taka funkcja, która zawiera n wolnych zmiennych zdaniowych i staje się zdaniem, gdy za wszystkie te zmienne podstawimy zdania.
Przykłady. Funkcja zdaniowa trzech zmiennych zdaniowych: - *jeżeli (p lub q) to r*. Funkcja zdaniowa czterech zmiennych zdaniowych: - *(jeżeli p, to q) lub (r or s)*.

Liczne, a łatwe do wykonania próby doprowadzają nas do stwierdzenia następujących reguł:

3.3.3.10. **Reguła dołączania**. Dołączając
- do funktora zdaniotwórczego od k argumentów zdaniowych, należącego do JPM
- k zmiennych zdaniowych[40], z których każde dwie są między sobą równokształtne

otrzymamy, jako wynik funkcję zdaniową jednej zmiennej zdaniowej, należącą do JPM.

Bez trudu można sformułować i uzasadnić regułę bardziej ogólną:

3.3.3.11. **Reguła dołączania**. Dołączając
- do funktora zdaniotwórczego od k argumentów zdaniowych, należącego do JPM,
- k zmiennych zdaniowych, wśród których mamy m zmiennych takich, że żadne dwie spośród nich nie są równokształtne otrzymamy, jako wynik funkcję zdaniową, co najmniej m zmiennych zdaniowych, należącą do JPM.

Oprócz reguł dołączania możemy, opierając się na łatwych do przeprowadzenia próbach, otrzymać reguły podstawiania.

3.3.3.21. **Reguła podstawiania**. Podstawiając
- do funkcji zdaniowej k zmiennych zdaniowych, należącej do JPM
- za niektóre przynajmniej zmienne zdaniowe
- zmienne zdaniowe

otrzymamy, jako wynik funkcję zdaniową nie więcej niż k zmiennych zdaniowych, należącą do JPM.

3.3.3.22. **Reguła podstawiania**. Podstawiając
- do funkcji zdaniowej k zmiennych zdaniowych, należących do JPM
- za zmienną zdaniową
- funkcję zdaniową m zmiennych zdaniowych, należącą do JPM

otrzymamy, jako wynik funkcję zdaniową nie więcej niż $(k - 1 + m)$ zmiennych zdaniowych, należącą do JPM.

Przejdziemy teraz do pojęcia funkcji zdaniowej zmiennych nazwowych.

3.3.3.30. **Wyjaśnienie**. „*Funkcja zdaniowa jednej zmiennej jednostkowo-nazwowej*", to taka i tylko taka funkcja, która zawiera jedną wolną zmienną jednostkowo nazwową i staje się zdaniem, gdy za tę zmienną jednostkowo nazwową podstawimy nazwę jednostkową lub pustą. Dołączając do funktora zdaniotwórczego od jednego argumentu jednostkowo nazwowego zmienną jednostkowo nazwową, otrzymamy funkcję zdaniową jednej zmiennej jednostkowo nazwowej.

[40] Odtąd, jeżeli w regule czy dyrektywie będziemy mówili o zmiennych, będziemy zawsze mieli na myśli zmienne wolne, chyba że wyrażenie napiszemy, że w danym wypadku wchodzą w grę zmienne związane.

Przykład. Funkcją zdaniową jednej zmiennej jednostkowo nazwowej wyrażenia „*x jest człowiekiem*".

Istnieją tez inne sposoby budowania funkcji zdaniowej jednej zmiennej jednostkowo nazwowej. Dołączmy do funktora zdaniotwórczego od dwóch argumentów jednostkowo nazwowych „*jest równe*" dwie równokształtne zmienne jednostkowo nazwowe, otrzymamy w wyniku funkcję zdaniową jednej zmiennej jednostkowo nazwowej.
Przykład. Funkcją zdaniową jednej zmiennej jednostkowo nazwowej jest wyrażenie: „*x jest równe x*".

3.3.3.31. **Wyjaśnienie**. „*Funkcja zdaniowa dwu zmiennych jednostkowo nazwowych*", to taka i tylko taka funkcja, która zawiera dwie zmienne jednostkowo nazwowe i staje się zdaniem, gdy za obie tę zmienne podstawimy nazwy jednostkowe lub puste. Dołączając do funktora zdaniotwórczego od dwóch argumentów jednostkowo nazwowych dwie (nie równokształtne) zmienne jednostkowo nazwowe, otrzymamy w wyniku funkcję zdaniową dwóch zmiennych jednostkowo nazwowych. Innych sposobów budowania funkcji zdaniowych dwóch zmiennych jednostkowo nazwowych nie będziemy tu omawiać.
Przykład. Funkcją zdaniową dwóch zmiennych jednostkowo nazwowych jest wyrażenie „*x jest młodszym bratem (dla) y*".

Uogólnimy teraz wyjaśnienia 3.3.3.30 oraz 3.3.3.31.

3.3.3.32. **Wyjaśnienie**. „*Funkcja zdaniowa n zmiennych jednostkowo nazwowych*", to taka i tylko taka funkcja, która zawiera n wolnych zmiennych jednostkowo nazwowych i staje się zdaniem, gdy za wszystkie te zmienne podstawimy nazwy jednostkowe lub puste.
Przykłady. Funkcją zdaniową trzech zmiennych jednostkowo nazwowych jest wyrażenie: „*x oraz y są rodzicami osoby z*". Funkcją zdaniową pięciu zmiennych jednostkowo nazwowych jest wyrażenie: „x, y, z *są dziećmi rodziców* a, b".

3.3.3.33. **Wyjaśnienie**. „*Funkcja zdaniowa n zmiennych ogólno nazwowych*", to taka i tylko taka funkcja, która zawiera n wolnych zmiennych ogólnonazwowych i staje się zdaniem, gdy za wszystkie te zmienne podstawimy dowolne nazwy.
Przykłady. Funkcje zdaniowe dwu zmiennych ogólno-nazwowych:
(1) każde x jest y,
(2) przynajmniej pewne x są y,
(3) tylko pewne x są y,
(4) żadne x nie jest y,
(5) przynajmniej pewne x nie są y,
(6) tylko pewne x nie są y.

W oparciu o wyjaśnienia 3.3.3.30, 3.3.3.31, 3.3.3.32 oraz 3.3.3.33 - łatwymi do wykonania próbami (doświadczeniami językowymi), można zbudować wiele reguł opisujących budowanie funkcji zdaniowych zmiennych jednostkowo - czy ogólno-nazwowych.

3.3.3.40. **Reguła dołączania**. Dołączając
- do funktora zdaniotwórczego od k argumentów jednostkowo nazwowych, należącego do JPM
- k zmiennych jednostkowo nazwowych, między sobą równo kształtnych,

otrzymamy jako wynik funkcję zdaniową jednej zmiennej jednostkowo nazwowej.
Przykład. Dołączamy
- do funktora zdaniotwórczego od dwu argumentów jednostkowo nazwowych: „*jest*

rówieśnikiem”,
- dwie zmienne jednostkowo nazwowe: x, x równokształtne,
otrzymamy jako wynik następującą funkcję zdaniową jednej zmiennej jednostkowo nazwowej: „*x jest rówieśnikiem x*”.

3.3.3.41. **Reguła dołączania**. Dołączając
- do funktora zdaniotwórczego od k argumentów, jednostkowych, należącego do JPM,
- k zmiennych jednostkowo nazwowych, wśród których mamy m zmiennych takich, że żadne dwie spośród nich nie są wzajemnie równokształtne, otrzymamy jako wynik funkcję zdaniową co najmniej $m$zmiennych jednostkowo nazwowych.

Przykład. Dołączamy
- do funktora zdaniotwórczego od trzech argumentów jednostkowo nazwowych: „*ojcem osoby ... oraz osoby ... jest osoba*”, trzy zmienne jednostkowo nazwowe: x, x, y, z których dwie nie są równokształtne, otrzymamy jako wynik funkcję zdaniową dwu zmiennych jednostkowo nazwowych: „*ojcem osoby x oraz osoby x jest osoba y”*.

3.3.3.50**. Reguła podstawiania**. Podstawiając
- do funkcji zdaniowej k zmiennych jednostkowo nazwowych, należącej do JPM
- za niektóre przynajmniej zmienne jednostkowo nazwowe
- zmienne jednostkowo-nazwowe,

otrzymamy jako wynik funkcję zdaniową najwyżej k zmiennych jednostkowo-nazwowych, należącą do JPM.

Przykład. Bierzemy pod uwagę następującą funkcję zdaniową trzech zmiennych jednostkowo-nazwowych:

(1) „*x jest krewnym osoby y oraz osoby z*”.

Podstawiamy teraz
- do funkcji zdaniowej (1), należącej do JPM
- za zmienne jednostkowo nazwowe: x, y, z
- zmienne jednostkowo nazwowe : a, b, c,

otrzymujemy jako wynik następującą funkcję zdaniową trzech zmiennych jednostkowo nazwowych, należącą do JPM:

(2) „*a jest krewnym osoby b oraz osoby c*”.

Podstawmy teraz
- do funkcji zdaniowej (1), należącej do JPM
- za zmienne jednostkowo nazwowe: x, y, z
- zmienne jednostkowo nazwowe: a, b, b,otrzymujemy jako wynik następującą funkcje

zdaniową dwu zmiennych jednostkowo nazwowych, należącą do JPM:

(3) „*a jest krewnym osoby b oraz osoby b*”.

Jasne jest przy tym, że wyrażenie (3) jest równoznaczne z następującym wyrażeniem, również należącym do JPM:

(4) „*a jest krewnym osoby b*”.

3.3.3.51. **Reguła podstawiania**. Podstawiając
- do funkcji zdaniowej k zmiennych zdaniowych, należącej do JPM
- za każdą zmienną zdaniową
- funkcję zdaniową m zmiennych jednostkowo nazwowych,

otrzymujemy jako wynik funkcję zdaniową o co najwyżej $(k \times m)$ zmiennych jednostkowo nazwowych, należącą do JPM.

Przykład. Bierzemy pod uwagę następującą funkcję zdaniową dwu zmiennych zdaniowych, należącą do JPM:

(1) *„Jeżeli p to q”*.

Podstawmy teraz

- do funkcji zdaniowej (1)
- za zmienne zdaniowe: p, q

odpowiednio - funkcje zdaniowe (każda jednej zmiennej jednostkowo nazwowej), należące do JPM:

x jest człowiekiem,

y jest ssakiem,

otrzymujemy jako wynik podstawienia funkcję zdaniową dwu zmiennych jednostkowo nazwowych, należącą do JPM:

„jeżeli (x jest człowiekiem), to (y jest ssakiem)”.

Funkcja powyższa jest funkcją dwu zmiennych. Podstawmy natomiast

- do funkcji zdaniowej (1)
- za zmienne zdaniowe: p, q
- odpowiednio funkcje zdaniowe (każda jednej zmiennej - jednostkowo nazwowej), należące do JPM:

x jest człowiekiem,

y jest ssakiem,

otrzymujemy jako wynik podstawienia funkcję zdaniową jednej zmiennej jednostkowo nazwowej, należącą do JPM:

(3) *„jeżeli (x jest człowiekiem), to (x jest ssakiem)”*.

Wyjaśniliśmy już, czym są funkcje zdaniowe zmiennych zdaniowych, funkcje zdaniowe zmiennych jednostkowo nazwowych i wreszcie funkcje zdaniowe zmiennych ogólno nazwowych; nie wyjaśniliśmy natomiast ogólnie rozumienia wyrażenia „funkcja zdaniowa”. Uzupełnimy teraz tę lukę.

3.3.3.60. **Wyjaśnienie**. Mówimy, że *dana funkcja jest funkcją zdaniową* zamiast mówić, że po dozwolonym podstawieniu stałych za wszystkie zmienne wolne zawarte w tej funkcji, funkcja ta staje się zdaniem.

Zajmiemy się jeszcze pewnym innym podziałem funkcji zdaniowych.

3.3.3.70. **Wyjaśnienie**. Mówimy, ze funkcja zdaniowa jest funkcją pustą, zamiast mówić, że po dozwolonym podstawieniu stałych za wszystkie występujące w tej funkcji zmienne wolne otrzymamy zawsze zdanie fałszywe.

Przykłady. Funkcje puste:

(1) *Fałszem jest, że* (*jeżeli* p, *to* p).

(2) *Fałszem jest, że [jeżeli (p oraz q), to (p oraz q)*].

(3) x *nie jest identyczne z* x.

Jak widać, funkcja (1) jest funkcją zdaniową jednej zmiennej zdaniowej, (2) jest funkcją zdaniową dwu zmiennych zdaniowych, (3) jest funkcją zdaniową jednej zmiennej jednostkowo nazwowej, a przy tym każda z tych trzech funkcji jest funkcją pustą.

3.3.3.71. **Wyjaśnienie**. Mówimy, ze funkcja zdaniowa jest funkcją pustą, zamiast mówić, że dla chociaż jednego dozwolonego podstawienia stałych za wszystkie występujące w tej funkcji zmienne wolne otrzymamy z niej zdanie prawdziwe.

3.3.3.72. **Wyjaśnienie**. Mówimy, że funkcja zdaniowa jest funkcją powszechnie prawdziwą, zamiast mówić, że po dozwolonym podstawieniu stałych za wszystkie występujące w tej funkcji zmienne wolne otrzymamy zawsze zdanie prawdziwe.
Przykłady. Funkcje zdaniowe, które są zarazem niepuste i nie są powszechnie prawdziwe:
(1) *Jeżeli p, to q*,
(2) *x jest mniejsze od y*.
Podstawmy do funkcji zdaniowej (1) za zmienną zdaniową „*p*" zdanie „*Sokrates jest człowiekiem*" zaś za zmienną zdaniową „*q*" zdanie „*Sokrates jest śmiertelny*"; otrzymamy jako wynik tego podstawienia zdanie (złożone):
(3) *Jeżeli (Sokrates jest człowiekiem), to (Sokrates jest śmiertelny)*.
Zdanie (3) jest prawdziwe, a więc funkcja zdaniowa (1) jest niepusta. Podstawmy teraz do funkcji (1) za zmienną zdaniową „*p*" zdanie „*Sokrates jest człowiekiem*", zaś za zmienną zdaniową „*q*" podstawmy zdanie „*Sokrates jest niemową*"; otrzymamy jako wynik tego nowego podstawienia zdanie (złożone):
(4) *Jeżeli (Sokrates jest człowiekiem), to (Sokrates jest niemową)*.
Zdanie (4) jest fałszywe, a więc funkcja zdaniowa (1) nie jest powszechnie prawdziwa. Zatem funkcja zdaniowa (1) jest niepusta i zarazem nie jest powszechnie prawdziwa. Podstawmy teraz do funkcji (2) za zmienną jednostkowo nazwową „*x*" nazwę jednostkową „*jeden*", a za zmienną jednostkowo nazwową „*y*" - nazwę jednostkową „*dwa*"; otrzymamy jako wynik tego podstawienia zdanie:
(5) *Jeden* - jest mniejsze od *dwu*.
(Sprawę deklinacji pomijamy, jako dla nas nieistotną). Zdanie (5) jest prawdziwe, a więc funkcja zdaniowa (2) jest niepusta. Podstawmy teraz do funkcji (2) za zmienną jednostkowo nazwową „*x*" nazwę jednostkową „*dwa*", zaś za zmienną jednostkowo nazwową „*y*" nazwę jednostkową „*jeden*", otrzymamy jako wynik tego podstawienia zdanie:
(6) *Dwa* - jest mniejsze od *jednego*.
(Sprawę deklinacji znowu pomijamy). Zdanie (6) jest fałszywe, a więc funkcja zdaniowa (2) nie jest powszechnie prawdziwa.

Podamy teraz przykłady funkcji powszechnie prawdziwych:
(1) *jeżeli p, to p*,
(2) jeżeli (x jest identyczne z y), to (y jest identyczne z x).

Piśmiennictwo: *Greniewski H.* G.2.1.

3.3.4. FUNKCJE NAZWOWE

3.3.4.00. **Wyjaśnienie**. „Funkcja jednostkowo nazwowa k zmiennych jednostkowo nazwowych", to taka i tylko taka funkcja, która zawiera k wolnych zmiennych jednostkowo nazwowych i staje się nazwą jednostkową lub pustą zawsze, gdy za każdą z tych zmiennych podstawimy nazwę jednostkową lub pustą. Przykłady na funkcje jednostkowo nazwowe zmiennych jednostkowo nazwowych najłatwiej czerpać z języka algebry szkolnej.
Przykłady. Funkcje jednostkowo nazwowe jednej zmiennej jednostkowo nazwowej:

$$x, x^2, x^3, x + x, x^x, 2^x, \log x, \sin x, \cos x.$$

Funkcje jednostkowo nazwowe dwu zmiennych jednostkowo nazwowych:

$$x + y, x - y, x^y, x^2+y^2, x^2+ 2xy + y^2.$$

Funkcje jednostkowo nazwowe trzech zmiennych jednostkowo nazwowych:

$$x + y + z, x - y + z, x + y - z, x^2+ y^2+ z^2.$$

Również w innych językach sztucznych czy mieszanych, znaleźć można przykłady na funkcje jednostkowo nazwowe zmiennych jednostkowo nazwowych.
Przykład. Funkcja jednostkowo nazwowa trzech zmiennych jednostkowo nazwowych: trójkąt o wierzchołkach a, b, c.

Nie trudno w drodze prostych doświadczeń językowych przekonać się o trafności następujących reguł:

3.3.4.00. **Reguła dołączania**. Dołączając
- do funktora jednostkowo nazwotwórczego od k argumentów jednostkowo nazwowych
- k zmiennych jednostkowo nazwowych, wśród których mamy m takich, że żadne dwie z nich nie są równokształtne,

otrzymujemy jako wynik funkcję jednostkowo nazwową co najmniej m zmiennych jednostkowo nazwowych.
Przykład (zaczerpnięty z języka algebry szkolnej). Dołączając do funktora jednostkowo nazwotwórczego: $+ \ldots - \ldots + \ldots -$; pięć zmiennych jednostkowo nazwowych: x, y, x, z, x; z których trzy mają tę własność, że nie są między sobą równokształtne, otrzymujemy jako wynik następującą funkcję jednostkowo nazwową trzech zmiennych jednostkowo-nazwowych, należącą do języka algebry szkolnej:

$$x + y - x + z - x$$

3.3.4.11. **Reguła podstawiania**. Podstawiając
- do funkcji jednostkowo nazwowej k zmiennych jednostkowo-nazwowych
- za jedną ze zmiennych
- funkcję jednostkowo nazwową m zmiennych jednostkowo-nazwowych, otrzymujemy, jako wynik funkcję jednostkową najwyżej $(k - 1 + m)$ zmiennych jednostkowo-nazwowych.

Przykład (zaczerpnięty z języka algebry szkolnej). Podstawiamy:
- do funkcji jednostkowo nazwowej trzech zmiennych jednostkowo nazwowych:

$$x^2 + y^2 + z^2$$

- za zmienną: x
- funkcję jednostkowo nazwową dwu zmiennych jednostkowo nazwowych $(a^2 + b^2)$,

otrzymujemy jako wynik funkcję jednostkowo nazwową czterech zmiennych jednostkowo nazwowych: $(a^2 + b^2)^2 + y^2 + z^2$
3.3.4.20. **Wyjaśnienie**. „Funkcja jednostkowo nazwowa k zmiennych ogólno nazwowych", to taka i tylko taka funkcja, która zawiera k wolnych zmiennych ogólno nazwowych i staje się nazwą jednostkową lub pustą zawsze, gdy za każdą z tych zmiennych podstawimy nazwę.
Przykład. Funkcją jednostkowo nazwową dwu zmiennych ogólno-nazwowych jest wyrażenie: najlżejszy spośród x oraz y. Po podstawieniu stałych za zmienne (na przykład): najlżejszy spośród pierwiastków chemicznych i związków chemicznych.

3.3.4.30. **Wyjaśnienie**. „*Funkcja ogólno nazwowa k zmiennych jednostkowo nazwowych*", to taka i tylko taka funkcja, która zawiera k wolnych zmiennych jednostkowo nazwowych i staje się nazwą zawszę, gdy za każdą z, tych zmiennych podstawimy nazwę jednostkową lub pustą.
Przykłady: *Przodek osoby x*. *Dziecko obywatela x i obywatelki y*.

3.3.4.40. **Wyjaśnienie**. „*Funkcja ogólno-nazwowa k zmiennych ogólno nazwowych*" to taka i tylko taka funkcja, która zawiera k wolnych zmiennych ogólno-nazwowych i staje się nazwą, gdy za każdą z tych zmiennych podstawimy nazwę.

Przykład. Funkcją ogólno-nazwową dwu zmiennych ogólno-nazwowych jest każde z obu następujących wyrażeń należących do JPM: a lub b, a i zarazem b.

Piśmiennictwo: *Greniewski H.* G.2.1.

3.3.5. Inne funkcje

Poznaliśmy wiele rodzajów funkcji, przy czym chcemy tu jeszcze podkreślić, że dalecy jesteśmy od wyczerpania tematu. Istnieją jeszcze oprócz funkcji zdaniowych i nazwowych różne rodzaje funkcji funktorowych (*funkcje funktorowe zmiennych zdaniowych, funkcje funktorowe zmiennych nazwowych, funkcje funktorowe zmiennych funktorowych itp.*). Istnieją też *funkcje operatorowe*.

Piśmiennictwo: *Greniewski H.* G.2.1.

3.3.6. Pojęcie tezy

3.3.6.00. **Wyjaśnienie**. Mówimy krótko „*teza*", zamiast mówić „*zdanie lub funkcja zdaniowa, lecz nie definicja*" (wyjaśnienie 3.4.0.00).
Przykłady. Weźmy pod uwagę następujące wyrażenia:
(1) Sokrates jest człowiekiem.
(2) x jest człowiekiem.
Wyrażenie (1) jest zdaniem, a więc jest tezą; wyrażenie (2) jest funkcją zdaniową, a więc również jest tezą. W dalszym ciągu zastosujemy konwencję zaczerpniętą z języka XML i będziemy zapisywać tezy w sposób następując: `<teza>` Sokrates jest człowiekiem `</teza>`

3.3.6.01. **Wyjaśnienie**. Mówimy krótko „*teza prawdziwa*", zamiast mówić „*zdanie prawdziwe lub funkcja zdaniowa powszechnie prawdziwa*".

Istnieje dość rozpowszechnione a błędne mniemanie, że każde twierdzenie matematyczne jest zdaniem. Rzeczywiście, wyrażenie:

$$2 + 2 = 4$$

jest zdaniem. Natomiast znany wzór na kwadrat sumy liczb:

$$(a + b)^2 = a^2 + 2ab + b^2$$

nie jest zdaniem (a więc nie jest zdaniem prawdziwym), lecz jest funkcją zdaniową, mianowicie funkcją powszechnie prawdziwą, jeżeli tylko przyjąć, że za występujące zmienne wolne można podstawić jedynie jednostkowe nazwy liczb. Można natomiast powiedzieć, że każde twierdzenie matematyczne jest tezą.

Piśmiennictwo: *Greniewski H.* G.2.1.

3.3.7. Operatory

Wiemy już, co to są zmienne i funkcje. Pozostały nam jeszcze do omówienia operatory. W wielu językach sztucznych i mieszanych występują wyrażenia takiego rodzaju, jak:
(1) *dla każdego x,*
(2) *istnieje chociaż jedno x takie, że,*
(3) *istnieje najwyżej jedno x takie, że,*
(4) *istnieje dokładnie jedno x takie, że,*
(5) *dla żadnego x,*
(6) *jedyne x takie, że.*
Żadne z powyższych wyrażeń nie jest zmienną (choć każde zawiera w sobie jakąś zmienną), żadne z tych wyrażeń nie jest funkcją (gdyż w żadnym z nich, jak łatwo się przekonać, nie występuje ani jedna zmienna wolna). Każde z tych wyrażeń jest operatorem.

Wyjaśnimy teraz ogólnie, co to są operatory.

3.3.7.00. **Wyjaśnienie**. Mówimy, że dane wyrażenie jest *operatorem*, zamiast mówić, że spełnione są wszystkie warunki następujące:

1) Wyrażenie dane zawiera, chociaż jedną zmienną związaną i nie zawiera żadnej zmiennej wolnej;
2) Istnieje - chociaż jedna funkcja taka, że dla każdej zmiennej związanej występującej w danym wyrażeniu istnieje - chociaż jedna zmienna zawarta w tej funkcji i równokształtna ze zmienną związaną, występującą w danym wyrażeniu;
3) Dołączając do danego wyrażenia funkcję, o której mowa w warunku 2), otrzymujemy wyrażenie.

3.3.7.01. **Wyjaśnienie**. Mówimy, ze funkcja jest *argumentem operatora*, zamiast mówić, ze jest to taka funkcja, która dołączona do tego operatora daje łącznie z nim wyrażenie.

Zajmiemy się teraz wyróżnieniem różnego rodzaju operatorów. Jak widać z wyjaśnienia 3.3.7.00, można rozróżniać rozmaite rodzaje operatorów w zależności od następujących danych:

1) Rodzaju wyrażenia, jakie otrzymamy - jako wynik dołączenia operatora do funkcji (argumentu);
2) Rodzaju i ilości zmiennych związanych występujących w operatorze;
3) Rodzaju funkcji, które można dołączyć do operatora, czyli rodzaju argumentu.

Ad 1). Będziemy odróżniali *operatory zdaniotwórcze, nazwotwórcze* oraz *funktorotwórcze*.
Ad 2). Będziemy odróżniali *operatory wiążące zmienne zdaniowe, zmienne nazwowe i zmienne funktorowe*.
Ad 3). Będziemy odróżniali *operatory stosowalne do funkcji zdaniowych, funkcji nazwowych* i *funkcji funktorowych*.

3.3.7.10. **Wyjaśnienie**. Mówimy, że *operator* (1) *jest zdaniotwórczy*, (2) *wiąże jedną zmienną zdaniową* i (3) *jest stosowalny do funkcji zdaniowej zmiennych zdaniowych*, zamiast mówić, ze spełnione są wszystkie warunki następujące:

1) Operator dołączony do argumentu będącego funkcją zdaniową jednej zmiennej zdaniowej zawsze daje łącznie zdanie;
2) Operator zawiera dokładnie jedną zmienną zdaniową;
3) Argumentem operatora jest stale funkcja zdaniowa zmiennych zdaniowych.

Przykłady. Operatory zdaniotwórcze wiążące jedną zmienną zdaniową stosowalne do funkcji zdaniowej zmiennych zdaniowych:
(1) *przy wszelkim p*,
(2) *przy pewnym p*.
Przekonamy się przede wszystkim, ze wyrażenie (1) i wyrażenie (2) zawierają zmienne związane. W tym celu przypomnijmy sobie wyjaśnienie 3.3.1.60 i podstawmy do wyrażenia (1) i do wyrażenia (2) za zmienną zdaniową „*p*" zmienną zdaniową „*q*"; otrzymujemy jako wynik podstawienia wyrażenia należące nadal do JPM:
(3) *przy wszelkim q*,
(4) *przy pewnym q*.
Podstawmy teraz do wyrażenia (1) i do wyrażenia (2) za zmienną zdaniową „*p*" zdanie „Sokrates jest człowiekiem"; otrzymamy jako wynik podstawienia napisy nie należące do naszego języka (bezsensy):
(5) *przy wszelkim Sokrates jest człowiekiem,*

(6) *przy pewnym Sokrates jest człowiekiem*.

Będzie tak zawsze, ilekroć do wyrażenia (1) czy do wyrażenia (2) podstawimy cokolwiek innego niż zmienną zdaniową. W wyrażeniu (1) i w wyrażeniu (2), występuje zatem związana zmienna zdaniowa. Dołączmy teraz do wyrażenia (1) a także do wyrażenia (2) następującą funkcję zdaniową zmiennej zdaniowej:

fałszem jest, że p.

Jako wynik tego dołączenia otrzymamy zdania (pierwsze fałszywe, drugie prawdziwe):
(7) *Przy w wszelkim p, fałszem jest, że p*,
(8) *Przy pewnym p, fałszem jest, że p*.

Okazuje się więc, że wyrażenie (1), a także wyrażenie (2) jest operatorem (wyjaśnienie 3.3.7.00). Weźmy teraz pod uwagę następującą funkcję zdaniową dwu zmiennych zdaniowych:

jeżeli p, to q

i dołączmy ją do wyrażenia (1), a następnie do wyrażenia (2), otrzymany następujący wynik:
(9) *Przy wszelkim p, (jeżeli p, to q).*
(10) *Przy pewnym p, (jeżeli p, to q)*.

Łatwo zauważyć, że każde z tych wyrażeń jest funkcją zdaniową jednej zmiennej zdaniowej. Podobny wynik da dołączenie do naszych operatorów funkcji zdaniowej więcej niż dwu zmiennych zdaniowych (w tym jedna zmienna równokształtna z „*p*"). Opierając się na wyjaśnieniu 3.3.7.10, możemy teraz stwierdzić, że wyrażenie (1), a także wyrażenie (2) jest operatorem zdaniotwórczym wiążącym jedną zmienną zdaniową i stosowalnym do funkcji zdaniowej zmiennych zdaniowych.

3.3.7.11. **Reguła**. Do operatora zdaniotwórczego wiążącego jedną zmienną zdaniową i stosowalnego do funkcji zdaniowej zmiennych zdaniowych - dołączmy funkcję zdaniową nie jednej, lecz $(n + 1)$ zmiennych zdaniowych, z których jedna jest równokształtna ze zmienną w operatorze. Otrzymamy - jako wynik tego dołączenia nie zdanie, lecz funkcję zdaniową n zmiennych zdaniowych.

Łatwo teraz uogólnić wyjaśnienie 3.3.7.10.

3.3.7.12. **Wyjaśnienie**. Mówimy, że operator (1) jest zdaniotwórczy, (2) wiąże n zmiennych zdaniowych i (3) jest stosowalny do funkcji zdaniowej przynajmniej n zmiennych zdaniowych, zamiast mówić, że spełnione są wszystkie warunki następujące:

1) Operator dołączony do argumentu będącego funkcją zdaniową n zmiennych zdaniowych daje łącznie zdanie;
2) Operator zawiera dokładnie n zmiennych zdaniowych;
3) Argumentem operatora jest stale funkcja zdaniowa przynajmniej n zmiennych zdaniowych.

Przykłady. Operatory zdaniotwórcze wiążące dwie zmienne zdaniowe i stosowalne do funkcji zdaniowej przynajmniej dwu zmiennych zdaniowych:
(1) przy wszelkich p, q,
(2) przy pewnych p, q,
(3) przy każdym p i dla pewnego q,
(4) przy pewnym p to dla każdego q.
Zajmiemy się obecnie dalszymi rodzajami operatorów zdaniotwórczych. Wejdą teraz w grę operatory wiążące zmienne jednostkowo nazwowe.

3.3.1.20. **Wyjaśnienie**. Mówimy, że dany operator (1) jest zdaniotwórczy (2) wiąże jedną zmienną jednostkowo nazwową i (3) jest stosowalny do funkcji zdaniowej zmiennych jednostkowo nazwowych, zamiast mówić, że spełnione są wszystkie warunki następujące:

1) Operator dołączony do argumentu będącego funkcją zdaniową jednej zmiennej jednostkowo nazwowej daje łącznie zdanie;
2) Operator zawiera dokładnie jedną zmienną jednostkowo nazwową;
3) Argumentem operatora jest stale funkcja zdaniowa zmiennych jednostkowo nazwowych.

Przykłady. Operatory zdaniotwórcze wiążące jedną zmienną jednostkowo nazwową i stosowalne do argumentu, którym jest funkcja zdaniowa jednej zmiennej jednostkowo nazwowej:
(1) *przy wszelkim* x,
(2) *przy żadnym* x,
(3) *przy przynajmniej pewnym* x,
(4) *przy tylko pewnych* x.

Weźmy teraz pod uwagę funkcję zdaniową jednej zmiennej nazwowej:
(5) x *jest człowiekiem*.

Dołączając do któregokolwiek z operatorów (1) - (4) funkcję zdaniową (5) otrzymujemy zawsze zdanie:
(6) Przy wszelkim x, (x jest człowiekiem).
(7) Przy żadnym x, (x jest człowiekiem)[41].
(8) Przy przynajmniej pewnym x, (x jest człowiekiem).
(9) Przy tylko pewnych x, (x jest człowiekiem).
Opierając się na wyjaśnieniach 3.3.7.20 i na dostatecznie licznych próbach (o charakterze doświadczalnym) można uzasadnić trafność reguły następującej:

3.3.7.21. **Reguła**. Dołączając
- do operatora zdaniotwórczego wiążącego jedną zmienną jednostkowo nazwową i stosowalnego do funkcji zdaniowej zmiennych jednostkowo nazwowych
- funkcję zdaniową $(n + 1)$ zmiennych jednostkowo nazwowych, z których jedna jest równokształtna ze zmienną zawartą w operatorze,

otrzymujemy - jako wynik funkcję zdaniową n zmiennych jednostkowo nazwowych.
Przykład. Bierzemy pod uwagę znany już operator:
(1) przy wszelkim x,
oraz funkcję zdaniową dwu zmiennych jednostkowo nazwowych:
(2) x jest mniejsze od y,
jako wynik dołączenia operatora (1) do funkcji zdaniowej (2) otrzymamy wyrażenie (3):
(3) Przy wszelkim x, (x jest mniejsze od y).
Łatwo zauważyć, że ostatnie wyrażenie nie jest zdaniem, lecz funkcją zdaniową jednej zmiennej jednostkowo nazwowej (mianowicie zmiennej wolne „y”).

Uogólnijmy teraz wyjaśnienie 3.3.7.20.

3.3.7.22. **Wyjaśnienie**. Mówimy, że dany operator (1) jest zdaniowo twórczy, (2) wiąże n zmiennych jednostkowo nazwowych i (3) jest stosowalny do funkcji zdaniowej przynajmniej n

[41] Ze względów wyłącznie tradycyjnych w języku polskim mówi się i pisze „przy żadnym x, x nie jest człowiekiem” zamiast „przy żadnym x, x jest człowiekiem”.

zmiennych jednostkowo-nazwowych, zamiast mówić, że spełnione są wszystkie warunki następujące:

1) Operator dołączony do argumentu będącego funkcją zdaniową n zmiennych jednostkowo nazwowych daje łącznie zdanie;
2) Operator zawiera dokładnie n zmiennych jednostkowo nazwowych;
3) Argumentem operatora jest stale funkcja zdaniowa przynajmniej n zmiennych jednostkowo nazwowych.

Przykłady. Operatory zdaniotwórcze wiążące dwie zmienne jednostkowo-nazwowe oraz stosowalne do funkcji zdaniowe o przynajmniej dwu zmiennych jednostkowo-nazwowych:

(1) przy wszelkich x, y,
(2) przy żadnych x, y,
(3) przy wszelkich x i niektórych tylko y.

3.3.7.30. **Wyjaśnienie**. Mówimy, że dane wyrażenie jest *kwantyfikatorem*, zamiast mówić, ze jest ono operatorem zdaniotwórczym stosowalnym do funkcji zdaniowej.

Zajmiemy się teraz operatorami nazwotwórczymi.

3.3.7.40. **Wyjaśnienie**. Mówimy, że dany operator (1) jest nazwotwórczy, (2) wiąże jedną zmienną jednostkowo nazwową i (3) jest stosowalny do funkcji zdaniowej zmiennych jednostkowo nazwowych, czyli mówimy krótko, że dany operator jest deskryptem, zamiast mówić, ze spełnione, są wszystkie warunki następujące:

1) Operator dołączony do argumentu będącego funkcją zdaniową jednej zmiennej jednostkowo nazwowej daje łącznie nazwę jednostkową lub pustą;
2) Operator zawiera dokładnie jedną zmienną jednostkowo nazwową;
3) Argumentem operatora jest stale funkcja zdaniowa zmiennych jednostkowo nazwowych.

Przykład. Deskryptem jest wyrażenie:
(1) *jedyny x taki, że,*
Weźmy teraz pod uwagę funkcję zdaniową jednej zmiennej jednostkowo nazwowej:
(2) *x jest bratem Piotra*.
Dołączmy wyrażenie (1) do wyrażenia (2), otrzymamy jako wynik nazwę:
(3) *jedyny x taki, że x jest bratem Piotra*.

Jeżeli Piotr ma (miał czy będzie miał) dokładnie jednego brata, to sprawa przedstawia się jasno, nazwa (3) jest nazwą jednostkową. Jeżeli natomiast Piotr nigdy w swym życiu nie miał i nie będzie miał ani jednego brata, czy też miał, ma lub będzie miał więcej niż jednego brata, to nie jest na pierwszy rzut oka jasne, jaką nazwą jest wyrażenie (3). Myślę jednak, że bez jawnego pogwałcenia naszej praktyki językowej można przyjąć, że zarówno w razie nieistnienia braci, jak też w razie istnienia więcej niż jednego brata wyrażenie (3) jest nazwą pustą.

Teraz jeszcze kilka słów o operatorach funktorotwórczych.

3.3.7.50. **Wyjaśnienie**. Mówimy, ze dany operator (1) jest zdaniowo funktorotwórczy, (2) wiąże jedną zmienną jednostkowo nazwową i (3) jest stosowalny do funkcji zdaniowej zmiennych jednostkowo nazwowych, zamiast mówić, że spełnione są wszystkie warunki następujące:

1) Operator dołączony do argumentu będącego funkcją zdaniową jednej zmiennej jednostkowo nazwowej daje łącznie funktor zdaniotwórczy od jednego argumentu jednostkowo nazwowego;
2) Operator zawiera dokładnie jedną zmienną jednostkowo nazwową;

3) Argumentem operatora jest stale funkcja zdaniowa zmiennych jednostkowo nazwowych.

Przykład. Operator zdaniowo – funktorotwórczy wiążący jedną zmienną jednostkowo nazwową i stosowalny do funkcji zdaniowej zmiennych jednostkowo nazwowych:
(1) *jest takim x, że.*
Weźmy teraz pod uwagę funkcję zdaniową jednej zmiennej nazwowej:
(2) x jest biały lub x jest czarny.
Dołączmy operator (1) do funkcji zdaniowej (2); otrzymamy wyrażenie:
(3) *jest taki x, że x jest biały lub x jest czarny.*
Łatwo zauważyć, że otrzymaliśmy funktor zdaniotwórczy od jednego argumentu jednostkowo nazwowego, występujący na przykład w zdaniu:
Każdy łabędź jest takim x, że x jest biały lub x jest czarny.

3.3.7.51. **Wyjaśnienie**. Mówimy, że dany operator (1) jest zdaniowo funktorotwórczy, (2) wiąże n zmiennych jednostkowo nazwowych i (3) jest stosowalny do funkcji zdaniowej przynajmniej n zmiennych jednostkowo nazwowych, czyli mówimy krótko, że dany operator jest symbolem abstrakcji, zamiast mówić, ze spełnione są wszystkie warunki następujące:

1) Operator dołączony do argumentu będącego funkcją zdaniową n zmiennych jednostkowo nazwowych zawsze daje łącznie funktor zdaniotwórczy od n argumentów jednostkowo nazwowych;
2) Operator zawiera dokładnie n zmiennych jednostkowo nazwowych;
3) Argumentem operatora jest stale funkcja zdaniowa przynajmniej n zmiennych jednostkowo nazwowych.

Przykład. Operator zdaniowo funktorotwórczy wiążący dwie zmienne jednostkowo nazwowe i stosowalny do funkcji zdaniowej przynajmniej dwóch zmiennych jednostkowo nazwowych:
(1) *są taką parą uporządkowaną* y, x, że …
Weźmy teraz pod uwagę funkcję zdaniową dwu zmiennych jednostkowo nazwowych:
(2) y, *jest młodszym bratem* x.
Dołączmy teraz operator (1) do funkcji zdaniowej (2); jako wynik dołączenia otrzymamy wyrażenie:
(3) *są taką parą uporządkowaną* y, x, *że* y *jest młodszym bratem* x.

Wyrażenie (3) jest funktorem zdaniotwórczym od dwu argumentów jednostkowo nazwowych. Łatwo się o tym przekonać dołączmy do funktora (3) dwie nazwy jednostkowe:
(4) $Adam$,
(5) $Bolesław$.
Wynikiem dołączenia do funktora (3) nazwy (4) i nazwy (5) będzie zdanie poniższe:
(6) $Adam, Bolesław$ są taką parą uporządkowaną y, x, że y jest młodszym bratem x.

Operator zdaniowo funktorotwórczy wiążący trzy zmienne jednostkowo nazwowe i stosowalny do funkcji zdaniowej przynajmniej trzech zmiennych jednostkowo nazwowych:

są taką trojką uporządkowaną x_1, x_2, x_3, że …

Analogiczny operator zawierający cztery zmienne:

są taką czwórką uporządkowaną x_1, x_2, x_3, x_4, że …

Wreszcie zajmiemy się tylko jednym rodzajem operatorów nazwowo funktorotwórczych.

Tablica 3.3.7.70					
Znak wyjaśnienia	Rodzaj operatorów	Charakterystyka operatorów ze względu na			Przykłady
		Wynik dołączeń	Zmienne związane	Argument	
I	*II*	*III*	*IV*	*V*	*VI*
3.3.7.12	Kwantyfikatory	Zdaniotwórczy	n zmiennych zdaniowych	Funkcja zdaniowa przynajmniej n zmiennych zdaniowych	1) przy wszelkim p 2) przy pewnym p 3) dla wszystkich p, q 4) dla pewnych p, q
3.3.7.22		Zdaniotwórczy	n zmiennych jednostkowo zdaniowych	Funkcja zdaniowa przynajmniej n zmiennych jednostkowo nazwowych	1) przy wszelkim x 2) przy żadnym x 3) przy tylko pewnych x 4) przy wszelkich x, y 5) przy żadnych x, y
3.3.7.40	Deskrypty	Nazwo twórczy	Jedna zmienna jednostkowo nazwowa	Funkcja zdaniowa zmiennych jednostkowo nazwowych	Jedyny x, taki że
3.3.7.51	Symbole abstrakcji	Zdaniowo funktoro-twórczy	n zmiennych jednostkowo zdaniowych	Funkcja zdaniowa przynajmniej n zmiennych jednostkowo nazwowych	Są taką n - tką uporządkowaną $x_1, \ldots, x_n$, że
3.3.7.60		Nazwowo funktoro-twórczy	Jedna zmienna jednostkowo nazwowa	Funkcja jednostkowo nazwowa zmiennych jednostkowo nazwowych	$\frac{d}{dx}$ $\int \ldots \mathrm{dx}$

3.3.7.60. **Wyjaśnienie**. Mówimy, że dany operator (1) jest nazwowo funktorotwórczy, (2) wiąże jedną zmienną jednostkowo nazwową i (3) jest stosowalny do funkcji jednostkowo nazwowej zmiennych jednostkowo nazwowych, zamiast mówić, że spełnione są wszystkie warunki następujące:

1) Operator dołączony do argumentu będącego funkcją jednostkowo nazwową jednej zmiennej jednostkowo nazwowej daje łącznie funktor jednostkowo nazwotwórczy od jednego argumentu jednostkowo nazwowego;
2) Operator zawiera dokładnie jedną zmienną jednostkowo nazwowych;

Argumentem operatora jest stale funkcja jednostkowo nazwowa zmiennych jednostkowo nazwowych.

Przykład. Operatorami nazwowo funktorotwórczymi wiążącymi jedną zmienną jednostkowo nazwową i stosowalnymi do funkcji jednostkowo nazwowej zmiennych jednostkowo nazwowych są dwa zasadnicze symbole analizy matematycznej: symbol pochodnej (nie cząstkowej) i symbol całki nieoznaczonej. Celem ułatwienia czytelnikowi orientację załączamy jeszcze tablicę zbitych rodzajów operatorów, o których wyżej była mowa (tablica 3.3.7.70).

Piśmiennictwo: *Greniewski H.* G.2.1. , *Kuratowski K.* K.7.1, *Mostowski A.* M.4.1.

3.3.8. „UBOGIE" JĘZYKI SZTUCZNE

Na początku rozdziału niniejszego (podrozdział 3.3.0), napisaliśmy: „W językach sztucznych i mieszanych znajdujemy obok wyrażeń spotykanych w językach naturalnych, mianowicie zdań, nazw i funktorów, także innego rodzaju wyrażenia, nie spotykane w ogóle w językach naturalny: zmienne, funkcje i operatory".

Z tego, że w językach sztucznych możemy spotkać takie rodzaje wyrażeń (np. zmienne, funkcje czy operatory), nie wynika, żebyśmy w każdym języku sztucznym musieli napotkać wyrażenia wszystkich tych rodzajów. W podrozdziale 3.0.2 – wymieniliśmy, jako przykład języka sztucznego - *międzynarodowy system znaków drogowych*. Otóż system ten nie zawiera ani jednej

zmiennej żadnego rodzaju. Wobec tego nie zawiera on ani funkcji, ani operatorów, kiedy przystępujemy do produkcji, gromadzimy najpierw środki potrzebne do jej rozpoczęcia. Gromadzenie środków zbędnych w danej produkcji jest marnotrawstwem. Otóż coś podobnego zachodzi przy budowaniu języków sztucznych.

Każdy język sztuczny jest tworzony świadomie (wyjaśnienie 3.0.2.10), dla określonych celów. Język sztuczny tworzymy w ten sposób, aby obsługując potrzeby, dla których zaspokojenia go tworzymy zawierał wszystkie wyrażenia do tego niezbędne i aby nie zawierał w miarę możliwości żadnych wyrażeń zbędnych. Staramy się, mówiąc inaczej, w taki sposób budować języki sztuczne, aby były one możliwie „ubogie".

Piśmiennictwo: *Greniewski H.* G.2.1.

3.3.9. XML - BARDZO WAŻNY SZTUCZNY JĘZYK

Teraz zajmiemy się pewnym językiem sztucznym o bardzo istotnym znaczeniu nie tylko dla informatyki. XML (skrót od *eXtensible Markup Language*) – to rozszerzalny język znaczników i jeden z najważniejszych języków sztucznych, jaki dotychczas opracowano. XML nie jest językiem programowania. XML nie jest protokółem transportu sieciowego. XML nie jest bazą danych. Natomiast XML oferuje możliwość formatowania danych; zapewnia ich prawdziwą wielo-platformowość i odporność na upływ czasu, a tym samym jest bardzo przydatnym narzędziem tworzenia dokumentacji i operowania dokumentami.

3.3.9.10. **Wyjaśnienie**. XML jest prostym, poprawnie udokumentowanym i przyjaznym w obsłudze formatem danych. Dokumenty XML są tekstem i dlatego każdy edytor plików tekstowych, jest w stanie przeczytać dokument XML. Tekstem są również znaczniki występujące w pliku XML'owym. Na ogół, aby dowiedzieć się, co zawiera dokument XML – wystarczy przeczytać nazwy znaczników. Znaczniki XML przypominają na pierwszy rzut oka znaczniki HTML. Występuje tu jednak podstawowa różnica: HTML operuje około 100 predefiniowanymi znacznikami; natomiast w XML użytkownik może użyć tylu różnych znaczników – ile mu aktualnie potrzeba. Warto podkreślić, że XHTML – jest tzw aplikacją XML, operującą predefiniowanymi znacznikami.

Zajmijmy się z kolei składnią języka XML:

3.3.9.11. **Reguła**. Znacznik początkowy zaczyna się od znaku „<", a znacznik końcowy zaczyna się od znaku „</". Po obu tych częściach składowych następuje nazwa znacznika, zwana elementem; oba znaczniki zamykane są znakiem >. Jednakże, w przeciwieństwie do XHTML, użytkownik w miarę potrzeb tworzy nowe znaczniki. Przykładowo dla opisania osoby używamy pary znaczników: `<osoba>` i `</osoba>` - pomiędzy którymi umieszczamy tekst - opisujący daną osobę. Na ogół, nazwy znaczników odzwierciadlają rodzaj zawartości znajdującej się wewnątrz elementu, a nie sposób, w jaki zawartość będzie formatowana.
Przykład: dokumentu XML częściowego opisu osoby (tzw. prostego drzewa XML):

```
<osoba>
      <nazwa>
            <imię>Alan</imię>
            <nazwisko>Turing</nazwisko>
      </nazwa>
            <zawód>informatyk</zawód>
            <zawód>matematyk</zawód>
```

```
            <zawód>kryptografik</zawód>
</osoba>
```

3.3.9.12. **Reguła**. Obok elementów zawierających tekst, XML umożliwia korzystanie z tzw. pustych elementów, czyli elementów bez zawartości. Te elementy są reprezentowane przez znacznik, zaczynający się od znaku `<`, a kończący się znakami `/>`. Na przykład elementy: nowego wiersza to `<br/>`, a linii poziomej to `<hr/>`.

3.3.9.13. **Reguła**. Elementy rodzicielskie i potomne. Dokument z powyższego przykładu zbudowany jest z jednego elementu osoba i czterech elementów potomnych: nazwa i trzech elementów zawód. Z kolei element nazwa, zawiera dwa elementy pochodne: imię i nazwisko. Element osoba nazywa się przodkiem nazwa i trzech elementów zawód.

3.3.9.14. **Reguła**. Element bazowy. Każdy dokument XML, zawiera jeden element nieposiadający przodka. Jest to pierwszy element w dokumencie, który jest przodkiem dla wszystkich pozostałych elementów danego dokumentu. W powyższym przykładzie elementem bazowym jest osoba.

3.3.9.15. **Reguła**. Atrybuty. Elementy XML mogą posiadać atrybuty. Atrybutem jest para nazwa-wartość, dołączona do początkowego znacznika elementu. Nazwy są oddzielone od wartości znakiem równości i opcjonalnie może się tam znaleźć spacja. Wartości są zawarte w pojedynczym lub podwójnym cudzysłowie. Wartością atrybutu jest ciąg znaków (tzw. sekwencja znaków).
Przykład: dokument XML częściowego opisu osoby z użyciem atrybutów:

```
<osoba urodzony="1912/06/23" zmarł="1954/06/07">
      <nazwa imię="Alan" nazwisko="Turing"/>
      <zawód value="informatyk"/>
      <zawód value="matematyk"/>
      <zawód value="kryptografik"/>
</osoba>
```

3.3.9.16. **Wyjaśnienie**. Nie podlega dyskusji, że każdy element może posiadać najwyżej jeden atrybut opatrzony daną nazwą. Narzuca to więc - ograniczenia na posługiwanie się atrybutami. Struktura oparta o elementy jest znacznie bardziej elastyczna – od struktury, w której elementy wyposażone są w atrybuty. Nie mniej jednak, w niektórych zastosowaniach – wykorzystywanie atrybutów jest celowe i wygodniejsze.

3.3.9.20. **Reguła**. Element oraz inne nazwy XML mogą zawierać dowolne znaki alfanumeryczne (tzn. litery alfabetu angielskiego od A do Z oraz od a do z, a także cyfry 0 do 9; nie angielskie litery, jak również trzy znaki interpunkcyjne: _ podkreślenie, - myślnik, . kropka). Nazwy XML nie mogą zawierać innych znaków interpunkcyjnych (tzn. cudzysłów, apostrof, znak dolara, znak daszka, symbole procentów i średnik). Dwukropek jest dozwolony, lecz jego użycie jest zarezerwowane dla przestrzeni nazw. Nazwy XML - mogą rozpoczynać się tylko od liter lub znaku podkreślenia. Nazwy XML nie mogą zaczynać się od: cyfry, myślnika oraz kropki.

3.3.9.21. **Reguła**. Tekst elementu podobnie jak nazwy XML może zawierać dowolne znaki alfanumeryczne (tzn. litery alfabetu angielskiego od A do Z oraz od a do z, a także cyfry 0 do 9; nie angielskie litery, jak również trzy znaki interpunkcyjne: _ podkreślenie, - myślnik, . kropka). Tekst nie może zawierać w formie jawnej znaków: `<`, `&`, `>`, `"`, `'`. W przypadku konieczności

użycia, któregoś z wymienionych znaków – należy posłużyć się tzw. odwołaniem do jednej z encji: `&alt;` - zamiast znaku <, `&` - zamiast znaku &, `>` - zamiast znaku >, `"` - zamiast znaku ", `'` - zamiast znaku '. Odwołania do encji spowodują wyświetlenie (wydrukowanie) w tekście, znaku odpowiadającego użytej encji.

3.3.9.22. **Wyjaśnienie**. Oprócz tych pięciu predefiniowanych odwołań do encji – wymienionych w wyżej podanej regule (3.3.8.21) – w definicji typu dokumentu możemy definiować inne odwołania do encji. Dalsze reguły określą jak to robić.

3.3.9.23. **Reguła** posługiwania się sekcją CDATA. Aby umieścić w dokumencie XML sekwencję znaków, która ma nie być przetwarzana przez procesor XML (np. jest to kod programu komputerowego, który jest umieszczony wewnątrz dokumentu XML), należy umieścić tę sekwencję znaków w sekcji CDATA. Sekcja ta ma format:

```
<![CDATA[ miejsce-na-sekwencję-znaków ]]>.
```

3.3.9.24. **Reguła** pisania komentarzy. Dokumenty XML mogą być opatrywane komentarzami, np. wyjaśniającymi przyjęte w dokumencie idee autora. Komentarze XML mają analogiczną składnię do komentarzy w HTML. Komentarz ma format: `<-- miejsce na komentarz -->`.

3.3.9.25. **Reguła** dotycząca instrukcji przetwarzania XML. Instrukcje przetwarzania mogą podobnie jak komentarze występować w dokumencie XML. W odróżnieniu jednak od komentarzy, instrukcja przetwarzania steruje pracą procesora XML w toku przetwarzania dokumentu XML. Najczęściej występującą instrukcją przetwarzania w dokumentach xml-stylesheet, dołączająca arkusz styli prezentacji dokumentu. Instrukcja ta ma następujący format dla arkusza styli CSS np. o nazwie osobiste.css:

```
<?xml-stylesheethref="osobiste.css" type="text/css"?>.
```

3.3.9.26. **Reguła** tworzenia deklaracji XML. Dokument powinien zaczynać się od deklaracji XML. Deklaracja - składa się z nagłówka oraz trzech atrybutów.

1. Nagłówek deklaracji wygląda następująco: `<?xml –miejsce-na-atrybuty- ?>`.
2. Atrybut version zawsze ma wartość 1.0: `version="1.0"`.
3. Atrybut `encoding` przyjmuje wartość jednego z kodów Unicode, np.: `encoding="UTF-8"`.
4. Atrybut `standalone` może przyjmować jedną z dwu wartości `no` albo `yes`.
   - Jeśli atrybut ma wartość `no`, oznacza to, że deklaracja DTD, o której dalej, jest umieszczona w pliku zewnętrznym w stosunku do danego dokumentu XML.
   - Jeśli atrybut ma wartość `yes`, oznacza to, że deklaracja DTD - jest umieszczona wewnątrz danego dokumentu XML.

Przykładowa deklaracja XML:

```
<?xml version="1.0" encoding="UTF-8" standelone="yes" ?>.
```

3.3.9.27. **Reguła** tworzenia definicji dokumentu DTD. Definicja DTD umożliwia określenia struktury dokumentu XML, na poziomie elementów i ich atrybutów, oraz nałożenie ograniczeń na postać dokumentu. Najprostszy model zawartości elementu – to model, który określa, że element zawiera tylko analizowane dane tekstowe, ale nie może zawierać elementów potomnych żadnego typu. W takim przypadku, model zawartości elementu zawiera jedynie słowo kluczowe `#PCDATA`. Na strukturę dokumentu składają się definicje elementów (każdy

element, może zawierać wiele elementów potomnych) i definicji atrybutów dla poszczególnych elementów:

1. Definicja elementu składa się ze słowa kluczowego `ELEMENT`, nazwy elementu oraz listy elementów bezpośrednio potomnych (zawartej w nawiasach) - dla danego elementu zwanej modelem zawartości elementu, np.:

   ```
   <!ELEMENT osoba (nazwa^, zawód^)>.
   ```

   Uwaga: symbol ^ oznacza: (a) brak symbolu – element podrzędny występuje dokładnie jeden raz; (b) symbol ma wartość ? – zezwala na zerowe lub jednokrotne wystąpienie elementu podrzędnego; (c) symbol ma wartość * - zezwala na zerowe lub wielokrotne wystąpienie elementu podrzędnego; (d) symbol ma wartość + - wymaga przynajmniej jednokrotnego wystąpienia elementu podrzędnego. Puste elementy (zapisane - jako pojedynczy znacznik z zamykającym go znakiem />), deklaruje się przy użyciu słowa kluczowego EMPTY, użytego w odniesieniu do modelu zawartości elementu, np.:

   ```
   <!ELEMENT obraz EMPTY>.
   ```

   Niedokładne definicje DTD czasami stwierdzają, że istniej element, ale nie wskazują, co może lub nie może on zawierać. W takim przypadku, jako modelu zawartości można użyć słowa kluczowego ANY, np.:

   ```
   <!ELEMENT strona ANY>.
   ```

2. Definicji elementu może towarzyszyć lista atrybutów danego elementu, z podaniem typu (np. `CDATA`), oraz konieczności wystąpienia (`REQUIRED`), albo możliwości wystąpienia (`IMPLIED`) lub (`FIXED`) - np.:

   ```
   <!ELEMENT osoba (nazwa, zawód*)>
   <!ELEMENT nazwa>
   <!ATTLIST nazwa imię CDATA #REQUIRED
                   nazwisko CDATA #REQUIRED>
   <!ELEMENT zawód>
   <!ATTLIST zawód value CDATA #IMPLIED>.
   ```

   Uwaga: w XML występuje dziesięć typów atrybutów (a) `CDATA`; (b) `NMTOKEN`; (c) `NMTOKENS`; (d) wyliczenie; (e) `ENTITY`; (f) `ENTITIES`; (g) `ID`; (h) `IDREF`; (i) `IDREFS`; (j) `NOTATION`. DTD nie może np. określać, że wartością atrybutu ma być liczba całkowita albo data kalendarzowa.

3.3.9.28. **Reguła** deklaracji typu dokumentu. Deklaracja typu dokumentu – określa:

1. Nazwę elementu bazowego danego dokumentu, np. osoba;
2. Gdzie znajduje się definicja `DTD` dotycząca danego dokumentu XML. Mamy dwie możliwości:
   - `DTD` jest częścią danego dokument, wówczas deklaracja typu dokumentu ma postać:

     ```
     <!DOCTYPE osoba [-miejsce-na-definicję-DTD-]>;
     ```

   - `DTD` znajduje się w innym pliku, wówczas podawany jest np. adres URI pliku:

     ```
     <!DOCTYPE osoba SYSTEM ”http://ibiblio.org/xml/osoba.dtd”>.
     ```

Przykład: dokument XML zawierającego `DTD` i deklarację typu - opisu osoby bez atrybutów:

```
<?xml version=”1.0” encoding=”UTF-8” standelone=”yes” ?>
<!DOCTYPE osoba [
<!ELEMENT osoba (nazwa, zawód*)>
<!ELEMENT zawód (#PCDATA)>
<!ELEMENT nazwa (imię, nazwisko)>
<!ELEMENT imię (#PCDATA)>
<!ELEMENT nazwisko (#PCDATA)>
]>
<osoba>
      <nazwa>
```

```
            <imię>Alan</imię>
            <nazwisko>Turing</nazwisko>
        </nazwa>
            <zawód>informatyk</zawód>
            <zawód>matematyk</zawód>
            <zawód>kryptografik</zawód>
</osoba>
```

Przykład: dokument XML zawierającego `DTD` i deklarację typu - opisu osoby z użyciem atrybutów:

```
<?xml version="1.0" encoding="UTF-8" standelone="yes" ?>
<!DOCTYPE osoba [
<!ELEMENT osoba (nazwa, zawód*)>
<!ELEMENT nazwa>
<!ATTLIST nazwa imię CDATA #REQUIRED
                nazwisko CDATA #REQUIRED
>
<!ELEMENT zawód>
<!ATTLIST zawód value CDATA #IMPLIED
>]>
<osoba >
            <nazwa imię="Alan" nazwisko="Turing"/>
            <zawód value="informatyk"/>
            <zawód value="matematyk"/>
            <zawód value="kryptografik"/>
        </osoba>
```

3.3.9.29. **Wyjaśnienie**. Ze względu na ograniczone potrzeby niniejszego wykładu, problematyka modeli zawartości elementów została ograniczona do opisania paru podstawowych wariantów. Dla zapoznania się z całością możliwości oferowanych przez XML odsyłamy do piśmiennictwa.

3.3.9.30. **Wyjaśnienie**. Przez przestrzeń nazw – rozumiana jest następująca opcja centralnej składnicy standardowych definicji DTD, do której swobodny dostęp mają użytkownicy języka XML. Poszczególne grupy definicji, są umieszczone w sieci pod określonymi adresami URI, do której odwołuje się użytkownik – zgonie z regułą 3.3.9.28. Dla zapewnienia unikalności nazw elementów i atrybutów, nazwy zadeklarowane w standardowych definicjach – zostają poprzedzone prefiksem zadeklarowanym przez użytkownika i oddzielone od oryginalnej nazwy ze standardowej definicji znakiem dwukropka.

3.3.9.31. **Wyjaśnienie**. W podrozdziale 3.0.1, przytoczyliśmy zabawną anegdotę dotyczącą wpływu intonacji na rozumienie treści listu. W języku naturalnym - intonacja jest jednym z istotnych narzędzi prezentacji akustycznej. W pewnym sensie, odpowiednikiem w języku XML intonacji, jest możliwość eksponowania wybranych fragmentów dokumentu – w toku prezentacji, korzystając z jednego z dwu dostępnych narzędzi SSL (*Style Sheet Language*) lub XSLT (*ExtensibleStylesheet Language Transformations*).

3.3.9.32. **Wyjaśnienie**. Język XML jest podstawą tzw. administracji elektronicznej. W celu, projektuje się odpowiednie formularze z wykorzystaniem funkcji skrótu szyfrowego i podpisu elektronicznego, dla zapewnienia autentyczności oryginalnego dokumentu.

3.3.9.33. **Wyjaśnienie**. Językowi XML towarzyszy wiele aplikacji – tworzonych na drodze wstępnego zdefiniowania zbioru znaczników dziedzinowych dla danej aplikacji. Są to przykładowo - następujące aplikacje XML:

1. XHTML – aplikacja, która zastąpiła HTML używany do budowania stron www.
2. SVG – aplikacja skalowalnej grafiki wektorowej, zalecana przez Konsorcjum W3C – standard kodowania rysunków, wykorzystywany w większości pakietów grafiki wektorowej.
3. MathML – aplikacja znaczników do zastosowań matematycznych, np. osadzania równań matematycznych na stronach www, zalecana przez Konsorcjum W3C.
4. CML – aplikacja znaczników do zastosowań chemicznych, zalecana przez Konsorcjum W3C.
5. RDF – aplikacja będąca schematem opisu zasobów, np. bibliotecznych, zalecana przez Konsorcjum W3C.

Piśmiennictwo: *Harold E.* H.2.1., *Mercer D.* M.4.1., *W3C* W.8.1.

3.4. DEFINIOWANIE

3.4.0. POJĘCIE DEFINICJI

Dotychczas interesowaliśmy się tylko rozróżnianiem rodzajów języków i odróżnianiem rodzajów wyrażeń w obrębie danego języka, pomijaliśmy natomiast uporczywie sprawę zmian zachodzących w obrębie poszczególnych języków.

Teraz zajmiemy się w pewnym stopniu takimi właśnie zmianami. Piszemy „w pewnym stopniu", gdyż badanie całości zmian, którym podlega język, wykracza poza kompetencje logiki. Zagadnienia zmian zachodzących w języku będą nas tu głównie interesowały w związku, z procesem definiowania. W tym celu, wyjaśnimy najpierw wyrażenie „definicja", a następnie wyrażenie „definiowanie".

3.4.0.00**. Wyjaśnienie**. Przez „*definicję*" rozumiemy każdy i zarazem tylko taki komunikat, który ma postać: *Używamy wyrażeń równokształtnych z następującym: ... zamiast wyrażeń równokształtnych z następującym : ...*

3.4.0.01. **Wyjaśnienie**. Skrót „$=_{Df}$" będzie używany zamiast rozwlekłego zwrotu: *Używamy wyrażeń równokształtnych z następującym: ... zamiast wyrażeń równokształtnych z: ...*

3.4.0.10. **Wyjaśnienie**. Przez „*definiens*" (w węższym rozumieniu) danej definicji rozumiemy wyrażenie występujące w danej definicji po skrócie równokształtnym z „$=_{Df}$".

3.4.0.11. **Wyjaśnienie**. Przez „*definiendum*" (w węższym rozumieniu) danej definicji rozumiemy wyrażenie występujące w danej definicji przed skrótem równokształtnym z „$=_{Df}$".

3.4.0.12. **Wyjaśnienie dodatkowe**. Wyraz „*po*" (występujący w wyjaśnieniu 3.4.0.10) i wyraz „*przed*" (występujący w wyjaśnieniu 3.4.0.11), należy - rozumieć odpowiednio do kolejności wyrażeń, obowiązującej w danym języku. Ponieważ w książce niniejszej obowiązuje kolejność „*od lewej do prawej*", więc wyraz „*po*" będziemy rozumieli - jako „*po prawej stronie*", zaś wyraz „*przed*" będziemy rozumieli - jako „*po lewej stronię*".

Przykłady. Definicje, w których łatwo wyróżniamy *definiens* i *definiendum* (w węższym rozumieniu obu tych wyrazów):

Definicja

(1) kwadrat $=_{Df}$ prostokąt równoboczny

Definiendum *Definies*

(w węższym rozumieniu) (w węższym rozumieniu)

Definicja

(2) $x \leq y$ to ($x < y$ lub $x = y$)

Definiendum $=_{Df}$ *Definies*

(w węższym rozumieniu) (w węższym rozumieniu)

W pierwszym z powyższych przykładów (definicja (1)) definiendum jest nazwą ogólną i definiens jest również nazwą ogólną, natomiast w przykładzie drugim (definicja (2)) definiendum jest funkcją zdaniową dwu zmiennych jednostkowo nazwowych (należącą do języka algebry szkolnej) i definiens jest również funkcją zdaniową dwu zmiennych jednostkowo nazwowych (należącą do tegoż języka).

3.4.0.30. **Wyjaśnienie**. Przez „*definiens* (w szerszym rozumieniu)"- danej *definicji* rozumiemy każde wyrażenie, które:

1) jest równo kształtne i równoznaczne z *definiensem* (w węższym rozumieniu)
   lub
2) jest prawidłowym (w danym języku) podstawieniem *definiensa* (w węższym rozumieniu).

3.4.0.31. **Wyjaśnienie**. Przez „*definiendum* (w szerszym rozumieniu)" - danej *definicji* rozumiemy każde wyrażenie, które:

1) jest równokształtne i równoznaczne z *definiendum* (w węższym rozumieniu)
   lub
2) jest - prawidłowym w danym języku, podstawieniem *definiendum* (w węższym rozumieniu).

Przykłady. Weźmy pod uwagę następującą definicję:

Koło $=_{Df}$ *elipsa o równych osiach*.

Wyrażenie następujące jest *definiensem* (w szerszym rozumieniu) tej definicji:

elipsa o równych osiach.

Natomiast wyrażenie następujące jest *definiendum* (w szerszym rozumieniu) tej definicji: *koło*.

Przejdźmy do następnej definicji przykładowej:

x jest rówieśnikiem y $=_{Df}$ (*x nie jest młodszy od y*) *oraz* (*x nie jest starszy od y*).

Każde z następujących czterech wyrażeń jest definiendum (w szerszym rozumieniu) tej definicji:

(1) x jest rówieśnikiem y,
(2) Adam jest rówieśnikiem y,
(3) x jest rówieśnikiem Bolesława,
(4) Adam jest rówieśnikiem Bolesława.

3.4.0.50. **Wyjaśnienie**. Mówimy, że ***A*** jest definicją wyrażenia ***B***, zamiast mówić, że wyrażenie ***B*** jest *definiendum* (w szerszym rozumieniu) definicji ***A***.

Przykład. Definicja ostatnio podana jest definicją wyrażenia „*x jest rówieśnikiem y*".

3.4.0.51. **Wyjaśnienie**. Pisząc krótko „*definiens*" czy też „*definiendum*" w obrębie dyrektyw zastępowania będziemy mieli zawsze na myśli *definiens* (ewentualnie *definiendum*) w szerszym rozumieniu.

3.4.0.60. **Wyjaśnienie**. Przez „*definiowanie*” rozumiemy budowanie definicji.

Doświadczenie uczy, że definicje mają poważne znaczenie dla rozbudowy języków sztucznych i mieszanych, a także dla analizy struktury tych języków. Pewne, znacznie skromniejsze znaczenie mają one w językach naturalnych i to dopiero w ich późnych stajach rozwojowych. Umiejętność definiowania jest w praktyce nader przydatna dla naukowca i nauczyciela. W życiu potocznym stosunkowo rzadko stosuje się definicje, chociaż bardzo często spotkać można ludzi podających rzekomo jakieś definicje czy powołujących się na jakąś definicję, którzy swym postępowaniem jaskrawo wykazują, że w ogóle nie rozumieją, czym są definicje.

Piśmiennictwo: *Greniewski H.* G.2.1.

3.4.1. Przydatność definicji w związku z konstruowaniem nowych wyrażeń

Bardzo często konstruujemy w danym języku (w sposób zgodny z jego regułami gramatycznymi) nowe wyrażenie, na przykład łączymy wyrażenia dotychczas w danym języku używane w nową nazwę ogólną. W wyniku tej konstrukcji otrzymujemy niejednokrotnie wyrażenie niezręczne (np. zbyt długie), którym trudno się posługiwać. W takich wypadkach formułujemy komunikat zwany definicją (podrozdział 3.4.0), w którym owo niezręczne wyrażenie jest *definiensem*, zaś *definiendum* jest zwięzłe, nowe wyrażenie dotychczas do danego języka nie zaliczone.

Przykład ilustrujący powyższe uwagi nie będzie fikcyjny. Kilkadziesiąt lat temu wprowadzono w wielu językach naturalnych, między innymi w języku Polskim, nowy wyraz, mianowicie „*aeroplan*”. Przedstawmy schematycznie proces wprowadzenia tego wyrazu do języka polskiego . Przed wprowadzeniem wyrazu „*aeroplan*" używane już były, oczywiście, w języku polskim wyrażenia następujące:

(1) nazwa ogólna „*aparat*”,
(2) nazwa ogólna „*służący do latania*”,
(3) nazwa ogólna „*cięższy od powietrza*”,
(4) funktor nazwotwórczy od 3 argumentów nazwowych „*i zarazem … i zarazem*”.

Z czterech wyżej wymienionych wyrażeń można w języku polskim zbudować zgodnie z regułami gramatyki następujące wyrażenie złożone:

Aparat i zarazem służący do latania i zarazem cięższy od powietrza.

Powyższe wyrażenie (będące nazwą ogólną) jest jednak zbyt rozwlekłe do częstego użytku, mówi się wobec tego krócej:

Aparat służący do latania, cięższy od powietrza.

Stąd przyjęto definicję:

Aeroplan $=_{\mathrm{Df}}$ Aparat służący do latania, cięższy od powietrza.

Można też podać innego rodzaju przykłady. Nie jest wygodnie używać stale rozwlekłych nieco nazw niektórych urzędów, wobec czego w Polsce używamy skrótów opartych na definicjach następujących:

MSZ $=_{\mathrm{Df}}$ Ministerstwo Spraw Zagranicznych.
MON $=_{\mathrm{Df}}$ Ministerstwo Obrony Narodowej.

Piśmiennictwo: *Greniewski H.* G.2.1.

3.4.2. PRZYDATNOŚĆ DEFINICJI W WYJAŚNIANIU UŻYWANYCH JUŻ WYRAŻEŃ

W podrozdziale poprzednim mieliśmy do czynienia z *wprowadzeniem do języka nowych wyrażeń* za pomocą *komunikatu zwanego definicją*. Nie jest to jednak jedyna rola pożyteczna, jaką spełniają definicje. Niejednokrotnie przydatne są definicje, w których jako definienda występują wcale nie nowe wyrażenia, lecz przeciwnie, wyrażenia z dawna do danego języka zaliczone.

3.4.2.00. **Wyjaśnienie**. Mówimy, że dana definicja jest merytorycznie ścisła zamiast mówić, że spełnione są oba warunki następujące:

1) Pewne definienda (w rozumieniu szerszym) tej definicji są już używane w odnośnym języku przed zbudowaniem danej definicji;
2) Dana definicja nie precyzuje rozumienia jej dotychczas używanych definiendów, lecz po prostu rozumienie to wykłada.

Przykłady: Definicjami merytorycznie ścisłymi są wszystkie następujące:

Dziadek $=_{Df}$ (ojciec ojca) lub (ojciec matki).

Wuj $=_{Df}$ brat matki.

Stryj $=_{Df}$ brat ojca.

Często definicje merytorycznie ścisłe przydają się do wyjaśnienia komuś niezrozumiałego dlań wyrażenia. W takich wypadkach komunikujemy osobie - nierozumiejącej danego wyrażenia definicję spełniającą wszystkie trzy warunki następujące:

1) Definiendum tej definicji jest właśnie owym wyrażeniem dla danej osoby niezrozumiałej;
2) Definiendum tej definicji jest wyrażeniem dla danej osoby zrozumiałym;
3) Definicja jest merytorycznie ścisła.

Przykład. Tłumaczymy dziecku, które rozumie już wyraz „ojciec" i wyraz „matka" a nie rozumie wyrazu „dziadek", że dziadek, to ojciec ojca lub ojciec matki. W ten sam sposób ułatwiamy nieraz cudzoziemcowi (nawet nie znając jego języka) rozszerzenie zasobu poznanych przez niego wyrażeń języka polskiego, w ten sam wreszcie sposób postępuje nieraz nauczyciel wobec ucznia.

3.4.2.20. **Wyjaśnienie**. Mówimy, że dana definicja jest merytorycznie uściślająca, zamiast mówić, że spełnione są trzy warunki następujące:

1) Przed zbudowaniem danej definicji używano już w odnośnym języku wyrażeń równokształtnych z definiendum (w rozumieniu węższym) danej definicji;
2) Każde z wyrażeń równokształtnych, o których mowa w punkcie 1), ma rozumienie zbliżone do rozumienia, które dana definicja nadaje swemu definiendum (w rozumieniu węższym);
3) Każde z wyrażeń równokształtnych, o których mowa w punkcie 1), ma rozumienie dla praktyki nie dość jasne lub zbyt chwiejne.

W językach naturalnych występuje sporo wyrażeń o mało sprecyzowanym rozumieniu. Istnieją różne metody uściślania takich wyrażeń; jedną z tych metod jest zbudowanie definicji danego wyrażenia. Taka definicja powinna być merytorycznie uściślająca. Wobec niejasności wyrażenia precyzowanego może się zdarzyć, że zbudujemy na przykład dwie definicje, z których każda nada wyrażeniu precyzowanemu inne rozumienie; każde z tych rozumień będzie wprawdzie zgodne z niejasnym rozumieniem potocznym, lecz obydwa będą niezgodne między sobą. Nie trudno tu o przykład i to dobrze znany czytelnikowi ze szkoły średniej: znamy wyrażenie potoczne „*kąt*" (ewentualnie „*kąt płaski*"); jest ono dla osób - nieznających geometrii zrozumiałe, ale trudno powiedzieć, że w pełni sprecyzowane. Sprecyzujemy to wyrażenie za pomocą definicji na gruncie planimetrii i na gruncie trygonometrii płaskiej. Każda z tych definicji jest zgodna z

potocznym rozumieniem wyrazu „*kąt*”, a mimo to definicje te są niezgodne między sobą. Definicje planimetryczną i trygonometryczną można sformułować w sposób następujący:

(1) Kąt (*płaski*) $=_{Df}$ *Część płaszczyzny zawarta między dwiema półprostymi mającymi wspólny koniec.*

(2) *Kąt* (*płaski*) $=_{Df}$ *Obrót na płaszczyźnie półprostej dookoła jej końca, odniesiony do półprostej o wspólnym końcu z obracającą się półprostą.*

W myśl definicji (1) istnieją kąty największe, mianowicie kąty pełne, natomiast w myśl definicji (2) nie istnieje żaden kąt największy. A więc obie definicje precyzują rozumienie wyrazu „*kąt*” w sposób różny i, co więcej, niezgodny między sobą.

Piśmiennictwo: *Greniewski H.* G.2.1.

3.4.3. Przydatność definicji w związku z rugowaniem wyrażeń

Jeżeli zbudujemy (merytorycznie ścisłą) definicję wyrażenia już uprzednio (tj. przed zdefiniowaniem) należącego do danego języka, definicja ta poucza nas, że wszelkie jej definienda (w szerszym rozumieniu) są w naszym języku teoretycznie zbędne, to znaczy mogą być wyrugowane z języka (w drodze zastąpienia przez definiens) bez zubożenia zasobu myśli dających się w tym języku wypowiedzieć. Takie stwierdzenie teoretycznej rugowalności danego wyrażenia z danego języka pogłębia naszą wiedzę o danym języku. Będziemy to mogli w pełni ocenić, gdy przystąpimy do systematycznego wykładu podstaw logiki formalnej (części 2, 3 i 4), w którym spotkamy się z tak zwanymi „prawami rugowania”. Każde z tych praw, będzie konkretnym przykładem stwierdzenia teoretycznej rugowalności pewnego wyrażenia.

Zajmowaliśmy się dotychczas zarówno sprawą definiowania wyrażeń nowych, jak i sprawą definiowania wyrażeń już w języku używanych. W związku z obu tymi sprawami nasuwa się jeszcze następująca uwaga:

Na pierwszy rzut oka wydawałoby się, że cóż łatwiejszego, jak budowanie definicji wyrażeń już uprzednio do języka wprowadzonych. W rzeczywistości sprawa przedstawia się inaczej: znacznie łatwiej jest prawidłowo zbudować definicję świeżo do języka wprowadzonego wyrażenia niż definicję wyrażenia już do danego języka należącego. Wymaga to nieco obszerniejszego objaśnienia:

1) Jeżeli za pomocą definicji chcemy wprowadzić do danego języka nowe wyrażenie, mamy wówczas dużą swobodę działania; definiendum - jest bowiem w danym języku „*nowicjuszem*” nie odgrywającym dotychczas żadnej roli. Dobrze tylko, abyśmy dbali o to, żeby nasz „*nowicjusz*” był przydatny w praktyce, a więc łatwy do zapamiętania, niezbyt długi, niedający się pomylić z wyrażeniami dotychczas do danego języka zaliczonymi.
2) Jeżeli natomiast chcemy zbudować definicję wyrażenia już do danego języka należącego, to znaczy, jeżeli chcemy zbudować zgodny z rzeczywistością komunikat (a więc zdanie prawdziwe) stwierdzający, że w danym języku takie a takie wyrażenie jest używane równoznacznie z innym- musimy wówczas „uszanować” dotychczasowe rozumienie naszego definiendum, a to w praktyce bynajmniej nie jest łatwe.

Piśmiennictwo: *Greniewski H.* G.2.1.

3.4.4. ZALEŻNOŚCI MIĘDZY DEFINICJAMI

3.4.4.00. **Wyjaśnienie**. Mówimy, że definicja ***B*** jest bezpośrednio zależna od definicji ***A***, zamiast mówić, że przynajmniej jedno z definiensów (w szerszym rozumieniu) definicji ***A*** jest zawarte w definiensie (w węższym rozumieniu) definicji ***B***.

Przykład (schematyczny). Definicja (2) bezpośrednio zależna - od definicji (1):

(1) $Y =_{Df} F_1 X,$

(2) $Z =_{Df} F_2 Y.$

3.4.4.01. **Wyjaśnienie**. Mówimy, ze definicja ***C*** usuwa bezpośrednią zależność definicji ***B*** od definicji ***A***, zamiast mówić, że:

1. definicja ***B*** jest bezpośrednio zależna od definicji ***A***
   oraz
2. definicja ***C*** powstaje z definicji ***B*** przez zastąpienie
   - w definiensie definicji ***B***
   - wszystkich części będących definiendami (w szerszym rozumieniu) definicji ***A***
   - odnośnymi definiensami (w szerszym rozumieniu) definicji ***A***.

Wyjaśnienie powyższe jest, być może, nieco trudne da zrozumienia. Należy przypuszczać, że poniższy przykład schematyczny ułatwi nieco zrozumienie wyjaśnienia 3.4.4.01.

Przykład (schematyczny). Definicja (3) usuwająca bezpośrednią zależność definicji (2) od definicji (1):

(1) $Y =_{Df} F_1 X.$

(2) $Z =_{Df} F_2 Y.$

(3) $Z =_{Df} F_2 F_1 X.$

3.4.4.10. **Wyjaśnienie**. Mówimy, że w danym zespole definicji definicja ***B*** jest zależna od definicji ***A***, zamiast mówić, ze wszystkie czy niektóre z definicji tego zespołu można ustawić kolejno w sposób spełniający warunki następujące:

1) Pierwsza w tej kolejności jest definicja ***A***;
2) Ostatnia w tej kolejności jest definicja ***B***;
3) Każda z definicji ustawionych w tej kolejności (z wyjątkiem pierwszej definicji) jest bezpośrednio zależna od definicji bezpośrednio ją poprzedzającej.

Przykład (schematyczny). Weźmy pod uwagę zespół definicji (1) - (4):

(1) $X_2 =_{Df} F_1 X_1.$

(2) $X_3 =_{Df} F_2 X_2.$

(3) $X_4 =_{Df} F_3 X_3.$

(4) $X_5 =_{Df} F_4 X_4.$

Łatwo zauważyć, że w zespole definicji (1) - (4) definicja (4) jest zależna od definicji (1).

3.4.4.11. **Wyjaśnienie**. Mówimy, że w danym zespole definicji definicja $\boldsymbol{C_n}$ usuwa zależność definicji $\boldsymbol{A_n}$ od definicji $\boldsymbol{A_0}$, zamiast mówić, że

1) Definicja $\boldsymbol{A_n}$ jest zależna od definicji $\boldsymbol{A_0}$
   oraz
2) Ustawiamy definicje danego zespołu według kolejności opisanej w wyjaśnieniu 3.4.4.10:

$$\boldsymbol{A_0, A_1, A_2, \ldots, A_{n-1}, A_n,}$$

następnie - budujemy definicję $\boldsymbol{C_1}$ usuwającą bezpośrednią zależność definicji $\boldsymbol{A_1}$ od definicji $\boldsymbol{A_0}$, następnie budujemy definicję $\boldsymbol{C_2}$ usuwającą bezpośrednią zależność definicji $\boldsymbol{A_2}$ od definicji $\boldsymbol{C_1}$

itd., aż wreszcie budujemy definicję $\boldsymbol{C_n}$ usuwając bezpośrednią zależność definicji $\boldsymbol{A_n}$ od definicji $\boldsymbol{C_{n-1}}$.

Przykład (schematyczny). Dany jest zespół definicji:

(A_0) $X_1 =_{Df} F_0 X_0,$

(A_1) $X_2 =_{Df} F_1 X_1,$

(A_2) $X_3 =_{Df} F_2 X_2,$

(A_3) $X_4 =_{Df} F_3 X_3.$

Budujemy definicję:

(C_1) $X_2 =_{Df} F_1 F_0 X_0.$

Definicja ta, jak łatwo zauważyć, usuwa bezpośrednią zależność definicji (A_1) od definicji (A_0). Następnie budujemy definicję:

(C_2) $X_3 =_{Df} F_2 F_1 F_0 X_0.$

Definicja (C_2) usuwa, oczywiście, bezpośrednią zależność definicji (A_2) od definicji (C_1). Wreszcie budujemy definicję:

(C_3) $X_4 =_{Df} F_3 F_2 F_1 F_0 X_0,$

która usuwa bezpośrednią zależność definicji (A_3) od definicji (C_2). Otóż w myśl wyjaśnienia 3.4.4.11 w zespole definicji (A_0) - (A_3) definicja (C_2) usuwa zależność definicji (A_3) od (A_0).

Piśmiennictwo: *Greniewski H.* G.2.1.

3.4.5. Błędne koło w definiowaniu

3.4.5.00. **Wyjaśnienie**. Mówimy, że dany zespół definicji zawiera bezpośrednie *błędne koło*, zamiast mówić, że jedno z jej *definiendów* (w szerszym rozumieniu) jest zawarte w jej *definiensie* (w węższym rozumieniu).

Przykłady. Definicja zawierająca bezpośrednie błędne koło (jest to przykład schematyczny):

$$X =_{Df} FX.$$

Definicja zawierająca bezpośrednie błędne koło:

$$\textit{Człowiek} =_{\mathrm{Df}} \textit{człowiek}$$

Definicja zawierająca bezpośrednie błędne koło (jest to przykład skonstruowany w języku algebry szkolnej):

(1) $$(a < b) =_{Df} (a - b < 0).$$

Przykład definicji (1) wymaga, jak się zdaje, paru słów wyjaśnienia. W myśl wyjaśnienia 3.4.0.31, przez *definiendum* (w szerszym rozumieniu) definicji (1) rozumiemy również każde prawidłowe podstawienie do *definiendum* (w węższym rozumieniu). W szczególności jednym z *definiendów* (w szerszym rozumieniu) *definicji* (1) jest podstawienie otrzymane w ten sposób, że do *definiendum* (w węższym rozumieniu) podstawiamy

funkcję nazwową dwu zmiennych	$a - b$
za zmienną nazwową	a

oraz

Nazwę	0
za zmienną nazwową	b

jako wynik otrzymujemy wyrażenie:

$$a - b < 0.$$

Każde wyrażenie równokształtne i równo znaczne z powyższym wynikiem podstawienia jest jednym z *definiendów* (w szerszym rozumieniu) definicji (1), w szczególności jednym z takich *definiendów* jest *definiens* (w węższym rozumieniu) definicji (1), a więc jedno z *definiendów* (w

szerszym rozumieniu) definicji (1) jest zawarte w jej *definiensie*, zatem (w myśl wyjaśnienia 3.4.5.00) - *definicja* (1) zawiera *bezpośrednie błędne koło*.

3.4.5.10. **Wyjaśnienie**. Mówimy, że dany zespół definicji zawiera pośrednie błędne koło, zamiast mówić, że można zbudować taką definicję, która:

1) Usuwa zależność jednej z definicji tego zespołu od innej definicji tegoż zespołu
   oraz
2) Zawiera bezpośrednie błędne koło.

Przykłady. Weźmy pod uwagę poniższy zespół złożony z dwu definicji:

(1) *Samolot* $=_{Df}$ *aeroplan*.

(2) *Aeroplan* $=_{Df}$ *samolot*.

Definicja ta zawiera bezpośrednie błędne koło.

Przechodzimy do przykładu schematycznego. Dany jest zespół definicji:

(1) $Y =_{Df} F_1 X.$

(2) $Z =_{Df} F_2 Y.$

(3) $X =_{Df} F_3 Z.$

Łatwo zauważyć, że definicja (3) jest bezpośrednio zależna od definicji (2), zaś definicja (2) jest bezpośrednio zależna od definicji (1). Wobec tego (wyjaśnienie 3.4.4.10), w zespole (1) - (3) definicja (3) jest zależna od definicji (1). Zbudujemy teraz definicję (4), która usuwa bezpośrednią zależność definicji (3) od definicji (2):

(4) $Y =_{Df} F_3 F_2 Y.$

Łatwo zauważyć, że definicja (4) jest bezpośrednio zależna od definicji (1). Budujemy definicję (5) usuwającą bezpośrednią zależność definicji (4) od definicji (1):

(5) $X =_{Df} F_3 F_2 F_1 X.$

W myśl wyjaśnienia 1.4.4.11 definicja (5) usuwa w zespole (1) - (3) zależność definicji (3) od definicji (1). Zauważmy jeszcze, ze definicja (5) zawiera bezpośrednie błędne koło (wyjaśnienie 1.4.5.00), a więc zespół (1) - (3) zawiera (wyjaśnienie 1.4.5.10),pośrednie - błędne koło.

Piśmiennictwo: *Greniewski H.* G.2.1.

3.4.6. ZASIĘG DEFINICJI

Możemy odróżnić dwa rodzaje definicji:

1) Definicje, które obowiązują (od chwili ich wprowadzenia) w całym języku[42];
2) Definicje, które obowiązują tylko w obrębie pewnego zespołu tez.

Definicje scharakteryzowane w punkcie 2) można znaleźć na przykład w Dzienniku Ustaw. Zdarza się nieraz, że ogłoszono w Dzienniku Ustaw ustawę, a następnie rozporządzenie wykonawcze do tej ustawy. We wstępnym artykule rozporządzenia wykonawczego możemy znaleźć nieraz definicję następującej postaci: przez „ustawę" rozumie się w rozporządzeniu niniejszym ustawę z dnia... Dz. U. RP ... poz. ... Definicja taka nadaje wyrazowi „ustawa" nowe, specjalne rozumienie, ale definicja ta nie obowiązuje w całym języku polskim; działa ona tylko w obrębie danego rozporządzenia wykonawczego. W związku z powyższym przyjmujemy następujące wyjaśnienie:

[42] Podobne zasady obowiązują w językach programowania, używamy terminu „deklaracja" w miejsce terminu „definicja". Deklarując zmienne programu, rozróżniamy „zmienne globalne" – obowiązujące w ramach całego programu i „zmienne lokalne" – obowiązujące w ramach danego bloku kodu programu lub podprogramu.

1.4.6.00. **Wyjaśnienie**. Mówimy, że dana definicja jest definicją o zasięgu Z, zamiast mówić, że definicja ta obowiązuje nie w całym języku, lecz tylko w zespole tez Z.

Piśmiennictwo: *Greniewski H.* G.2.1.

3.4.7. Reguły poprawności definiowania

Mieliśmy możność przekonać się, że definiowanie bywa pożytecznym środkiem rozbudowy języka i analizy rozumienia, ewentualnie niezbędności niektórych należących do języka wyrażeń. Należy teraz przestrzec czytelnika, że środek ten nie jest wcale tak niewinny, na jaki może wyglądać. Nie każda definicja jest poprawna, a za pomocą niepoprawnej definicji łatwo dojść do fałszywych wniosków. Co gorzej, w danym rozważaniu może się zdarzyć, że każda z występujących w nim definicji jest poprawna, ale zespół wszystkich tych definicji jest niepoprawny, co może pociągnąć za sobą konsekwencje wcale nie lepsze niż niepoprawność poszczególnej definicji. W związku z powyższymi ostrzeżeniami przyjmujemy następujące wyjaśnienia:

3.4.7.00. **Wyjaśnienie**. Mówimy, że dana funkcja jest stała ze względu na daną zmienną wolną, zamiast mówić, że każde dwa dowolne, byle prawidłowe, podstawienia za tę zmienną dają w wyniku wyrażenia równoznaczne.

3.4.7.10. **Wyjaśnienie**. Mówimy, że dana definicja jest poprawna, zamiast mówić, że spełnione są oba warunki następujące:

1) Definicja ta nie zawiera bezpośredniego błędnego koła;
2) Nie istnieje takie poprawne rozumowanie, które:
   a)doprowadza do fałszywego wyniku. Jeżeli użyć w nim danej definicji
   oraz
   b) nie doprowadza do fałszywego wyniku, jeżeli nie użyć w nim danej definicji.

Praktyka wskazuje, że chcąc budować jedynie poprawne definicje, należy stale przestrzegać przynajmniej poniższych reguł:

3.4.7.11. **Reguła definiowania**. Jeżeli w definiensie (w węższym rozumieniu) definicji poprawnej zawarta jest jakaś zmienna wolna, to zachodzi - chociaż jeden z trzech następujących warunków:

1) W definiendum (w węższym rozumieniu) zawarta jest zmienna wolna równo kształtna z tą zmienną zawartą w definiensie;
2) Definiens (w węższym rozumieniu) jest funkcją stałą ze względu na tę zawartą w nim zmienną wolną;
3) Definicja ma zasięg i obowiązuje dyrektywa, która zabrania dokonywać jakichkolwiek podstawień za tę zawartą w definiensie (w rozumieniu węższym) zmienną.

Reguła 3.4.7.11, niema zastosowania do języków naturalnych (w żadnym takim języku nie mamy w ogóle zmiennych), natomiast w wielu językach sztucznych i mieszanych ma ona duże znaczenie praktyczne.

3.4.7.12. **Reguła definiowania**. Jeżeli dana definicja poprawna ma zasięg, to o każdej tezie można orzec, czy należy ona do zasięgu, czy tez nie należy.

3.4.7.13. **Reguła definiowania**. Jeżeli dana definicja (1) jest poprawna oraz (2) jej definiendum (w węższym rozumieniu) nie jest operatorem ani nie zawiera w sobie operatora jako części

właściwej, to owo definiendum nie zawiera w sobie żadnego funktora uprzednio do odnośnego języka wprowadzonego ani żadnej funkcji uprzednio do tego języka wprowadzonej.

3.4.7.20. **Wyjaśnienie**. Mówimy, że dany zespół definicji jest poprawny (w rozumieniu szerszym), zamiast mówić, że nie istnieje takie poprawne rozumowanie, które:
1) Doprowadza do fałszywego wyniku, jeżeli użyć w nim, choć jednej definicji należącej do zespołu;
2) Nie doprowadza do fałszywego wyniku, jeżeli nie użyć w nim żadnej definicji należącej do tego zespołu.

W oparciu o wyjaśnienia 3.4.7.10 i 3.4.7.20, otrzymujemy natychmiast poniższą regułę:

3.4.7.21. **Reguła**. Jeżeli w danym zespole definicji - chociaż jedna definicja nie jest poprawna, pomimo że nie zawiera błędnego koła, to zespół ten nie jest poprawny (w rozumieniu szerszym).

3.4.7.30. **Wyjaśnienie**. Mówimy, ze dany zespół definicji jest poprawny (w rozumieniu węższym), zamiast mówić, że spełnione są oba warunki poniższe:
1) Zespół ten jest poprawny (w rozumieniu szerszym);
2) Zespół ten nie zawiera pośredniego błędnego koła.

W pracy naukowej i w systematycznym wykładzie jakiejkolwiek nauki zawsze wymagamy, aby wchodzący w grę najszerszy zespół definicji był poprawny w rozumieniu węższym, natomiast przy redagowaniu słownika (jednojęzycznego) wystarczy zadośćuczynić wymaganiu, aby całość słownika była zespołem definicji poprawnym w rozumieniu szerszym. Praktyka skłania nas do przyjęcia poniższej reguły definiowania:

3.4.7.31. **Reguła definiowania**. Jeżeli dany zespół definicji jest poprawny (w rozumieniu szerszym), to spełniony jest jeden z warunków następujących:
1) Żadne dwa *definienda* (w szerszym rozumieniu) definicji należących do tego zespołu nie są równokształtne;
2) Jeżeli jakieś *definienda* (w szerszym rozumieniu) dwu definicji należących do tego zespołu są równokształtne, to każda z obu definicji ma zasięg i nie istnieje teza, która by należała do obu tych zasięgów.

Piśmiennictwo: *Greniewski H.* G.2.1.

3.4.8. Nie uniwersalność definiowania

Wyjaśniliśmy już, na czym polega definiowanie, do czego ono służy i jakie są warunki poprawności definicji oraz poprawności zespołu definicji. Stwierdziliśmy w szczególności, że budujemy definicje chcąc osiągnąć jeden z czterech poniższych celów:
1) Wprowadzenie do języka nowego wyrażenia;
2) Uściślenie czy ustalenie strony znaczeniowej (niejasnej czy chwiejnej) wyrażenia już do danego języka należącego;
3) Wyjaśnienie komuś wyrażenia już do języka należącego, lecz dla danej osoby niezrozumiałego;
4) Zbadanie rugowalności (można by też powiedzieć „usuwalności") danego wyrażenia z danego języka.

Pozostaje jeszcze do wyjaśnienia, w jakich wypadkach definiowanie jest:

1) Niewykonalne
   lub
2) Nieopłacalne.

Jeśli chodzi o niewykonalność definiowania (w pewnym skrajnym zresztą wypadku), to podstawową informację - daje nam następująca zasada:

3.4.8.00. **Zasada**. Dla żadnego języka nie można zbudować takiego zespołu definicji, który by:
1) Był poprawny (w węższym rozumieniu)
   oraz
2) Definiował wszystkie wyrażenia danego języka.

Zasada ta staje się oczywista, gdy tylko uprzytomnimy sobie, że po to, by zdefiniować wszystkie wyrażenia danego języka, trzeba popełnić pośrednie błędne koło (a wtedy nasz zespół definicji nie będzie poprawny w rozumieniu węższym), albo też trzeba się cofać w nieskończoność, poszukując do danych *definiendów*, coraz to nowych *definiensów*, czyli popełnić tak zwany *regressusad infinitum*, co jest niewykonalne.

Niejednokrotnie poprawne zdefiniowanie pewnego wyrażenia jest praktycznie niewykonalne, w innych znów razach potrafimy wprawdzie zbudować definicję poprawną danego wyrażenia (a nawet, jeśli potrzeba -- definicję merytorycznie ścisłą czy uściślającą), ale definicja ta okazuje się tak skomplikowana czy niezręczna w użyciu, że przez to właściwie staje się nieopłacalna.

Jeżeli mamy zrealizować cel przedstawiony w punkcie 1), 2), ewentualnie 3), ale nie realizujemy celu przedstawionego w punkcie 4), to często - jako metoda konkurencyjna (w stosunku do metody wprowadzania, uściślania czy wyjaśniania wyrażenia za pomocą definicji) występuje metoda wprowadzania, uściślania czy wyjaśniania wyrażeń przez postulaty (metoda postulatowa).

3.4.8.10. **Wyjaśnienie**. Metoda postulatowa polega na podaniu *tez prawdziwych* (tzw. postulatów) zawierających obok wyrażeń o rozumieniu już niewątpliwym - wyrażenia, które wszystkie:

albo
1) do języka wprowadzamy,
   albo
2) uściślamy,
   albo
3) komuś wyjaśniamy.

Przykład. Jest dość trudno (gdyż wymaga to znacznej znajomości logiki formalnej) zdefiniować w języku arytmetyki liczb naturalnych którekolwiek z czterech wyrażeń:
1) Nazwę ogólną „$liczba\ naturalna$”;
2) Nazwę jednostkową „0” (zero);
3) Nazwę jednostkową „1” (jeden);
4) Funkcję jednostkowo nazwową dwu zmiennych jednostkowo nazwowych „$m + n$”.

Stosunkowo łatwo natomiast sformułować i przyjąć następujące postulaty:

3.4.8.11 0 - *jest liczbą naturalną*.

3.4.8.12 1 - *jest liczbą naturalną*.

3.4.8.13. Jeżeli [(m - *jest liczbą naturalną*) oraz (n - *jest liczbą naturalną*)], to ($m + n$) *jest też liczbą naturalną*.

3.4.8.14. Jeżeli (n - *jest liczbą naturalną*), to ($n + 0 = n$).

3.4.8.15. Jeżeli [(m - *jest liczbą naturalną*) oraz (n - *jest liczbą naturalną*)], to

$$[m + (n + 1) = (m + n) + 1].$$

Po wprowadzeniu przez postulaty do języka, wprowadzone wyrażenia pozwalają już bez trudności definiować niektóre dalsze wyrażenia należące do języka arytmetyki liczb naturalnych, na przykład:

$$2 =_{Df} 1 + 1$$
$$3 =_{Df} 2 + 1$$
$$4 =_{Df} 3 + 1 \quad \text{itd.}$$

Metoda postulatowa ma szeroki zakres zastosowania. Stosujemy ją poczynając od najbardziej abstrakcyjnych teorii matematycznych aż po wyjaśnianie cudzoziemcowi, gdy nie znamy jego języka, czy małemu dziecku najprostszych choćby wyrażeń. W ostatnim wypadku nasze postulaty zawierają często nazwy okazjonalne. Jeżeli małemu dziecku, które pragnie się dowiedzieć, co oznacza nazwa „drabina", chcemy wyraz ten wyjaśnić, to pokazujemy mu koleino kilka drabin, mówiąc za każdym razem - „to jest drabina".

Piśmiennictwo: *Greniewski H.* G.2.1.

3.5. TEORIA JĘZYKÓW I GRAMATYK SFORMALIZOWANYCH

3.5.1. WPROWADZENIE DO GRAMATYK JĘZYKÓW SFORMALIZOWANYCH

W podrozdziale 3.0.3 wprowadziliśmy pojęcie języka sformalizowanego i jego gramatyki. Obecnie, zajmiemy się bliżej tą tematyką. Język sformalizowany, to jedno z najważniejszych pojęć w informatyce teoretycznej, logice matematycznej oraz metodologii nauk dedukcyjnych. Podstawami każdego języka (jak wiemy z wcześniejszych rozważań), w tym również języka sformalizowanego, są: *alfabet*, *słownik* i *gramatyka*. Alfabet to zbiór znaków (symboli) dopuszczalnych w danym języku. Elementy słownika są zazwyczaj nazywane słowami, z kolei słowa są zbudowane ze znaków alfabetu, czyli symboli. Słowa tworzą zdania zgodnie z regułami określonymi przez gramatykę. Tym samym, gramatyka sztucznego języka pozwala nam uzyskać odpowiedz na pytanie: czy dany ciąg słów języka jest zdaniem, czyli tworem poprawnym gramatycznie, czy też nie.

Mówiąc poprzednio interesowaliśmy się wyrazami zbudowanymi z symboli *alfabetu*. Rozważać będziemy interesujący nas problem zwany poprawnością wyrazów. Czym jest język, zarówno naturalny, jak i język programowania? Jest to podzbiór zbioru wszystkich możliwych wyrazów, jakie można zbudować z liter przyjętego alfabetu. O poprawności wyrazów w danym języku decyduje *gramatyka języka*. Próby zbudowania matematycznych podstaw gramatyki języka naturalnego (w tym przypadku angielskiego) — podejmował *Noam Chomsky*, bez powodzenia ze względu na zbyt dużą komplikację i niekonsekwentną składnię języka naturalnego. W wyniku, *Chomsky* zbudował jednak podstawy lingwistyki matematycznej.

3.5.1.10. **Wyjaśnienie**. *Czym jest gramatyka?* Formalną definicję podamy dalej. Na razie można przyjąć:

1. Gramatyka to generator pozwalający produkować napisy, które są zgodnie z regułami języka i są poprawnym zdaniami. Te, których się nie da wygenerować są odrzucane. Proces generowania i odrzucania odbywa się automatycznie.
2. Gramatykę wykorzystujemy do konstrukcji translatora języków programowania, który tłumaczy program napisany przez człowieka na język komputera używając matematycznych definicji języków programowania.

Pierwszy krok przy tworzeniu gramatyki, to tworzenie wykazu symboli do budowy zdań języka czyli alfabetu terminalnego. Wygenerowane napisy zawierające choć jeden symbol spoza alfabetu terminalnego są odrzucane.

3.5.1.20. **Wyjaśnienie**. *Symbolom terminalnym*, z których buduje się zdania odpowiadają wyrazy języka naturalnego (nie zaś poszczególne litery). *Nie każdy napis zbudowany z poprawnych wyrazów jest poprawnym zdaniem — jak w języku naturalnym*. Gramatyka *zawiera reguły* pozwalające na poprawną strukturę zdań. Do budowy tych reguł używa się zbioru pomocniczych symboli czyli *alfabetu nieterminalnego*.

3.5.1.30. **Wyjaśnienie**. Symbole nieterminalne nie wchodzą w skład języka. Należą one do *metajęzyka*. W matematyce da się oddzielić *język reguł gramatyki* od języka podlegającego opisowi. Tego nie da się zrobić w języku naturalnym, np. reguły gramatyki języka polskiego podaje się w języku polskim.

3.5.1.40. **Wyjaśnienie**. Gramatykę (czyli tzw. gramatykę generacyjną), języka sztucznego lub naturalnego, można określić, jako czwórkę uporządkowaną:

$$G =_{\mathrm{Df}} (V, T, P, S);$$

gdzie: V — zbiór skończony słów terminalnych, czyli kończących zdanie, danego języka; T — zbiór niepusty słów nie-terminalnych; P — lista dopuszczalnych produkcji, czyli skończony zbiór reguł gramatycznych danego języka; S — symbol początkowy oznaczający klasę obiektów językowych, dla opisu, których przeznaczona jest gramatyka. Noam Chomsky w pracy: *Struktura logiczna teorii lingwistycznej* (1975) wyróżnia cztery typy gramatyk. Gramatyki te wyodrębnia się przez nakładanie coraz silniejszych ograniczeń na układ reguł gramatycznych P:

- Gramatyka klasy 0 – o ogólnej postaci reguł gramatycznych, bez konieczności używania symboli początkowych S;
- Gramatyka klasy 1 – zwana gramatyką kontekstową, o narzuconych ograniczeniach wymagających używania symboli początkowych S;
- Gramatyka klasy 2 - zwana gramatyką bez-kontekstową, która w układzie reguł dopuszcza stosowanie jedynie produkcji - typu $A \to b$, gdzie $A \in S, b \in V$;
- Gramatyka klasy 3 – zwana gramatyką regularną, która w układzie reguł dopuszcza stosowanie jedynie reguł postaci $A \to bB$ (gramatyki prawostronnie regularne) albo $A \to Bb$ (gramatyki lewostronnie regularne), gdzie $A \in S, B \in S, b \in T$.

Nas interesują języki sztuczne – dokładniej mówiąc języki sformalizowane, których rozwój jest związany z informatyką. A są to odpowiednio:

1. Języki o gramatyce bez-kontekstowej – krótko nazywane językami bez-kontekstowymi;
2. Języki o gramatyce regularnej – krótko zwane językami regularnymi.

3.5.1.50. **Wyjaśnienie**. Języki bez-kontekstowe są wykorzystywane do definiowania języków programowania, w tym do formalizacji pojęcia analizy syntaktycznej i do upraszczania translacji

języków programów napisanych w językach programowania do postaci kodów rozkazów komputerowych.

3.5.1.60. **Wyjaśnienie**. Język regularny (*regular language*) to język formalny taki, że istnieje automat o skończonej liczbie stanów potrafiący zdecydować (patrz podrozdziały 3.7.3 i 3.7.4), czy dane słowo należy do języka. Wszystkie języki regularne są jednocześnie językami bez-kontekstowymi. Każdy język regularny można zapisać w postaci gramatyki formalnej – takiej gramatyki, że po lewej stronie każdej reguły jest jeden symbol nie-terminalny, po prawej zaś dowolna liczba symboli terminalnych, po których występuje, co najwyżej jeden symbol nie-terminalny.

3.5.1.70. **Wyjaśnienie**. Jak zatem będziemy opisywać języki programowania lub ich elementy? Składnię języków będziemy opisywali za pomocą powszechnie znanej notacji BNF, odpowiadającej gramatykom bez-kontekstowym. Semantykę rozmaitych konstrukcji będziemy na ogół opisywali w sposób „naiwny", czyli zwyczajnie, po polsku. Notacja BNF (*Backus Naur Form*) jest powszechnie używana właśnie do opisu składni języków programowania[43]. Stworzona została w trakcie prac nad językami Fortran i Algol na przełomie lat pięćdziesiątych i sześćdziesiątych XX wieku

Przykładowa definicja języka w notacji BNF, jest realizowana z pomocą – zbioru czternastu reguł:

1. Poszczególne reguły mają postać <symbol> ::= <definicja symbolu>.
2. Sens takiej reguły jest następujący: symbol występujący po lewej stronie znaku ::= można zastąpić tym, co pojawia się po prawej stronie.
3. Innymi słowy, stwierdzamy, że to, co stoi po lewej stronie, może wyglądać jak to, co stoi po prawej.
4. Symbole pojawiające się po lewej stronie reguł zwane są symbolami nie-terminalnymi.
5. Symbole pojawiające się wyłącznie po prawej stronie to symbole terminalne.
6. Generalnie symbole terminalne to symbole z alfabetu definiowanego języka, a zatem „docelowe"; symbole nie-terminalne spełniają natomiast rolę pomocniczą przy jego definiowaniu.
7. Pionowa kreska `|` oznacza alternatywne warianty reguły, np. `typ ::= char | int | float | double`
8. Nawiasy kwadratowe `[...]` oznaczają opcjonalną część reguły, np. `instr_warunkowa ::= if wyr_logiczne then instr [ else instr ]`
9. Nawiasy klamrowe `{...}` oznaczają fragment, który może być powtórzony dowolnie wiele razy (być może, że dokładnie zero krotnie, czyli jest pominięty), np. `lista_arg ::= arg { "," arg }`
10. Zwykłych nawiasów okrągłych `(...)` używa się do grupowania alternatywnych fragmentów definicji, np.
    `liczba_ze_znakiem ::= ("+" | "-") liczba_bez_znaku`
11. Jednoznakowe symbole terminalne umieszcza się w cudzysłowie, dla odróżnienia ich od symboli samej notacji BNF.
12. Symbole terminalne pisze się czcionką wytłuszczoną; nie jest wówczas konieczne pisanie nawiasów kątowych wokół symboli nie-terminalnych.

[43] Ze względu na ograniczenia notacji BNF przy opisie struktur rekurencyjnych, wprowadzony został EBNT (Extended BNF).

13. Chcąc opisać powtarzające się elementy, możemy stworzyć definicję rekurencyjną lub wykorzystać nawiasy klamrowe.
14. Zdefiniujmy dla przykładu niepustą listę identyfikatorów, rozdzielonych przecinkami:
    `lista_identyfikatorów ::= identyfikator | lista_identyfikatorów "," identyfikator`
    Alternatywnie piszemy:
    `lista_identyfikatorów ::= identyfikator { "," identyfikator }`

Klasyczne zastosowanie notacji BNF to opis składni notacji BNF:

```
Składnia::= { reguła }
reguła ::= identyfikator "::=" wyrażenie
wyrażenie::= składnik { "|" składnik }
składnik ::= czynnik { czynnik }
czynnik ::=
identyfikator|symbol_zacytowany| "(" wyrażenie ")"| "[" wyrażenie "]"|
"{" wyrażenie "}"
identyfikator ::= litera { litera | cyfra }
symbol_zacytowany ::= """ { dowolny_znak } """
```

Zauważmy, że niektóre symbole, formalnie terminalne, są właściwie niedodefiniowanymi symbolami nie-terminalnymi. Gwoli ścisłości powinniśmy, zatem dopisać:

```
litera ::= "A" | "B" | "C" | ...
cyfra ::= "0" | "1" | ... | "9"
dowolny_znak ::= ...
```

Jak widać, użyliśmy kolejnego niedodefiniowanego symbolu, a mianowicie wielo-kropka. BNF to, zatem w istocie sposób zapisywania produkcji gramatyk bez-kontekstowych.

3.5.1.80. **Wyjaśnienie**. *Wyrażenia regularne* (*regular expressions*, w skrócie *regex* lub *regexp*) – *wzorce*, które opisują *łańcuchy symboli*. Teoria wyrażeń regularnych jest związana z teorią języków regularnych. Wyrażenia regularne mogą określać zbiór pasujących łańcuchów, mogą również wyszczególniać istotne części łańcucha.

Wyrażenia regularne to w informatyce teoretycznej ciągi znaków pozwalające opisywać języki regularne. W praktyce znalazły bardzo szerokie zastosowanie, pozwalają - bowiem w łatwy sposób opisywać wzorce tekstu, natomiast istniejące *algorytmy* w efektywny sposób określają, czy podany ciąg znaków pasuje do wzorca lub wyszukują w tekście wystąpienia wzorca. Wyrażenia regularne w praktycznych zastosowaniach są zapisywane za pomocą bogatszej i łatwiejszej w użyciu składni niż ta stosowana w rozważaniach teoretycznych. Co więcej, opisane niżej powszechnie wykorzystywane wsteczne referencje, czyli użycie wcześniej dopasowanego fragmentu tekstu, jako części wzorca, powodują, że wyrażenie regularne je zawierające - może nie definiować języka regularnego.

Wyrażenia regularne stanowią integralną część narzędzi systemowych takich jak *sed*, *grep*, wielu edytorów tekstu, języków programowania przetwarzających tekst *AWK* i *Perl*, a także są dostępne, jako odrębne biblioteki dla wszystkich języków programowania obecnie używanych.

Dwie najpopularniejsze składnie wyrażeń regularnych to składnia typu *Unix* i składnia typu *Perl*. Składnia typu *Perl* jest znacznie bardziej rozbudowana. Jest ona używana nie tylko w języku *Perl*, ale także w innych językach programowania: *Ruby*, bibliotece *PCRE* do *C* i w narzędziu powłoki o

nazwie *pcregrep* (znanego też, jako *pgrep*). Znajomość wyrażeń regularnych jest niezbędna, do walidacji danych z formularzy wykorzystywanych na stronach *www*. Bez wyrażeń regularnych trudno jest sprawdzić przykładowo, poprawność adresów e-mailowych czy też adresów *www*.

Piśmiennictwo: *Cormen T.* C.4.1., *Hyde R.* H.5.1., *Ross K.* R.1.1., *Wirth N.* W.5.1.

3.5.2. AKSJOMATY I POPRAWNOŚĆ WYRAŻEŃ JĘZYKA SFORMALIZCWANEGO

3.5.2.10. **Wyjaśnienie**. *Aksjomat*: jeden z ważniejszych symboli metajęzykowych w matematycznym opisie gramatyki dotyczących poprawności. Dla języków programowania aksjomat — to "*prawidłowo zbudowana instrukcja*".

3.5.2.20. **Wyjaśnienie**. *Produkcja*: reguły przekształcania symboli metajęzykowych w napisy zawierające inne symbole metajęzyka lub wyrazy należące do języka. Produkcja nie prowadzi nigdy poza język.

3.5.2.30. **Wyjaśnienie**. Stosowane są *dwie metody badania poprawności zdania* w danym języku, czyli sprawdzanie tego, że dane zdanie należy do języka:

1. Metoda generacyjna — wychodzimy od aksjomatu i stosując ciąg produkcji generujemy pożądane zdanie;
2. Metoda redukcyjna — cofając się w ciągu produkcji można "zwinąć" zdanie do aksjomatu.

Jeśli obie powyższe metody zawodzą — to dane zdanie jest niepoprawne.

3.5.2.60. **Wyjaśnienie**. Każde poprawnie algebraicznie wyrażenie zawierające wymienione wyżej symbole terminalne będzie akceptowane przez naszą gramatykę. Nie będą akceptowane wyrażenia niepoprawne algebraicznie np.:

`x + * y)2 + y(xy + 22 + itp`.

Piśmiennictwo: *Cormen T.* C.4.1., *Hyde R.* H.5.1., *Ross K.* R.1.1., *Wirth N.* W.5.1.

3.5.3. GRAMATYKA GENERACYJNA

3.5.3.10. **Wyjaśnienie**. Formalna definicja *gramatyki generacyjnej G*: to jak już zostało powiedziane w wyjaśnieniu 3.5.1.40, to czwórka uporządkowana elementów (V, T, P, S) gdzie:

- V — zbiór elementów pomocniczych zwanych *nieterminalnymi;*
- T — zbiór niepusty symboli *terminalnych* — alfabet języka, który budujemy;
- P — lista dopuszczalnych *produkcji* — formalny opis składni języka, który chcemy zbudować;
- S — *symbol początkowy* (zwany również symbolem *startu*), od którego zaczyna się generacja.

3.5.3.20. **Wyjaśnienie**. Zestaw (lista) produkcji składa się z par *(A, w)*, gdzie *A* jest symbolem pomocniczym, zaś *w* słowem zbudowanym z symboli terminalnych i nieterminalnych. Często *produkcję* oznaczamy $A \rightarrow w$.

Rozważmy, z kolei, kilka przykładów prostych gramatyk, korzystając z przykładu gramatyki generacyjnej:

3.5.3.30. **Przykład.** Gramatyka generuje słowa złożone tylko z jednego symbolu a, który ma w słowie wystąpić zawsze *parzystą liczbę razy*, co symbolicznie oznaczamy a^{2n}. Niech:

`G = { {S}, {a}, P, S}, gdzie P ={S → aaS, S → aa}`

1. Wygenerujemy słowo: aaaaaa.
2. W tym celu wykonamy: S ⇒ aa; aaS ⇒ aaaa; aaaaS ⇒ aaaaaa;

Użyliśmy 2 razy pierwszej produkcji i 1 raz drugiej, otrzymaliśmy słowo spełniające wymaganie, czyli należące do języka.

3.5.3.40. **Przykład.** Gramatyka generuje słowa złożone z dwóch symboli a i b, które mają w nich wystąpić tę samą ilość razy czyli w postaci zapisanej symbolicznie jako $a^n b^n$.

Niech G = {{S}, {a, b}, P, S}, gdzie P = {S → aSb, S → ab}

1. Wygenerujemy z kolei słowo: aaabbb
2. S ⇒ aSb; aSb ⇒ aaSbb; aaSbb ⇒ aaabbb

Użyliśmy 2 razy pierwszej produkcji i 1 raz drugą, otrzymaliśmy słowo należące do języka.

3.5.3.50. **Wyjaśnienie**. Każde słowo zbudowane z *symboli terminalnych* i otrzymane przez wielokrotne zastosowanie produkcji począwszy od symbolu *początkowego* — to *słowo generowane przez gramatykę*. Zbiór wszystkich słów generowany przez daną gramatykę nazywamy *językiem generowanym przez tę gramatykę*.

3.5.3.60. **Wyjaśnienie**. Zgodnie z powyższym, język polski to ogół zdań generowanych zgodnie z regułami gramatyki. Zaś język programowania C, to zbiór wszystkich tekstów wygenerowanych według reguł budowy tego języka (czyli przez gramatykę języka C). Uwaga**:** Interesuje nas tu tylko składnia, a nie znaczenie wypowiedzi - bardziej przecież istotne.

Piśmiennictwo: *Cormen T.* C.4.1., *Hyde R.* H.5.1., *Ross K.* R.1.1., *Wirth N.* W.5.1.

3.5.4. Podział gramatyk formalnych

Dotąd rozważaliśmy produkcje, gdzie po lewej stronie „→" (lub zapis produkcji w notacji BNF „::=") występował pojedynczy symbol nieterminalny. Zatem odpowiedniego podstawiania można było dokonać bez względu na kontekst, w jakim ów symbol nieterminalny wystąpił. Takie gramatyki noszą nazwę *bezkontekstowych*. Istnieje znacznie szersza klasa tzw. *gramatyk kontekstowych*, gdzie o możliwości użycia danej produkcji decyduje kontekst, w jakim podstawiany symbol występuje.

3.5.4.10. **Wyjaśnienie**. Czasem można uprościć *gramatykę bezkontekstową*. Polega to na takim uproszczeniu produkcji, że po prawej stronie „→ " występuje *tylko pojedynczy symbol terminalny* albo *para: symbol terminalny — symbol nieterminalny*. Taką gramatykę nazywamy *regularną*.

Rozważmy zatem *gramatykę regularną* w postaci:

3.5.4.20 G = {{S, T}, {a}, P, S}, gdzie P = {S → aT, T → aS, T → a}

Wygenerujmy nią poprzednio generowane gramatyką bezkontekstową słowo aaaaaa:

S ⇒ AT; AT ⇒ aaS; aaS ⇒ aaaT; aaaT ⇒ aaaaS;

aaaaS ⇒ aaaaaT; aaaaaT ⇒ aaaaaa.

Użyliśmy 3 razy pierwszą produkcję, 2 razy drugą i 1 raz trzecią. Udało się nam znaleźć prostszą gramatykę generującą ten sam język.

3.5.4.30. **Wyjaśnienie**. Język nazywamy określeniem najprostszej gramatyki, która go generuje. Zatem język słów o parzystej liczbie liter a (zapis symboliczny a^{2n}) jest *regularny*. Dla języka

postaci $a^n b^n$ nie da się zbudować gramatyki *regularnej*, jest on więc *bezkontekstowy* (bo taka była nasza gramatyka).

3.5.4.40. **Wyjaśnienie**. Istnieją języki np. typu $a^n b^n c^n$, dla których nie da się znaleźć gramatyki bezkontekstowej. Musimy zatem rozważyć możliwości zbioru produkcji. Niech będzie to nadal skończony zbiór reguł postaci $w \to t$, gdzie w jest dowolnym słowem zawierającym choć jeden symbol nieterminalny (*otoczenie tego symbolu nazywamy kontekstowym*), zaś t dowolnym słowem zawierającym symbole *terminalne i nieterminalne*. Podczas generowania możemy zastępować zamiast pojedynczych symboli nieterminalnych całe lewe strony produkcji.
Przykład. Niech $G = \{N, T, P, S\}$, gdzie $N = \{S, B, C\}$, $T = \{a, b, c\}$, $P = \{S \to aSBC, S \to abc, cB \to Bc, bB \to bb, cC \to cc\}$.
Spróbujmy wygenerować słowo aaabbbccc.
$SaSBC \Rightarrow aaSBCBC \Rightarrow aaabcBCBC \Rightarrow aaabBcCBC \Rightarrow aaabbcCBC \Rightarrow aaabbccBC \Rightarrow aaabbcBcC \Rightarrow aaabbBccC \Rightarrow aaabbbccC \Rightarrow aaabbbccc$.
Zatem istotnie była tu potrzebna *gramatyka kontekstowa*. Spośród używanych języków takiej gramatyki kontekstowej wymagał język ALGOL (bardzo elastyczny, ale skomplikowany). Dlatego między innymi ALGOL 60 został wyparty przez znacznie prostrzy, chociaż w dużym stopniu wzorowany na ALGOL'u - język PASCAL.

3.5.4.50. **Wyjaśnienie**. *Analizator składniowy* czyli *parser* – program dokonujący analizy składniowej danych wejściowych w celu określenia ich struktury gramatycznej w związku z określoną gramatyką formalną. Parser dokonuje przekształcenia formuł arytmetycznych i logicznych zapisanych w formie wyrażeń z nawiasami do postaci bez-nawiasowej, zwanej *odwrotną notacją polska*[44]. Nazwa *analizator składniowy* podkreśla analogię z analizą składniową stosowaną w gramatyce i językoznawstwie. Analizator składniowy umożliwia przetworzenie tekstu czytelnego dla człowieka w strukturę danych w zapisie bez-nawiasowym prostą do bezpośredniego przekształcenia na kod komputera.

3.5.4.60. **Wyjaśnienie**. Budowa kompilatorów języków programowania, wykracza poza zakres niniejszej książki. Zaineresowanych odsyłamy do bogatej literatury tematu.

Piśmiennictwo: *Cormen T.* C.4.1., *Hyde R.* H.5.1., *Ross K.* R.1.1., Sipser M. S.7.1., *Wirth N.* W.5.1.

3.6. WNIOSKOWANIE

3.6.0. WYNIKANIE

Przystępujemy teraz do jednego z najważniejszych i trudniejszych zarazem zadań każdego kursu logiki formalnej. Zadanie to polega na wyjaśnieniu rozumienia funktora zdaniotwórczego od dwu argumentów jednostkowo nazwowych:

... wynika z ...,

ewentualnie - polega ono na wyjaśnieniu rozumienia funkcji zdaniowej dwu zmiennych jednostkowo nazwowych: *B wynika z A*.

Przed przystąpieniem do realizacji tego zadania należy je jeszcze uściślić. Rzecz polega na tym, iż potocznie funktor „wynika z" jest używany w języku naturalnym w różnych rozumieniach. Jedno

[44] *Odwrotna notacja polska* wynaleziona przez polskiego logika Jana Łukasiewicza w latach dwudziestych XX w.

z tych rozumień to tak zwane rozumienie przyczynowe. Tadeusz Kotarbiński podaje następujący przykład na rozumienie przyczynowe funktora „*wynika z*": „ ... do ciasnej izby wprowadziły się dwie rodziny i stąd wynikły między nimi nieporozumienia"[45]. Nam zależy natomiast na wyjaśnieniu tak zwanego węższego rozumienia logicznego. Zależy nam mianowicie na tym, by uchwycić takie rozumienie wyrażenia „wynika z", z którym mamy do czynienia w przykładach następujących.
Przykłady: Bierzemy pod uwagę trzy zdania poniższe:
(1) Sokrates jest człowiekiem.
(2) Każdy człowiek jest śmiertelny.
(3) Sokrates jest śmiertelny.

Stwierdzamy, że zdanie (3) wynika ze zdań (1) i (2). Przejdźmy do drugiego przykładu na wynikanie. Tym razem bierzemy pod uwagę dwie tezy następujące[46]:
(1) Jeżeli x ma gorączkę, to x jest chory;
(2) Jeżeli Jan ma gorączkę, to Jan jest chory.

Łatwo zauważyć, że z tezy (1) wynika teza (2). Trzeci przykład wynikania przedstawia się następująco, Bierzemy pod uwagę tezy[47]:
1) Jeżeli n jest liczbą naturalną, to $(n + 1)$ jest liczbą naturalną;
2) Jeżeli $2m$ jest liczbą naturalną, to $(2m + 1)$ jest liczbą naturalną.

Widać natychmiast, że z tezy (1) wynika teza (2), wreszcie podamy czwarty przykład wynikania, przykład bardzo prosty, ale może mniej codzienny. Bierzemy pod uwagę tezy[48]:
(1) Przy wszelkim x, x jest identyczne z x;
(2) Przy wszelkim y, y jest identyczne z y.
Teza (2) wynika z tezy (1) i co, więcej – teza (1) wynika z tezy(2).

Mając na uwadze powyższe przykłady, wracamy do postawionego sobie na początku zadania. Bezpośrednie jego wykonanie napotyka dwie przeszkody; pierwszą z nich jest potrzeba wprowadzenia do naszych rozważań pewnych pojęć pomocniczych. Wprowadźmy przede wszystkim te pojęcia, a następnie dopiero pomyślmy o przeszkodzie drugiej.

3.6.0.00. **Wyjaśnienie**. Wyrażenie „zespół tez" rozumiemy tu w ten sposób, że spełnione są wszystkie warunki poniższe:
1) Cokolwiek należy do takiego zespołu, jest tezą lub definicją;
2) Zespół złożony, z jednej tylko tezy jest identyczny z tą tezą;
3) Zespół złożony z jednej tylko definicji jest identyczny z tą definicją.

3.6.0.01. **Wyjaśnienie**. Mówi my, że zespół tez Z_1 jest podzespołem zespołu tez Z_2, zamiast mówić, że cokolwiek należy do zespołu tez Z_1 należy też do zespołu tez Z_2.
Przykład. Bierzemy pod uwagę następujące tezy i definicje:
(1) Warszawa leży nad Wisłą.
(2) Paryż leży nad Sekwaną.
(3) Kwadrat $=_{\mathrm{Df}}$ prostokąt równoboczny.

[45] Kotarbiński, K.9.2, s. 57.
[46] Litera „x" jest tu wolną zmienną jednostkowo nazwową.
[47] Każda z liter „m" oraz „n" jest tu wolną zmienną jednostkowo nazwową, za którą wolno podstawiać tylko te nazwy jednostkowe, których desygnatami są liczby naturalne.
[48] W tym przykładzie każda z liter „x" oraz „y" jest związaną zmienną jednostkowo nazwową.

(4) Koło $=_{Df}$ elipsa o równych osiach.

Zespół złożony z (1), (3) i (4) jest podzespołem własnym i ponadto jest podzespołem zespołu złożonego z (1), (2), (3) i (4).

3.6.0.02. **Wyjaśnienie**. Mówimy, że zespół tez Z_1 jest permutacją zespołu tez Z_2, zamiast mówić, że spełnione są wszystkie warunki następujące:

1) Każdej tezie czy definicji należącej do zespołu tez Z_1 odpowiada jedna równokształtna i równoznaczna z nią teza czy definicja należąca do zespołu tez Z_2;
2) Odwrotnie, każdej tezie czy definicji należącej do zespołu tez Z_2 odpowiada jedna i tylko jedna równokształtna i równoznaczna z nią teza czy definicja należąca do zespołu tez Z_1.

Zauważmy teraz, że wyjaśnienie 3.6.0.02, nie wymaga od nas, aby w obrębie dwu zespołów tez, z których jeden jest permutacją drugiego, przestrzegana była ta sama kolejność, wobec czego możemy podać następujący przykład zespołu tez Z_1 i zespołu tez Z_2 oraz stwierdzić, że każdy z tych zespołów jest permutacją drugiego:

<zespół tez Z_1>
<teza> Warszawa leży nad Wisłą </teza>
<teza> Paryż leży nad Sekwaną </teza>
</zespół tez Z_1>
<zespół tez Z_2>
<teza> Paryż leży nad Sekwaną</teza>
<teza> Warszawa leży nad Wisłą </teza>
</zespół tez Z_2>

Opierając się na wyjaśnieniu 3.6.0.02, stwierdzamy obowiązywanie - następujących reguł permutacji:

3.6.0.03. **Reguła**. Jeżeli Z_1 jest zespołem tez, to zespół Z_2 jest permutacją zespołu Z_1.

3.6.0.04. **Reguła**. Jeżeli zespół tez Z_1 jest permutacją zespołu tez Z_2, to zespół Z_2 jest permutacją zespołu Z_1.

3.6.0.05. **Reguła**. Jeżeli (1) zespół tez Z_1 jest permutacją zespołu tez Z_2 oraz (2) zespół tez Z_2 jest permutacją zespołu tez Z_3, to zespół Z_1 jest permutacją zespołu Z_3.

Trzeba będzie teraz rozróżnić między sobą rozmaite rodzaje zespołów tez:

3.6.0.10. **Wyjaśnienie**. Mówimy, że zespół tez jest bez definicyjny, zamiast mówić, że nie należy do niego ani jedna definicja.

3.6.0.11. **Wyjaśnienie**. Mówimy, że zespół tez jest częściowo definicyjny, zamiast mówić, że należy do niego - chociaż jedna definicja oraz że należy do niego, chociaż jedna teza.

3.6.0.12. **Wyjaśnienie**. Mówimy, że zespół tez jest czysto definicyjny, zamiast mówić, ze składa się on tylko z definicji.
Przykłady:

<bez definicyjny zespół tez>
<teza>Warszawa leży nad Wisłą</teza>
<teza>Petersburg leży nad Newą</teza>
</bez definicyjny zespół tez>

<częściowo definicyjny zespół tez>
<teza>Kraków leży nad Wisłą</teza>
<teza>Lublin nie leży nad Wisłą</teza>
<teza>Kwadrat $=_{Df}$ prostokąt równoboczny</teza>
<częściowo definicyjny zespół tez>

<czysto definicyjny zespół tez>
<teza>Dziadek $=_{Df}$ ojciec ojca lub ojciec matki</teza>
<teza>Babka $=_{Df}$ matka ojca lub matka matki</teza>
<teza>Stryj $=_{Df}$ brat ojca</teza>
</czysto definicyjny zespół tez>

Opierając się na wyjaśnieniach: 3.5.0.00. 3.5.0.10, 3.5.0.11, oraz 3.5.0.12, otrzymujemy poniższą regułę:

3.6.0.13. **Reguła**. Każdy zespół tez jest
albo
1) bez definicyjny,
albo
2) częściowo definicyjny,
albo
3) czysto definicyjny.

3.6.0.20. **Wyjaśnienie**. Mówimy, że zespół tez jest czysto zdaniowy, zamiast mówić, że spełnione są wszystkie warunki następujące:
1) Żadna teza należąca do tego zespołu nie jest funkcją zdaniową;
2) Albo zespół ten jest bez definicyjny, albo też w żadnej definicji należącej do tego zespołu nie ma definiensa (w rozumieniu węższym) ani definiendum (w rozumieniu węższym) będącego funkcją.

Przykłady. Weźmy teraz pod uwagę następujące zespoły tez:
<zespół tez Z_1>
<teza 1> Warszawa leży nad Wisłą</teza 1>
<teza 2> Kraków leży nad Wisłą</teza 2>
<teza 3>Petersburg leży nad Newą</teza 3>
</zespół tez Z_1>

<zespół tez Z_2>
<teza 4> Paryż leży nad Sekwaną</teza 4>
<teza 5> Ziemia jest planetą</teza 5>
<teza 6> Kwadrat $=_{Df}$ prostokąt równoboczny</teza 6>
<teza 7> Koło $=_{Df}$ elipsa o równych osiach</teza 7>
</zespół tez Z_2>

<zespół tez Z_3>
<teza 8> Merkury jest planetą</teza 8>
<teza 9> Jeżeli a jest mniejsze od b, to b jest większe od a</teza 9>
</zespół tez Z_3>

<zespół tez Z_4>
<teza 10>$2 < 3$ </teza 10>
<teza 11>$(a \leq b) =_{Df} [(a < b)\ lub\ (a = b)]$</teza 11>
<zespół tez Z_4>

<zespół tez Z_5>
<teza 12> 2 < 3 </teza 12>
<teza 13> 0 $=_{Df}$ (a – a) </teza 13>
</zespół tez Z_5>

Zespół tez Z_1 jest czysto zdaniowy, ponieważ wyrażenie (teza 1) jest zdaniem, wyrażenie (teza 2) jest zdaniem, a także wyrażenie (teza 3) jest zdaniem. Jest to zarazem zespół bez definicyjny. Zespół Z_2 jest czysto zdaniowy, gdyż wyrażenia (teza 4) oraz (teza 5) są zdaniami i ani definiendum, ani definiens definicji (teza 6) nie jest funkcją; to samo dotyczy definicji (teza 7). Zespół tez Z_2 jest ponadto częściowo definicyjny, ponieważ należy do niego definicja (teza 6) oraz definicja (teza 7). Zespół Z_3 nie jest czysto zdaniowy: wyrażenie (teza 8) jest wprawdzie zdaniem, lecz wyrażenie(teza 9) jest funkcją zdaniową. Jest to zresztą zespół bez definicyjny. Zespół tez Z_4 nie jest czysto zdaniowy: wyrażenie (teza 10) jest wprawdzie zdaniem, ale zarówno definiendum (w rozumieniu węższym) jak i definiens w (rozumieniu węższym) definicji (teza 11) są funkcjami. Zespół tez Z_5 również nie jest czysto zdaniowy; wyrażenie (teza 12) jest wprawdzie zdaniem, lecz definiens definicji (teza 13) jest funkcją.

3.6.0.30. **Wyjaśnienie**. Mówimy, że zespół tez jest prawdziwy, zamiast mówić, ze spełnione są wszystkie warunki następujące:

1) Jeżeli jakaś teza należy do tego zespołu, to jest ona tezą prawdziwą;
2) Jeżeli zespół ten zawiera jedną i tylko jedną definicję, to definicja ta jest poprawna;
3) Jeżeli zespół ten zawiera - chociaż dwie definicje, to podzespół wszystkich definicji należących do danego zespołu tez jest poprawnym (w rozumieniu szerszym) zespołem definicji.

3.6.0.31. **Wyjaśnienie**. Mówimy, że zespół tez jest nieprawdziwy, zamiast mówić, że nie jest on prawdziwy. Warto zdać sobie sprawę, że możemy już teraz rozróżnić dwanaście rodzajów zespołów tez. Informuje nas o tym tablica 3.6.0.32. W ten sposób pokonaliśmy pierwszą przeszkodę; dysponujemy już dostatecznym zasobem pojęć pomocniczych. Zajmiemy się teraz przeszkodą drugą. Czytelnicy zapewne zauważyli, że nasze wyjaśnienia mają na ogół postać nader zbliżoną do definicji i mogłyby zostać z łatwością przerobione na definicje. Co prawda, nie wszystkie nasze wyjaśnienia mają tę własność: na przykład wyjaśnienie 3.5.0.00, to raczej wyjaśnienie za pomocą postulatów (o którym była mowa w rozdziale poprzednim) niż definicja.

Tablica 3.6.0.32

Zespoły tez	Zespoły tez			
	czysto zdaniowe		nie czysto zdaniowe	
	nieprawdziwe	prawdziwe	nieprawdziwe	prawdziwe
I.	II.	III.	IV.	V.
Bez definicyjne	Bez definicyjne, nieprawdziwe i czysto zdaniowe	Bez definicyjne, prawdziwe i czysto zdaniowe	Bez definicyjne, nieprawdziwe i nie czysto zdaniowe	Bez definicyjne, prawdziwe i nie czysto zdaniowe
Częściowo definicyjne	Częściowo definicyjne, nieprawdziwe i czysto zdaniowe	Częściowo definicyjne, prawdziwe i czysto zdaniowe	Częściowo definicyjne, prawdziwe i nie czy to zdaniowe	Częściowo definicyjne, prawdziwe i nie czysto zdaniowe
Czysto definicyjne	Czysto definicyjne, nieprawdziwe i czysto zdaniowe	Czysto definicyjne, prawdziwe i czysto zdaniowe	Częściowo definicyjne, nieprawdziwe i nie czy to zdaniowe	Czysto definicyjne, prawdziwe i nie czysto zdaniowe

Otóż podanie definicji czy też, zbliżonego do definicji wyjaśnienia dla funktora:

… wynika z …

czy też dla funkcji zdaniowej:

Z_2 wnika z Z_1,

jest zadaniem trudnym, a w zakresie kursu elementarnego chyba niewykonalnym. Będziemy, więc wyjaśniali, co rozumiemy przez, „wynikanie", za pomocą wyjaśnień częściowych.

3.6.0.40. **Wyjaśnienie częściowe**. Jeżeli Z_2 wynika z Z_1, to Z_1 jest zespołem tez oraz Z_2 jest zespołem tez; ponadto o ile zespół Z_1 jest prawdziwy, to zespół Z_2 też jest prawdziwy.

3.6.0.41. **Wyjaśnienie częściowe**. Jeżeli (1) Z_3 wynika z Z_2 oraz (2) Z_2 wynika z Z_1, to Z_3 wynika z Z_1.

Przykład (do wyjaśnienia częściowego 3.6.0.41). Bierzemy pod uwagę trzy następujące zespoły tez:

<zespół tez Z_1>
<teza> Adam jest człowiekiem</teza>
<teza> Bolesław jest człowiekiem</teza>
<teza> Każdy człowiek jest ssakiem</teza>
<teza> Każdy ssak jest kręgowcem</teza>
<teza> Każdy kręgowiec ma kręgosłup </teza>
</zespół tez Z_1>

<zespół tez Z_2>
<teza> Adam jest ssakiem</teza>
<teza> Bolesław jest ssakiem </teza>
<teza> Każdy ssak ma kręgosłup</teza>
<zespół tez Z_2>

<zespół tez Z_3>
<teza> Adam ma kręgosłup</teza>
<teza> Bolesław ma kręgosłup</teza>
<zespół tez Z_3>

Zespół tez Z_3 wynika z zespołu tez Z_2, zespół tez Z_2 wynika z zespołu tez Z_1 i co więcej, zespół tez Z_3 wynika z zespołu tez Z_1.

3.6.0.42. **Wyjaśnienie częściowe**. Jeżeli (1) Z_3 jest podzespołem zespołu tez Z_2 oraz (2) Z_2 wynika z Z_1, to Z_3 wynika z Z_1.

Przykład. Bierzemy pod uwagę trzy następujące zespoły tez:

<zespół tez Z_1>
<teza> Adam jest uczulony</teza>
<teza> Bolesław jest uczulony</teza>
<teza> Adam jest bratem Bolesława </teza>
<teza> Bolesław jest bratem Czesława </teza>
<teza> Czesław ma tylko dwu braci </teza>
</zespół tez Z_1>
<zespół tez Z_2>
<zespół tez Z_3>
<teza 1> Adam jest bratem Czesława</teza 1>
<teza 2> Choć jeden z braci Czesława jest uczulony </teza 2>
<zespół tez Z_3>
<teza 3> Wszyscy bracia Czesława są uczuleni </teza 3>
</zespół tez Z_2>

Zespół tez Z_3 jest podzespołem zespołu Z_2, zespół tez Z_2 wynika z zespołu tez Z_1 i co więcej, zespół Z_3 wynika z zespołu tez Z_1.

3.6.0.43. **Wyjaśnienie częściowe**. Jeżeli (1) Z_3 wynika z Z_2 oraz (2) Z_2 jest podzespołem zespołu tez Z_1, to Z_3 wynika z Z_1.
Przykład. Bierzemy pod uwagę trzy następujące zespoły tez:

<zespół tez Z_1>
<zespół tez Z_2>
<teza 1>Adam jest niższy od Bolesława</teza 1>
<teza 2>Bolesław jest niższy od Czesława </teza 2>
<teza 3>Czesław jest niższy od Dobiesława </teza 3>
</zespół tez Z_2>
<teza 4>Dobiesław jest niższy od Edwarda </teza 4>
<teza 5>Edward jest niższy od Ferdynanda</teza 5>
<zespół tez Z_1>

<zespół tez Z_3>
<teza 6> Adam jest niższy od Czesława</teza 6>
<teza 7>Bolesław jest niższy od Dobiesława</teza 7>
<teza 8>Adam jest niższy od Dobiesława<teza 8>
</zespół tez Z_3>

Zespół tez Z_3 wynika z zespołu tez Z_2, zespół tez Z_2 jest podzespołem zespołu tez Z_1 i co więcej, zespół tez Z_3 wynika z zespołu tez Z_1.

3.6.0.44. **Wyjaśnienie częściowe**. Jeżeli zespół tez Z_1 jest permutacją zespołu tez Z_2, to Z_2 wynika z Z_1, (oraz Z_1 wynika z Z_2).
Przykład. Bierzemy pod uwagę poniższe dwa zespoły tez:

<zespół tez Z_1>
<teza> Jeżeli x jest mniejsze od y, to y jest większe od x</teza>
<teza> Jeżeli x jest nierówne y, to y jest nierówne x</teza>
<teza> Liczba 0 nie równa się liczbie 1</teza>
<zespół tez Z_1>

<zespół tez Z_2>
<teza> Liczba 0 nie równa się liczbie 1 </teza>
<teza> Jeżeli x jest mniejsze od y, to y jest większe od x</teza>
<teza> Jeżeli x jest nierówne y, to y jest nierówne x</teza>
<zespół tez Z_2>

Otóż zespół Z_1 jest permutacją zespołu Z_2 i ponadto zespół tez Z_2 wynika z zespołu tez Z_1.

3.6.0.45. **Wyjaśnienie częściowe**. Dla każdego zespołu tez Z_1 istnieje taki zespół Z_2, że spełnione są oba warunki następujące:

1) Z_2 wynika z Z_1;
2) Znając zespół Z_1 można zbudować zespół tez Z_2 stosując do tez czy definicji, należących do zespołu Z_1, choć jedną albo niektóre, albo wszystkie z niżej opisanych czynności: kopiowanie, zastępowanie (definiensa przez definiendum), podstawianie (za zmienne), odrywanie i dołączanie (zwykle kwantyfikatora).

Uwaga: Czynności, o których mowa w wyjaśnieniu 3.6.0.45, poznaliśmy już w rozdziale 3.1.
Przykłady. Bierzemy pod uwagę dwa poniższe zespoły tez:

<zespół tez Z_1>
<teza 1> Każdy prostokąt równoboczny jest prostokątem</teza 1>
<teza 2> kwadrat $=_{Df}$ prostokąt równoboczny <teza 2>

</zespół tez Z_1>

<zespół tez Z_2>
<teza> Każdy kwadrat jest prostokątem</teza>
<zespół tez Z_2>

Otóż, jak łatwo zauważyć, zespół Z_2 złożony z jednej tezy wynika z zespołu tez Z_1 i co więcej, jedyną tezę zespołu Z_2 otrzymujemy zastępując w zdaniu (1) definiens (w szerszym rozumieniu) definicji (2) przez jej definiendum. Przejdźmy do przykładu drugiego. Tym razem bierzemy pod uwagę poniższe zespoły tez (każdy złożony tylko z jednej tezy):
<zespół tez Z_1> Jeżeli każde A jest B, to pewne A jest B</zespół tez Z_1>
<zespół tez Z_2> Jeżeli każda stolica jest miastem,
to pewna stolica jest miastem </zespół tez Z_2>

Zespół Z_2 wynika z zespołu Z_1. Łatwo też zauważyć, że tezę Z_2 otrzymujemy z tezy Z_1 podstawiając do niej:

nazwy ogólne	stolica	miasto
za zmienne ogólno-nazwowe	A	B

Jeszcze jeden przykład do częściowego wyjaśnienia 3.5.0.45. Zwróćmy uwagę na dwa następujące zespoły tez (pierwszy z nich składa się z dwu tez, drugi - tylko z jednej):
<zespół tez Z_1>
<teza 1>Jeżeli Adam ma gorączkę, to Adam jest chory</teza 1>
<teza 2> Adam ma gorączkę <teza 2>
</zespół tez Z_1>

<zespół tez Z_2>Adam jest chory </zespół tez Z_2>

Zauważmy, że zespół Z_2 wynika z zespołu Z_1, a przy tym Z_2 powstaje ze zdania (teza 1) przez oderwanie od niego zdania (teza 2) z pominięciem funktora „jeżeli ... to".

Wreszcie ostatni przykład do częściowego wyjaśnienia 1.5.0.45. Mamy dwa poniższe zespoły, oba jednotezowe:
<zespół tez Z_1> $(x+y)^2 = x^2+2xy+y^2$ </zespół tez Z_1>
<zespół tez Z_2> $Dla\ pewnych\ x, y\quad (x+y)^2 = x^2+2xy+y^2$ </zespół tez Z_2>

Z_2 wynika z Z_1, a ponadto zauważmy, że zdanie Z_2 powstaje z funkcji zdaniowej Z_1 przez dołączenie do niej operatora zdaniotwórczego (kwantyfikatora) „$dla\ pewnych\ x, y$". Na tym kończymy przykłady do częściowego wyjaśnienia 3.6.0.45.

3.6.0.46. **Wyjaśnienie częściowe**. Istnieją takie zespoły tez Z_1, Z_2, że spełnione są wszystkie trzy warunki następujące:
1) *Z_1 nie jest permutacją Z_2;*
2) *Z_2 wynika z Z_1*oraz
3) *Z_1 wynika z Z_2.*

Przykład. Bierzemy pod uwagę dwa zespoły tez[49]:
<zespół tez Z_1>
<teza> Jeżeli x jest czerwone, to x jest barwne</teza>
<teza> Jeżeli x jest niebieskie, to x nie jest czerwone </teza>

[49]W tym przykładzie dwu zespołów tez każda z liter „x" oraz „y" jest wolną zmienną jednostkowo nazwową.

</zespół tez Z_1>
<zespół tez Z_2>
<teza> Jeżeli y jest niebieskie, to y nie jest czerwone</teza>
<teza> Jeżeli y jest czerwone, to y jest barwne </teza>
</zespół tez Z_2>

3.6.0.47. **Wyjaśnienie częściowe**. Istnieją takie zespoły tez Z_1, Z_2, że spełnione są wszystkie cztery warunki poniższe:

1) *Z_1 jest zespołem bez definicyjnym;*
2) *Z_2 jest zespołem bez definicyjnym;*
3) *Z_2 nie wynika z Z_1;*
4) *Z_1 nie wynika z Z_2.*

Przykład. Bierzemy pod uwagę dwa zespoły tez:

<zespół tez Z_1>
<teza> *Każdy człowiek jest ssakiem*</teza>
<teza> *Każdy ssak jest kręgowcem*</teza>
</zespół tez Z_1>

<zespół tez Z_2>
<teza> Każda żaba jest płazem </teza>
<teza> Każdy płaz jest kręgowcem </teza>
<zespół tez Z_2>

Tablica 3.6.0.50.		
Zespół tez		Funkcja zdaniowa „Z_2 wynika z Z_1" staje się zdaniem prawdziwym
Z_1	Z_2	
I.	*II.*	*III.*
Bez definicyjny	Bez definicyjny	dla niektórych tylko Z_1, Z_2
	Częściowo definicyjny	dla żadnych Z_1, Z_2
	Czysto definicyjny	
Częściowo definicyjny	Bez definicyjny	dla niektórych tylko Z_1, Z_2
	Częściowo definicyjny	
	Czysto definicyjny	
Czysto definicyjny	Bez definicyjny	dla żadnych Z_1, Z_2
	Częściowo definicyjny	
	Czysto definicyjny	dla niektórych tylko Z_1, Z_2

Podamy teraz jeszcze trzy dalsze (i ostatnie) wyjaśnienia funktora „wynika z". Te trzy wyjaśnienia sformułujemy w postaci tablic (patrz 3.6.0.50, 3.6.0.51, 3.6.0.52). Wyjaśnienie częściowe 3.6.0.52 - informuje nas, że zdarza się, iż zespół prawdziwy tez wynika z zespołu prawdziwego tez. Zdarza się też, że zespół prawdziwy wynika z zespołu nieprawdziwego. Zdarza się również, że zespół nieprawdziwy wynika z nieprawdziwego, natomiast nigdy zespół nieprawdziwy nie wynika z zespołu prawdziwego.

Tablica 3.6.0.51.		
Zespół tez		*Funkcja zdaniowa „Z_2 wynika z Z_1" staje się zdaniem prawdziwym*
Z_1	*Z_2*	
I.	*II.*	*III.*
Czysto zdaniowy	*Czysto zdaniowy*	*dla niektórych tylko Z_1, Z_2*
	Nie czysto zdaniowy	
Nie czysto zdaniowy	*Czysto zdaniowy*	
	Nie czysto zdaniowy	

Przykład. Zespół nieprawdziwy tez, z którego wynika zespół prawdziwy:

<zespół tez Z_1>
<teza> Każda liczba parzysta jest podzielna przez 4 </teza>
<teza> 8 - jest liczbą parzystą </teza>
<teza> 12 - jest liczbą parzystą </teza>
</zespół tez Z_1>

<zespół tez Z_2>
<teza> 8 - jest podzielne przez 4 </teza>
<teza>12 - jest podzielne przez 4 </teza>
</zespół tez Z_2>

Tablica 3.6.0.52.		
Zespół tez		Funkcja zdaniowa „Z_1 wynika z Z_2" staje się zdaniem prawdziwym
Z_1	Z_2	
$I.$	$II.$	$III.$
Nieprawdziwy	Nieprawdziwy	dla niektórych Z_1, Z_2
	Prawdziwy	
Prawdziwy	Nieprawdziwy	dla żadnych Z_1, Z_2
	Prawdziwy	dla niektórych Z_1, Z_2

3.6.0.60. **Wyjaśnienie**. Mówimy, że Z_1 wynika z Z_2, zamiast mówić, że Z_2 wynika z Z_1.

3.6.0.61. **Wyjaśnienie**. Mówimy, że Z_2 jest następstwem Z_1, zamiast mówić, że Z_2 wynika z Z_1.

Piśmiennictwo: *Ajdukiewicz K.* A.2.1., A.2.2., *Greniewski H. G.2.1.* ,*Tarski A*. T.2.1.

3.6.1. UZNAWANIE

Weźmy pod uwagę dwa następujące funktory należące do graficznego języka polskiego:
(1) $stwierdza, że,$[50]
(2) $przypuszcza, że.$[51]
Funktory te, są powszechnie stosowane w systemach ekspertowych, a w szczególności w tzw. systemach informacyjnych Zdzisława Pawlaka, którymi będziemy się dalej zajmować (podrozdział 4.3.3). Oba te funktory należą do rodzaju, któregośmy dotychczas nie badali; są to funktory zdaniotwórcze od jednego argumentu nazwowego i jednego argumentu zdaniowego. Świadczą o tym następujące przykłady zdań:
(3) *Adam stwierdza, że Bolesław wyzdrowiał.*
(4) *Adam przypuszcza, ze Bolesław wyzdrowiał.*

Nie trudno zbudować w JPM funkcje zdaniowe za pomocą omawianych funktorów. Będą to funkcje następujące:
(5) $x\ stwierdza, że\ p.$
(6) $x\ przypuszcza, że\ p.$

Gdybyśmy mieli miejsce i czas, żeby zająć się tu pewnym działem logiki zwanym *erystyką* (= teoria sporu), to zainteresowalibyśmy się bliżej funktorami (1) i (2), ewentualnie funkcjami zdaniowymi (5) i (6). W *erystyce* bowiem, badamy właśnie takie sytuacje, w których występują dwie strony (osoby) niezgodnie między sobą stwierdzające, a więc sytuacje o schemacie poniższym:

[50] Na funktorze tym opiera się powszechnie stosowane przez informatyków upraszczanie algorytmów lub programów komputerowych.
[51] Na funktorze tym opiera się tzw. *refinament* – czyli doskonalenie lub poprawianie wymagań funkcjonalny na system informatyczny.

(7) $x\ jest\ różny\ od\ y.$
(8) $x\ stwierdza, że\ p.$
(9) $y\ stwierdza, że\ fałszem\ jest, że\ p.$

Ponieważ brak nam, niestety, miejsca i czasu na *erystykę*, wobec tego tezy logiki formalnej będziemy stwierdzali jednostronnie, to jest stwierdzać je będzie autor, a miejmy nadzieję, że zgodnie z nim - i czytelnik. Toteż bardziej niż funktory dwuargumentowe (1) i (2) będą nas interesowały funktory jedno-argumentowe:
(10) $stwierdzamy, że,$
(11) $przypuszczamy, że$
i odpowiadające im funkcje zdaniowe:
(12) $stwierdzamy, że\ p,$
(13) $przypuszczamy, że\ p.$

Przydatne tu będą skróty, które wprowadzimy za pomocą poniższych definicji:

3.6.1.00 $(\vdash p) =_{\mathrm{Df}} (Stwierdzamy, że\ p).$

3.6.1.01 $(\Vdash p) =_{\mathrm{Df}} (Przypuszczamy, że\ p).$

Należy starannie odróżniać funkcję zdaniową (12) od poniższej funkcji zdaniowej:
(14) $Stwierdzamy\ tezę\ x.$

Niemniej starannie należy odróżniać funkcję zdaniową (13) od następującej funkcji zdaniowej:
(15) $Przyjmujemy\ hipotezę\ x.$

Łatwo zauważyć, że (12) jest funkcją zdaniową jednej zmiennej zdaniowej, natomiast (14) jest funkcją zdaniową jednej zmiennej jednostkowo nazwowej. Analogicznie - wyrażenie (13) jest funkcją zdaniową jednej zmiennej zdaniowej, wyrażenie (15) zaś jest funkcją zdaniową jednej zmiennej jednostkowo nazwowej. Chcąc uzyskać zdanie z funkcji zdaniowej (12) podstawiamy:
- do funkcji zdaniowej (12)
- za zmienną zdaniową p
- zdanie.

Chcąc natomiast uzyskać zdanie z funkcji zdaniowej (14) podstawiamy:
- do funkcji zdaniowej (14)
- za zmienną jednostkowo nazwową x,
- nazwę jednostkową zdania (czy też - ogólniej sprawę ujmując – nazwę jednostkową tezy).

Analogicznie, chcąc uzyskać zdanie z funkcji zdaniowej (13) podstawiamy:
- do funkcji zdaniowej (13)
- za zmienną zdaniową p
- zdanie.

Jeżeli chcemy natomiast uzyskać zdanie z funkcji zdaniowej (15), podstawiamy:
- do funkcji zdaniowej (15)
- za zmienną jednostkowo nazwową x
- nazwę jednostkową zdania (czy też - ogólniej sprawę ujmując – nazwę jednostkową tezy).

Być może, że przydatny tu będzie konkretny przykład. Weźmy pod uwagę zdanie:

(16) Warszawa jest stolicą Polski.

Zauważmy, że po lewej stronie tego zdania napisaliśmy numer „(16)”. Numer ten nie jest częścią naszego zdania. jest on natomiast - jego okazjonalną nazwą jednostkową. Podstawiamy teraz
- do funkcji zdaniowej (12)
- za zmienną zdaniową p
- zdanie (16),

otrzymujemy jako wynik podstawienia zdanie następujące:
(17) $Stwierdzamy, że$ Warszawa jest stolicą Polski.

Definicja 1.5.1.00 pozwala nam przy tym zastąpić zdanie (17) krótszym zdaniem poniższym:
(18) $\vdash$ Warszawa jest stolicą Polski.

Dokonajmy teraz nowego podstawienia, podstawmy mianowicie:
- do funkcji zdaniowej (14)
- za zmienną jednostkowo nazwową x
- nazwę jednostkową zdania (16),

otrzymujemy jako wynik podstawienia zdanie poniższe:
(19) $Stwierdzamy\ tezę$ (16).

Piśmiennictwo: *Ajdukiewicz K.* A.2.1, A.2.2, *Greniewski H.* G.2.1.

3.6.2. WNIOSKOWANIE

W podrozdziale poprzednim (tj. 3.6.1) zajmowaliśmy się różnymi rodzajami asercji; interesowało nas samo stwierdzanie, uznawanie tezy, ale nie badaliśmy, na jakiej podstawie pewną tezę czy pewien zespół tez stwierdzamy. Często, ale nie zawsze, zdarza się, że uznajemy pewną tezę na podstawie innej tezy już poprzednio uznanej czy na podstawie zespołu innych tez już poprzednio uznanych. Nie zawsze tak się dzieje, gdyż niekiedy uznajemy tę czy inną tezę na tej podstawie, że jest ona wiernym opisem naszego doświadczenia, niekiedy zaś na tej podstawie, że odnośna teza jest oczywista[52].

Nas będą tu interesowały jedynie te sytuacje, w których uznajemy tezę (czy zespół tez) na podstawie zespołu tez. W związku z tym zajmiemy się tu obu funkcjami:
(1) $(p_1, \dots, p_n), więc\ q.$
(2) $(p_1, \dots, p_n), więc\ zapewne\ q.$

Systemy ekspertowe opierają się zarówno na regule wnioskowania (1), jak również, formułując hipotezę na regule (2).

3.6.2.00. **Wyjaśnienie**. Mówimy „$q, ponieważ\ (p_1, \dots, p_n)$”, zamiast mówić „$(p_1, \dots, p_n), więc\ q$”.

3.6.2.01. **Wyjaśnienie**. Mówimy „$przypuszczalnie\ q, poniewż\ (p_1, \dots, p_n)$", zamiast mówić „$(p_1, \dots, p_n), więc\ zapewne\ q$”.
Okazuje się celowe przyjęcie następujących definicji i schematów definicyjnych:

3.6.2.10 $[(p_1, \dots, p_m) \vdash q] =_{Df} [(p_1, \dots, p_m), więc\ q],$

3.6.2.11 $[(p_1, \dots, p_m) | \vdash q] =_{Df} [(p_1, \dots, p_m), więc\ zapewne\ q],$

[52] Ubocznie zwracamy uwagę, że poczucie oczywistości niejednokrotnie okazuje się podstawą nader zwodną.

3.6.2.12 $[(p_1, \ldots, p_m) \vdash (q_1, \ldots, q_{n+1})] =_{Df}$
$\{[(p_1, \ldots, p_m) \vdash (q_1, \ldots, q_n)]\ oraz\ [(p_1, \ldots, p_m) \vdash q_{n+1}]\}$,

3.6.2.13 $[(p_1, \ldots, p_m)| \vdash (q_1, \ldots, q_{n+1})] =_{Df}$
$\{[(p_1, \ldots, p_m)| \vdash (q_1, \ldots, q_n)]\ oraz\ [(p_1, \ldots, p_m)| \vdash q_{n+1}]\}$.

Uznawanie tezy na podstawie innych tez ma pewne własności opisane w zasadach poniższych:

3.6.2.20. **Zasada**. *Jeżeli* $(p_1, \ldots, p_m) \vdash q, to \vdash q$[53].

3.6.2.21. **Zasada**. *Jeżeli* $(p_1, \ldots, p_m) \Vdash q, to \Vdash q$.

Pierwsza z powyższych zasad ustala związek między stwierdzaniem, którym teraz się zajmujemy (tj. stwierdzaniem opartym na innych tezach niż teza stwierdzana), a stwierdzaniem (niemotywowanym), o którym była mowa poprzednio. Analogiczną rolę odgrywa zasada 3.6.2.21; ustala ona związek - między przypuszczaniem tezy, opartym na innych tezach, a przypuszczaniem niemotywowanym.

Przyjmujemy jeszcze definicje następujące:

3.6.2.30. $[(q_1, \ldots, q_n) \dashv (p_1, \ldots, p_m)] =_{Df} [(p_1, \ldots, p_m) \vdash (q_1, \ldots, q_n)]$.

3.6.2.31. $[(q_1, \ldots, q_n) -\!\| (p_1, \ldots, p_m)] =_{Df} [(p_1, \ldots, p_m) \Vdash (q_1, \ldots, q_n)]$.

Możemy teraz krótko wyjaśnić rozumienie wyrażeń: wnioskowanie, przesłanka, wniosek.

3.6.2.40. **Wyjaśnienie**. Przez „wnioskowanie" rozumiemy tu wszelki i tylko taki komunikat, który ma jedną z dwu następujących postaci:

$$(q_1, \ldots, q_n) \dashv (p_1, \ldots, p_m)$$
$$(q_1, \ldots, q_n) -\!\| (p_1, \ldots, p_m)$$

3.6.2.41. **Wyjaśnienie**. Przez „wniosek (w rozumieniu węższym)", będziemy rozumieli każdą i tylko taką tezę, która we wnioskowaniu poprzedza znak „$\dashv$", ewentualnie znak „$-\!\|$".

3.6.2.42. **Wyjaśnienie**. Przez „przesłankę (w rozumieniu węższym)" będziemy rozumieli każdą i tylko taką tezę, która we wnioskowaniu następuje po znaku „$\dashv$", ewentualnie po znaku „$-\!\|$".

3.6.2.43. **Wyjaśnienie dodatkowe**. Wyrażenia „poprzedza" i „następuje po", są w poprzednich wyjaśnieniach użyte zgodnie z kolejnością, która w odnośnym języku obowiązuje.

Podamy teraz dwa przykłady schematyczne i dwa konkretne na wnioskowanie.

Przykłady (schematyczne):

Wnioskowanie		Wnioskowanie	
$(q_1, q_2, q_3) \dashv$	(p_1, p_2, p_3, p_4)	$(q_1, q_2) -\!\|$	(p_1, p_2, p_3)
Wnioski (w rozumieniu węższym)	Przesłanki (w rozumieniu węższym)	Wnioski (w rozumieniu węższym)	Przesłanki (w rozumieniu węższym)

Przykłady (konkretne):

<u>Wnioskowanie</u>

(Adam jest chory) $\dashv$ [(Jeżeli Adam ma gorączkę, to Adam jest chory), (Adam ma gorączkę)

Wniosek (w rozumieniu węższym) Przesłanka (w rozumieniu węższym) Przesłanka (w rozumieniu węższym)

[53] Czytamy te zasadę: Jeżeli $[(p_1, \ldots, p_n), więc\ q]$, to stwierdzamy, że q.

Wnioskowanie

(*Jutro będzie deszcz*)	(*Mamy pochmurny wieczór*)
Wniosek	Przesłanka
(w rozumieniu węższym)	(w rozumieniu węższym)

3.6.2.44. **Wyjaśnienie**. Przez „wniosek (w rozumieniu szerszym)" - będziemy rozumieli wszelką i tylko taką tezę, która jest równokształtna i zarazem równoznaczna z wnioskiem (w rozumieniu węższym).

3.6.2.45. **Wyjaśnienie**. Przez „wniosek (w rozumieniu szerszym)" - będziemy rozumieli wszelką i tylko taką tezę, która jest równokształtna i zarazem równoznaczna z przesłanką (w rozumieniu węższym).

Już obecnie zaczyna się zarysowywać pewne podobieństwo między nauką o definicjach a nauką o wnioskowaniu. Wkrótce przekonamy się, że podobieństwo to sięga dość daleko. Należy starannie odróżniać:

1) Wnioskowanie od wynikania;
2) Przesłankę od racji;
3) Wniosek od następstwa.

Powyższe rozróżnienia - staną się w pełni jasne, gdy wyjaśnimy, na czym polegają dwa szczególnie doniosłe rodzaje wnioskowań: dedukcja i redukcja (nastąpi to wkrótce). Chwilowo poprzestaniemy na niezbyt ścisłej, ale dość dobitnej uwadze: każde wynikanie to obiektywny związek zachodzący między dwoma zespołami tez: związek taki ma zawsze pewien odpowiednik w rzeczywistości materialnej, czy też (inaczej się wyrażając) zapis wynikania jest zawsze pośrednio opisem jakiegoś fragmentu rzeczywistości materialnej. Natomiast każde wnioskowanie jest wysłowieniem procesu myślowego odbywającego się w którymś z mózgów. Taki proces zachodzący w mózgu - może nie być wiernym odbiciem rzeczywistości materialnej, mimo do czynienia z wnioskowaniem. Można by powyższe streścić w ten sposób (jeszcze bardziej nieścisły, ale jeszcze bardziej dobitny): wynikanie - to związek logiczny, wnioskowanie to wysłowienie procesu psychicznego.

Mając dane dwa zespoły Z_1 i Z_2 natkniemy się zawsze na jedną z czterech sytuacji poniższych:

1) Z zespołu Z_1 nie wynika zespół Z_2 i ponadto z zespołu Z_1 nie wnioskujemy zespołu Z_2;
2) Z zespołu Z_1 nie wynika zespół Z_2,lecz z zespołu Z_1 wnioskujemy zespół Z_1 (dzieje się tak na przykład w razie wnioskowania błędnego oraz niekiedy, gdy dokonujemy wnioskowania redukcyjnego, o którym mowa niżej);
3) Z zespołu Z_1 wynika zespół Z_2, lecz z zespołu Z_1 nie wnioskujemy zespołu Z_2 (wynikanie nie odkryte, czy też nie wykorzystane);
4) Z zespołu Z_1 wynika zespół Z_2 i z zespołu Z_1 wnioskujemy zespół Z_2 (np. wnioskowanie dedukcyjne, o którym mowa niżej).

Wyróżnimy teraz wśród ogółu wnioskowań dwa szczególnie doniosłe: dedukcję i redukcję.

3.6.2.50. **Wyjaśnienie**. Mówimy, że dane wnioskowanie jest dedukcją, zamiast mówić, że z zespołu wszystkich przesłanek tego wnioskowania wynika zespół wszystkich jego wniosków.

3.6.2.51. **Wyjaśnienie**. Mówimy, że dane wnioskowanie jest redukcją, zamiast mówić, że:

1) Z zespołu wszystkich wniosków tego wnioskowania wynika zespół wszystkich jego przesłanek
   oraz

2) Wnioskowanie ma postać: $(q_1, \ldots, q_n) \dashv (p_1, \ldots, p_m)$.
W oparciu o powyższe wyjaśnienia - otrzymujemy reguły:

3.6.2.52. **Reguła**. W każdej dedukcji zespół wszystkich przesłanek jest racją, zespół wszystkich wniosków – następstwem.

3.6.2.53. **Reguła**. W każdej redukcji zespół wszystkich przesłanek jest następstwem, zespół wszystkich wniosków - racją.

Obie te reguły formułuje się tez w sposób mniej wprawdzie ścisły, ale bardziej obrazowy:
1) Dedukcja przebiega zgodnie z kierunkiem wynikania (tj. od racji do następstwa);
2) Redukcja przebiega niezgodnie z kierunkiem wynikania (tj. od następstwa do racji).

Jak wiemy z podrozdziału 3.6.0, z nieprawdziwej racji wynika niekiedy prawdziwe następstwo, toteż redukcja jest wnioskowaniem niepewnym; znaczy to, że niejednokrotnie prowadzi ona, od zespołu przesłanek (następstwo) prawdziwych do nieprawdziwego zespołu wniosków (racja).

Podział wnioskowań dla dedukcje i redukcje jest nierozłączny i niezupełny (zwrócił na to uwagę - Kazimierz Ajdukiewicz). Chcemy przez to powiedzieć, że:
1) Istnieją takie wnioskowania, z których każde jest dedukcją i zarazem redukcją;
2) Istnieją takie wnioskowania, z których żadne nie jest ani dedukcją, ani redukcją.

Podamy teraz kilka przykładów wnioskowania, zachowując przy tym kolejność następującą:
1) Dedukcja niebędąca redukcją;
2) Dedukcja będąca redukcją;
3) Redukcja niebędąca dedukcją.
Przykłady. Przyjmijmy, że c jest jakaś liczbą całkowitą. Zamiast pisać, że liczba całkowita c_1 jest dzielnikiem liczby całkowitej c_2, będziemy krótko pisali „$(c_1|c_2)$". Możemy teraz zwięźle sformułować zapowiedziane przykłady.
Dedukcja, lecz nie redukcja:

$$(10 \mid c) \dashv \mid [(6 \mid c), (35 \mid c)].$$

Dedukcja i zarazem redukcja:

$$(10 \mid c), (21 \mid c)] -\!\!\| \ [(6 \mid c), (35 \mid c)].$$

Redukcja lecz nie dedukcja:

$$[(10 \mid c), (21 \mid c)] \dashv \mid (6 \mid c).$$

Wszystkie powyższe przykłady zostały zbudowane wedle pewnej „recepty", którą teraz „ujawnimy":
1. Bierzemy pod uwagę cztery tezy: x_1, x_2, y_1, y_2.
2. Zespół złożony wyłącznie z tez x_1, x_2 oznaczamy „$\{x_1, x_2\}$".
3. Zespół złożony wyłącznie z tez y_1, y_2 oznaczamy „$\{y_1, y_2\}$".
4. Tezy o których mowa w punkcie 1), są tak dobrane, że:
   a) y_1 wynika z $\{x_1, x_2\}$, lecz nie odwrotnie;
   b) x_1 wynika z $\{y_1, y_2\}$, lecz nie odwrotnie;
   c) $\{x_1, x_2\}$ wynika z $\{y_1, y_2\}$ i odwrotnie.
5. Zamiast tez: x_1, x_2, y_1, y_2 będziemy pisali odpowiednio p_1, p_2, q_1, q_2.

Po powyższych ustaleniach przejdziemy do zapowiedzianych „*recept*" (schematów) przykładów wynikania:

Przykłady (schematyczne). Dedukcja, lecz nie redukcja:

$$q_1 \dashv (p_1, p_2).$$

Dedukcja, lecz nie redukcja:

$$q_1 -\!\!\| (p_1, p_2).$$

Dedukcja i zarazem redukcja:

$$(q_1, q_2) -\!\!\| (p_1, p_2).$$

Redukcja, lecz nie dedukcja:

$$(q_1, q_2) -\!\!\| p_1.$$

Wykazaliśmy za pomocą przykładu, że podział ogólny wnioskowań na dedukcje i redukcje nie jest rozłączny. Daliśmy - bowiem przykład wnioskowania, które jest zarazem redukcją i dedukcją. Podamy teraz przykład wnioskowania, które nie jest dedukcją ani redukcją; wykażemy tym samym, że ów podział nie jest zupełny.
Przykład. Weźmy pod uwagę następujące wnioskowanie: Może jutro będzie ładna pogoda, ponieważ dziś wieczór jest bezchmurny. Wynikanie nie łączy w tym przykładzie ani przesłanki z wnioskiem, ani wniosku z przesłanką, a więc podany został przykład wnioskowania, które nie jest dedukcją i nie jest redukcją.

Oprócz redukcji i dedukcji tradycyjnie wyróżnia się jeszcze pewien rodzaj wnioskowań, mianowicie *indukcję*. Na ogół przyjmuje się, że indukcja to tyle, co wnioskowanie zawierające tylko jeden wniosek (w węższym rozumieniu) i to taki, który jest bardziej ogólny niż którakolwiek z przesłanek (w węższym rozumieniu) tegoż wnioskowania. Nie będziemy tutaj starali się poprawić czy uściślić - tego wyjaśnienia, które powinno stanowić punkt wyjścia dla *logiki indukcji*, nauki doniosłej zarówno z punktu widzenia przyrodoznawstwa jak i techniki. Pamiętajmy jednak, że zarówno dedukcja jak również indukcja, są podstawowymi metodami wykorzystywanymi w systemach ekspertowych, jak również w eksploracji danych.

Piśmiennictwo: *Ajdukiewicz K.* A.2.1., A.2.2., *Greniewski H.* G.2.1.

3.6.3. ZADANIA A WNIOSKOWANIE

Poznaliśmy już pewien podział ogółu wnioskowań na dwa rodzaje: dedukcje i redukcje. Jest to podział tradycyjny, o znacznej po dziś dzień doniosłości, pomimo że nie jest on rozłączny i nie jest zupełny. Obecnie poznamy w zarysie nowy podział ogółu wnioskowań, zaproponowany przez Kazimierza Ajdukiewicza. Podział ten polega na odróżnieniu wnioskowań spontanicznych od wnioskowań kierowanych przez postawione zadanie, a następnie na rozróżnieniu rozmaitych rodzajów wnioskowań kierowanych w zależności od postawionego zadania.

3.6.3.00. **Wyjaśnienie**. Przez „*wnioskowanie spontaniczne*" rozumiemy takie i tylko takie wnioskowanie, które nie jest kierowane przez uprzednio postawione zadanie.

3.6.3.10. **Wyjaśnienie**. Przez „*dowodzenie*" rozumiemy takie i tylko takie wnioskowanie, które jest kierowane przez zadanie postaci: *wykaż, że p*.

3.6.3.11. **Wyjaśnienie**. Przez „*sprawdzanie*" rozumiemy takie i tylko takie wnioskowanie, które jest kierowane przez zadanie postaci: *sprawdź, czy p*.

3.6.3.12. **Wyjaśnienie**. Przez „*tłumaczenie*" rozumiemy takie i tylko takie wnioskowanie, które jest kierowane przez zadanie postaci: *stwierdziliśmy, że p*, *ale dlaczego p*?

Tablica 3.6.3.20		
Rodzaj wnioskowania	Zadanie kierujące	Wniosek
$I.$	$II.$	$III.$
Dowodzenie	Wykaż, że p!	p
Sprawdzanie	Sprawdź, czy p!	p Fałszem jest, że p
Tłumaczenie	Stwierdzamy, że p, ale dlaczego p?	$p \dashv (q_1, \dots, q_n)$

Zgodnie z powyższymi wyjaśnieniami każde dowodzenie, każde sprawdzanie i każde tłumaczenie jest wnioskowaniem mającym tylko jeden wniosek (w węższym rozumieniu). Wniosek taki ma przy tym postać z góry wyznaczoną przez zadanie kierujące. Zależność wniosku i rodzaju wnioskowania od zadania kierującego podaje tablica 3.5.3.20.

Piśmiennictwo: *Ajdukiewicz K.* A.2.1., A.2.2., *Greniewski H.* G.2.1. , *Kotarbiński T.* K.5.1., K.5.2.,K.5.3.

3.6.4. Przydatność wnioskowania

W rozdziale 3.4, zajmowaliśmy się, między innymi, przydatnością definicji. Obecnie zajmiemy się pokrótce przydatnością wnioskowań. Zauważymy zresztą, że oba tematy wykazują niejedno podobieństwo:

1) Często używamy deklinacji do wzbogacenia naszego języka; często również posługujemy się wnioskowaniem celem wzbogacenia naszej wiedzy.
2) Niejednokrotnie używamy definicji celem usunięcia wątpliwości, jak rozumieć dane wyrażenie; niejednokrotnie też, używamy wnioskowania dla usunięcia wątpliwości, czy uznać daną tezę.
3) Niejednokrotnie używamy definicji w tym celu, aby komuś zakomunikować rozumienie dotychczas niezrozumiałego dlań wyrażenia; niejednokrotnie również posługujemy się wnioskowaniem, aby przekonać kogoś o prawdziwości tezy, której dotychczas ten ktoś nie stwierdził.
4) Niekiedy posługujemy się definicją po to, żeby wykazać teoretyczną zbędność pewnych wyrażeń w języku, czy też, inaczej mówiąc, aby teoretycznie wyrugować pewne wyrażenia naszego języka zredukować je do innych wyrażeń traktowanych - jako wyrażenia bardziej podstawowe. Niekiedy posługujemy się także wnioskowaniem (redukcją), aby wykazać teoretyczną zbędność pewnych hipotez, zredukować je do innych, traktowanych - jako bardziej podstawowe.
5) Wnioskowanie jest podstawą działania systemów ekspertowych.[54]

Piśmiennictwo: *Ajdukiewicz K.* A.2.1., A.2.2., *Greniewski H.* G.2.1.

3.6.5. Zależność między wnioskowaniami

3.6.5.00. **Wyjaśnienie**. Mówmy, że wnioskowanie ***B*** jest bezpośrednio zależne od wnioskowania ***A***, zamiast mówić, że przynajmniej jeden z wniosków (w szerszym rozumieniu) wnioskowania ***A*** jest przesłanką (w węższym rozumieniu) wnioskowania ***B***.

Przykład (schematyczny). Wnioskowanie (2) bezpośrednio zależne od wnioskowania (1):

(1) $q_1 \dashv (p_1, p_2)$.

(2) $r \dashv (q_1, q_2)$.

[54] Informatyka dała dodatkowy asumpt do formalizacji wnioskowania.

Łatwo zauważyć podobieństwo zachodzące między wyjaśnieniem 3.4.4.00, bezpośrednia zależność między definicjami, a wyjaśnieniem powyższym (bezpośrednia zależność między wnioskowaniami).

3.6.5.01. **Wyjaśnienie**. Mówimy, ze wnioskowanie ***C*** usuwa bezpośrednią zależność wnioskowania ***B*** od wnioskowania ***A***, zamiast mówić, że:

1) Wnioskowanie ***B*** jest bezpośrednio zależne od wnioskowania ***A***
   oraz
2) Wnioskowanie ***C*** powstaje z wnioskowania ***B*** przez zastąpienie:
   - w przesłankach (w węższym rozumieniu) wnioskowania ***B***
   - tych przesłanek, które są wioskami (w szerszym rozumieniu) wnioskowania ***A***
   - kompletem przesłanek (w szerszym rozumieniu) wnioskowania ***A***.

Przykład (schematyczny).
Wnioskowanie (3) usuwające zależność wnioskowania (2) od wnioskowania (1):

(1) $q_1 \dashv (p_1, p_2)$.
(2) $r \dashv (q_1, q_2)$.
(3) $r \dashv (p_1, p_2, q_2)$.

I tym razem podobieństwo między wyjaśnieniem 3.4.4.01 (usuwanie bezpośredniej zależności między definicjami), a wyjaśnieniem powyższym (usuwanie bezpośredniej zależności między wnioskowaniami) jest łatwe do zauważenia.

3.6.5.10. **Wyjaśnienie**. Mówimy, że w danym zespole wnioskowań wnioskowanie ***B*** jest zależne od wnioskowania ***A***, zamiast mówić, że wszystkie czy niektóre wnioskowania należące do tego zespołu można ustawić kolejno w sposób spełniający warunki następujące:

1) Pierwsze w tej kolejności jest wnioskowanie ***A***;
2) Ostatnie w tej kolejności jest wnioskowanie ***B***;
3) Każde z wnioskowań ustawionych w tej kolejności (z wyjątkiem wnioskowania pierwszego) jest bezpośrednio zależne od wnioskowania bezpośrednio je poprzedzającego. (Analogiczne wyjaśnienie 3.4.4.10).

Przykład (schematyczny). Wnioskowanie (2) zależne w danym zespole od wnioskowania (1):

(1) $q_1 \dashv (p_1, p_2)$.
$r_1 \dashv (q_1, q_2)$.
(2) $s \dashv (r_1, r_2)$.

3.6.5.11. **Wyjaśnienie**. Mówimy, że w danym zespole wnioskowań wnioskowanie C_n usuwa zależność wnioskowania A_n od wnioskowania A_0, zamiast mówić, że:

1) Wnioskowanie A_n jest zależne od wnioskowania A_0
   oraz
2) Ustawiamy wnioskowania danego zespołu według kolejności opisanej w wyjaśnieniu poprzednim:
   $A_0, A_1, \ldots, A_{n-1}, A_n$
   następnie zapisujemy wnioskowanie C_1 usuwając bezpośrednią zależność wnioskowania A_1 od wnioskowania A_0, następnie zapisujemy wnioskowanie C_2 usuwające bezpośrednią zależność wnioskowania A_2 od wnioskowania C_1 itd., aż wreszcie zbudujemy wnioskowanie C_n usuwając bezpośrednią zależność wnioskowania A_n od wnioskowania C_{n-1}. (Analogiczne wyjaśnienie 3.4.4.11).

Przykład (schematyczny). Weźmy ,pod uwagę zespół wnioskowań:

(A_0) $q_1 \dashv (p_1, p_2).$
(A_1) $r_1 \dashv (q_1, q_2).$
(A_2) $s_1 \dashv (r_1, r_2).$
(A_3) $t \dashv (s_1, s_2).$

Zapiszemy teraz wnioskowania usuwające zależność bezpośrednią:

(C_1) $r_1 \dashv (p_1, p_2, q_2).$
(C_2) $s_1 \dashv (p_1, p_2, q_2, r_2).$
(C_3) $t \dashv (p_1, p_2, q_2, r_2, s_2).$

Łatwo zauważyć, że (C_3) usuwa zależność wnioskowania (A_3) od wnioskowania (A_0).

Piśmiennictwo: *Ajdukiewicz K.* A.2.1., A.2.2., *Greniewski H.* G.2.1.

3.6.6. Błędne koło we wnioskowaniu

3.6.6.00. **Wyjaśnienie**. Mówimy, że dane wnioskowanie zawiera, bezpośrednie błędne koło, zamiast mówić, że jeden jego wniosków (w szerszym rozumieniu) jest przesłanką (w węższym rozumieniu) tegoż wnioskowania.

Przykład (schematyczny). Bezpośrednie błędne koło we wnioskowaniu:

$$(q_1, q_2, q_3) \dashv (p_1, p_2, q_3)$$
$$\uparrow \text{----------} \uparrow$$
równokształtne i równoznaczne!

Wydaje się celowe porównanie wyjaśnienia poniższego z wyjaśnieniem 3.4.5.00, w pełni bowiem - uwydatni się wówczas głębokie podobieństwo, między bezpośrednim błędnym kołem w definiowaniu a bezpośrednim błędnym kołem we wnioskowaniu.

3.6.6.10. **Wyjaśnienie**. Mówimy, że dany zespół wnioskowań zawiera pośrednie błędne koło, zamiast mówić, że można zbudować takie wnioskowanie, które:

1) Usuwa zależność jednego z wnioskowań tego zespołu od innego wnioskowania tego zespołu oraz
2) Zawiera bezpośrednie błędne koło.

Podamy teraz dwa przykłady pośredniego błędnego koła; pierwszy z tych przykładów będzie miał charakter schematyczny, drugi - konkretny.

Przykład (schematyczny). Weźmy pod uwagę zespół złożony z trzech schematów wnioskowań:

(1) $q_1 \dashv (p_1, p_2).$
(2) $r_1 \dashv (q_1, q_2).$
(3) $p_1 \dashv (r_1, r_2).$

Łatwo teraz zauważyć, że schemat wnioskowania:

(4) $p_1 \dashv (p_1, p_2, q_2, r_2),$

usuwa zależność (3) od (1) i zawiera bezpośrednie błędne koło, wobec czego zespół (1), (2), (3) zawiera pośrednie błędne koło. Przejdziemy do następnego przykładu; na wstępie jednak wzbogacimy język. w którym będziemy formułowali przykład (JPM), dwoma skrótami:

3.6.6.20 $x < y \;=_{\mathrm{Df}}\; x \textit{ jest mniejsze od } y.$
3.6.6.21 $x > y \;=_{\mathrm{Df}}\; x \textit{ jest większe od } y.$

Przykład (konkretny). Chcemy otrzymać w drodze wnioskowania następujący zespół tez:

<zespól tez Z_1>
<teza (1)> $Jeżeli (x < y \text{ orazy } < z), to\ x < z$ </teza (1)>
<teza (2)> $x < y\ wtedy\ i\ tylko\ wtedy, jeżeli y >\ x$ </teza (2)>
</zespól tez Z_1>

Wnioskowanie nasze oprzemy na następującym zespole tez:
<zespól tez Z_2>
<teza (3)> $Jeżeli\ (x > y\ oraz\ y > z), to\ x > z$ </teza (3)>
<teza (4)> $x > y\ wtedy\ i\ tylko\ wtedy, jeżeli\ y < x$ </teza (4)>
</zespól tez Z_2>

Podstawmy do tezy (4)

zmienne	x	y
za zmienne	y	x

otrzymujemy wynik:
(5) $y > x\ wtedy\ i\ tylko\ wtedy, jeżeli\ x < y.$
Z tezy (5) wynika teza:
(6) $x < y\ wtedy\ i\ tylko\ wtedy, jeżeli\ y > x.$
(Zauważmy tu, że teza (6) jest równokształtna i równoznaczna z tezą (2)).
Z tezy (4) wynika ponadto:
(6) $Jeżeli\ y < x, to\ x > y.$

Podstawmy teraz do tezy (7)

zmienne	z	y
za zmienne	y	x

otrzymujemy wynik:
(8) $Jeżeli\ z < y, to\ y > z.$
Z tez (7) i (8) wynika:
(9) $Jeżeli\ (y < x\ oraz\ z < y), to\ (x > y\ oraz\ y > z).$
Z tezy (9) wynika:
(10) $Jeżeli\ (z < y\ oraz\ y < x), to\ (x > y\ oraz\ y > z).$
Z tezy (10) i (3) wynika:
(11) $Jeżeli\ (z < y\ oraz\ y < x), to\ x > z.$
Podstawmy do tezy (4)

zmienną	z
za zmienną	y

otrzymujemy wynik:
(12) $x > z\ wtedy\ i\ tylko\ wtedy, jeżeli\ z < x.$
Z tezy (12) wynika:
(13) $Jeżeli\ x > z, to\ z < x.$
Z tezy (11) i (13) wynika:
(14) $Jeżeli\ (z < y\ oraz\ y < x), to\ z < x.$
Podstawmy do tezy (14)

zmienne	z	x
za zmienne	x	z

otrzymujemy wynik:

(15) $Jeżeli\ (x < y\ oraz\ y < z), to\ x < z.$
[Zauważmy, że teza (15) jest równokształtna i równoznaczna z tezą (1)]. Wnioskowanie doprowadziło nas do zespołu tez (Z_1) (teza (6) i teza (15)). Co prawda, wnioskowanie to oparte było na zespole tez (Z_2) nigdy przedtem niestwierdzonym, ale co z tego? Przecież możemy naprawić nasze niedopatrzenie i w drodze wnioskowania doprowadzić do stwierdzenia zespołu tez (Z_2). Podstawmy do tezy (2)

zmienne	x	y
za zmienne	y	x

otrzymujemy wynik:

(16) $y < x$ *wtedy i tylko wtedy, jeżeli* $x > y$.

Z tezy (16) wynika teza:

(17) $x > y$ *wtedy i tytko wtedy, jeżeli* $y < x$.

(Zauważmy tu, że teza (17) jest równokształtna i zarazem równoznaczna z tezą (4)). Z tezy (2) wynika ponadto:

(18) *Jeżeli* $y > x$, *to* $x < y$.

Podstawmy teraz do tezy (18)

zmienne	z	y
za zmienne	y	x

otrzymujemy wynik:

(19) *Jeżeli* $z > y$, *to* $y < z$.

Z tez (18) i (19) wynika:

(20) *Jeżeli* $(y > x$ *oraz* $z > y)$, *to* $(x < y$ *oraz* $y < z)$.

Z tezy (20) wynika:

(21) *Jeżeli* $(z > y$ *oraz* $y > x)$, *to* $(x < y$ *oraz* $y < z)$.

Z tezy (21) i (1) wynika:

(22) *Jeżeli* $(z > y$ *oraz* $y > x)$, *to* $x < z$.

Podstawmy do tezy (2)

zmienna	z
za zmienną	y

otrzymujemy wynik:

(23) $x < z$ *wtedy i tylko wtedy, jeżeli* $z > x$.

Z tezy (23) wynika:

(24) *Jeżeli* $x < z$, *to* $z > x$.

Z tezy (22) i (24) wynika:

(25) *Jeżeli* $(z > y$ *oraz* $y > x)$, *to* $z > x$.

Podstawmy do tezy (25)

zmienne	z	x
za zmienne	x	z

otrzymujemy wynik:

(26) *Jeżeli* $(x > y$ *oraz* $y > z)$, *to* $x > z$.

[Zauważmy, że teza (26) jest równokształtna i równoznaczna z tezą (3)]. Wnioskowanie doprowadziło, więc nas do zespołu tez (Z_2) (teza (17) i teza (26)). Ten, komu wyjaśniamy, na czym polega błędne koło, uważa je zwykle za tak naiwny błąd, że nie bardzo może uwierzyć, jak w ogóle ludzie mogą taki błąd popełniać. Zazwyczaj - bywa jednak, że ten kto uważa błędne koło za naiwny błąd, sam niejednokrotnie błąd ten popełnia.

Piśmiennictwo: *Ajdukiewicz K.* A.2.1., A.2.2., *Greniewski H.* G.2.1.

3.6.7. Schematy wnioskowania

Dotychczas mówiąc o wnioskowaniu dokonywaliśmy zapisu w postaci wynikającej z zasad używanych w potocznym języku. Począwszy od późnych lat dwudziestych ubiegłego wieku, logicy wprowadzili sformalizowany zapis reguł wnioskowania (dedukcji), nazywany schematami wnioskowania.

3.6.7.10. **Wyjaśnienie**. Każda z reguł wnioskowania, będąca częścią schematu system dedukcji, jest używana do przeprowadzenia dowodu analitycznego. Poszczególne reguły wnioskowania zapisujemy jak niżej:

$$\frac{\textit{założenie1... założenie2}}{\textit{wniosek}}\ [\textit{nazwa reguły || założenia}]$$

wynikający z przyjętych założeń

Lista założeń – bywa czasami pusta. Znaczenie tego rodzaju reguły wynika z faktu, że prawdziwość wniosku – wynika z prawdziwości przyjętych założeń.

Przykładowe dwie reguły wnioskowania, zapisane w postaci schematu dedukcji są *modus ponens i uogólnienie* mają postać jak niżej:

3.6.7.11 $$\frac{\vdash A \longrightarrow B \quad \vdash A}{\vdash B}[\textit{modus ponens}] \qquad \frac{\vdash A(a)}{\vdash \forall x A(x)}\ [\textit{uogólnienie}]$$

Jako przykładu użycia schematu wnioskowania, przeprowadzimy dowód analityczny dobrze znanej tautologii, w której poprzednik implikacji będący koniunkcją pary zmiennych logicznych implikuje trzecią zmienną logiczną, możemy zastąpić podwójną implikacją zachodzącą pomiędzy wzmiankowanymi zmiennymi:

3.6.7.20 $$(p \wedge q \rightarrow r) \rightarrow (p \rightarrow (q \rightarrow r))$$

W celu pokazania jak można przeprowadzić dowód analityczny powyższej tautologii, rozważmy fragment końcowy drzewa dowodu:

3.6.7.21 $$\frac{\vdots}{(p \wedge q \rightarrow r) \rightarrow (p \rightarrow (q \rightarrow r))}$$

Głównymi łącznikami naszej tautologii, jest implikacja, rozważmy jak możemy wyprowadzić wyrażenie znajdujące się po prawej strony dowodzonej implikacji. W tym celu zróbmy założenie 1, iż prawdziwa jest implikacja $p \wedge q \rightarrow r$; zachodząca pomiędzy koniunkcją dwóch zmiennych logicznych, a trzecią zmienną. Umieśćmy to założenie na szczycie drzewa dowodu, a następnie usuwając pierwszy z członów dowodzonej tautologii, korzystając z przyjętego założenia 1, wprowadzamy przedostatni człon drzewa dowodowego:

3.6.7.22 $$\frac{\begin{array}{c}[p \wedge q \rightarrow r]^{[1]} \\ \vdots \\ \hline (p \rightarrow (q \rightarrow r))\end{array}}{(p \wedge q \rightarrow r) \rightarrow (p \rightarrow (q \rightarrow r))}\ [\rightarrow - \text{intro } 1]$$

W wyniku dotychczasowych rozważań, powstaje podwójna implikacja zachodząca pomiędzy trzema zmiennymi logicznymi $p \rightarrow (q \rightarrow r)$. Powtórzmy poprzednio zastosowaną procedurę, przyjmijmy założenie 2, że występuje zmienna logiczna p; otrzymamy w wyniku:

3.6.7.23 $$\frac{\begin{array}{c}\begin{array}{c}[p \wedge q \rightarrow r]^{[1]} \\ [p]^{[2]} \\ \vdots \\ \hline q \rightarrow r\end{array} \\ \hline (p \rightarrow (q \rightarrow r))\end{array}}{(p \wedge q \rightarrow r) \rightarrow (p \rightarrow (q \rightarrow r))}\ [\rightarrow - \text{intro } 1]$$

$[\rightarrow -$ intro 2]

Teraz w kolejnym wyniku pośrednim, otrzymamy implikację $q \to r$. Stosując po raz trzeci tę samą procedurę, przyjmujemy jako założenie 3, że występuje zmienna logiczna q; otrzymamy w wyniku:

$$
\begin{array}{c}
[p \wedge q \to r]^{[1]} \\
[p]^{[2]} \\
[q]^{[3]} \\
\vdots \\
\hline
r \\
\hline\hline
q \to r \quad [\to - \text{ intro } 3] \\
\hline\hline
(p \to (q \to r)) \quad [\to - \text{ intro } 2] \\
\hline\hline
(p \wedge q \to r) \to (p \to (q \to r)) \quad [\to - \text{ intro } 1]
\end{array}
$$

3.6.7.24

W zbudowanym dotychczas obszernym fragmencie drzewa dowodowego, nie mamy już do czynienia ze strukturą wyrażeń logicznych, tylko z pojedynczą zmienną logiczną r. Nastał więc czas, żeby rozpocząć postępowanie wychodząc z przyjętych dotychczas trzech założeń. Dodajmy kolejną linię dowodu, przyjmując że z założenia 1 wyeliminujemy poprzednik implikacji, a w wyniku zostanie nam zmienna logiczna r:

$$
\begin{array}{c}
[p \wedge q \to r]^{[1]} \\
[p]^{[2]} \\
[q]^{[3]} \\
\vdots \\
\hline
[p \wedge q \to r]^{[1]} \quad p \wedge q \\
\hline\hline
r \quad [\to - \text{ elim}] \\
\hline\hline
q \to r \quad [\to - \text{intro } 3] \\
\hline\hline
(p \to (q \to r)) \quad [\to - \text{ intro } 2] \\
\hline\hline
(p \wedge q \to r) \to (p \to (q \to r)) \quad [\to - \text{ intro } 1]
\end{array}
$$

3.6.7.25

Teraz jest już jasne, jak możemy zakończyć nasz dowód. W tym celu wystarczy połączyć przyjęte w założeniach 2 i 3 zmienne logiczne koniunkcją:

$$
\begin{array}{c}
[p]^{[2]} \quad [q]^{[3]} \\
\hline\hline
[p \wedge q \to r]^{[1]} \quad p \wedge q \quad [\wedge - \text{ intro}] \\
\hline\hline
r \quad [\to - \text{ elim}] \\
\hline\hline
q \to r \quad [\to - \text{intro } 3] \\
\hline\hline
(p \to (q \to r)) \quad [\to - \text{ intro } 2] \\
\hline\hline
(p \wedge q \to r) \to (p \to (q \to r)) \quad [\to - \text{ intro } 1]
\end{array}
$$

3.6.7.26

cbdo.

Piśmiennictwo: *Ben-Ari M*. B.2.1., *Pacholski L*. P.1.1.

3.5.8. POJĘCIE ANTYNOMII

3.5.8.00. **Wyjaśnienie**. Mówimy że, zdanie ***B*** jest zaprzeczeniem zdania ***A***, zamiast mówić, że zdanie ***B*** powstaje przez dołączenie funktora „*fałszem jest, że*" (czy też funktora „*nie jest prawdą, że*") do zdania ***A*** .

Przykład. Bierzemy pod uwagę zdania następujące:

(1) Ziemia jest planetą.

(2) *Fałszem jest, że* Ziemia jest planetą.

Zdane (2), jak łatwo zauważyć, jest zaprzeczeniem zdania (1).

3.5.8.01**. Wyjaśnienie**. Mówi my, że zespół tez ***A*** jest *antynomię*, zamiast mówić, że spełnione są oba warunki następujące:

(1) Stwierdzamy, z poczuciem oczywistości zespół tez ***A***;

(2) Istnieje taki zespół tez ***B*** i takie zdanie ***C***, że ***B*** wynika z ***A***, zdanie ***C*** należy do zespołu ***B*** oraz zaprzeczenie zdania ***C*** należy do zespołu ***B***.

Przykład (*antynomii*). Nazwijmy ≪zwykłym≫ wszelki taki zbiór, który nie zawiera siebie samego - jako elementu (takim jest np. zbiór wszystkich liczb całkowitych: jego elementami są liczby, a nie zbiory liczb). Zbiory zawierające siebie - jako element nazwijmy ≪niezwykłymi≫; zbiorem ≪niezwykłym≫ będzie zbiór wszystkich zbiorów, ponieważ z jego definicji wynika, że zawiera on - jako elementy wszystkie zbiory, a zatem także i siebie. Jest rzeczą oczywistą, że każdy zbiór jest albo ≪zwykłym≫, albo ≪niezwykłym≫. Rozpatrzmy zbiór wszystkich zbiorów ≪zwykłych≫. Jeżeli jest on zbiorem ≪zwykłym≫, to zgodnie ze swą definicją musi zawierać siebie samego - jako element (gdyż zawiera wszystkie zbiory ≪zwykłe≫) - a zatem, jest zbiorem≪niezwykłym≫. Zatem przypuszczenie, że zbiór nasz jest zbiorem ≪zwykłym≫, doprowadza do sprzeczności, a więc zbiór ten nie może być ≪zwykłym≫. Z drugiej strony nie może on także być zbiorem ≪niezwykłym≫, gdyż zbiór ≪niezwykły≫ zawiera sam siebie - jako element, a elementami naszego zbioru są tylko zbiory ≪zwykłe≫. Tak, więc - zbiór wszystkich zbiorów ≪zwykłych≫ nie może być ani zbiorem ≪zwykłym≫, ani ≪niezwykłym≫.

Jest to znana, antynomia Bertranda Russella, podaliśmy ją w nieco zmienionym sformułowaniu popularnym wśród matematyków. Będziemy mieli jeszcze okazję w omówić znaczenie powyższej antynomii w rozwoju logiki, a także poznać jeszcze inne antynomie (część 5).

Piśmiennictwo: *Ajdukiewicz K.* A. 2.1., A.2.2., *Greniewski H.* G.2.1. , *Kotarbiński T*. K.5.1., *Kuratowski K*. K.7.1, *Mostowski A.* M.5.1.

3.6.9. NIE UNIWERSALNOŚĆ WNIOSKOWANIA

W rozdziale 3.4 powiedzieliśmy, że nie można zdefiniować wszystkich wyrażeń danego języka bez popełnienia błędnego koła w definiowaniu. Obecnie powiedzmy sobie jeszcze, że nie można w drodze samego wnioskowania uzyskać wszystkich tez naszej wiedzy bez popełnienia błędnego koła w rozumowaniu. Definiowanie nie jest uniwersalnym środkiem budowania wrażeń języka, wnioskowanie zaś nie jest uniwersalnym środkiem budowania naszej wiedzy. W każdym systematycznie ułożonym języku napotykamy w końcu wyrażeni a niezdefiniowane (niezbyt może szczęśliwie zwane „*wyrażeniami pierwotnymi*"). W każdej systematycznie wyłożonej nauce napotykamy tezy stwierdzone, z żadnych innych tez niewywnioskowane (tak zwane „*tezy pierwotne*").

3.6.9.00. **Zasada**. Dla żadnego etapu rozwojowego wiedzy nie-można zbudować takiego zespołu wnioskowań, żeby:

1) Nie zawierał on pośredniego błędnego koła
   i zarazem
2) Uzasadniał każdą tezę na tym etapie uznaną.

I jeszcze jedno - jeśli każda nauka musi się w końcu opierać na tezach pierwotnych, które nie są wywnioskowane z innych już stwierdzonych, to - na czym opierają się prawdziwe tezy pierwotne? Czy są one dowolne, czy umowne, a może stwierdzenie ich, jest jakąś koniecznością naszego umysłu? Źródłem prawdziwych tez pierwotnych jest zawsze - w ostatniej instancji - praktyka, doświadczenie wykonywane w obrębie świata materialnego.

Piśmiennictwo: *Ajdukiewicz K.* A.2.1., A.2.2., *Greniewski H.* G.2.1.

3.7. O GRAFACH, ZBIORACH I RELACJACH

3.7.0. UWAGI WSTĘPNE

Dotychczasowe rozważania w znacznym stopniu dotyczyły klasycznej już dzisiaj tematyki, poprzedzającej wykład logiki formalnej i jedynie odwoływały się do problematyki informatycznej. Niniejszy rozdział poświęcimy problematyce - blisko związanej z informatyką (jak: grafy otwarte, czyli drzewa matematyczne, teoria zbiorów i teorii relacji). Kolejnymi tematami, które omówimy w tym rozdziale są: sekwencje czasowe zdarzeń i ich związki z logiką temporalną, czyli logiką uwzględniającą zależności czasowe.

W żadnym przypadku, nie zdołamy przedstawić tej tematyki w sposób wyczerpujący. Ograniczmy się jedynie, do naszkicowania tych aspektów, które naszym zdaniem są najistotniejsze dla dalszych rozważań dotyczących algorytmów i obliczalności.

Piśmiennictwo: *Cormen T.* C.2.1.

3.7.1. GRAFY A STRUKTURY DANYCH

Graf to – w uproszczeniu – zbiór *wierzchołków*, które mogą być połączone *krawędziami*, w taki sposób, że każda krawędź kończy się i zaczyna w którymś z wierzchołków (ilustracja po prawej stronie). Grafy to podstawowy obiekt rozważań *teorii grafów*. Za pierwszego teoretyka i badacza grafów uważa się *Leonarda Eulera*, który rozstrzygnął *zagadnienie mostów w Królewcu*.

Wierzchołki grafu zwykle są numerowane i czasem stanowią reprezentację jakichś obiektów, natomiast krawędzie mogą wówczas obrazować relacje między takimi obiektami. Krawędzie mogą mieć wyznaczony kierunek, a graf zawierający takie krawędzie jest *grafem skierowanym*. Krawędź może posiadać także wagę, to znaczy przypisaną liczbę, która określa na przykład odległość między wierzchołkami. W grafie skierowanym wagi mogą być zależne od kierunku przechodzenia przez krawędź (przykładowo, jeśli graf reprezentuje trud poruszania się po jakimś terenie, to droga pod górkę będzie miała przypisaną większą wagę niż z górki).

Graf, graf prosty lub graf nieskierowany to *uporządkowana para* $G \coloneqq (V, E)$ gdzie:

- V- jest *niepustym zbiorem*. Elementy tego zbioru nazywamy *wierzchołkami*,
- E- jest rodziną dwuelementowych podzbiorów zbioru wierzchołków V, zwanych *krawędziami*: $E \subseteq \{\{u, v\}: u, v \in V, u \neq v\}$.[55]

[55] Notacja - patrz podrozdziały: następny 3.7.2. oraz 3.7.3.

Wierzchołki należące do krawędzi nazywane są jej *końcami*. Zazwyczaj V (i co za tym idzie E) są określane, jako zbiory skończone. Jednak powyższa definicja tego nie wymaga i w praktyce rozważa się też czasami grafy o nieskończonej liczbie wierzchołków (wtedy liczba krawędzi może być skończona, lub nieskończona). Grafy można podzielić ze względu na różne własności, zazwyczaj zachowane w obrębie izomorfizmów danego grafu. Najczęściej dotyczą one tylko grafów prostych (nie zawierających pętli i krawędzi wielokrotnych). Nas szczególnie będzie interesować spójny graf acykliczny – zwany drzewem. Mówiąc językiem obrazowym, z każdego wierzchołka drzewa można dotrzeć do każdego innego wierzchołka (spójność) i tylko jednym sposobem (acykliczność, czyli brak możliwości chodzenia w "kółko").

Graf prosty G jest drzewem jedynie, jeśli spełnia jeden z warunków:
- Dowolne dwa wierzchołki łączy dokładnie jedna ścieżka prosta;
- G jest acykliczny i dodanie krawędzi łączącej dowolne dwa wierzchołki utworzy cykl;
- G jest spójny i usunięcie dowolnej krawędzi spowoduje, że G przestanie być spójny.

Drzewo, w którym jest wyróżniony jeden z wierzchołków nazywamy drzewem ukorzenionym, a wyróżniony wierzchołek - korzeniem. Na takim drzewie możemy również określić relacje „rodzinne" pomiędzy wierzchołkami. Dla dowolnej ścieżki prostej rozpoczynającej się od korzenia i zawierającej wierzchołek v:
- Wierzchołki występujące w ścieżce przed v nazywamy jego przodkami, a wierzchołki występujące po v–potomkami;
- Wierzchołek bezpośrednio przed v nazywamy rodzicem lub ojcem, a bezpośrednio po - dzieckiem lub synem;
- Wierzchołki mające wspólnego ojca nazywamy braćmi;
- Wierzchołki, które nie mają synów nazywamy liśćmi drzewa.

Najdłuższą ścieżkę w drzewie nazywamy *średnicą drzewa*. Jej długość liczymy stosując metody programowania dynamicznego. W informatyce bardzo często przyjmuje się, że synowie tworzą nie zbiór, lecz listę[56] uporządkowaną. Taki twór, co prawda nie jest matematycznie grafem, jednak jest równoważny grafowi i ma ogromne znaczenie praktyczne w dziedzinie informatyki.

Podstawowe operacje na drzewach to:
- Wyliczenie wszystkich elementów drzewa;
- Wyszukanie konkretnego elementu;
- Dodanie nowego elementu w określonym miejscu drzewa;
- Usunięcie elementu.

W informatyce wiele *struktur danych* jest konkretną realizacją drzewa matematycznego. Wierzchołki drzewa reprezentują konkretne dane (liczby, napisy albo bardziej złożone struktury danych). Odpowiednie *ułożenie* danych w drzewie może ułatwić i przyspieszyć ich wyszukiwanie. Znaczenie tych struktur jest bardzo duże i ze względu na swoje własności drzewa są stosowane praktycznie w każdej dziedzinie informatyki (np. algorytmika, kryptografia, bazy danych, grafika komputerowa, przetwarzanie tekstu, telekomunikacja).

[56] Lista – to np. twór będący zbiorem par uporządkowanych (lista jednokierunkowa), albo zbiorem trójek uporządkowanych. W pierwszym przypadku: pierwszym elementem pary jest wierzchołek drzewa, zaś drugim elementem jest adres następnej pary uporządkowanej. W drugim przypadku; pierwszym elementem jest adres poprzedzającej trójki uporządkowanej, drugim elementem pary jest wierzchołek drzewa, zaś trzecim elementem jest adres następnej trójki uporządkowanej.

Specjalne znaczenie w informatyce mają *drzewa binarne* (liczba dzieci ograniczona do dwóch) i ich różne odmiany, np. *drzewa AVL, drzewa czerwono-czarne - BST*. Drzewa, które posiadają więcej niż dwoje dzieci są nazywane *drzewami wyższych rzędów*.

Piśmiennictwo: *Cormen T.* C.2.1., *Ross K.* R.1.1.

3.7.2. Zbiory

Pewne idee matematyczne, mimo ich pozornej prostoty, stanowią praktycznie niewyczerpalne zbiorniki pojęć, zasilających różnorodne koncepcje. Teoria zbiorów (zwana w Polsce - teorią mnogości), jest jedną z takich idei. Podstawy teorii mnogości stworzył *Georg Cantor* (1845 – 1918), nazywanej dzisiaj prymitywną teorią zbiorów. Teoria zbiorów jest częścią fundamentów nowoczesnych teorii matematycznych, dostarczając język i symbolikę umożliwiającą syntezę nowych i starych idei, sprawdzenie znanych wcześniej koncepcji i spojrzenie na dalszy rozwój. W 1902 roku, filozof brytyjski *Bertrand Russsel* (1872 - 1930) zauważył, że teoria mnogości jest sprzeczna, bowiem prowadzi do antynomii (patrz podrozdział 3.6.8.). Trzeba zauważyć, że twórca współczesnej logiki, niemiecki matematyk *Gottlob Frege* (1848 - 1925) uważał, że matematyka powinna być oparta na pojęciu zbioru, a nie na pojęciach logiki. *Frege* stworzył aksjomatyczny system logiki, ale przez współczesnych nie był doceniony. Dopiero w latach trzydziestych XX wieku nastąpił gwałtowny rozwój logiki, przy zasadniczym udziale polskiej szkoły logiki.

Sprawa poprawienia teorii mnogości, poprzez wyeliminowanie antynomii, stała się istotnym problemem dla podstaw matematyki. Twórcami zmodernizowanej wersji teorii mnogości z typami, są *Ernst Zermelo* i *Abraham Fränkel* (teoria ZF). Uczeni ci wzorując się na fizykalnym opisie rzeczywistości materialnej, uniknęli występowania szeregu antynomii (sprzeczności), w szczególności *antynomii Russela*, zakładając, że do zbioru mogą należeć tylko elementy o tym samym typie.

Jak już zostało powiedziane, każdy element danego zbioru - posiada jednoznacznie określony typ. W przypadku np. elementów posiadających interpretację fizyczną, typem jest jednostka miary służącej do mierzenia danego elementu lub zbiorowości elementów. Zbiór w sensie teorii *Zermelo – Fränkela*, jest dobrze zdefiniowaną kolekcją elementów jednego typu. „Dobrze zdefiniowana" oznacza w tym przypadku, że istnieje możliwość stwierdzenia, czy dany element należy do danego zbioru, czy też nie. Zwykle przyjmuje się, że zbiory oznaczamy dużymi literami alfabetu lub nazwami zaczynającymi się od dużej litery alfabetu. Przykładowe nazwy zbiorów: $A, B, Z, DYSK, LAMPA$. Elementy zbiorów oznaczamy małymi literami lub nazwami zaczynającymi się od małych liter. Przykładowe nazwy elementów: $a, b, c, x, y, z, alfa, delta$. Zbiór może być również kolekcją zbiorów, pod warunkiem jednak, że wszystkie zbiory wchodzące w skład kolekcji są tego samego typu.

3.7.2.01. **Definicja**. Zbiór w rozumieniu teorii mnogości – to grupa elementów traktowanych, jako całość. Zbiory mogą się składać z elementów jednego typu (czyli elementów, posiadających wspólną jednostkę miary), np. liczb, symboli albo innych zbiorów. Elementy składające się na dany zbiór nazywamy składnikami tego zbioru. Zbiory można opisywać formalnie na kilka sposobów. Jednym z opisów zbiorów o skończonej liczbie elementów, to wypisanie w nawiasach klamrowych wszystkich elementów zbioru. Jak np. zbiór, który zawiera sześć elementów: 2, 3, 5, 7, 11, 13:

$$\{2, 3, 5, 7, 11, 13\}$$

Symbole $\in$ oraz $\notin$ oznaczają odpowiednio przynależność do zbioru oraz brak przynależności do danego zbioru.

3.7.2.02. **Definicja**. *Universum* (z łaciny ogół, wszystko, wszechświat) – pojęcie w matematyce oznaczające klasę wszystkich elementów danego modelu matematycznego. *Universum* modelu teorii mnogości stanowią wszystkie zbiory.

3.7.2.03. **Definicja**. Zbiór pusty - to zbiór, który nie zawiera ani jednego elementu. Zbiór pusty oznaczamy symbolem $\emptyset$.

3.7.2.11. **Definicja**. Gdy mamy dwa zbiory A i B - złożone z elementów tego samego typu, to mówimy, że zbiór A jest podzbiorem zbioru B, co zapisujemy $A \subseteq B$, jeżeli każdy element zbioru A, jest także elementem zbioru B. Mówimy, że zbiór A, jest właściwym podzbiorem zbioru B, co zapisujemy $A \subset B$, gdy zbiór A jest podzbiorem zbioru B i nie jest równy zbiorowi B.

3.7.2.12. **Wyjaśnienie**. Kolejność, w jakiej występują elementy zbioru, niema znaczenia, podobnie jak nieistotne są powtórzenia elementów. Jeśli chcemy uwzględnić liczbę wystąpień elementu w zbiorze, to mamy do czynienie z innym tworem niż zbiór, a mianowicie z tak zwanym zbiorowiskiem. Tak więc {7, 7} i {7} oznaczają dwa różne zbiorowiska, ale identyczne zbiory.

3.7.2.13. **Definicja**. Zbiór poza-skończony zawiera nieskończenie wiele elementów. Definiując zbiór nieskończony nie można wypisać wszystkich elementów zbioru, więc czasem używamy znaku wielokropka „ … ”, by zaznaczyć, że ciąg elementów ma być kontynuowany w podany sposób „w nieskończoność”. Np. zbiór liczb naturalnych $\mathbb{N}$ możemy zapisać jako:

$$\{1, 2, 3, \ldots\}.$$

Inną, formalnie poprawną formą zapisu zbiorów poza-skończonych, jest zapis oparty o wykorzystanie predykatu określającego spełnianie warunku przynależności elementu x do zbioru$A$. Uogólniony opis analityczny zbioru wygląda następująco:

$$A = \{x : X \mid P(x) \bullet \ jawna\ postać\ elementu \ \in \ zbioru\ A\};$$

gdzie $P(x)$ *predykat*[57] narzucający ograniczenia na zmienną, do którego podstawiono element $x \in X$ (tzw. *dziedzinę zmiennej* x).

3.7.2.21. **Definicja**. Mając dwa zbiory A i B - złożone z elementów tego samego typu, możemy mówić o sumie zbiorów; oznaczając ją $A \cup B$, czyli o zbiorze, który otrzymamy łącząc elementy obu zbiorów w jeden zbiór wynikowy.

3.7.2.22. **Definicja**. Część wspólna dwu zbiorów A i B - złożone z elementów tego samego typu, nazywamy iloczynem skalarnym (przekrojem) zbiorów; oznaczając ją $A \cap B$, czyli tworzymy zbiór z elementów, które równocześnie należą zarówno do zbioru A, jak i zbioru B.

3.7.2.31. **Definicja**. Iloczyn kartezjański dwu zbiorów A i B, oznaczamy $A \times B$, jest to zbiór wszystkich par, których pierwszy element pochodzi ze zbioru A, a drugi ze zbioru B. Elementy poszczególnych zbiorów argumentów iloczynu kartezjańskiego mogą różnych typów. Przykłady: Dla zbiorów A={1, 2} i B={x, y, z} mamy $A \times B$={(1, x), (1, y), (1, z), (2, x), (2, y), (2, z)}; Zbiór $\mathbb{N}^2$ to iloczyn kartezjański zbiorów $\mathbb{N} \times \mathbb{N}$ i składa się ze wszystkich par liczb naturalnych. Można go zapisać jako $\{(i, j) \bullet i, j \ \leq 1\}$.

[57] *Predykat* – funkcja przyjmująca wartości 0 albo 1 – w zależności od wyniku testu wartości argumentu/ów (patrz 4.5.5).

3.7.2.32. **Definicja**. Zbiór potęgowy zbioru A, to zbiór wszystkich podzbiorów zbioru A. Przykład: dla zbioru A={0, 1} zbiór potęgowy A - to zbiór {∅, {0}, {1}, {0, 1}}.

3.7.2.40. **Wyjaśnienie**. Każdy element należący do danego zbioru ma swój typ. Np. zbiór złożony z liczb całkowitych $\{1, 2, \ldots\}$, którego wszystkie elementy są typu $\mathbb{Z}$ (gdzie przez ten symbol oznaczony typ wszystkich liczb), a zbiór liczb całkowitych jest również typu $\mathbb{Z}$. Innym przykładem jest zbiór kolorów - typu COLOR. Natomiast elementy zbioru kolorów, np.: *czerwony*, *żółty*, *zielony* mają typ COLOR.

Istnieją jednak obszary zastosowań, w których teoria zbiorów z typami, jest niestety niewystarczająca. Takim obszarem jest np. mechanika kwantowa. W 1965 roku *Lotfi Zadeh* – profesor Uniwersytetu w Berkley zaproponował inne pojęcie zbioru, w którym elementy mogą należeć do zbioru z pewnym prawdopodobieństwem, ale nie definitywnie, jak to ma miejsce w teorii zbiorów Zermelo – Fränkela. Propozycję tą nazwano *teorie zbiorów rozmytych* (*fuzzy set theory*) i znalazła ona szereg zastosowań, ale również wywołała falę krytyki – wśród matematyków.

Kolejne rozszerzenie teorii mnogości – tzw. *teoria zbiorów przybliżonych*, zawdzięczamy *Zdzisławowi Pawlakowi* (1982). Wersji teorii mnogości z typami, którą zawdzięczamy *Zermelo* i *Fränkelowi*, podobnie jak wcześniejsza wersja zawdzięczana *Cantorowi*, zakłada ostry podział elementów *Universum* – na należące do danego zbioru i nienależące do danego zbioru. Jest to podział odpowiadający logice dwuwartościowej. *Pawlak* zwrócił uwagę, na fakt trudności, na jakie napotyka matematyka operując jedynie ostrym podziałem na przynależność lub brak przynależności do danej klasy. Np. pewne kobiety są uznawane za piękne, inne zaś nie. Istnieją również kobiety, o których urodzie zdania są podzielone. W tym miejscu zawodzi podział dwuwartościowy. *Pawlak* zaproponował wersję teorii mnogości opartą o logikę trójwartościową. W tym celu *Pawlak* wprowadził pojęcie *zbioru przybliżonego*.

3.7.2.51. **Definicja**. *Zbiór przybliżony* jest ograniczony parą zbiorów ostrych (w rozumieniu teorii ZF), zwanych granicami dolną i górną zbioru przybliżonego. Jeśli jednak zbiory tworzące granice dolną i górną zbioru przybliżonego pokrywają się (są identyczne), to wówczas zbiór przybliżony nie istnieje i mamy do czynienia z klasycznym zbiorem teorii ZF.

3.7.2.52. **Definicja**. Podstawową ideą zbiorów przybliżonych jest zastąpienie każdego pojęcia nieostrego parą pojęć ostrych, zwanych górnym i dolnym przybliżeniem tegoż pojęcia. Różnica pomiędzy granicami górną i dolną zbioru przybliżonego jest *obszarem brzegowym*, do którego należą wszystkie elementy, które nie mogą być prawidłowo zaklasyfikowane do jednej z obu granic. Im większy jest obszar brzegowy, czyli im więcej istnieje przypadków, których nie możemy zakwalifikować do granic dolnej lub górnej zbioru przybliżonego, z tym bardziej nieostrym zbiorem przybliżonym mamy do czynienia.

Teoria zbiorów przybliżonych znalazła już i dalej znajduje szereg zastosowań. Z pomocą zbiorów przybliżonych określono działania przydatne do analizy niepewnych danych. Przykładowo - dane pacjentów klinik są zapisywane, jako zbiory pewnych cech, takich jak wiek, płeć, temperatura, ciśnienie krwi, wyniki badań analitycznych i biologicznych, określonych objawów itp. Lekarze są zainteresowani wyodrębnieniem tych cech, które zawsze występują przy danej jednostce chorobowej, od tych cech występujących czasami oraz cech występujących sporadycznie. Dotyczy to takich chorób jak: leczenie wrzodów dwunastnicy, ostrego zapalenia trzustki itd.

Przykładami innych zastosowań teorii zbiorów przybliżonych jest farmakologia (badanie związków pomiędzy strukturą leku a jego aktywnością mikrobiologiczną), meteorologia (analiza czynników powodujących zmiany klimatyczne), analiza wibracji urządzeń i mechanizmów (przemysł lotniczy i kosmiczny), jak również badanie zjawisk zachodzących w świecie finansów i rachunkowości. Teoria zbiorów przybliżonych znajduje zastosowanie przy rozpoznawaniu głosu, analizy obrazów, rozpoznawania ręcznego pisma. *Peter Apostoli* i *Akiro Kanda* profesorowie Uniwersytetu w Toronto, opublikowali 2002 roku pracę, w której twierdzą, że teoria zbiorów przybliżonych uwalnia teorię mnogości *Cantora* od sprzeczności.

Piśmiennictwo: *Kuratowski K.* K.7.1., *Pawlak Z.* P.2.2., P.2.3.

3.7.3. CIĄGI, KROTKI, RELACJE I ZWIĄZKI RELACJI

3.7.3.11. **Definicja**. Ciąg obiektów, to lista obiektów przedstawiona w określonym porządku. Ciąg zapisujemy zwykle, jako listę ujętą w nawiasy okrągłe. W zbiorze kolejność elementów nie ma znaczenia, ale w ciągu jest istotna.
Przykład: ciąg trzy liczbowy 7, 13, 19 – zapisujemy, jako (7, 13, 19).

3.7.3.12. **Wyjaśnienie**. Podobnie jak w przypadku zbiorów, ciągi mogą być skończone oraz nieskończone.

3.7.3.20. **Definicja**. Ciągi skończone często nazywamy krotkami. Ciąg złożony z k elementów nazywamy k-krotką. Ciąg dwuelementowy (2-krotkę), nazywamy parą uporządkowaną. Przykład: ciąg (7, 13, 19) jest 3-krotką.

3.7.3.21. **Wyjaśnienie**. Iloczyn kartezjański kolejnych k-zbiorów, daje w wyniku zbiór k-krotek.

Podstawowymi jednostkami w wielu działach matematyki są relacje. Relacje wyrażają własności i warunki łączenia rozważanych obiektów. Przykładami relacji, o których już mówiliśmy, są relacje rodzinne. Innym przykładem są relacje przynależności, czy też leżenia (w rozumieniu, że dane miasto leży nad określoną rzeką). O funkcjach, pod kątem potrzeb logiki formalnej, mówiliśmy w paragrafie 3.3.2. Pojęcie funkcji i w kolejnych podrozdziałach. Korzystając z terminologii teorii zbiorów, funkcję $y = f(x)$ możemy utożsamić ze zbiorem:

$$R_{\mathrm{f}} = \{(x, y) \in S \times T \bullet y = f(x)\}.$$

Jest to relacja, określona na zbiorze $S \times T$. Oczywiście nie wszystkie relacje są funkcjami. Jeżeli traktujemy funkcję, jako relację to funkcja odwzorowująca zbiór S w zbiór T jest szczególnym przypadkiem relacji R na zbiorze $S \times T$, mianowicie jest relacją taką, że dla każdego $x \in S$ istnieje dokładnie jeden $y \in T$ taki, że $(x, y) \in R$. Zatem funkcje są to relacje, dla których ma sens zapis funkcyjny: $f(x)$ jest jedynym elementem zbioru T takim, że para $(x, f(x))$ należy do R_{f}. Z punktu widzenia informatyki, szczególną rolę odgrywa teoria relacji - rozwinięta w latach siedemdziesiątych XX wieku dla potrzeb baz danych. Relacyjne bazy danych, których idea oparta została przez *Edgara Franka Codda*[58] na mnogościowej teorii relacji (mająca swoje źródła w pracach *Cantora*), składa się z wzajemnie powiązanych, tzw. związkami (*relationships*) tabel – zwanych relacjami (*relations*), zbudowanych z kolei z k-krotek (*tuples*).

[58] Założenia te zostały opublikowane po raz pierwszy w 1970 roku w pracy *A Relational Model of Data for Large Shared Data Banks.* Praca ta opisuje podstawowe zależności, jakie mogą występować pomiędzy danymi trwałymi, oraz wprowadza główne założenia dotyczące modelu relacyjnego dla danych wraz z propozycją formalnych operatorów przeszukiwania danych. W 1972 roku, w pracy pt. *Relational Completeness of Data Base Sublanguages* - Codd uszczegółowił opis modelu oraz przedstawił dwa modele formalne odpytywania (przeszukiwania) danych. Tu właśnie po raz pierwszy pojawiły się terminy algebra relacji oraz rachunek relacyjny. Codd pokazał, że oba modele są równoważne.

3.7.3.30. **Definicja**. Relacja (*relation*) w rozumieniu *Codda*, to tabela o n – kolumnach i m – wierszach, gdzie poszczególne kolumny tabeli nazywamy atrybutami (*attributes*), z których każdy ma unikalną nazwę, zaś kolumny są k-krotkami (*tuples*), przy czym k jest liczbą atrybutów. Relacja spełnia następujące warunki:

1. Elementami relacji znajdującymi się na przecięciu kolumny i wiersza relacji pojedynczymi nazwami lub liczbami (czy nie są tablicami lub powtarzającymi się grupami znaków);
2. Elementy pojedynczej kolumny są tego samego rodzaju;
3. Każda kolumna relacji ma unikalną nazwę, będącą nazwą atrybutu danej relacji (przy czym nazwy kolumn w różnych relacjach mogą się powtarzać);
4. Żadne dwie kolumny relacji nie mogą być identyczne.

Tabela 3.7.3.31

Identyfikator	Nazwisko	Imię	Wydział	Stanowisko
20451	Kowalski	Jan	Remontowy	Ślusarz
23245	Adamczyk	Tomasz	Obsługi klientów	Referent
24300	Nowak	Jerzy	Remontowy	Spawacz
31202	Maliniak	Bożena	Rachunkowości	Księgowa

Przykład**:** tabela 3.7.3.31, zawiera przykład relacji w rozumieniu *Codda*, gdzie klucz główny = Identyfikator, zaś pozostałe atrybuty to: Nazwisko, Imię, Wydział, Stanowisko.

3.7.3.32. **Wyjaśnienie**. Relacyjny model baz danych opracowany przez *Codda* posługuje się własną terminologią, a mianowicie:

1. K-krotka (*tuple*) – wiersz relacji;
2. Liczebność (*cardinality*) – liczba wierszy relacji;
3. Atrybut (*attribute*) – kolumna relacji;
4. Domena (*domain*) – zbiór wartości, jaki może przebiegać atrybut relacji;
5. Stopień (*degree*) – Liczba atrybutów w relacji;
6. Kandydat na klucz (*candidate to key*) – atrybut lub kilka atrybutów unikalnie identyfikujących k-krotki;
7. Klucz główny (*primary key*) – wybrany kandydat na klucz;
8. Obcy klucz (*foreign key*) – atrybut danej relacji przebiegający tę samą domenę, co klucz główny innej relacji;
9. Relacja jednostkowa (*unary relation*) – relacja z jednym tylko atrybutem;
10. Relacja binarna (*binary relation*) – relacja z dwoma atrybutami;
11. Relacja bazowa (*base relation*) – relacja liniowo niezależna od pozostałych relacji danej bazy danych;
12. Relacja pochodna (*derive drelation*) – relacja, która została utworzona na podstawie relacji bazowych (zwana również *view*).

3.7.3.33. **Definicja**. Związek relacji 1 do 1 (w rozumieniu *Codda*), to związek między parą relacji, zwanych odpowiednio *relacją bazową* i *relacją pochodną*, polegający na tym, że jednym z atrybutów w *relacji bazowej* jest *obcy klucz*, którego wartości odpowiadają poszczególnym wartościom *klucza głównego relacji pochodnej*. Przykład: tabela 3.6.3.34, zawiera przykładowy związek relacji 1 do 1 w rozumieniu *Codda*.

Tabela 3.7.3.34

Dzieci na utrzymaniu			
Klucz główny	Obcy klucz	Imię dziecka	Rok
1021	20451	Piotr	2003
1022	23245	Paweł	2007
1045	24300	Anna	2009
1100	31202	Małgorzata	2001

Pracownicy			
Klucz główny	Nazwisko	Imię	Wydział
20451	Kowalski	Jan	Remontowy
23245	Adamczyk	Tomasz	Obsługi
24300	Nowak	Jerzy	Remontowy
31202	Maliniak	Bożena	Rachunkowości

3.7.3.35. **Definicja**. Związek relacji 1 do N (w rozumieniu *Codda*), to to związek między parą relacji, zwanych odpowiednio *relacją bazową* i *relacją pochodną*, polegający na tym, że jednym z atrybutów w *relacji bazowej* jest *obcy klucz*, którego wartości odpowiadają poszczególnym wartościom, co najmniej *dwu atrybutowego klucza głównego relacji pochodnej*.
Przykład: tabela 3.6.3.36, zawiera przykładowy związek relacji 1 do N w rozumieniu Codda.

Tabela 3.7.3.36

Dzieci na utrzymaniu			
Klucz główny	Obcy klucz	Imię dziecka	Rok
1021	23245	Piotr	2003
1022	23245	Paweł	2007
1045	23245	Anna	2009
1100	31202	Małgorzata	2001

Pracownicy			
Klucz główny	Nazwisko	Imię	Wydział
20451	Kowalski	Jan	Remontowy
23245	Adamczyk	Tomasz	Obsługi
24300	Nowak	Jerzy	Remontowy
31202	Maliniak	Bożena	Rachunkowości

3.7.3.41. **Wyjaśnienie**. Przez *integralność relacji* ze względu na jej *klucz główny* (wg *Codda*) rozumiemy zapewnienie brak powtórzeń *wartości klucza głównego* - w obrębie każdej relacji należącej do zestawy relacji tworzących bazę danych.

3.7.3.42. **Wyjaśnienie**. Przez *integralność referencyjną relacji* ze względu na jej *obce klucze* (wg *Codda*) rozumiemy zapewnienie istnienia w relacji, do której odwołuje się *dany klucz obcy* krotki o danej wartości *klucza głównego* - w obrębie każdej relacji należącej do zestawy relacji tworzących bazę danych.

Piśmiennictwo:. *Codd E.* C.4.1., *Cormen T.* C.5.1., *Jakubowski A.* J.2.1.

3.7.4. Sekwencje (czasowe)

3.7.4.00. **Definicja**. Przez czas dyskretny rozumiemy czas podzielony na odcinki, które często są nazywane kwantami czasu, lub precyzyjniej *chronami*, w odróżnieniu od tzw. czasu ciągłego, typowego np. dla fizyki klasycznej.

3.7.4.10. **Wyjaśnienie**. Długość *chronu* (kwantu czasu) jest sprawą umowną i zależy głównie od rozważanego modelu opisu rzeczywistości materialnej.
Przykład: dla potrzeb opisu działania procesora mówimy o *chronach* odpowiadających długości impulsów zegarowych (przypadek *synchroniczny* – równych długości trwania chronów) lub przy działaniu *asynchronicznym* – sterowanie zdarzeniami (przypadek *asynchroniczny* – zmienna długość trwania chronów), natomiast dla potrzeb kalendarza mówimy o kwantach czasowych odpowiadających poszczególnym dniom, itp.

3.7.4.20. **Definicja**. Sekwencja to ciąg o okręconej nazwie, którego kolejne elementy tworzą listę wartości funkcji w kolejnych - wybranych kwantach czasu, ujęty w nawiasy trójkątne ⟨⟩.
Przykład**:** $workday = \langle monday, tuesday, wednesday, thursday, friday \rangle$.

Sekwencje są ściśle związane z opisami układów (czy systemów) w czasie, czyli pokazywaniem zmian stanów układu wraz z upływem czasu. Omawiając dalej funkcjonalność automatów skończonych oraz maszyn *Turinga*, będziemy mieli do czynienia z sekwencjami stanów tych systemów. Wracając do problematyki logiki formalnej, należy zauważyć, że problematyka formalnego ujęcia czasu była interesująca dla filozofów już w starożytności (zmiana w czasie, determinizm), jednakże, w okresie Średniowiecza problematyka ta traktowana była marginalnie, pewne ślady można znaleźć przykładowo w pracach świętego *Augustyna* i *Ockhama*.

Piśmiennictwo: *Cormen T.* C.5.1., *Hyde R.* H.5.1,

3.8. O ALGORYTMACH I OBLICZALNOŚCI

3.8.0. UWAGI WSTĘPNE

W kolejnych rozdziałach części trzeciej, spróbujemy nieco uporządkować wiedzę o podstawach współczesnej informatyki. Dotychczas mówiliśmy o językach, teraz skoncentrujmy się na rozważaniach dotyczących podstaw informatyki, w rozumieniu współczesnym. Jednym z istotnych pojęć informatyki, jest *algorytm*. Zastanowimy się skąd się wzięło i kiedy zostało sformułowane pojęcie *algorytmu*. Nie chodzi tu o sam termin *algorytm*, który to termin jest znacznie starszy od współczesnego rozumienia pojęcia *algorytmu*. W Europie wczesnego Średniowiecza *termin algorytm* – nie był używany. Termin ten pojawił się gdzieś na przełomie X i XI wielu, razem z tzw. liczbami arabskimi. Przez kilka wieków termin algorytm - oznaczał opis wykonywanie czterech działań arytmetycznych na *liczb arabskich* (w odróżnieniu od *abacism* - przy pomocy *abakusa*) i pochodzi od nazwiska, które nosił *Muhammad ibn Musa al-Chuwarizmi* (أبو عبد الله محمد بن موسى الخوارزمي), matematyk perski z IX wieku. W kolejnych wiekach następowało wzbogacanie zakresu pojęcia *algorytmu*. Ale dopiero w XX wieku, nabrał on swego dzisiejszego znaczenia. W czasach nowożytnych, algorytm oznaczał metody obliczeniowe inżynierii budowlanej, w szczególności inżynierii budowli obronnych. Równolegle do kształtowania się terminu *algorytm*, nabierał znaczenia nowy termin *obliczalność*. Na przełomie XIX i XX wieku, nie było jeszcze pojęcia *algorytmu* - we współczesnym rozumienia tego terminu, posługiwano się za to pojęciami metody mechanicznej, efektywnej procedury, itp.

Mimo braku sprecyzowania pojęcia *algorytmu*, w starożytnych pracach matematycznych można znaleźć wiele opisów algorytmów dla różnych zadań, takich jak znajdowanie liczb pierwszych czy największego wspólnego dzielnika. Nikt jednak nie nazywał tego algorytmami. Spotykane wcześniej algorytmy były nazwane procedurami lub przepisami.

Piśmiennictwo: *Banachowski L.* B.1.1., *Cormen T.* C.2.1., *Ross K.* R.1.1., *Sipser M.* S.7.1., *Wirth N.* W.5.1.

3.8.1. SKĄD SIĘ WZIĘŁO WSPÓŁCZESNE POJĘCIE ALGORYTMU?

W roku 1900, podczas drugiego Światowego Kongresu Matematycznego odbywającego się w Paryżu, *Dawid Hilbert* ogłosił problemy, w których wskazał kierunki dalszego rozwoju matematyki w XX wieku. Łącznie *Hilbert* przedstawił dwadzieścia trzy problemy matematyczne, które uznał za wyzwanie na nadchodzący wiek. Dziesiąty problem na liście *Hilberta* dotyczył algorytmów. Zanim przedstawimy ten problem, omówmy krótko wielomiany. Wielomian to suma jednomianów, gdzie jednomian to iloczyn pewnych zmiennych i stałej nazywanej współczynnikiem. Na przykład:

$$6 \cdot x \cdot x \cdot x \cdot y \cdot z \cdot z = 6x^3yz^2$$

jest jednomianem ze współczynnikiem 6, a

$$6x^3yz^2+3xy^2-x^3-10$$

jest wielomianem z czterema jednomianami i zmiennymi x, y oraz z. Będziemy rozważać tylko wielomiany ze współczynnikami będącymi liczbami całkowitymi. *Pierwiastkiem wielomianu* jest takie przypisanie wartości zmiennym, przy którym *wartość wielomianu wynosi zero*. Przedstawiony wyżej wielomian ma pierwiastek dla x = 5, y = 3 oraz z = 0. Jest to pierwiastek całkowity, gdyż wszystkie zmienne mają w nim przypisane wartości całkowite. Niektóre wielomiany mają pierwiastki całkowite, a inne - nie.

Jednym z tych problemów, było zadanie znalezienia metody rozstrzygania w skończonej liczbie kroków istnienia rozwiązania równań diofantycznych[59]. Dziesiąty problem Hilberta polegał na znalezieniu algorytmu, który sprawdziłby, czy wielomian ma pierwiastki całkowite. Hilbert nie użył słowa algorytm, mówił raczej o „*procesie opisanym za pomocą skończonej liczby operacji*". Interesujące jest, ze w swoim oryginalnym sformułowaniu problemu Hilbert mówi o „znalezieniu" algorytmu. Najwyraźniej więc zakłada, że taki algorytm istnieje - ktoś musi go jedynie odnaleźć. Dziś wiemy, że dla tego zadania nie istnieje żaden algorytm - jest on *algorytmicznie nierozwiązywalny*. Dla matematyków z tego okresu dojście do takiego wniosku przy intuicyjnym pojęciu algorytmu było praktycznie niemożliwe. Intuicyjne rozumienie pojęcia algorytmu może być dobre przy opisywaniu algorytmów dla wybranych zagadnień, ale jest bezużyteczne przy próbie wykazania, że dla określonego zadania nie istnieje żaden algorytm. Dowód nieistnienia algorytmu wymaga klarownej jego definicji. Rozwiązanie dziesiątego problemu Hilberta musiało poczekać, aż taka definicja się pojawiła.

A definicja algorytmu pojawiła się dopiero w 1936 roku, w pracach *Alonso Churcha* i *Alana Turinga*. *Church* wykorzystał notację nazywaną rachunkiem λ. Do zdefiniowania algorytmu. *Turing* uczynił to za pomocą „swoich maszyn". Później pokazano, że obie definicje są równoważne. Powiązanie nieformalnego pojęcia algorytmu z precyzyjną definicją nazwano *tezą Churcha-Turinga*. Teza Churcha-Turinga dostarcza definicję algorytmu potrzebną do rozwiązania dziesiątego problemu Hilberta. W 1970 roku *Yuri Matijasević*, opierając się na pracy *Martina Davisa*, *Hilarego Putnama* oraz *Julii Robinson*, pokazał, że nie istnieje algorytm sprawdzający, czy wielomian ma całkowite pierwiastki.

Wyraźmy dziesiąty problem *Hilberta*, używając naszych pojęć. Pomoże nam to wprowadzić zagadnienia, którymi dalej zajmiemy się. Niech:

D = { p | p jest wielomianem z całkowitym pierwiastkiem}.

Zasadnicze pytanie w dziesiątym problemie *Hilberta* dotyczy rozstrzygalności zbioru D. Odpowiedz jest negatywna.

3.8.1.10. **Wyjaśnienie**. Według dzisiejszego rozumienia tego pojęcia: algorytm – w *matematyce* oraz *informatyce* to skończony ciąg jednoznacznie zdefiniowanych czynności, koniecznych do wykonania pewnego rodzaju zadań. Algorytm we współczesnym rozumieniu, ma przeprowadzić *system* z pewnego *stanu początkowego* do pożądanego *stanu końcowego*. Badaniem algorytmów zajmuje się *algorytmika*. Algorytm może zostać *zaimplementowany* w postaci *programu komputerowego* lub dla innego urządzenia. Kiedy podczas tego procesu programista popełni błąd (*bug*), może to doprowadzić do poważnych konsekwencji - np. błędy w implementacji algorytmów bezpieczeństwa, mogą ułatwić popełnienie przestępstwa komputerowego.

3.8.1.30. **Wyjaśnienie**. Jednym z istotnych dla praktycznych zastosowań teorii obliczeń, są algorytmy mające postać tzw. funkcji obliczalnych. Dla potrzeb niniejszej książki przyjmiemy, że *funkcje obliczalne* to:

(1) Funkcje elementarna, począwszy od: funkcji o wartości stałej, następnika, funkcji trygonometrycznych, logarytmicznych, wykładniczych itp., których wartość można obliczyć z dowolną dokładnością $\varepsilon > 0$, po skończonej liczbie *n* kroków;

[59] Równanie diofantyczne to równanie algebraiczne z wieloma zmiennymi o współczynnikach całkowitych rozwiązywane w zbiorze liczb całkowitych. Równania diofantyczne często pojawiają się często przy rozwiązywaniu różnych zadań matematycznych.

(2) Funkcje wielu zmiennych rzutowania na podprzestrzenie, w szczególności na osie układu współrzędnych;
(3) Funkcje rekursywne[60] – o wartości obliczanej według poniższego schematu:

(Rek)	$h(\vec{x}, \vec{y}_0) = f(\vec{x})$
	$h(\vec{x}, \vec{y}_{n+1}) = g(\vec{x}, \vec{y}_n, h(\vec{x}, \vec{y}_n))$

gdzie $\vec{x} \epsilon \mathbb{R}^k$, $\vec{y}_0, \ldots, \vec{y}_n, \vec{y}_{n+1} \epsilon \mathbb{R}^m$ oraz $k > 0$, $m \geq n+1$ [61].
(4) Funkcje otrzymywane z funkcji elementarnych i rekursywnych poprzez składanie: $f(\vec{x}) = f(g_1(\vec{x}), g_2(\vec{x}), \ldots, g_s(\vec{x}))$; gdzie $s > 0$;
(5) Funkcje $f(\vec{x}) = \min \{y : g(y, \vec{x}) = 0\}$; czyli, że y przyjmuje najmniejszą wartość z pośród pierwiastków równania $g(y, \vec{x}) = 0$, gdzie funkcja g jest jedną z funkcji typu określonego w punktach $1 \div 4$.

W wyniku badań zainicjowanych pracami nad wzmiankowanym problemem *Hilberta*, okazało się, że nie wszystkie problemy matematyczne dają się rozwiązać na drodze wykonania kolejnej sekwencji kroków obliczeniowych. Okazało się, że zasadniczego przełomu dokonał w roku 1933 młody wiedeński matematyk *Kurt Gödel* (wzmiankowany wcześniej w części dwa), *dowodząc nierozstrzygalność złożonych systemów logicznych*. Innymi słowy, w złożonych systemach logicznych występują wyrażenia, o których, nie można rozstrzygnąć, czy są prawdziwe czy fałszywe. Jak następnie pokazano, fakt nierozstrzygalności ma swoje konsekwencje w odniesieniu do algorytmów służących do rozwiązywania różnorakich zadań danej teorii. Istnienie algorytmu służącego do przeprowadzenia systemu ze stanu początkowego danego systemu (teorii), do pożądanego stanu końcowego, jest równoważne z rozstrzygalnością zadania - znalezienia rozwiązania zadania polegającego na zbudowaniu procesu pozwalającego na przejście od stanu początkowego do pożądanego stanu.

3.8.1.40. **Wyjaśnienie**. Zgodnie z hipotezą *Churcha - Turinga*, funkcjami obliczalnymi są dokładnie te funkcje, które można obliczyć używając urządzenia maszynowego mając nieskończenie wiele czasu oraz przestrzeni pamięciowej. Równoważnie twierdzenie to oznacza, że *każda funkcja dająca się wyrazić przez algorytm jest obliczalna*.

W dalszych rozważaniach, zajmiemy się pomocniczą dla dalszych rozważań tematyką automatów skończonych, a następnie kluczową dla tematyki algorytmiki oraz obliczalności - Maszyną Turinga i jej konsekwencjami dla obliczalności.

Piśmiennictwo: *Banachowski L.* B.1.1., *Cormen T.* C.5.1., *Ross K.* R.1.1., *Sipser M.* S.7.1., *Wikipedia* W.2.2., *Wirth N.* W.5.1.

3.8.2. AUTOMATY SKOŃCZONE

3.8.2.00. **Wyjaśnienie**. Automat skończony (*Finite State Machine*, FSM) – abstrakcyjny, matematyczny, iteracyjny model zachowania *systemu dynamicznego* oparty na tablicy dyskretnych przejść między jego kolejnymi stanami (*diagram stanów*) wewnętrznymi, stanami wejścia i stanem wyjścia automatu skończonego. Dodatkowym wymaganiami stawianymi automatowi skończonemu jest wyróżnienie tzw. stanu początkowego i stanu akceptującego, czyli stanu, w którym znajdzie się automat skończony po zakończeniu przetwarzania danych wejściowych. Innymi słowy, automat skończony jest przykładem układu względnie odosobnionego, czyli układu kontaktującego się z otoczeniem – jedynie za pośrednictwem

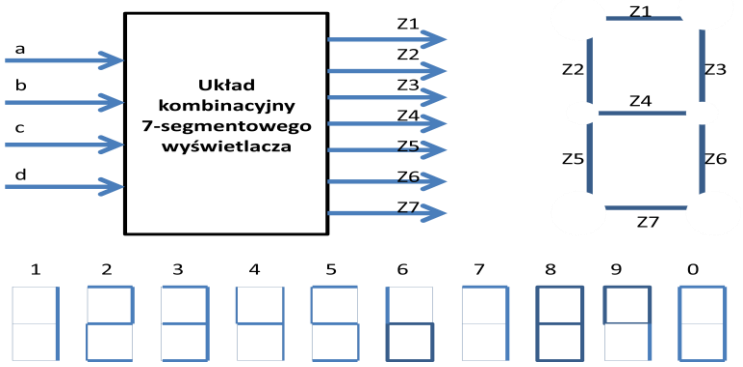
[60] W polskiej literaturze, często używany jest termin *rekurencja* zamiast *rekursja*.
[61] Dla skrócenia zapisu użyto oznaczenia $\vec{x} \equiv x_0, x_1, \ldots x_k$;

swojego wejścia i wyjścia. Iteracyjne zachowanie automatu skończonego przebiega w czasie, co oznacza, że stan wejść i stan wewnętrzny automatu skończonego w chwili n – decyduje o stanie wyjść i stanie wewnętrznym automatu skończonego w chwili $(n + 1)$.

Prostszym urządzeniem od automatu skończonego jest tzw. układ kombinacyjny[62], którego bieżący stan wyjścia określa aktualny stan wejścia. Układ kombinacyjny jest również układem względnie odosobnionym, ale pozbawionym właściwości posiadania i pamiętania stanu wewnętrznego, za którego pośrednictwem, działanie układu kombinacyjnego w chwili $(n + 1)$ jest zależne od działania układu kombinacyjnego w chwili n. Przykładem układu kombinacyjnego jest układ wyświetlacza siedmiosegmentowego (patrz rys. 3.8.3.01).Tablica dyskretnych przejść stanów czterobitowego wejścia w siedmiosegmentowe wyjście jest pokazana poniżej (tablica 3.8.2.02).

Przykładem układu sekwencyjnego, którego działanie przebiega iteracyjnie w kolejnych chwilach czasu, a który posiada tylko dwa stany wyróżnione, jest układ automatu otwierania drzwi wejściowych do budynku (patrz rys. 3.8.2.03). Jest to jeden z najprostszych automatów skończonych, przyjmując, że stanem początkowym są „drzwi zamknięte", a stanami akceptacyjnymi: dwa stany „drzwi otwarte" i „drzwi zamknięte".

Tablica dyskretnych przejść stanów, pokazująca zależności pomiędzy stanami obu wejść (panele wewnętrzny i zewnętrzny) oraz stanu drzwi (otwarte, zamknięte) w chwili n, a zachowaniem się drzwi w chwili (n+1), pokazuje tabela 3.8.2.04.

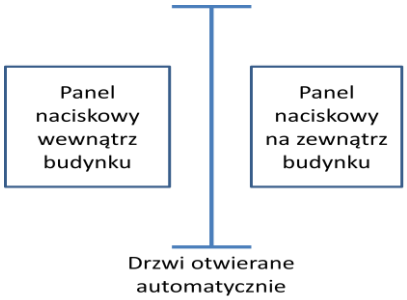


Rysunek 3.8.2.01 – Przykład układu kombinacyjnego

Ze względu na charakter przejść między stanami automatu skończonego, wyróżnia się *deterministyczne* i *niedeterministyczne* automaty skończone. Automaty skończone, są ważnym narzędziem teoretycznym (*modelem procesu obliczeniowego*) informatyki, wykorzystywanym w wytworzeniu i testowaniu oprogramowania. Jako modele szerszych procesów znajdują także swoje zastosowanie w różnych dziedzinach, jak: *matematyka i logika*, *lingwistyka, filozofia*, czy też *biologia*. Deterministyczny automat skończony (*Deterministic Finite-state Automaton* - DFA) to *abstrakcyjna maszyna o skończonej liczbie stanów*, która zaczynając w stanie początkowym czyta kolejne symbole pewnego słowa, po przeczytaniu każdego zmieniając swój stan na stan będący wartością funkcji jednego przeczytanego symbolu oraz stanu aktualnego. Jeśli po przeczytaniu całego słowa maszyna znajduje się w którymś ze stanów oznaczonych, jako akceptujące (końcowe), słowo należy do *języka regularnego,* dla rozpoznawania - którego, automat jest zbudowana. Deterministyczny automat skończony, podobnie jak inne automaty skończone może być reprezentowany za pomocą tabeli przejść pomiędzy stanami lub diagramu stanów.

[62] Patrz podrozdział 3.5.2.

Tabela 3.8.2.02 Stanu 7-segmentów wyjścia w funkcji stanu 4-bitowego wejścia.								
abcd	**cyfra**	**Z1**	**Z2**	**Z3**	**Z4**	**Z5**	**Z6**	**Z7**
0000	0	1	1	1	0	1	1	1
0001	1	0	0	1	0	0	1	0
0010	2	1	0	1	1	1	0	1
0011	3	1	0	1	1	1	0	1
0100	4	0	1	1	1	0	1	0
0101	5	1	1	0	1	0	1	1
0110	6	0	1	0	1	1	1	1
0111	7	1	0	1	0	0	1	0
1000	8	1	1	1	1	1	1	1
1001	9	1	1	1	1	0	1	0
1010	-	0	0	0	0	0	0	0
1011	-	0	0	0	0	0	0	0
1100	-	0	0	0	0	0	0	0
1101	-	0	0	0	0	0	0	0
1110	-	0	0	0	0	0	0	0
1111	-	0	0	0	0	0	0	0

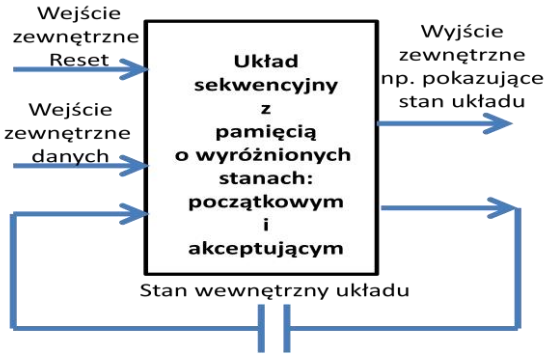


Rysunek 3.8.2.03 – Schematyczna prezentacja drzwi automatycznych.

Legenda: n – chwila bieżąca; $(n+1)$ – chwila następna; S – stan drzwi {otwarte, zamknięte}; P – panel zewnętrzny drzwi; T – panel wewnętrzny drzwi; O – włączenie silnika otwierania; Z – włączenie silnika zamykania.

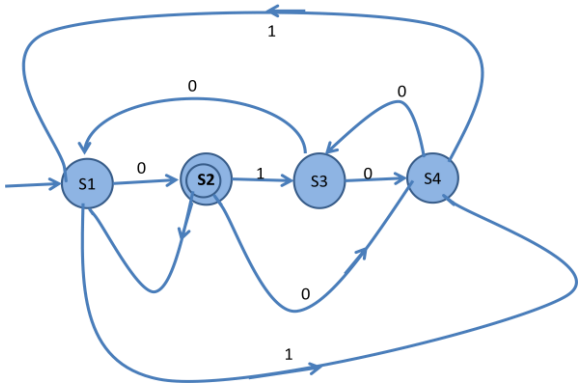
Tabela 3.8.2.04. Zależności pomiędzy stanami w chwili n, a działaniem w chwili $(n+1)$.					
S(n)	**P(n)**	**T(n)**	**S(n+1)**	**O(n+1)**	**Z(n+1)**
0	**0**	**0**	**1**	**0**	**1**
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	1	0	0
1	0	1	0	1	0
1	1	0	0	1	0
1	1	1	0	1	0

Uogólniając dotychczasowe rozważania, możemy automat skończony przedstawić graficznie jak na rys. 3.8.2.05.



Rysunek 3.8.2.05 – Graficzna prezentacja automatu skończonego.

Przykład diagramu przejść dla automatu skończonego o czterech wyróżnionych stanach: S1, S2, S3 i S4 pokazany jest na rys. 3.8.2.06. Stan S1 tego automatu jest *stanem początkowym* automatu (oznaczony specjalną strzałką – symbolizującą wejście danych). Stan S2 jest *stanem akceptującym* automatu (oznaczony podwójnym kółkiem). Na wejściu automatu może pojawiać się ciąg zer i jedynek. Ciąg danych wejściowych może się składać z $n \geq 1$ bitów.



Rysunek 3.8.2.06 – Diagram przejść: czterostanowego automatu skończonego.

Strzałki pomiędzy stanami pokazują jak diagram stanów określa nową wartość stanu po przeczytaniu kolejnego bitu danych wejściowych. Automat zaczyna czytać ciąg danych wejściowych znajdując się w stanie początkowym. Jeśli po przeczytaniu ostatniego bitu danych wejściowych w stanie akceptującym, to automat wysyła komunikat akceptuję. W pozostałym przypadku automat wysyła komunikat odrzucam. Tabela 3.8.3.07 pokazuje przejścia opisane przez diagram przejść pokazany na rys. 3.7.3.06.

Tabela 3.8.2.07 – Przykład przejść przez stany automatu skończonego.

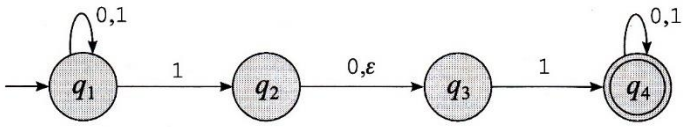
Bit wejścia	0	1	0	1	0	1	0	1
Stan kroku n	S1	S1	S2	S2	S3	S3	S4	S4
Stan kroku (n+1)	S2	S4	S1	S3	S4	S1	S3	S1
Ewentualny komunikat	akceptuję	odrzucam	odrzucam	odrzucam	odrzucam	odrzucam	odrzucam	odrzucam

3.8.2.10. **Definicja**. Deterministyczny automat skończony jest jednoznacznie opisany przez piątkę uporządkowaną (A, Q, q_0, F, δ):
<deterministyczny automat skończony>
<alfabet> A, czyli zbiór znaków akceptowanych przez automat </alfabet>
<zbiór stanów> Q, czyli zbiór stanów przyjmowanych przez automat </zbiór stanów>
<wyróżniony stan początkowy> $q_0 \in Q$ </wyróżniony stan początkowy>
<zbiór stanów końcowych> $F \subset Q$, odpowiadających słowom wyróżnionym </zbiór stanów końcowych>
<funkcja przejścia>$\delta(a, p, q)$: $Q \times A \rightarrow Q$, gdzie $a \in A$, zaś $p, q \in Q$ </funkcja przejścia>
</deterministyczny automat>

3.8.2.11. **Definicja**. Funkcja przejścia $\delta(a, p, q)$ - automatu skończonego opisuje przejścia od stanu $q \in Q$ do stanu $p \in Q$, po przeczytaniu symbolu $a \in A$. Inaczej mówiąc δ jest *funkcją przejścia* (od stanu q do stanu p), przypisującą parze (q, a) nowy stan p, w którym znajdzie się automat znajdujący się w stanie q, po przeczytaniu symbolu a. Funkcja δ może być częściowo określona. To znaczy mogą istnieć takie pary (q, a), dla których nie jest określony nowy stan.

3.8.2.21. **Wyjaśnienie**. *Niedeterminizm* (*Nondeterminism*) – jest jednym z głównych koncepcji teoretycznej informatyki. Jak pisze *Mordechai Ben-Ari*, pojęcie to wprowadzili *Rabin* oraz *Scott* w

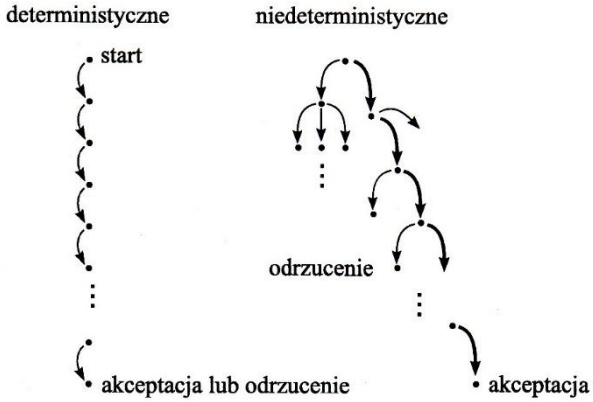
ich *niedeterministycznym automacie skończonym*. *Niedeterminizm* daje się stosunkowo prosto sprowadzić do determinizmu, co pozwala na zwiększenie efektywności algorytmów. *Niedeterminizm* pojawia się często w takich aplikacjach jak: gramatyki języków programowania, algorytmy, sklejanie modeli współbieżnych. *Niedeterminizm* jest przydatnym pojęciem, które wywarło istotny wpływ nie tylko na teorię obliczeń, ale również na informatykę. Deterministyczny automat skończony, gdy znajduje się w określonym stanie i czyta kolejny znak z wejścia, to wiemy, na podstawie funkcji przejścia, jaki automat przyjmie nowy stan. Jak już wiemy, takie działanie automatu - nazywamy działaniem deterministycznym. W dalszym ciągu omówimy kilka zasadniczych pojęć niedeterminizmu powtórzymy za *Michaelem Sipserem*:



Rys. 3.8.2.20. Niedeterministyczny automat skończony N.

3.8.2.22. **Wyjaśnienie**. *Niedeterminizm* jest uogólnieniem determinizmu, zatem każdy automat skończony deterministyczny jest również automatem skończonym niedeterministycznym. Jak widać na rys. 3.8.2.20, niedeterministyczny automat skończony może mieć dodatkowe możliwości.

Różnica między deterministycznym automatem skończonym, oznaczanym skrótem DFA (*deterministic finite automaton*), a niedeterministycznym automatem skończonym, oznaczanym skrótem NDFA (*nondeterministic finite automaton*), jest widoczna na pierwszy rzut oka. Po pierwsze, każdy stan DFA ma zawsze dokładnie jedno przejście dla każdego symbolu alfabetu. W automacie nie-deterministycznym przedstawionym na rys. 3.8.2.23 zasada ta jest naruszona. W stanie q_1 jest jedno wyjście dla symbolu 0, ale dwa dla symbolu 1; w stanie q_2 jest jedno wyjście, dla 0, ale nie ma żadnego dla 1. W NDFA z każdego stanu może być zero, jedno lub dowolnie wiele wyjść dla każdego symbolu z alfabetu. Po drugie, w DFA etykietami przy strzałkach przejść są



Rys. 3.8.2.23. Przetwarzanie deterministyczne, a niedeterministyczne.

symbole z alfabetu. Przedstawiony NDFA ma natomiast etykietę ε. W ogólności, strzałki przejść w NFA mogą być etykietowane symbolami alfabetu lub słowem pustym *ε*. Z każdego stanu może wychodzić zero, jedna lub wiele strzałek etykietowanych symbolem ε.

3.8.2.24. **Wyjaśnienie**. W jaki sposób działa NDFA? Przypuśćmy, że uruchomiliśmy NFA dla pewnego słowa wejściowego i doszliśmy do stanu, z którego możemy kontynuować

przetwarzanie na wiele sposobów. Na przykład, jesteśmy w stanie q_1 automatu N, a kolejnym symbolem wejściowym jest 1. Po jego przeczytaniu automat rozdziela się na kilka kopii, kontynuując równolegle wszystkie możliwe przetwarzania. Każda z kopii automatu wybiera jedną z dalszych ścieżek i działa jak zwykły NDFA. Jeżeli w trakcie przetwarzania pojawią się kolejne wybory automat ponownie rozdziela się na kilka kopii. Jeżeli dla kolejnego symbolu wejściowego pewna kopia automatu nie ma przejścia z aktualnego stanu, to kasujemy tę kopię wraz ze ścieżką przetwarzania, którą ona podjęła. Jeżeli na koniec którakolwiek z kopii automatu osiągnie stan akceptujący w chwili dojścia do końca słowa wejściowego, to NFA akceptuje słowo wejściowe.

Jeżeli automat jest w stanie, dla którego istnieje strzałka przejścia z etykietą ε, to dzieje się coś podobnego jak poprzednio. Nie czytając symbolu z wejścia, automat rozdziela się na kilka kopii, z których jedna pozostaje w aktualnym stanie, a pozostałe podążą wzdłuż strzałek etykietowanych symbolami ε. Dalej przetwarzanie przebiegaj niedeterministycznie, jak poprzednio.

3.8.2.25. **Wyjaśnienie**. Niedeterminizm można traktować, jako rodzaj przetwarzania równoległego, w którym wiele niezależnych procesów czy wątków działa jednocześnie. Gdy NFA rozdziela się, by śledzić kilka ścieżek przetwarzania, mamy zjawisko analogiczne do rozdzielenia procesu na kilka procesów potomnych realizowanych niezależnie. Jeśli ostatecznie jeden z tych procesów akceptuje, całe przetwarzanie jest także akceptujące.

3.8.2.30. **Wyjaśnienie**. Innym sposobem spojrzenia na niedeterminizm jest potraktowanie go, jako drzewa przetwarzania. Korzeń drzewa to początek przetwarzania. Każde rozgałęzienie w drzewie odpowiada momentowi przetwarzania, w którym automat ma wybór dalszej drogi działania. Automat akceptuje słowo wejściowe, gdy co najmniej jedna z gałęzi przetwarzania kończy się w stanie akceptującym, jak jest to pokazane na rys. 3.8.2.31.

Rozważmy kilka przykładowych przetwarzań NDFA N przedstawionego na rys. 3.8.2.20. Przetwarzanie N dla wejścia 01O110 jest pokazane na rys. 3.8.2.31. Dla danych wejściowych 010110 rozpoczynamy przetwarzanie w stanie q_1 automatu N i czytamy pierwszy symbol - 0. Ze stanu q_1 można, czytając 0, przejść tylko w jedno miejsce - z powrotem do stanu q_1, więc pozostajemy w tym stanie. Następnie czytamy kolejny symbol - 1. W stanie q_1 z symbolem 1_ mamy wybór: albo pozostajemy w q_1, albo przechodzimy do q_2. To oznacza, że w przetwarzaniu niedeterministycznym automat rozdziela się na dwie kopie, z których każda pójdzie inna ścieżką. Śledzimy to przetwarzanie, zaznaczając każdy ze stanów, w którym może być automat. Mamy, więc zaznaczone stany q_1 i q_2. Ze stanu, q_2 wychodzi strzałka z etykietą ε, więc automat rozdziela się ponownie; do zaznaczonych stanów dochodzi q_3 (q_2 nadal pozostaje w gronie zaznaczonych). Mamy, więc zaznaczone q_1, q_2 i q_3.

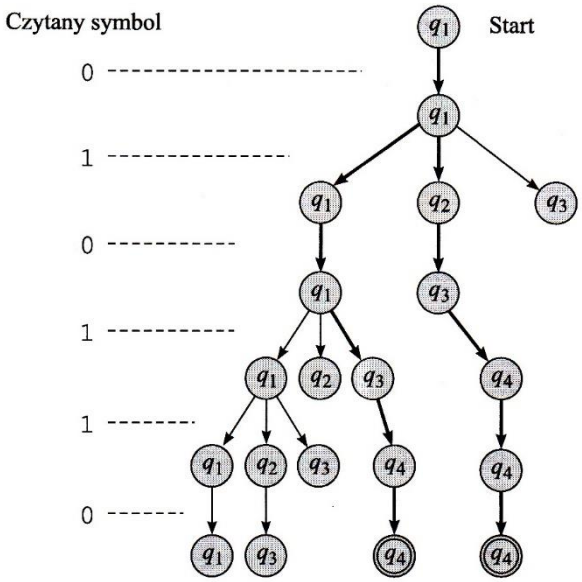
Czytając trzeci symbol - 0, patrzymy na każdy z zaznaczonych stanów. Stan q_1 pozostaje zaznaczony, q_2 zastępujemy przez q_3, a q_2 usuwamy z grona zaznaczonych. Z ostatniego stanu nie ma, bowiem wyjścia przy symbolu 0, co oznacza, że proces po prostu „wygasa". W tej chwili mamy, więc zaznaczone stany q_1 i q_3.

Po przeczytaniu czwartego symbolu - 1 zaznaczony stan q_1 rozbijamy na stany q_1 i q_2, po czym (nowo zaznaczony) stan q_2 rozbijamy na q_2 i q_3, aby uwzględnić przejście krawędzią etykietowaną symbolem ε; ze stanu q_3 przechodzimy natomiast do q_4. W tym momencie wszystkie cztery stany są zaznaczone.

Po przeczytaniu piątego symbolu - 1, zaznaczone stany q_1 i q_3 powodują zaznaczenie stanów q_1, q_2, q_3 i q_4, jak przekonaliśmy się w przypadku czwartego symbolu. Stan q_2 zostaje usunięty z grona zaznaczonych. Stan q_4 powoduje zaznaczenie q_4. W ten sposób stan q_4 jest zaznaczony dwukrotnie – usuwamy jedno „zaznaczenie", bo musimy pamiętać jedynie, że q_4 jest zaznaczony, nieważne, ile razy.

Po przeczytaniu ostatniego, szóstego symbolu - 0 stan q_1 pozostaję zaznaczony, stan q_2 powoduje zaznaczenie q_3, usuwamy oznaczenie q_3, pozostawiamy niezmienione zaznaczenie q_4. Jesteśmy na końcu słowa wejściowego, więc zaakceptujemy, jeśli w gronie zaznaczonych mamy stan akceptujący. Mamy zaznaczone stany q_1, q_2, q_3 i q_4, ale stan q_4 jest akceptujący, czyli automat N akceptuje dane wejściowe.

Co automat N robi dla wejścia 010? Zaznaczmy stan q_1. Po przeczytaniu 0 zaznaczony pozostaje tylko stan q_1, ale po przeczytaniu 1 zaznaczamy q_1, q_2 oraz q_3 (z powodu ε-przejścia). Po trzecim symbolu 0 usuwamy zaznaczenie z q_1, q_2 powoduje zaznaczenie q_3, a zaznaczenie q_1 pozostaje niezmienione. W tym momencie dotarliśmy także do końca danych wejściowych i żaden stan akceptujący nie jest zaznaczony, więc N odrzuca te dane wejściowe.



Rys. 3.8.2.31. Przetwarzanie przez automat N dla słowa wejściowego 010110.

Przeprowadzając podobne eksperymenty, przekonamy się, że automat N akceptuje wszystkie słowa zawierające pod-słowo 101 lub 11.

Niedeterministyczne automaty skończone są przydatne z kilku powodów. Można pokazać, że każdy NFA można przekształcić w równoważny mu DFA, a konstrukcja NDFA jest często łatwiejsza niż konstrukcja DFA. Automat NDFA może być znacznie mniejszy niż jego deterministyczny odpowiednik, a jego działanie może być łatwiejsze do zrozumienia, Niedeterminizm automatów skończonych jest także dobrym wprowadzeniem do nie-determinizmu w silniejszych modelach przetwarzania, gdyż działanie automatów skończonych stosunkowo łatwo zrozumieć.

Na zakończenie rozważań dotyczących niedeterminizmu, warto podkreślić, że:

1. Błędy w sprzęcie informatycznym, można z formalnego punktu widzenia traktować, jako działanie danego układu cyfrowego, jako automat NDFA.

2. Podobnie, w przypadku błędów w programach reaktywnych, polegających na pojawianiu się nieprzewidywanych zdarzeń, można traktować takie nieprzewidywane efekty, jako automat NDA.

3.8.2.40. **Wyjaśnienie**. Każdemu niedeterministycznemu automatowi skończonemu odpowiada deterministyczny automat skończony akceptujący dokładnie te same słowa. Możemy go uzyskać dokonując *determinizacji automatu skończonego*. Automat skończony jest realizacją języka regularnego, inaczej mówiąc, każdemu *językowi regularnemu* odpowiada realizujący ten język *automat skończony* i każdemu *automatowi skończonemu* odpowiada charakterystyczny *język regularny*.

Piśmiennictwo: Ben-Ari Mordechai B.2.2., Sipser Michael S.7.1.

3.8.3. AUTOMATY SKOŃCZONE ZE STOSEM

Automat skończony ze stosem (*puch down automation*, PDA) – abstrakcyjny, matematyczny, iteracyjny model zachowania systemu dynamicznego wyposażony w stos, oparty na tablicy dyskretnych przejść między jego kolejnymi stanami (*diagram stanów*), posiadający dodatkowo możliwość zapisywania w stosie symboli, które odczytuje później, w toku obliczenia. Zapisanie symbolu w stosie (*operacja **push***), powoduje „zepchnięcie w dół stosu" wszystkich pozostałych symboli zapisanych w stosie. W dowolnym momencie można odczytać symbol ze szczytu stosu – usuwając go (*operacja **pop***), powoduje „podniesienie w górę stosu" wszystkich pozostałych symboli zapisanych w stosie. Stos jest strukturą pamięci typu LIFO (*last in, first out* – czyli „ostatni na wejściu, pierwszy na wyjściu). Urządzenie stosu przypomina zasadę działania magazynku w powtarzalnej broni strzeleckiej. Obok funkcjonalności stosu, automat skończony ze stosem realizuje funkcjonalność typową dla automatu skończonego. Podstawowa formalna różnica pomiędzy automatem skończonym a automatem ze stosem, to postać funkcji przejścia, która w przypadku automatu ze stosem jest dwu-wymiarową funkcją wektorową (tzw. wektor funkcja przejścia), a nie funkcją skalarną.

3.8.3.10. **Definicja**. Deterministyczny automat skończony ze stosem jest jednoznacznie opisany przez szóstkę uporządkowaną $(A, Q, G, q_0, F, \delta)$:
<deterministyczny automat skończony ze stosem>
<alfabet wejścia>A, czyli zbiór znaków akceptowanych przez automat </alfabet wejścia>
<zbiór stanów>Q, czyli zbiór stanów przyjmowanych przez automat </zbiór stanów>
<alfabet stosu>G, czyli zbiór symboli, zapisywanych na stosie </alfabet stosu>
<wyróżniony stan początkowy> $q_0 \in Q$ </wyróżniony stan początkowy>
<zbiór stanów końcowych> $F \subset Q$ </zbiór stanów końcowych>
<wektor funkcja przejścia>
$\delta(a, p, q, r): Q \times A \times G \rightarrow Q \times G$, gdzie $a \in$ A, zaś $p, q \in$ Q oraz $r \in G$
</wektor funkcja przejścia>
</deterministyczny automat ze stosem>

Automat ze stosem $M = (A, Q, G, q_0, F, \delta)$ działa w następujący sposób. Akceptuje wejście w, jeżeli można je przedstawić w postaci sekwencji $w = \langle w_0, w_1, \ldots, w_m \rangle$, gdzie każde $w_i \in A$, oraz istnieje sekwencja stanów $r = \langle r_0, r_1, \ldots, r_m \rangle$, gdzie każde $r_i \in Q$, oraz istnieje sekwencja zapisów w stosie $s = \langle s_0, s_1, \ldots, s_m \rangle$, gdzie każde $s_i \in G$,które spełniają trzy poniższe warunki:

1. Zachodzą następujące zależności $r_0 = q_0$ oraz s_0 = *empty*. Oznacza to, że automat M rozpoczyna działanie z pustym stosem.

2. Dla i = 0, … , m-1 wektor nowego stanu oraz nowej zawartości stosu w chwili (i+1), czyli (r_{i+1}, s_{i+1}) = $\delta(w_i, r_i, s_i)$. Warunek ten definiuje przejście automatu M na podstawie wejścia w chwili i, zawartości stosu w chwili i oraz stanu automatu w chwili i.
3. Zachodzi zależność $r_m \in R$, co oznacza, że automat po przeczytaniu słowa r złożonego z sekwencji znaków $\langle r_0, r_1, \ldots, r_m \rangle$ jest w stanie akceptującym.

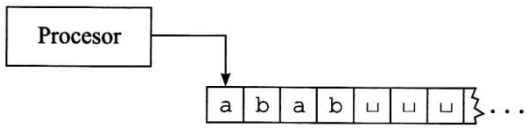
Uogólnieniem *deterministycznego automatu ze stosem* jest *niedeterministyczny automat ze stosem*. W przeciwieństwie do automatów skończonych, *niedeterministyczny automat ze stosem* nie daje się sprowadzić do równoważnemu mu *deterministycznego automatu skończonego*. Innymi słowy, okazuje się, że wprowadzenie losowości w proces zmiany stanów, w miejsce *determinizmu* zmiany stanów, prowadzi do nowej klasy automatów ze stosem, tzw. *niedeterministycznych automatów ze stosem*, znacznie ważniejszych teoretycznie od automatów deterministycznych.

3.8.3.20. **Wyjaśnienie**. *Niedeterministyczny automat ze stosem* jest realizacją *języka bezkontekstowego*, inaczej mówiąc, każdemu językowi bezkontekstowemu odpowiada realizujący ten język niedeterministyczny automat ze stosem i każdemu niedeterministycznemu automatowi ze stosem odpowiada charakterystyczny język bezkontekstowy.

Piśmiennictwo: Sipser M. S.5.1.

3.8.4. Maszyna Turinga i Uniwersalna Maszyna Turinga

Przejdziemy do zaproponowanego w 1936 roku przez *Alana Turinga* silnego modelu obliczeń, nazwanego Maszyną Turinga (w skrócie TM od *Turing Machine*). Podobna do automatu skończonego, ale dysponująca pamięcią o nieograniczonym rozmiarze oraz swobodnym dostępie. Dziś możemy powiedzieć, że maszyna Turinga jest w pewnym sensie modelem komputera służącego do ogólnych zastosowań. Maszyna Turinga może, zasadzie robić wszystko to, co może robić współczesny komputer. Jednak istnieją problemy, których ona także nie może rozwiązać. Są to problemy niemieszczące się nawet teoretycznie w granicach możliwości obliczeniowych. W modelu *Maszyny Turinga* jako nieograniczona pamięć jest używana taśma o nieskończonej długości. *Maszyna Turinga* ma głowicę, która może przesuwać się po taśmie oraz czytać i zapisywać na niej symbole (rys. 3.8.4.00). Początkowo taśma zawiera tylko *słowo wejściowe* - poza nim jest wypełniona *symbolami pustymi*. Gdy maszyna musi zapamiętać jakąś informację, może zapisać ją na taśmie. Aby odczytać zapisaną informację, maszyna może z powrotem przesunąć nad nią głowicę. Maszyna kontynuuje obliczenia do momentu, aż zdecyduje się wypisać wynik. Wynikiem jest informacją *akceptuj* lub *odrzuć* – przekazana przez wejście maszyny powodująca przejście w stan *akceptujący* lub *odrzucający*. Jeśli maszyna w żadnym momencie nie osiągnie stanu akceptacji lub odrzucenia, to może kontynuować obliczenia w nieskończoność, nigdy się nie zatrzymując (jest to tak zwany *problem stopu*).



Rysunek 3.8.4.00 – Schematyczne przedstawienie Maszyny Turinga

Poniżej podsumowanie różnic między automatem skończonym oraz maszyną Turinga.

(1) Maszyna Turinga może zarówno zapisywać na taśmie, jak i z niej czytać.

(2) Głowica czytająco-pisząca może przesuwać się nad taśmą w obu kierunkach.
(3) Taśma jest nieskończona.
(4) Wejście maszyny w stan *akceptujący* jak również w stan *odrzucający* natychmiast powoduje zatrzymanie maszyny.

A oto, jak radzi sobie Maszyna Turinga M_1 sprawdzająca przynależność słowa do języka:

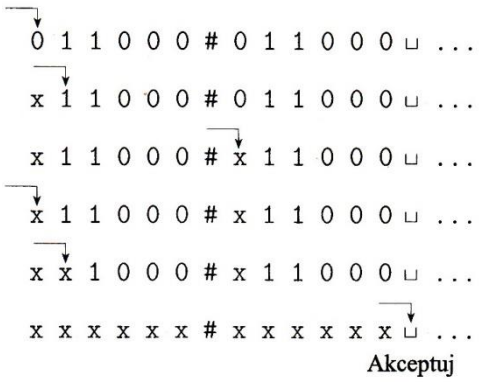
$$B = \{w \# w \mid w \in \{0, 1\}^*\}.$$

Chcielibyśmy, by M_1 akceptowała słowo, jeśli należy ono do B, i odrzucała w przeciwnym przypadku. Aby lepiej zrozumieć działanie M_1, postawmy się w jej sytuacji. Stoimy więc przed długą na kilometr taśmą zawierającą miliony znaków - słowo wejściowe. Nasze zadanie polega na podjęciu decyzji, czy słowo to jest elementem języka B, czyli sprawdzeniu, czy składa się ono z dwu identycznych słów oddzielonych znakiem #.

Może być, że słowo jest długie, zapamiętane w całości na taśmie (jest to tzw. słowo wejściowe), możemy przesuwać się po nim w obie strony i zaznaczać wybrane miejsca. Pierwszy przychodzący na myśl sposób rozpoznania słowa, to poruszanie się po słowie „zygzakiem", tak by odwiedzić kolejno wszystkie odpowiadające sobie pozycje po obu stronach znaku # i sprawdzić, czy stoją na nich jednakowe znaki. Odwiedzone miejsca możemy zaznaczać, umieszczając w nich odpowiednie znaczniki.

Zaprojektujemy maszynę M_1, która będzie działać właśnie w taki sposób. Będzie ona wielokrotnie przesuwała głowicę nad słowem wejściowym. Po każdym przejściu będzie dopasowywać parę znaków leżących po różnych stronach znaku #. Aby zapamiętać, które znaki zostały już porównane, maszyna M1 będzie zamazywać każdy sprawdzony znak. Gdy wszystkie symbole zostaną zamazane, będzie to oznaczało, że udało się dopasować wszystkie znaki i maszyna M_1 może wejść w stan akceptujący. Jeśli maszyna znajdzie niezgodność, to wchodzi w stan odrzucający. Ostatecznie, algorytm dla Ml jest następujący:

(1) Przejdź wzdłuż taśmy między odpowiadającymi sobie pozycjami po różnych stronach znaku #, by sprawdzić, czy na pozycjach tych stoją jednakowe znaki. Jeśli znaki te są różne lub gdy w słowie nie ma znaku #, to odrzuć słowo wejściowe. W przeciwnym razie wykreśl porównane symbole, by zapamiętać miejsce, do którego zostały porównane fragmenty słowa.
(2) Po wykreśleniu wszystkich symboli na lewo od znaku # sprawdź, czy na prawo od tego znaku pozostały jakieś symbole. Jeśli pozostały, to odrzuć słowo wejściowe, w przeciwnym razie – zaakceptuj je.



Rysunek 3.8.4.01 – Stan taśmy w kolejnych krokach przetwarzania słowa 011000#011000

Na rysunku 3.8.4.01, jest pokazanych kilka stanów taśmy maszyny M_1 na których znajduje się ona w czasie obliczeń dla słowa wejściowego 011000#011000. Podany opis maszyny Turinga M_1 to zarys sposobu jej działania, w którym pominięto wiele szczegółów. Dokładny opis możemy sporządzić, przedstawiając formalny opis, podobnie jak w przypadku automatów skończonych czy automatów ze stosem. W takim opisie specyfikujemy każdy z elementów formalnej definicji maszyny Turinga, którą zaprezentujemy niżej. W rzeczywistości prawie nigdy nie opisuje się formalnie działania maszyn Turinga – ponieważ opisy takie są zbyt długie.

3.8.4.10. **Definicja**. *Maszyny Turinga*. Istotą definicji jest funkcja przejścia δ, która określa, w jaki sposób maszyna przechodzi z jednego stanu do kolejnego. Dla *Maszyny Turinga* jest to funkcja postaci:

$$Q \times \Gamma \rightarrow Q \times \Gamma \times \{a, R|L\}.$$

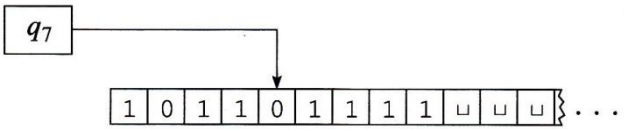
To oznacza, że kiedy maszyna jest w stanie q, $r \in Q$, jej głowica jest nad „komórką" taśmy wejściowej zawierająca symbol a oraz $\delta(q, a) = (r, b, L)$ – gdzie $a, b \in \Gamma$, wtedy maszyna w miejsce a pisze symbol b i przechodzi do stanu r. Trzecia składowa funkcji przejścia, czyli L lub R, określa, czy głowica nad taśmą przesunie się w lewo, czy w prawo. W podanym przypadku L oznacza, że głowica przesunie się w lewo.

Maszyna Turinga to siódemka uporządkowana $(Q, \Sigma, \Gamma, \delta, q_0, q_{akceptuj}, q_{odrzuć})$, gdzie Q, Σ i Γ są zbiorami skończonymi oraz:

(1) Q jest zbiorem stanów;
(2) Σ jest alfabetem wejściowym, do którego nie należy znak pusty ⊔;
(3) Γ jest alfabetem taśmy, gdzie $\sqcup \in \Gamma$ oraz $\Sigma \subseteq \Gamma$;
(4) $\delta: Q \times \Sigma \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ jest funkcją przejścia;
(5) $q_0 \in Q$ jest stanem początkowym;
(6) $q_{akceptuj} \in Q$ jest stanem akceptującym;
(7) $q_{odrzuć} \in Q$ jest stanem odrzucającym, gdzie $q_{akceptuj} \neq q_{odrzuć}$.

Maszyna Turinga $(Q, \Sigma, \Gamma, \delta, q_0, q_{akceptuj}, q_{odrzuć})$ działa w następujący sposób. Na początku słowo wejściowe $<w> = <w_1w_2...w_n> \in \Sigma$ znajduje się w początkowych n komórkach taśmy, a pozostała część taśmy jest pusta (tzn. wypełniona symbolami pustymi). Głowica jest umieszczona nad pierwszą komórką taśmy. Zauważmy, że symbol pusty nie należy do alfabetu Σ, zatem pierwszy symbol pusty na taśmie oznacza koniec słowa wejściowego. Maszyna rozpoczyna działanie, które przebiega na podstawie funkcji przejścia. Gdy maszyna M próbuje przesunąć głowicę na lewo od początku taśmy, głowica pozostaje w miejscu, nawet gdy funkcja przejścia nakazuje przesunąć ją w lewo. Maszyna działa do momentu, aż wejdzie w jeden ze stanów: *akceptujący* lub *odrzucając*y - w chwili gdy to nastą) a, maszyna kończy działanie (stop). Jeśli to nie nastąpi, to maszyna M działa bez końca. Podczas obliczeń maszyny Turinga zmieniają się jej: *stan maszyny*, *zawartość taśmy* oraz *pozycja głowicy* nad taśmą. Ustawienie tych trzech elementów nazywamy *konfiguracją Maszyny Turinga*.

Konfiguracje zazwyczaj zapisuje się w następującej postaci: Dla stanu q oraz słów u i v złożonych z symboli alfabetu taśmowego Γ, przez $u\ q\ v$ oznaczamy konfigurację, gdy maszyna jest w stanie q, zawartością taśmy jest słowo uv, a głowica znajduje się nad pierwszym symbolem słowa u. Poza słowem uv (czyli za ostatnim znakiem v) taśma zawiera jedynie symbole puste. Na przykład, zapis $101q_701111$ oznacza konfigurację maszyny, w której na taśmie znajduje się słowo 101101111, aktualnym stanem jest q_7, a głowica jest aktualnie nad drugim zerem. Konfiguracja ta jest przedstawiona na rys. 3.8.4.02.



Rysunek 3.8.4.02 – Maszyna Turinga w konfiguracji $1011q_701111$

Zapiszmy formalnie przedstawione dotychczas intuicyjnie obliczenia *Maszyny Turinga*. Mówimy, że maszyna z konfiguracji C_1 przechodzi do konfiguracji C_2, gdy *Maszyna Turinga* może w jednym kroku, zgodnie z zasadami działania, zmienić konfigurację z C_1 na C_2. Formalnie zapiszemy to następująco. Niech *a*, *b* i *c* będą symbolami z alfabetu Γ, słowa *u* i *v* niech będą słowami z Γ* i niech q_i oraz q_j będą stanami. Wówczas *ua* q_i *bv* oraz *u* q_j *acv* są dwiema konfiguracjami. Powiemy, że z konfiguracji:

ua q_i *bv maszyna przechodzi do konfiguracji u* q_j *acv, jeśli* $\delta(q_i, b) = (q_j, c, L)$.

Jest to przypadek, gdy maszyna przesuwa głowicę w lewo. Gdy przesuwa głowicę w prawo, to z konfiguracji:

ua q_i *bv maszyna przechodzi do konfiguracji uac* q_j *v, jeśli* $\delta(q_i, b) = (q_j, c, R)$.

Gdy głowica jest nad krańcową pozycją na taśmie, mamy do czynienia z przypadkami szczególnymi. Jeśli znajduje się nad skrajnie lewym symbolem, to z konfiguracji q_i *bv* przechodzi do konfiguracji q_j *cv* przy ruchu głowicy w lewo (wynika to stąd, że głowica nie może wyjść poza lewy kraniec taśmy) oraz do konfiguracji c q_j *v* przy ruchu głowicy w prawo. Gdy maszyna jest nad skrajnie prawym symbolem, to konfiguracja *ua* q_i jest równoważna konfiguracji *ua* q_i ⊔, ponieważ założyliśmy, że naprawo od słowa na taśmie znajdują się same symbole puste. Przypadek ten możemy więc rozważyć analogicznie jak zwykły przypadek, gdy głowica nie znajdowała się nad skrajną prawą pozycją taśmy. *Konfiguracją początkową* - maszyny M dla wejścia *w* jest konfiguracja q_0 *w*, która oznacza, że maszyna jest w stanie początkowym q_0, a jej głowica znajduje się nad pierwszym od lewej symbolem na taśmie. *Konfiguracją akceptującą* - jest konfiguracja, dla której wartością stanu jest $q_{akceptuj}$. *Konfiguracją odrzucającą* - jest konfiguracja, dla której wartością stanu jest $q_{odrzuć}$. Konfiguracje akceptująca i odrzucająca są *konfiguracjami końcowymi* - maszyna nie przechodzi z nich do kolejnych konfiguracji. Przyjmując, że maszyna zatrzymuje się po przejściu w stan $q_{akceptuj}$ albo $q_{odrzuć}$, jednocześnie powinniśmy zdefiniować funkcję przejścia w bardziej formalny sposób:

$\delta: Q' \times \Gamma \to Q \times \Gamma \times \{L, R\}$, *gdzie* Q' *to zbiór* Q *bez stanów* $q_{akceptuj}$ i $q_{odrzuć}$.

Maszyna Turinga M akceptuje wejście *w*, jeśli istnieje ciąg konfiguracji $C_1, C_2, \ldots, C_k$ takich, że:

(1) C_1 jest konfiguracją początkową maszyny M dla słowa wejściowego *w*;
(2) z każdej konfiguracji C_i maszyna przechodzi do C_{i+1};
(3) C_k jest konfiguracją akceptującą.

Zbiór słów które maszyna M akceptuje, to język maszyny M lub *język rozpoznawany* przez maszynę M. Oznaczamy go L(M) (w niektórych źródłach języki takie są nazywane *rekurencyjnie przeliczalnymi*).

Działanie *Maszyny Turinga* dla słowa wejściowego może skończyć się na jeden z trzech sposobów. Maszyna może *zaakceptować*, *odrzucić* lub się *zapętlić* (działać w nieskończoność). Przez *zapętlenie się* rozumiemy sytuację, gdy maszyna po prostu działa w nieskończoność. Zapętlenie maszyny może powodować proste lub skomplikowane zachowanie - ważne jest, ze nie prowadzi ono do stanu końcowego (*akceptuje* lub *odrzucam*). *Maszyna Turinga* M może nie

zaakceptować słowa wejściowego, wchodząc w stan $q_{odrzuć}$ i jawnie to słowo odrzucając. Może także nie zaakceptować słowa, zapętlając się. Czasami trudno jest odróżnić maszynę, która się pętli, od maszyny, która po prostu działa bardzo długo. Z tego powodu lepiej jest mieć do czynienia z maszynami, które zatrzymują się dla każdego wejścia; takie maszyny nigdy się nie zapętlają. Nazywamy je *maszynami rozstrzygającymi*, gdyż zawszę podejmują decyzję, czy słowo zostało zaakceptowane) czy odrzucone. Mówimy także, że język jest *rozstrzygany* przez maszynę, jeśli rozpoznaje go *maszyna rozstrzygająca*.

3.8.4.20. **Definicja**. Język narywamy *rozpoznawalnym w sensie Turinga*, jeżeli jest rozpoznawany przez pewną *Maszynę Turinga*[63].

Podamy kilka przykładów języków rozstrzygalnych. Każdy język rozstrzygalny jest rozpoznawalny w sensie Turinga.

3.8.4.30. Przykłady *Maszyn Turinga* (wg Sipser'a). Podobnie jak to uczyniliśmy w przypadku automatów skończonych i automatów ze stosem, możemy formalnie opisać *Maszynę Turinga*, określając każdy z jej siedmiu elementów. Wnikanie w szczegóły na tak niskim poziomie może jednak być kłopotliwe dla wszystkich *Maszyn Turinga* poza najprostszymi. Nie będziemy więc poświęcać wiele miejsca takim opisom, Zazwyczaj będziemy podawać opisy na wyższym poziomie - wystarczająco precyzyjne dla naszych celów i znacznie łatwiejsze do zrozumienia. Warto jednak pamiętać, że każdy opis na wysokim poziomie jest w rzeczywistości tylko skrótem opisu formalnego. Poświęcając odpowiednio wiele czasu i uwagi, każdą z przedstawionych *Maszyn Turinga* można opisać ze wszystkimi formalnymi szczegółami.

Aby ułatwić powiązanie opisu formalnego i opisu na wysokim poziomie, w pierwszych dwóch przedstawionych przykładach załączymy diagramy stanów. Analizę tychże diagramów stanów można pominąć, jeśli powiązanie to jest już dla czytelnika oczywiste.

Przykład 1. Opiszemy *Maszynę Turinga* M_2 (TM),która rozstrzyga język $A = \{0^{2^n} \mid n \geq 0\}$, czyli język złożony ze wszystkich ciągów zer długości równej pewnej potędze dwójki.
M_2 = „Dla słowa wejściowego <*w*>:
(1) Przeczytaj słowo wejściowe od lewej do prawej, wykreślając co drugie zero.
(2) Jeśli po wykonaniu powyższego punktu na taśmie pozostało jedno zero, zaakceptuje wejście.
(3) Jeśli po wykonaniu pierwszego punktu na taśmie pozostało więcej niż jedno zero i liczbach zer jest nieparzysta, odrzuć wejście.
(4) Przesuń głowicę nad lewą skrajną pozycję.
(5) Powróć do punktu (1)."

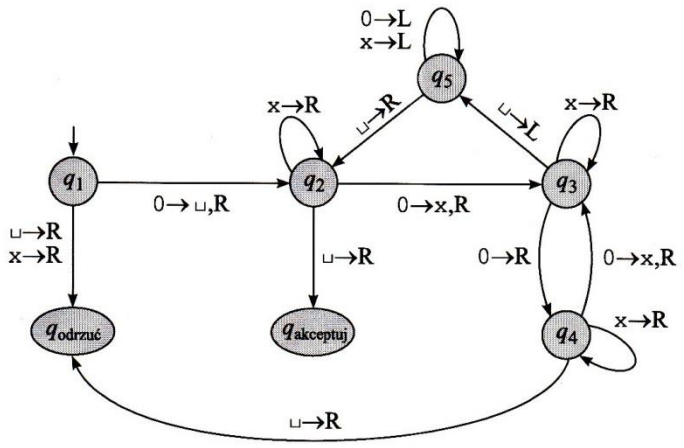
Każde wykonanie procedury opisanej w punkcie (1) zmniejsza liczbę zer o połowę. Maszyna, czytając całe słowo na tym etapie, pamięta, czy zobaczyła dotychczas parzystą, czy nieparzystą liczbę zer. Jeśli na końcu (tej procedury) liczba ta jest nieparzysta i większa od jeden, to liczba zer w słowie wejściowym, nie mogła być potęgą dwójki. W takiej sytuacji maszyna odrzuca wejście. Jeśli jednak liczba przeczytanych zer jest równa jeden, to liczba zer w pierwotnym słowie wejściowym musiała być potęgą dwójki. W takim przypadku maszyna akceptuje wejście.

3.8.4.31. **Definicja**. Podajmy teraz formalny opis $M_2 = (Q, \Sigma, \Gamma, \delta, q_1, q_{akceptuj}, q_{odrzuć})$:

- $Q = \{ q_1, q_2, q_3, q_4, q_5, q_{akceptuj}, q_{odrzuć} \}$;

[63] W pewnych źródłach języki takie są nazywane rekurencyjnymi.

- $\Sigma = \{ 0 \}$;
- $\Gamma = \{ 0, x, \sqcup \}$
- Funkcja przejścia δ jest przedstawiona za pomocą diagramu przejścia na rys. 3.8.4.02.
- Stanami początkowym, akceptującym i odrzucającym są odpowiednio q_1, $q_{akceptuj}$, $q_{odrzuć}$.



Rysunek 3.8.4.02 – Diagram stanów *Maszyny Turinga* M_2

W diagramie stanów z rys. 3.8.4.02 przy strzałce przejścia ze stanu q_1 do q_2 występuje etykieta 0 → ⊔, R. Oznacza ona, że jeśli maszyna jest w stanie q_1 i widzi symbol 0, to przechodzi w stan q_2, zapisuje na taśmie symbol ⊔ i przesuwa głowicę nad taśmą w prawo. Innymi słowy $\delta(q_1, 0) = (q_2, \sqcup, R)$, Przy przejściu ze stanu q_3 do q_4 użyliśmy krótszego zapisu 0 → R, by zaznaczyć, że maszyna w stanie q_3, widząc zero na taśmie, przesuwa głowicę w prawo, nie zmieniając symbolu na taśmie, czyli:

$$\delta(q_3, 0) = (q_4, 0, R).$$

Maszyna rozpoczyna od zapisania symbolu pustego w miejscu pierwszego z lewej zera, tak by w czasie dalszego działania, w kroku 4, mogła rozpoznać lewy koniec taśmy. Podczas gdy w innych przypadkach jako znacznika lewego końca taśmy użylibyśmy bardziej wyróżniającego się symbolu, na przykład #, tutaj użyliśmy symbolu pustego, by nie zwiększać alfabetu taśmowego i, tym samym, diagramu stanów. Uwaga: w przykładzie 3 podamy inny sposób znajdowania lewego końca taśmy.

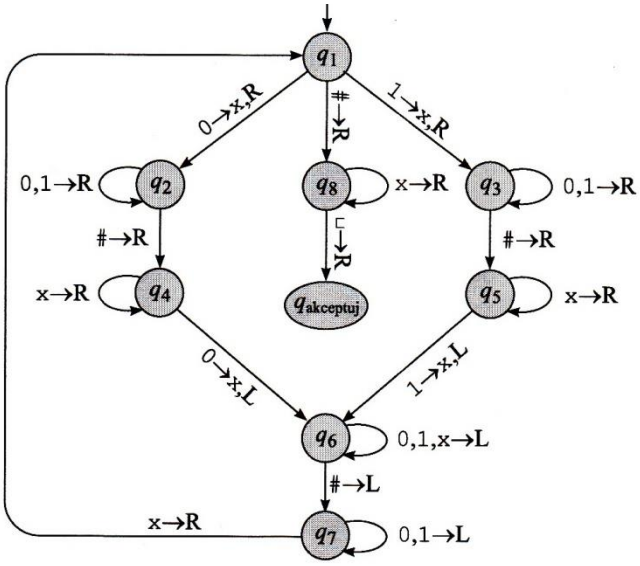
Zobaczmy, jak działa maszyna M_2 na słowie wejściowym 0 0 0 0. Konfiguracja początkowa jest równa q_1 0 0 0 0. Potem maszyna wchodzi kolejno w następujące konfiguracje (konfiguracje są zapisane kolejno w kolumnach od góry do dołu; w lewej, środkowej i prawej).

q_1 0 0 0 0	⊔ q_5 x 0 x ⊔	⊔ x q_5 x x ⊔
⊔ q_2 0 0 0	q_5 ⊔ x 0 x ⊔	⊔ q_5 x x x ⊔
⊔ x q_3 0 0	⊔ q_2 x 0 x ⊔	q_5 ⊔ x x x ⊔
⊔ x 0 q_4 0	⊔ x q_2 0 x ⊔	⊔ q_2 x x x ⊔
⊔ x 0 x q_3 ⊔	⊔ x x q_3 x ⊔	⊔ x q_2 x x ⊔
⊔ x 0 q_5 x ⊔	⊔ x x x q_3 ⊔	⊔ x x q_2 x ⊔
⊔ x q_5 0 x ⊔	⊔ x x q_5 x ⊔	⊔ x x x q_2 ⊔
		⊔ x x x ⊔$q_{akceptuj}$

3.8.4.32. **Definicja**. Opiszemy formalnie maszynę $M_1 = (Q, \Sigma, \Gamma, \delta, q_1, q_{akceptuj}, q_{odrzuć})$, którą nieformalnie opisaliśmy już wcześniej (patrz s. 258), a która rozstrzyga język B = { w # w | w ∈ {0, 1}*}.

- $Q = \{ q_1, q_2, q_3, q_4, q_5, q_{akceptuj}, q_{odrzuć} \}$;
- $\Sigma = \{ 0, 1, \# \}$ oraz $\Gamma = \{ 0, 1, \#, x, \sqcup \}$;
- Funkcja przejścia δ jest przedstawiona za pomocą diagramu przejścia na rys.3.8.5.03;
- Stanami początkowym, akceptującym i odrzucającym są odpowiednio q_1, $q_{akceptuj}$, $q_{odrzuć}$.

W diagramie stanów *Maszyny Turinga* M_1 z rys. 3.8.4.03 przy strzałce przejścia ze stanu q_3 do q_3 występuje etykieta 0, 1 → R. Oznacza ona, że jeśli maszyna jest w stanie q_3 i widzi symbol 0 lub 1, to pozostaje w stanie q_3 i przesuwa głowicę nad taśmą w prawo. Nie zmienia przy tym symbolu na taśmie.



Rysunek 3.8.4.03 – Diagram stanów *Maszyny Turinga* M_1

Stany od q_1 do q_6 służą do implementacji operacji z fazy pierwszej, a pozostałe stany do implementacji operacji z fazy drugiej. Aby uprościć diagram, pominęliśmy stan odrzucający i przejścia do niego. Takie przejścia występują niejawnie wszędzie tam, gdzie brak wyjścia ze stanu przy pewnym symbolu. Na przykład, ponieważ ze stanu q_5 nie wychodzi żadna strzałka z etykietą #, to jeśli w stanie q_5 głowica znajdzie się nad symbolem #, wówczas maszyna przejdzie do stanu $q_{odrzuć}$. Dodajmy jeszcze, że w każdym z przejść do stanu odrzucającego głowica przesuwa się w prawo.

Przykład 3. Przedstawimy teraz *Maszynę Turinga* M_3 wykonującą elementarne operacje arytmetyczne. Rozstrzyga ona język $C = \{ a^i b^j c^k \mid i \times j = k \text{ dla } i, j, k \geq 1 \}$.
M_3 = „Dla słowa wejściowego *w*:

(1) Przeczytaj słowo wejściowe od lewej do prawej, sprawdzając , czy jest postaci $a^+b^+c^+$; jeśli nie jest, to je odrzuć.
(2) Przesuń z powrotem głowicę nad lewą skrajną pozycję.
(3) Wykreśl znak *a* i przesuń głowicę w prawo do pierwszego symbolu *b*. Przesuwając głowicę wahadłowo między symbolami *b* i *c*, wykreślaj po jednym symbolu *b* i *c*. Jeśli wszystkie symbole *c* zostały wykreślone, a pozostaną symbole *b*, to odrzuć słowo wejściowe.

(4) Odtwórz wykreślone symbole *b* i powtórz krok 3, jeśli na taśmie pozostaje jeszcze symbole *a*. Jeśli wszystkie symbole *a* zostały wykreślone, sprawdź, czy także wszystkie symbole *c* zostały wykreślone. Jeśli tak, to zaakceptuj, w przeciwnym razie odrzuć słowo wejściowe."

Przeanalizujemy dokładniej cztery kroki pracy maszyny M_3. W pierwszym kroku maszyna działa jak automat skończony. Nie musi nic pisać na taśmie, tylko czyta dane wejściowe od lewej do prawej, wykorzystując stany, by sprawdzić, czy słowo to ma właściwą postać. Krok drugi wygląda równie prosto, ale kryje pewną subtelność. W jaki sposób *Maszyna Turinga* ma odnaleźć lewy koniec taśmy? Znalezienie prawego końca słowa jest proste, ponieważ na prawo od niego występuje symbol pusty. Jednak po lewej stronie słowa wejściowego nie ma początkowo żadnego szczególnego znacznika. Jednym ze sposobów rozwiązania tego problemu jest postawienie specjalnego znaku na początku słowa w chwili, gdy maszyna rozpoczyna obliczenia. Wówczas zawsze, gdy maszyna chce przesunąć głowicę nad lewy kraniec taśmy, może przesuwać się w lewo do momentu, aż znajdzie znacznik. Rozwiązanie to jest zaprezentowane w przykładzie 3, gdzie symbol pusty odgrywa rolę znacznika lewego końca taśmy.

Również lewy koniec taśmy możemy znajdować sprytniej, wykorzystując definicję *Maszyny Turinga*. Przypomnijmy, że maszyna, która chce przesunąć głowicę poza lewy koniec taśmy, pozostaje w tym samym miejscu. Możemy to wykorzystać, by skonstruować mechanizm wykrywania lewego końca taśmy. Wyobraźmy sobie, ze maszyna zawsze zapisuje na taśmie specjalny symbol na bieżącej pozycji, a symbol przez niego zamazany zapamiętuje w stanie. Następnie usiłuje przesunąć głowicę w lewo. Jeśli po tej operacji nadal widzi pod głowicą specjalny symbol, to oznacza, że ruch w lewo nie powiódł się, a więc głowica musiała być nad lewym końcem taśmy. Jeśli natomiast widzi symbol inny niż specjalny, to oznacza, że nie była wcześniej nad lewym końcem taśmy. Może więc przesuwać się dalej w lewo, ale wcześniej musi odtworzyć symbol zapisany w stanie i zamazany przez symbol specjalny.

Implementacja kroku 3 i 4 jest bezpośrednim przełożeniem opisanych procedur. Każda z nich wymaga tylko kilku dodatkowych stanów.

Przykład 3. Opiszemy teraz *Maszynę Turinga* M_4 rozwiązującą problem, który nazwiemy problemem rozróżniania elementów. Maszyna ta dostaje listę słów nad alfabetem {0, 1} oddzielonych znakami # i jej zadaniem jest zaakceptowanie, jeśli wszystkie słowa z listy są różne. Język to

$$E = \{ \#x_1\#x_2\#\ldots\#x_i|, \text{ gdzie każde } x_i \in \{0, 1\}^* \text{ oraz } x_i \neq x_j \text{ dla każdego } i \neq j \}.$$

Maszyna M_4 będzie porównywać słowo x_1 kolejno z $x_2, x_3, \ldots, x_i$; następnie będzie porównywać słowo x_2 kolejno z $x_3, x_4, \ldots, x_i$ itd. Nieformalny opis *Maszyny Turinga* M_4 jest następujący:

M_4 = „Dla słowa wejściowego w:

(1) Umieść znacznik w miejscu pierwszego symbolu na taśmie. Jeśli był to symbol pusty to zaakceptuj. Jeśli był to symbol #, to przejdź do kolejnego kroku. W przeciwnym przypadku odrzuć.

(2) Przesuń głowicę w prawo do kolejnego znaku # i napisz na jego miejscu kolejny znacznik - jeśli nie znajdziesz symbolu # przed końcem słowa (symbolem pustym), to oznacza, że na liście było tylko słowo x_1, więc zaakceptuj.

(3) Przechodząc tam i z powrotem, porównaj dwa słowa zapisane na prawo od zaznaczonych symboli #. Jeśli słowa te są równe, to odrzuć.
(4) Usuń dragi znacznik (z prawego z zaznaczonych symboli #) i wpisz go w miejscu kolejnego (na prawo) znaku #. Jeśli takiego kolejnego znaku # nie ma, to przesuń pierwszy znacznik (z lewego z zaznaczonych symboli #) na kolejny symbol # na prawo. Jeśli tym razem nie ma znaku # przed prawym końcem słowa, to oznacza, że wszystkie słowa zostały porównane, więc można zaakceptować.
(5) Wróć do kroku 3."

Przedstawiona maszyna ilustruje technikę umieszczania znaczników na taśmie. W kroku 2 maszyna pisze znacznik na miejscu innego symbolu, tym razem znak #. W przedstawionej implementacji maszyna ma dwa różne symbole: # oraz $\dot{\#}$ w alfabecie taśmowym. Mówiąc, że maszyna pisze znacznik w miejscu znaku #, mamy na myśli zastąpienie znaku # znakiem $\dot{\#}$. Usunięcie znacznika oznacza, że znak z kropką jest zastępowany pierwotnym. Możemy oczywiście chcieć zaznaczać różne symbole na taśmie. Aby to zrealizować wystarczy dołączyć do alfabetu taśmowego „zaznaczone" wersje wszystkich symboli taśmowych.

Z przedstawionych przykładów wynika, że języki A, B, C i E są rozstrzygalne. Wszystkie języki rozstrzygalne są rozpoznawalne w sensie *Turinga*, wymienione języki są więc rozpoznawalne w sensie *Turinga*. Wykazanie, że język jest rozpoznawalny w sensie *Turinga*, ale nie jest rozstrzygalny, jest trudniejsze.

3.8.4.40. **Wyjaśnienie**. Rodzaje *Maszyn Turinga* (wg Sipser'a). Istnieje wiele alternatywnych definicji *Maszyn Turinga*, w tym wersje z wieloma taśmami lub też z możliwością niedeterministycznych wyborów. Nazywamy je wariantami modelu *Maszyny Turinga*. Pierwotny model i jego podstawowe warianty mają taką samą moc obliczeniową rozpoznają tę samą klasę języków. W tym rozdziale przedstawimy kilka takich wariantów i udowodnimy ich równoważność pod względem mocy. Niezależność definicji od wyboru wariantu modelu nazywamy odpornością. Zarówno automaty skończone, jak i automaty ze stosem są modelami do pewnego stopnia odpornymi, jednak *Maszyny Turinga* są pod tym względem naprawdę nadzwyczajne. Aby zademonstrować odporność modelu *Maszyny Turinga*, zmieńmy typ dopuszczalnej funkcji przejścia. W przedstawionej definicji funkcja przejścia wymusza ruch głowicy w lewo lub w prawo po każdym kroku; głowica nie może pozostać nigdy w miejscu. Przypuśćmy, że damy *Maszynie Turinga* możliwość pozostania na miejscu. Funkcja przejścia ma wówczas postać $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$. Czy dodana własność umożliwia *Maszynom Turinga* rozpoznawanie nowych języków? Innymi słowy, czy zwiększa ona moc modelu? Oczywiście nie, ponieważ potrafimy przekształcić każdą *Maszynę Turinga*, która może pozostawiać głowicę w miejscu, w maszynę, która nie ma tej możliwości. Wystarczy zastąpić każdy krok, po którym głowica pozostaje w miejscu, przez dwa kroki: jeden w prawo i drugi, z powrotem, w lewo. Na tym małym przykładzie widzimy, jaka jest podstawowa metoda wskazywania równoważności wariantów *Maszyn Turinga*. Aby pokazać, że dwa modele są równoważne, wystarczy tylko pokazać, że potrafimy na jednym modelu symulować drugi.

3.8.4.41. **Wyjaśnienie**. Wielotaśmowe *Maszyny Turinga* (wg Sipser'a). *Wielotaśmowa Maszyna Turinga* to zwykła *Maszyna Turinga* z wieloma taśmami. Każda taśma ma własną głowicę czytająco-piszącą. Na początku obliczeń na pierwszej taśmie znajduje się słowo wejściowe, a pozostałe taśmy są puste. Funkcja przejścia jest zmieniona tak, by umożliwić jednoczesne

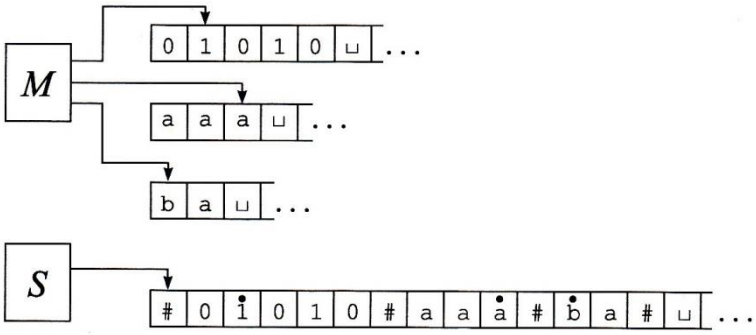
przesuwanie wielu (wszystkich lub niektórych) głowic oraz czytanie i zapisywanie na wielu taśmach. Formalnie:

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k,$$

gdzie k jest liczbą taśm. Wyrażenie

$$\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, L, P. L)$$

oznacza, że maszyna, która jest w stanie q^i widzi na kolejnych taśmach symbole $a_1, \ldots, a_k$ przechodzi do startu q_i, zapisuje na odpowiednich taśmach symbole $b_1, \ldots, b_k$ i przesuwa każdą z głowic w podanym kierunku.



Rysunek 3.8.4.04 – Zapisywanie trzech taśm na jednej taśmie

Wielotaśmowe maszyny Turinga mogą wydawać się silniejsze niż zwykłe *Maszyny Turinga*, ale pokażemy, że są tak samo mocne. Przypomnijmy, że dwie maszyny są równoważne, gdy rozpoznają ten sam język.

3.8.4.42. **Twierdzenie**. Dla każdej wielotaśmowej maszyny Turinga istnieje równoważna jej jednotaśmowa maszyna Turinga.

Dowód (wg Sipser'a). Pokażemy, jak przekształcić wielo taśmową maszynę Turinga M w równoważną jej jedno taśmową maszynę Turinga S. Istotą dowodu jest symulacja M przez S. Powiedzmy, że M ma k taśm. Wówczas S symuluje efekt posiadania k taśm, przechowując informacje ze wszystkich taśm na jednej. Maszyna S wykorzystuje symbol # do rozdzielenia zawartości poszczególnych taśm. Oprócz zawartości taśm, trzeba także pamiętać położenie głowic. Robimy to, zaznaczając kropką symbol, nad którym stoi głowica. W ten sposób otrzymujemy coś na kształt wirtualnych taśm i głowic. Oznaczone kropką symbole to, jak poprzednio, nowe symbole dodane do alfabetu taśmowego. Na rysunku 3.8.5.04 jest przedstawiony sposób zapisania trzech taśm na jednej.

S = „Dla słowa wejściowego $w = w_1 \ldots w_n$:

(1) Na początku maszyna S przekształca zapis na swojej taśmie w zapis reprezentujący wszystkie k taśm maszyny M. Jest on następujący:

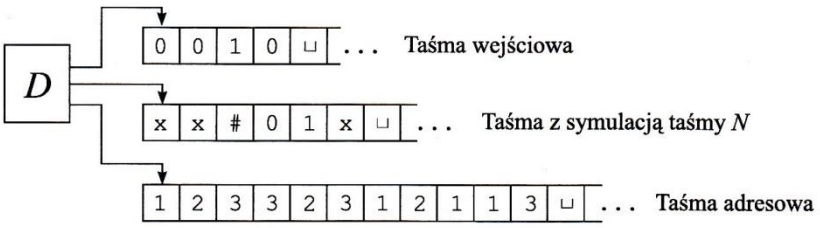
$$\# \dot{w}_1 w_2 \ldots w_n \# \dot{\sqcup} \# \dot{\sqcup} \# \ldots \#$$

(2) Aby zasymulować jeden krok, maszyna S czyta swoją taśmę od pierwszego znaku #, który oznacza lewy kraniec taśmy, do (k + 1)-szego znaku #, który oznacza prawy kraniec taśmy. Po przeczytaniu maszyna zna symbole widziane przez wirtualne głowice. Wówczas S wykonuje kolejne przejście przez całą taśmę, by zmienić zawartość taśm zgodnie z funkcją przejścia maszyny M.

(3) Jeśli w jakimkolwiek momencie maszyna S przesuwa jedną z wirtualnych głowic na prawo od znaku #, to oznacza, że maszyna M przesunęła odpowiednią głowicę nad pusty niewidziany wcześniej fragment taśmy. Maszyna S zapisuje więc symbol pusty w komórce

taśmy widzianej przez odpowiednią głowicę. Następnie przesuwa o jeden w prawo całą zawartość taśmy - od bieżącej pozycji do ostatniego po prawej znaku #. Po tej operacji maszyna S może podjąć dalej symulację." cbdo.

3.8.4.43. **Wniosek**. Język jest rozpoznawalny w sensie *Turinga* wtedy i tylko wtedy, gdy jest rozpoznawany przez pewną wielotaśmową maszynę Turinga.
Dowód (wg Sipser'a). Język rozpoznawalny w sensie Turinga jest rozpoznawany przez zwykłą (jednotaśmową) maszynę Turinga, która jest szczególnym przypadkiem wielotaśmowej maszyny Turinga. To kończy dowód wniosku w jedną stronę. Dowód wniosku w drugą stronę wynika z twierdzenia 3.8.4.42.



Rysunek 3.8.4.05 – Deterministyczna *Maszyna Turinga* D
symuluje niedeterministyczną *Maszynę Turinga* N

3.8.4.44. **Wyjaśnienie**. *Niedeterministyczna Maszyna Turinga*. Możemy spodziewać się, jak wygląda definicja *niedeterministycznej Maszyny Turinga*. W każdym momencie obliczeń maszyna ma kilka możliwości ruchu. Funkcja przejścia *niedeterministycznej Maszyny Turinga* ma następującą postać:

$$\delta : Q \times \Gamma \rightarrow P(Q \times \Gamma \times \{L, R\}).$$

Diagram wykonania obliczenia niedeterministycznej *Maszyny Turinga* to drzewo, którego gałęzie odpowiadają różnym sposobom prowadzenia obliczeń maszyny. Jeśli jakaś gałąź obliczeń prowadzi do *stanu akceptującego*, to maszyna *akceptuje słowo wejściowe*. Pokażemy, że niedeterminizm nie wpływa na moc modelu *Maszyny Turinga*.

3.8.4.45. **Twierdzenie**. Dla każdej niedeterministycznej *Maszyny Turinga* istnieje równoważna jej deterministyczna *Maszyna Turinga*.
Schemat Dowodu (wg Sipser'a). Każdą niedeterministyczną *Maszynę Turinga* N możemy symulować na deterministycznej *Maszynie Turinga* D. Ogólnie mówiąc, symulacja będzie polegać na przejściu przez maszynę D wszystkich ścieżek niedeterministycznego obliczenia maszyny N. Jeśli D znajdzie na jakiejś ścieżce stan akceptujący, to akceptuje słowo wejściowe. W przeciwnym razie maszyna D nigdy nie kończy symulacji. Obliczenie N dla słowa *w* ma postać drzewa. Każde z rozgałęzień drzewa reprezentuje niedeterministyczne rozgałęzienie obliczeń. Każdy węzeł drzewa to konfiguracja maszyny N. Korzeń drzewa to konfiguracja początkowa. Maszyna D poszukuje w drzewie konfiguracji akceptującej. Uważne wykonanie tego przeszukania jest istotne, jeśli chcemy, by maszyna D przeszukała całe drzewo. Kuszące, ale niestety złe, rozwiązanie to przeszukanie drzewa metodą w głąb (*DFS* od *depth-first search*). Przeszukanie w głąb polega na zejściu w dół jedną gałęzią do samego końca, a następnie cofnięciu się w górę drzewa do kolejnej gałęzi do przeszukania. Gdyby maszyna D przechodziła przez drzewo w ten sposób, to mogłoby się zdarzyć, że weszłaby w nieskończoną gałąź i nigdy nie przeszłaby na inną gałąź, na której mogłaby znajdować się konfiguracja akceptująca. Dlatego zaprojektujemy maszynę D tak, by przeszukiwała graf wszerz (BFS od *breadth-first search*). W tej strategii przeszukujemy wszystkie gałęzie do określonej, tej samej głębokości. Dopiero po zakończeniu

tego etapu zwiększamy głębokość i kontynuujemy przeszukanie. Taki sposób daje gwarancję, że D znajdzie konfigurację akceptującą jeśli jest ona w drzewie.

Dowód (wg Sipser'a). Deterministyczna *Maszyna Turinga* D symulująca maszynę nie-deterministyczną ma trzy taśmy. Z twierdzenia 3.8.4.45 wiemy, że taka maszyna jest równoważna maszynie z jedną taśmą. Maszyna D wykorzystuje swoje trzy taśmy w określony sposób, pokazany na rys. 3.8.5.05. Taśma pierwsza zawiera słowo wejściowe i jej zawartość nie jest nigdy zmieniana. Taśma druga zawiera kopię zawartości taśmy maszyny N będącej na pewnej ścieżce obliczeń. Na taśmie trzeciej znajduje się informacja o pozycji maszyny D w drzewie niedeterministycznych obliczeń maszyny N. Rozważmy najpierw sposób zapisu informacji na taśmie trzeciej. Kiedy wierzchołek w drzewie obliczeń może mieć najwyżej *b* dzieci, gdzie *b* jest rozmiarem największego zbioru możliwości do wyboru z funkcji przejścia maszyny N. Przypiszmy każdemu wierzchołkowi drzewa adres będący słowem nad alfabetem Σ_b : {*1, 2, .., b*}. Adres 231 przypiszemy wierzchołkowi, do którego dotrzemy, startując w korzeniu drzewa i przechodząc do jego drugiego dziecka; potem, z węzła, w którym jesteśmy, przechodzimy do jego trzeciego dziecka; na koniec, z węzła, w którym jesteśmy, przechodzimy do jego pierwszego dziecka. Każdy symbol w adresie mówi nam, jakiego kolejnego wyboru dokonać, symulując jeden krok niedeterministycznych obliczeń maszyny N. W niektórych przypadkach może nie istnieć wybór odpowiadający danemu symbolowi - dzieje się tak, gdy w danej konfiguracji jest zbyt mało możliwości do wyboru. W takim przypadku adres jest niepoprawny i nie odpowiada żadnemu węzłowi drzewa. Na taśmie trzeciej mamy słowo nad alfabetem Σ_b. Reprezentuje ono węzeł w drzewie obliczeń maszyny N, dla którego jest adresem, o ile tylko słowo to jest poprawnym adresem. Słowo puste jest adresem korzenia drzewa. cbdo.

3.8.4.46. **Definicja**. Teraz możemy opisać maszynę D (wg Sipser'a).

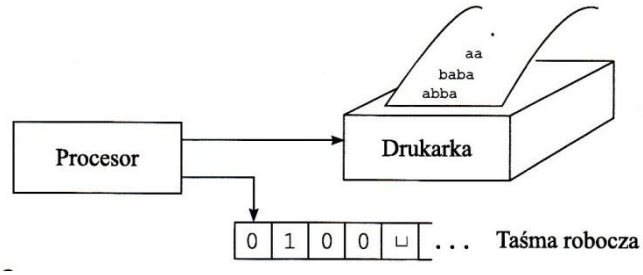
(1) Początkowo taśma pierwsza zawiera słowo wejściowe *u*, a pozostałe taśmy: druga i trzecia, są puste.
(2) Skopiuj słowo z taśmy pierwszej na drugą.
(3) Użyj drugiej taśmy do symulacji jednej ścieżki obliczeń niedeterministycznych maszyny N na słowie *w*. Przed każdym wyborem dokonywanym przez N przeczytaj kolejny symbol z taśmy trzeciej, by sprawdzić, który ruch wykonać spośród dozwolonych przez funkcję przejścia N. Jeśli jesteś już na końcu słowa na trzeciej taśmie lub aktualny symbol opisuje niepoprawny ruch, to porzuć tę ścieżkę obliczeń i przejdź do punktu (4). Do tego samego punktu przejdź, jeśli natrafisz na konfigurację odrzucającą. Jeśli natrafiłeś na konfigurację akceptującą to akceptuj słowo wejściowe.
(4) Zastąp słowo na taśmie trzeciej kolejnym słowem w porządku leksykograficznym. Rozpocznij symulację kolejnej ścieżki obliczeń maszyny N, przechodząc do punktu (2).

3.8.4.47. **Twierdzenie**. Język jest rozpoznawalny w sensie *Turinga* wtedy i tylko wtedy, gdy jest rozpoznawany przez pewną *niedeterministyczną Maszynę Turinga*.

Dowód (wg Sipser'a). Każda maszyna deterministyczna jest automatycznie maszyną niedeterministyczną więc dowód wniosku w jedną stronę jest natychmiastowy. Dowód w drugą stronę wynika z twierdzenia 3.8.4.45. Możemy zmodyfikować konstrukcję z dowodu twierdzenia 3.8.4.45 tak, by maszyna D kończyła obliczenia, jeśli maszyna N kończy obliczenia na wszystkich ścieżkach obliczeń. *Niedeterministyczną Maszynę Turinga* nazywamy *maszyną rozstrzygającą*, jeśli dla wszystkich słów wejściowych zatrzymuje się na wszystkich ścieżkach obliczeń. Można wykazać następujący wynikający z niego wniosek.

3.8.4.48. **Wniosek**. *Język jest rozstrzygalny* wtedy i tylko wtedy, gdy jest *rozstrzygany przez pewną niedeterministyczną Maszynę Turinga*.

3.8.4.50. **Wyjaśnienie**. *Enumerator* (wg Sipser'a). Wspomnieliśmy już, że czasami ubywa się określenia język rekurencyjnie przeliczalny zamiast język rozpoznawalny w sensie *Turinga*. Nazwa ta pochodzi od wariantu *Maszyny Turinga* nazwanego enumeratorem. W skrócie można powiedzieć, że enumerator to *Maszyna Turinga* z drukarką. Maszyna może używać drukarki do wypisywania wyniku - listy słów. Za każdym razem, gdy maszyna chce dodać słowo do listy wysyła je na drukarkę. W na rys. 3.8.4.06 jest przedstawiony jego schemat. Enumerator E rozpoczyna pracę z czystą taśmą wejściową. Jeśli się nie zatrzyma, może wydrukować nieskończoną listę słów. Język wyliczany przez enumerator E to zbiór wszystkich słów, które ta maszyna ostatecznie wydrukuje. Maszyna E może generować słowa w dowolnej kolejności i z powtórzeniami. Powiążemy teraz enumeratory i języki rozpoznawalne w sensie Turinga.



Rysunek 3.8.4.06 – Schemat enumeratora

3.8.4.51. **Twierdzenie**. *Język jest rozpoznawalny w sensie Turinga* wtedy i tylko wtedy, gdy jest wyliczany przez pewien enumerator.

Dowód (wg Sipser'a). Najpierw pokażemy, że jeśli mamy enumerator E, który wylicza język A, to istnieje *Maszyna Turinga* M, która rozpoznaje język A. Maszyna M działa następująco.
M – „Dla słowa wejściowego *w*:
(1) Uruchom maszynę E. Za każdym razem, gdy E drukuje słowo, porównaj je ze słowem *w*.
(2) Jeśli słowo *w* pojawi się na liście słów wydrukowanych przez E, to listę zaakceptuj."

Oczywiście widać, że maszyna M akceptuje te słowa, które pojawiają się na liście maszyny E. Przeprowadźmy teraz dowód w drugą stronę. Jeśli istnieje *Maszyna Turinga* M, która rozpoznaje język A, to możemy skonstruować następujący enumerator E dla A. Niech $s_1, s_2, s_3, \ldots$ będzie listą wszystkich słów ze zbioru Σ*.

E = „Nie zwracaj uwagi na słowo wejściowe:
(1) Powtórz poniższe punkty dla i = 1, 2, 3, ...
(2) Uruchom M na i kroków na każdym ze słów $s_1, s_2, \ldots, s_i$.
(3) Jeśli którekolwiek z tych obliczeń zakończy się akceptacją to wydrukuj na wyjściu słowo s_j, dla którego było wykonywane to obliczenie."

Jeśli maszyna M akceptuje słowo *s*, to pojawi się ono w końcu na liście generowanej przez E. W rzeczywistości słowo to pojawi się nieskończenie wiele razy, ponieważ maszyna M rozpoczyna obliczenia od początku dla każdego słowa za każdym razem, gdy powraca do punktu (1). Działanie procedury można porównać do równoległego uruchomienia maszyny M na wszystkich możliwych słowach wejściowych.

3.8.4.52. **Wyjaśnienie**. Równoważność z innymi modelami *Maszyny Turinga* (wg Sipser'a). Przedstawiliśmy do tej pory kilka wariantów modelu *Maszyny Turinga* i pokazaliśmy, że mają one jednakową moc. Istnieje także wiele innych ogólnych modeli obliczeń, Niektóre z nich bardzo przypominają *Maszyny Turinga*, ale inne wyraźnie się od nich różnią. Wszystkie mają jednak jedną cechę wspólną z *Maszyną Turinga* - nieograniczony dostęp do pamięci nieograniczonego rozmiaru, co różni je od słabszych modeli, takich jak automaty skończone i automaty ze stosem. Warto zaznaczyć, że wszystkie modele mające tę własność mają równoważną moc obliczeniową jeśli tylko spełniają pewne dodatkowe, sensowne założenia[64].

3.8.4.53. **Wyjaśnienie**. Aby pojąć to zastanawiające zjawisko, rozważmy analogiczną sytuację dla języków programowania. Wiele z nich, na przykład C czy JAVA różni się od siebie zasadami działania i strukturą. Powstaje pytanie: *Czy istnieją algorytmy, które można zaprogramować w jednym z języków, a nie można w drugim?* Oczywiście, że nie - przecież możemy przetłumaczyć program napisany w JAVA na program w C i na odwrót. To oznacza, że oba te języki opisują dokładnie takie same klasy algorytmów; podobnie jak wszystkie pozostałe, sensowne języki programowania. Powody równoważności szerokiej klasy modeli obliczeniowych są dokładnie takie same. *Każde dwa modele obliczeniowe spełniające pewne rozsądne założenia mogą się nawzajem symulować, a więc mają tę samą moc.* Konsekwencją opisanej równoważności jest ważny wniosek natury filozoficznej. Mimo, że możemy wyobrazić sobie wiele różnych modeli obliczeniowych, to *klasa algorytmów* przez nie opisywana *pozostaje niezmieniona*. Podczas gdy każdy model obliczeń ma pewne charakterystyczne, swoiste cechy, to związana z nim *klasa algorytmów* jest tą samą *naturalną klasą*, która jest związana ze wszystkimi pozostałymi modelami. Zjawisko to ma poważne konsekwencje w matematyce, co pokażemy w kolejnym podrozdziale.

3.8.4.60. **Wyjaśnienie**. Kolejnym krokiem, było zaprojektowanie – *Uniwersalnej Maszyna Turinga* w 1938 roku, przez *Alana Turinga*, abstrakcyjny model maszyny logicznej (jakbyśmy dzisiaj powiedzieli - rodzaju komputera), o zmiennej strukturze logicznej (liczbie stanów i funkcji przejścia – kształtowanych w zależności od potrzeb) służącej do wykonywania *algorytmów*. Uniwersalna Maszyna Turinga składająca się z:

(1) układu sterowania przyjmującego pewną liczbę nieograniczoną stanów Q (nie mniej jednak niż trzy stany: początkowy q_1, akceptujący q_{akceptuj} i odrzucający $q_{\text{odrzuć}}$) w zależności od potrzeb zdania;
(2) głowicy czytająco piszącej – która wykonuje ruch {R, L, S};
(3) alfabetu ∑ zawierającego co najmniej dwa znaki, z którego tworzone są słowa;
(4) zbioru słów języka budowanego z ciągów znaków alfabetu;
(5) nieskończenie długiej taśmy podzielonej na kolejne komórki (pola):
(6) programowanej funkcji przejścia od stanu do stanu, którą realizuj układ sterowania głowicą:
(7) znacznika komórki pustej ⊔, nie należącego do alfabetu.

Maszyna Turinga powstała - jako notacja umożliwiająca opisywanie algorytmów i badanie ich wykonywania. Taśma może być nieskończona jednostronnie lub obustronnie. Każda komórka

[64] Takim- założeniem jest na przykład, wymaganie, by maszyna mogła w jednym kroku wykonać tylko określoną skończoną pracę.

taśmy, może zawierać: znak pusty albo zawierać wpisany znak alfabetu ∑. Głowica maszyny zawsze jest ustawiona nad jednym z pól.

Stan układu sterowania maszyny określa jeden ze stanów $q_i \in Q$. Nowy stan układu sterowania po wykonaniu instrukcji zależnie od kombinacji:

d. Aktualnie wykonywanej instrukcji;
e. Stanu układu sterowania maszyny oraz
f. Zawartości komórki, nad którym znajduje się głowica.

Wykonując instrukcję – *Uniwersalna Maszyna Turinga*: (i) - odczytuje zawartość komórki taśmy nad którą znajduje się aktualnie głowica; (ii) – analizuje stan układu sterowania q_i; (iii) - zapisuje nową wartość w komórce; (iv) – zmienia stan układu sterowania na stan q_j; a następnie (v) może przesunąć głowicę: jedno pole w prawo R, albo w lewo L, albo pozostaje na dotychczasowym miejscu S.

Taka operacja nazywana jest *instrukcją funkcji przejścia* δ. *Uniwersalna Maszyna Turinga* jest sterowana programem, wykonywanym przez układ sterowania, zawierającym dowolną skończoną liczbę takich instrukcji[65]. Maszyna rozpoczynając działanie każdego ze swoich programów od stanu początkowego q_1, w którym znajduje się układ sterowania maszyny, oraz pierwszego znaku – tzw. słowo wejściowego (zapisanego na pierwszych *m* polach taśmy), w pozostałych polach taśmy wpisany jest znak ⊔ (słowo wejściowe – zapisane na pierwszych polach taśmy, jest odpowiednikiem słowa na wejściu automatu skończonego). Maszyna kończy działanie każdego programu, albo w jednym z dwu stanów układów sterowania: stan akceptujący q_a albo stan odrzucający q_r.

3.8.4.70. **Definicja**. Uniwersalna Maszyna Turinga opisuje się, jako zbiór uporządkowany,

$$UMT = \{ \sum, Q, q_0, q_a, q_r, \sqcup, M(R, L, S), \delta \}$$ - gdzie:

<maszyna UMT>
<alfabet >∑</alfabet >
<skończony zbiór stanów> Q </skończony zbiór stanów>
<stan początkowy > $q_1 \in Q$ </stan początkowy>
<stan akceptujący > $q_{akceptuj} \in Q$ </stan akceptujący>
<stan odrzucający > $q_{odzuć} \in Q$, gdzie $q_{akceptuj} \neq q_{odzuć}$ </stan odrzucający>
<zawartość pusta komórki taśmy> $\sqcup \notin \sum$ </zawartość pusta komórki taśmy>
<ruchy głowicy > $m \in M$, powodujących przesunięcie głowicy taśmy
<głowica>
<komórka taśmy w lewo> $R \in M$ </komórka taśmy w lewo>
<komórka taśmy w prawo> $L \in M$ </komórka taśmy w prawo>
<komórka taśmy bez zmian> $S \in M$ </komórka taśmy bez zamian>
</głowica>
</ruchy głowicy>
<funkcja przejścia>$\delta : \sum \times Q \rightarrow Q \times \sum \times M$ </funkcja przejścia>
</maszyna UMT>

Uwaga: Oznacza to, że δ jest funkcją pobierającą aktualny stan układu sterowania maszyny oraz symbol alfabetu odczytany z komórki taśmy, decyduje - jaki symbol alfabetu ma być zapisany w danej komórce taśmy, jaki jest kolejny stan układu sterowania maszyny oraz jakie ma być

[65] W odróżnieniu jednak od komputerów o architekturę *von Neumann'a*, *Uniwersalna Maszyna Turinga* nie może wykonywać instrukcji zmieniających strukturę działania swego sterowania.

przesunięcie głowicy maszyny nad nieskończoną taśmą (L - lewo, R - prawo lub S - bez przesunięcia).

3.8.4.71. **Wyjaśnienie**. Przykładowa instrukcji *Uniwersalnej Maszyny Turinga* (a, q_0, b, q_0,M), działa w sposób następujący:

1. Odczytanym przez głowicę symbolem z taśmy jest litera **a**, a układ sterowania znajduje się w stanie q_1, to głowica zamieni ten symbol na **b**, stan wewnętrzny nie zmieni się (pozostanie dalej q_1), a głowica przesunie się do sąsiedniej komórki po prawej stronie.
2. Pierwsze dwa argumenty – określają jednoznacznie instrukcję. Pozostałe trzy określają, co w ramach tej instrukcji należy zrobić, czyli jaki symbol umieścić w bieżącej komórce taśmy, w jaki nowy stan przechodzi układ sterowania i w którą stronę przesunąć głowicę M = {L, R}.

Ogólnie można powiedzieć:

- Programem dla *Uniwersalnej Maszyny Turinga*, jest tablica (wbudowaną w układ sterowania głowicą), w której określamy wszystkie wykonywalne przez nią instrukcje. Kolejność występowania tych instrukcji w żaden sposób nie jest istotna.
- *Uniwersalna Maszyna Turinga* - rozpoznaje instrukcje po symbolu z taśmy i swoim stanie wewnętrznym. Jeśli w tablicy (wbudowanej w układ sterowania głowicą) zabraknie dla tej kombinacji odpowiedniej instrukcji, to program zatrzymuje się.

3.8.4.72. **Wyjaśnienie**. W celu lepszego zrozumienia działania *Uniwersalnej Maszyny Turinga* o trzech wyróżnionych stanach układu sterowania: {q_1,$q_{akceptuj}$, $q_{odrzuć}$} oraz o alfabecie złożonym z dwu znaków: {0, 1}. Głowica maszyny znajduje się nad skrajną prawą komórką słowa startowego. Rozważmy następujący program złożony z trzech instrukcji - wbudowanych w układ sterowania głowicą. Program ten zastępuje zawartości komórek w których zapisane jest 0 przez 1 i odwrotnie (działanie poniższego programu, patrz tablica 3.8.4.01):

<program>
<inst. 1>δ(0, q_0 ,1, q_0, L) - bit = 0 zamień na 1 </inst. 1>
<inst. 2>δ(1, q_0, 0, q_0, L) - bit = 1 zamień na 0 </inst. 2>
<inst. 3>δ(⊔, q_0,⊔,q_a, L) - bit = ⊔ zmień stan z q_0 na q_a i zakończ pracę </inst. 3>
</program>

Tablica 3.8.4.01 – Przykład działania Uniwersalnej Maszyny Turinga.

Zawartość taśmy	Znak odczytany	Stan bieżący	Wykonywana operacja
⊔ 1 0 1 1 **<u>0</u>** ⊔	0	q_1	Kombinacja odczytanego znaku i stanu q_0 wyznacza instrukcję 0, q_1,1,q_1, L. Zatem znak w bieżącej komórce maszyna Turinga umieści symbol 1, stanu nie zmieni (wciąż pozostanie w q_1) i przemieści głowicę do sąsiedniej komórki po lewej stronie.
⊔ 1 0 1 **<u>1</u>** 1 ⊔	1	q_1	Teraz kombinacja odczytanego znaku i stanu wewnętrznego wyznacza instrukcję 1, q_1, 0, q_1, L. Znak w bieżącej komórce taśmy zostanie zastąpiony znakiem 0, stan wewnętrzny nie zmieni się i głowica będzie przesunięta w lewo do następnej komórki.
⊔ 1 0 **<u>1</u>** **0** 1 ⊔	0	q_1	Instrukcja 0, q_1,1, q_1, L
⊔ 1 **<u>1</u>** 0 0 1 ⊔	1	q_1	Instrukcja 1, q_1, 0, q_1, L
⊔ **<u>0</u>** 1 0 0 1 ⊔	⊔	q_1	Instrukcja ⊔,q_1,⊔,$q_{akceptuj}$, L - zmień stan z q_1 na $q_{akceptuj}$ i zakończ pracę!

Dotychczas mówiliśmy jedynie o trzech podstawowych stanach q_1, $q_{akceptuj}$ oraz $q_{odrzuć}$. W rzeczywistości Uniwersalna Maszyna Turinga może tworzyć wiele różnych stanów w toku wykonywania programu.

Kolejna przykładowa Uniwersalnej Maszyny Turinga, jest przeznaczona do przesuwania bitów liczby binarnej o 1 w lewo, pojawia się w niej konieczność użycia dodatkowego stanu, który nazwiemy q_2 (patrz tablica 3.8.4.02.). Program ten zawiera sześć instrukcji:

<program przesuwania o 1 bit w lewo>

<inst. 1> $\delta(0, q_1, 0, q_1, L)$ stan q_0 (zapamiętujemy 0 – dalej q_1) </inst. 1>

<inst. 2> $\delta(1, q_1, 0, q_2, L)$ stan q_0(zapamiętujemy 1 – nowy stan q_1) </inst. 2>

<inst. 3> $\delta(0, q_2, 1, q_1, L)$ zapisujemy zapamiętane 1 i zapamiętujemy 0 (stan q_0) </inst. 3>

<inst. 4> $\delta(1, q_2, 1, q_2, L)$ zapisujemy zapamiętane 1 i zapamiętujemy 1 (stan q_1) </inst. 4>

<inst. 5> $\delta(\sqcup, q_1, 0, q_{akceptuj}, L)$ zapisujemy zapamiętane 0, nowy stan q_a - koniec </inst. 5>

<inst. 6> $\delta(\sqcup, q_2, 1, q_{akceptuj}, L)$ - zapisujemy zapamiętane 1, nowy stan q_a– koniec </inst. 6>

</program przesuwania o 1 bit w lewo>

Tablica 3.8.4.02 – Przykład działania programu na Maszynie Turinga.

Taśma z głowicą	Odczytany znak	Stan bieżący	Wykonywana instrukcja	Zapisywany znak	Nowy stan
⊔⊔ 1 1 0 1 0 **1** ⊔	1	q_1	$\delta(1, q_1, 0, q_2, L)$	0	q_1
⊔⊔ 1 1 0 1 **0** 0 ⊔	0	q_2	$\delta(0, q_2, 1, q_1, L)$	1	q_0
⊔⊔ 1 1 0 **1** 1 0 ⊔	1	q_1	$\delta(1, q_1, 0, q_2, L)$	0	q_1
⊔⊔ 1 1 **0** 0 1 0 ⊔	0	q_2	$\delta(0, q_2, 1, q_1, L)$	1	q_0
⊔⊔ 1**1** 1 0 1 0 ⊔	1	q_1	$\delta(1, q_1, 0, q_2, L)$	0	q_1
⊔⊔**1**0 1 0 1 0 ⊔	1	q_2	$\delta(1, q_2, 1, q_2, L)$	1	q_1
⊔⊔1 0 1 0 1 0 ⊔	⊔	q_2	$\delta(\sqcup, q_2, 1, q_{akceptuj}, L)$	1	$q_{akceptuj}$

3.8.4.80. **Wyjaśnienie**. *Uniwersalna Maszyna Turinga* A posiadająca zdolność symulacji działania dowolnej innej *Maszyny Turinga* B (opisanej, jako dane wejściowe dla maszyny A) na dowolnych danych wejściowych dla maszyny B. Praktycznym przybliżeniem realizacji *Uniwersalnej Maszyny Turinga* jest *komputer*, będący w stanie wykonać dowolny *program* na dowolnych danych. Jednak komputery nie są *Uniwersalnymi Maszynami Turinga*, w sensie pierwotnej definicji, ponieważ ilość danych, które mogą przechowywać i przetwarzać jest skończona, tak więc - dla każdego komputera istnieje tylko skończona liczba programów, które może wykonać. Mimo że liczba ta jest niewyobrażalnie wielka i w praktyce często wystarczająca, to bez względu na rozmiar pamięci, zawsze będzie istnieć program, którego maszyna nie będzie w stanie wykonać, ponieważ jego kod (opis) po prostu nie mieści się w tej pamięci.

Mimo że *Uniwersalna Maszyna Turinga* jest jedynie abstrakcją o teoretycznej dużej mocy obliczeniowej (większej ze względu na nieograniczoną pojemność pamięci, niż dowolny komputer), istnieje wiele problemów (np. *problem stopu*), których nie da się na niej rozwiązać. John von Neuman projektując swój komputer (prototyp współczesnego komputera) zwrócił uwagę, że w przeciwieństwie do Uniwersalnej Maszyny Turinga, jego komputer, podobnie jak wszystkie współczesne komputery, może przetwarzać nie tylko dane, ale również własne programy – sterujące pracą tegoż komputera.

Piśmiennictwo: *Kisielewicz A.* K.1.1., *Sipser M.* S.7.1.

3.8.5. ZŁOŻONOŚĆ CZASOWA I ANALIZA ALGORYTMÓW

Nawet gdy problem jest rozstrzygalny, a więc zasadniczo rozwiązywalny w sensie obliczeniowym, w praktyce może być nierozwiązywalny, jeśli rozwiązanie wymaga niezwykle długiego czasu lub niezwykle wielkiej pamięci. Zasadniczy jest jednak czas i dlatego pominiemy sprawę ograniczeń wynikających z pojemności pamięci. W tym podrozdziale omówimy *teorię obliczeniowej złożoności czasu* - czyli badania czasu wymaganego do rozwiązania problemów

obliczalnych. Naszym celem będzie zaprezentowanie podstaw tzw *teorii złożoności czasowej*. Na początku zajmiemy się metodą pomiaru czasu wykorzystywanego do rozwiązywania problemu. Następnie pokażemy, jak klasyfikować problemy w zależności od ilości czasu wymaganego do ich rozwiązania. Potem przeanalizujemy hipotezy o istnieniu problemów rozstrzygalnych, których rozwiązanie wymaga najprawdopodobniej nieskończenie długiego czasu, zastanowimy się, jak rozpoznać, czy mamy do czynienia z takim problemem.

3.8.5.10. **Wyjaśnienie**. *Pomiar złożoności wg Sipser'a*. Rozpocznijmy od przykładu. Rozważmy język A : $\{0^k1^k|k \leq 0\}$[66]. Oczywiście język A jest rozstrzygalny. Ile czasu potrzebuje *Maszyna Turinga*, by go rozstrzygnąć? Przeanalizujmy następującą jedno taśmową *Maszynę Turinga* M_1 dla języka A. Podamy opis tej maszyny na niskim poziomie, uwzględniając przesunięcia głowicy na taśmie, tak byśmy mogli policzyć liczbę kroków, które wykonuje maszyna M_1 w czasie pracy realizując poniższy algorytm - „Dla słowa wejściowego *w*:

(1) Przejdź przez całą taśmę i odrzuć, jeśli na prawo od jedynek znajduje się jakieś zero.
(2) Dopóki na taśmie pozostają zera i jedynki, powtarzaj:
(3) Przejrzyj taśmę, wykreślając jedno zero i jedną jedynkę.
(4) Jeśli po wykreśleniu wszystkich jedynek na taśmie pozostało jakieś zero, to odrzuć. Postąp podobnie, jeśli po wykreśleniu wszystkich zer na taśmie pozostała jakaś jedynka. Jeżeli na taśmie nie zostało ani zera, ani jedynki, to zaakceptuj."

Przeanalizujmy algorytm *Maszyny Turinga* M_1 rozstrzygającej język A, by określić, jaki jest jego czas działania. Liczba kroków, które algorytm wykonuje dla danego słowa wejściowego, może zależeć od kilku parametrów. Jeśli, na przykład, dane wejściowe to graf, wówczas liczba kroków może zależeć od liczby wierzchołków, liczby krawędzi, maksymalnego stopnia wierzchołka w grafie lub też kombinacji tych czynników. Dla uproszczenia czas działania algorytmu będziemy obliczać jako funkcję długości słowa wejściowego, nie zwracając uwagi na pozostałe parametry. W analizie złożoności najgorszego przypadku, którą tu będziemy rozpatrywać, przyjmujemy złożoność najdłuższy czas działania algorytmu dla wszystkich słów określonej długości. W analizie złożoności uśredniego przypadku - przyjmuje się za złożoność, średni czas działania algorytmu, obliczony dla wszystkich słów określonej długości.

3.8.5.11. **Definicja** pojęcia czas działania wg Sipser'a. Niech M będzie deterministyczną *Maszyną Turinga*, która zatrzymuje się dla wszystkich wejść. Czas działania lub złożoność czasowa maszyny M to funkcja $f : N \rightarrow N$, gdzie f(n) jest maksymalną liczbą kroków, które wykonuje maszyna M dla dowolnego słowa wejściowego długości n. Jeśli f(n) jest czasem działania maszyny M, to mówimy, że M działa w czasie f(n) i że jest f(n) - czasową *Maszyną Turinga*. Do oznaczenia długości słowa wejściowego używamy litery n.

3.8.5.12. **Wyjaśnienie**. *Notacja duże* O *i małe* o. Zazwyczaj dokładny czas działania algorytmu jest opisany złożonym wyrażeniem, często więc poprzestajemy na jego przybliżeniu. Jeden z wygodnych sposobów szacowania czasu działania algorytmu, nazywany analizą asymptotyczną, polegającym na zrozumieniu, ile czasu działa algorytm dla jak długich słów wejściowych. Efekt ten uzyskujemy, rozwijając najbardziej znaczące składniki (termy) w wyrażeniu opisującym czas działania algorytmu, pomijając zarówno współczynniki stałe przy tym składniku, jak i składniki mniej znaczące. Wynika to ze spostrzeżenia, że składniki znaczące istotnie dla dużych argumentów - dominują nad pozostałymi. Na przykład, funkcja $f(n) = 6n^3 + 2n^2 + 20n + 45$; ma

[66] *Znacznik k* – oznacza symboliczny zapis krotności wystąpienie zer i jedynek w słowach należących do języka.

cztery termy i najbardziej znaczący z nich to $6n^3$. Pomijając współczynnik 6, możemy powiedzieć, że funkcja *f* jest asymptotycznie ograniczona przez n^3. *Ograniczenie asymptotyczne* lub *notacja z dużym* O umożliwia opisanie rzędu wielkości tej funkcji jako $f(n) = O(n^3)$. Sformalizujemy ten zapis w poniższej definicji. Niech R^+ będzie zbiorem nieujemnych liczb rzeczywistych. Intuicyjnie, $f(n) = O(g(n)))$ oznacza, że funkcja f jest mniejsza lub równa od funkcji g, jeśli przy porównaniu nie będziemy zwracać uwagi na stałe współczynniki. Można uważać symbol O za reprezentację pominiętej stałej. Dla większości funkcji, z którymi będziemy się spotykać w praktyce, najbardziej znaczący składnik h jest łatwy do rozpoznania. W takich przypadkach będziemy pisać, że $f(n) = O(g(n))$, gdzie g jest składnikiem h bez stałych współczynników.

Przykład 1 (wg Sipser'a). Niech $f_1(n)$ będzie funkcją $5n^3 + 2n^2 + 22n + 6$. Wybierając najbardziej znaczący składnik $5n^3$ oraz pomijając współczynnik 5, dostajemy $f_1(n) = O(n^3)$. Sprawdźmy, że jest to zgodne z formalną definicją. W tym celu przyjmijmy $c = 6$ i $n_0 = 10$. Wówczas:

$$5n^3 + 2n^2 + 22n + 6 \leq 6n^3 \text{ dla każdego } n \geq 10.$$

Dodatkowo $f_1(n) = O(n^4)$, ponieważ funkcja n^4 jest większa niż n^3, więc także jest asymptotycznym górnym ograniczeniem f_1. Jednak $f_1(n)$ nie jest $O(n^2)$, bowiem niezależnie od tego, jakie wartości przypisalibyśmy c i n_0, nie da się spełnić w tej sytuacji warunków definicji.

3.8.5.13. **Definicja**. Niech f i g będą funkcjami $f, g : N \to R^+$. Powiemy, że $f(n) = O(g(n))$, jeśli istnieją dodatnie liczby całkowite c i n_0 takie, że dla każdej liczby całkowitej $n \geq n_0$ zachodzi

$$f(n) \leq cg(n).$$

Gdy $f(n) = O(g(n))$, mówimy, że g(n) jest górnym ograniczeniem f (n) czy też dokładniej, że g(n) jest *asymptotycznym górnym ograniczeniem* f (n), by podkreślić, że nie bierzemy pod uwagę stałych współczynników.

Przykład 2 (wg Sipser'a). Notacja O działa ciekawie w przypadku logarytmów. Zazwyczaj, pisząc funkcję logarytmiczną musimy podać jej podstawę, na przykład $x = \log_2 n$. Podstawa 2 oznacza, że w tym przypadku zachodzi równość $n = 2^x$. Zmiana podstawy logarytmu powoduje zmianę jego wartości o stały czynnik, co wynika z równości $\log_b n = \log_2 n / \log_2 b$. Stąd, pisząc $f(n) = O(\log n)$, nie musimy określać podstawy logarytmu, bo stały współczynnik i tak jest pomijany. Niech $f_2(n)$ będzie funkcją:

$$3\, n \log_2 n + 5\, n \log_2(\log_2 n) + 2.$$

W tym przypadku zachodzi $f_2(n) = O(n \log n)$, gdyż funkcja log n przeważa nad log (log n). Notacji wielkie O możemy używać także w wyrażeniach arytmetycznych, jak w przykładzie:

$$f(n) = O(n^2) + O(n).$$

W takim przypadku z każdym wystąpieniem symbolu O wiąże się inna pominięta stała. Ponieważ składnik $O(n^2)$ dominuje nad składnikiem $O(n)$, więc podane wyrażenie jest równoważne następującemu: $f(n) = O(n^2)$. Symbol O może także występować w wykładniku, jak w przypadku $f(n) = 2^{O(n)}$. Wówczas ma takie samo znaczenie jak poprzednio - oznacza pominiętą stałą czyli podane wyrażenie jest górnym ograniczeniem dla 2^{cn}, gdzie c jest dowolną stałą. Wyrażenie $f(n) = 2^{O(\log n)}$ może pojawić się w pewnych przypadkach analizy złożoności. Wykorzystując równość $n = 2^{\log n}$, widzimy, $2^{O(\log n)}$ jest górnym ograniczeniem dla n^c dla dowolnej stałej c. Wyrażenie $n^{O(1)}$ jest innym przedstawieniem tego samego ograniczenia. ponieważ wyrażenie O(1) reprezentuje wartość, która jest nie większa niż pewna ustalona stała. Często w analizie pojawiają się ograniczenia postaci n^c dla pewnej stałej $c > 0$. Takie ograniczenia nazywamy ograniczeniami wielomianowymi. Ograniczenia postaci $2^{n\wedge 8}$ nazywamy ograniczeniami wykładniczymi, gdy δ jest liczbą rzeczywistą większą niż 0. Z notacją duże O jest

związana notacja małe o. Symbol duże O mówi nam, że funkcja jest asymptotycznie nie większa niż inna funkcja. Aby powiedzieć, że jedna funkcja jest asymptotycznie mniejsza niż inna funkcja, stosujemy notację małe o. Różnica między notacją duże O i małe p jest analogiczna do różnicy między relacjami $\leq$ i <.

3.8.5.14. **Definicja**. Definicja. Niech f i g będą funkcjami f, g: $N \to R^+$. Powiemy, że f(n) = o(g(n)), jeśli

$$\lim_{n \to \infty} [f(n)/g(n) = 0]$$

Inaczej mówiąc, f (n) = o(g(n)) oznacza, że dla dowolnej liczby rzeczywistej c > 0 istnieje liczba n_0 taka, że f (n) $\leq$ c g(n) dla wszystkich n $\geq n_0$.
Przykład 3. Następujące równości można łatwo sprawdzić:
(1) $\sqrt{n}$ = o(n);
(2) n = o(n log (log n));
(3) n log (log n) = o(n log n);
(4) n log n = o(n^2);
(5) n^2 = o(n^3).
Jednak nigdy nie zachodzi równość f (n) = o(f (n)).

3.8.5.15. **Wyjaśnienie**. Analiza algorytmów wg Sipser'a. Przeanalizujmy algorytm *Maszyny Turinga*, który podaliśmy dla języka A = $\{0^k1^k | k \geq 0\}$. Dla wygody przypomnimy algorytm ponownie w tym miejscu:
M_1 = „Dla słowa wejściowego w:
(1) Przejrzyj całą taśmę i odrzuć, jeśli na prawo od jedynek znajduje się jakieś zero.
(2) Dopóki na taśmie pozostają zera i jedynki, powtarzaj.
(3) Przejrzyj taśmę, wykreślając jedno zero i jedną jedynkę.
(4) Jeśli po wykreśleniu wszystkich jedynek na taśmie pozostało jakieś zero, to odrzuć. Postąp podobnie, jeśli po wykreśleniu wszystkich zer na taśmie pozostała jakaś jedynka. Jeżeli na taśmie nie zostaje ani zera, ani jedynki, to zaakceptuj."

Aby przeanalizować działanie M_1, rozważmy oddzielnie każdy z czterech kroków. W kroku (1) maszyna przechodzi całą taśmę, sprawdzając, czy słowo wejściowe ma postać 0*1*. Wykonuje to w n krokach - wspomnieliśmy wcześniej, że długość danych wejściowych będziemy zazwyczaj oznaczać przez n. Przestawienie głowicy na początek taśmy wymaga kolejnych n kroków. W sumie wykonaliśmy więc 2n kroków. Stosując notację wielkie O, powiemy, że w tej fazie wykonaliśmy O(n) kroków. Zauważmy, że w opisie maszyny nie wspomnieliśmy o przestawieniu głowicy na początek taśmy. Notacja asymptotyczna pozwalała nam bowiem pomijać szczegóły opisu maszyny, które zmieniają jej czas działania jedynie o stały czynnik.

W fazach (2) i (3) maszyna powtarza przeglądanie taśmy, przesuwając głowicę w przód i z powrotem oraz wykreślając po jednym zerze i jedynce w każdym przejściu. Każde przejście wymaga O(n) kroków. W każdym przejściu wykreślamy dwa symbole, więc może wystąpić najwyżej n/2 przejść. Całkowity czas wymagany do realizacji fazy (2) i (3) to (n/2) O(n) = O(n^2) kroków. W fazie (4) maszyna jednokrotnie przegląda taśmę, by zdecydować, czy należy zaakceptować, czy odrzucić. Zajmuje to najwyżej czas O(n).

Przyjrzyjmy się teraz parzystości/nieparzystości liczby zer i jedynek w każdym z wykonań fazy (3). Rozważmy ponownie sytuację początkową z 13 zerami i 13 jedynkami. W pierwszym wykonaniu fazy (3) stwierdzimy, że liczba zer jest nieparzysta (ponieważ 13 jest liczbą

nieparzystą) i liczba jedynek jest nieparzysta. W kolejnym wykonaniu pojawia się liczba parzysta 6, potem liczba nieparzysta 3 i na koniec liczba nieparzysta 1. Dla zerowej liczby zer i jedynek nie wykonujemy już kroku (3), gdyż warunek pętli podany w fazie (2) spowoduje jej zakończenie. Dla napotkanego w trakcie wykonania algorytmu ciągu liczb występujących w fazie (3) możemy zastąpić wystąpienia liczb parzystych zerami, a nieparzystych - jedynkami. Po odwróceniu otrzymanego w ten sposób ciągu dostaniemy ciąg 1101, czyli binarną reprezentację liczby 13, która jest początkową liczbą zer i jedynek. Nie jest to przypadek, dzieje się tak zawsze, dla dowolnej początkowej liczby zer i jedynek. Gdy w fazie (3) sprawdzamy, czy całkowita liczba pozostałych zer i jedynek jest parzysta, w rzeczywistości sprawdzamy, czy liczba pozostałych zer i liczba pozostałych jedynek mają taką samą parzystość. Jeśli parzystość ta zgadza się w każdym wykonaniu kroku (3), to binarne reprezentacje liczb oznaczających liczbę zer i liczbę jedynek są jednakowe, więc liczby są sobie równe.

3.8.5.20. **Wyjaśnienie**. Aby zbadać czas działania maszyny M_2, zauważmy najpierw, że każdy krok wymaga czasu O(n). Określmy następnie, ile razy jest wykonywany każdy krok. Fazy (1) i (5) są wykonywane jednokrotnie, więc wymagają całkowitego czasu O(n). W fazie (4) za każdym razem wykreślamy co najmniej połowę zer i jedynek, możemy więc wykonać tę fazę najwyżej $1 + \log_2 n$ razy, zanim wykreślimy wszystkie znaki na taśmie. Sumaryczny czas wykonania faz (2), (3) i (4) jest więc równy $(1 + \log_2 n)\ O(n)$, czyli $O(n \log n)$. Czas działania maszyny M_2 wynosi zatem $O(n) + O(n \log n) + O(n \log n)$.

Można pokazać, że $A \in TIME(n^2)$, a teraz potrafimy otrzymać lepsze ograniczenie, mianowicie $A \in TIME(n \log n)$. Tego wyniku nie da się już poprawić dla jedno taśmowej maszyny Turinga. W rzeczywistości każdy język, który można rozstrzygnąć w czasie $o(n \log n)$ na jednotaśmowej maszynie Turinga, jest regularny. Język A możemy rozstrzygnąć w czasie O(n) (nazywanym także czasem liniowym), jeśli maszyna Turinga ma drugą taśmę. Następująca dwutaśmowa maszyna Turinga M_3 rozstrzyga język A w czasie liniowym. Maszyna M_3 działa inaczej niż poprzednie maszyny dla języka A. Po prostu kopiuje zera na drugą taśmę, a następnie porównuje je z jedynkami. M_3 = „Dla słowa wejściowego <w>:

(1) Przejrzyj całą taśmę i odrzuć, jeśli na prawo od jedynek znajduje się jakieś zero.
(2) Przejdź przez zera na taśmie pierwszej, aż zobaczysz pierwszą jedynkę. Przechodząc, kopiuj jednocześnie zera na drugą taśmę.
(3) Przeglądaj dalej pierwszą taśmę, aż dojdziesz do jej końca. Dla każdej jedynki z tej taśmy wykreśl zero na drugiej taśmie. Jeśli wykreślisz wszystkie zeta, zanim dojdziesz do końca jedynek, to odrzuć.
(4) Jeśli wszystkie zera zostały wykreślone, to zaakceptuj. Jeśli pozostaje jakieś zera, to odrzuć."

Przedstawioną maszynę łatwo przeanalizować. Każda z jej czterech faz wymaga O(n) kroków, więc całkowity czas działania wynosi O(n) i jest liniowy. Warto zauważyć, że jest to najlepszy możliwy czas, ponieważ maszyna musi wykonać przynajmniej n kroków, by przeczytać słowo wejściowe.

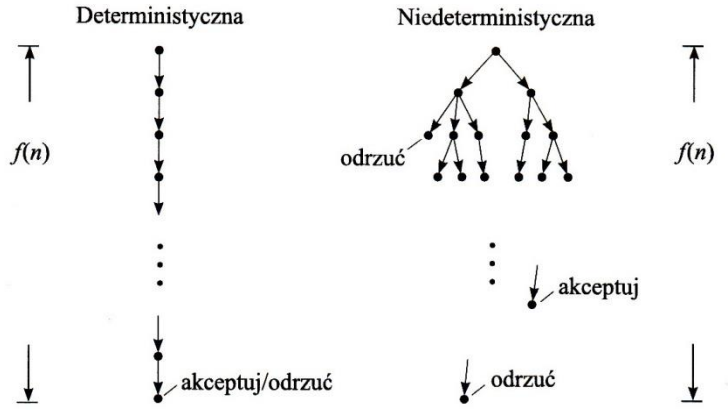
3.8.5.21. **Wyjaśnienie**. Podsumujmy spostrzeżenia Sipser'a dotyczące złożoności języka A, czyli czasu potrzebnego na rozstrzygnięcie tego języka. Pokazaliśmy jednotaśmową *Maszynę Turinga* M_1, która rozstrzyga język A w czasie $O(n^2)$, i szybszą jednotaśmową *Maszynę Turinga* M_2, która rozstrzyga język A w czasie $O(n \log n)$. Żadna jednotaśmowa *Maszyna Turinga* nie może tego zrobić szybciej. Następnie pokazaliśmy dwutaśmową *Maszynę Turinga* M_3, która rozstrzyga

język A w czasie O(n). Stąd widzimy, że złożoność czasowa języka A na jednotaśmowej *Maszynie Turinga* wynosi O(n log n), a na dwutaśmowej *Maszynie Turinga* - O (n). Złożoność języka A zależy zatem od wybranego modelu obliczeń.

3.8.5.22. **Wyjaśnienie**. Powyższe rozważania pokazują istotną różnicę pomiędzy *teorią złożoności* a *teorią obliczalności*. W *teorii obliczalności* z *tezy Churcha-Turinga* wynika, że wszystkie sensowne *modele obliczeń są sobie równoważne*, czyli wszystkie one rozstrzygają tę samą klasę języków. W *teorii złożoności* wybór *modelu ma wpływ na złożoność języka*. Języki rozstrzygalne na jednym modelu - w czasie, powiedzmy, liniowym, nie muszą być rozstrzygalne na innym modelu - w czasie liniowym.

W teorii złożoności klasyfikujemy problemy obliczeniowe ze względu na ich *złożoność czasową*. Jednak, którego z modeli urywamy do mierzenia czasu? Ten sam język może mieć różne wymagania czasowe na różnych modelach. Na szczęście wymagania czasowe nie różnią się jakoś znacząco dla typowych modeli deterministycznych. Jeśli więc nasza klasyfikacja nie będzie bardzo wrażliwa na stosunkowo małe różnice złożoności, to wybór modelu deterministycznego nie będzie istotny. Przeanalizujemy tę kwestię wnikliwiej dalej.

3.8.5.23. **Wyjaśnienie**. Wzajemna złożoność modelu to kolejny problem wymagający analizy. Przeanalizujemy, w jaki sposób wybór modelu obliczeń może wpłynąć na złożoność czasową języka. Rozważymy trzy modele: jednotaśmową maszynę Turinga, wielotaśmową maszynę Turinga oraz niedeterministyczną maszynę Turinga.



Rysunek 3.8.5.00 – Pomiar deterministycznego i niedeterministycznego czasu

3.8.5.24. **Twierdzenie**. Niech t(n) będzie funkcją dla której zachodzi t(n) ≥ n. Wówczas dla każdej t(n) - czasowej, wielotaśmowej *Maszyny Turinga* istnieje równoważna jej jedno taśmowa *Maszyna Turinga* działająca w czasie O (t^2(n)).

Schemat Dowodu wg Sipser'a. Idea dowodu tego twierdzenia jest bardzo prosta. Przypomnijmy, że w twierdzeniu 3.8.4.42 pokazaliśmy, jak przekształcić wielo taśmową *Maszynę Turinga* w symulującą ją jedno taśmową *Maszynę Turinga*. Przeanalizujemy teraz tę symulację, by zobaczyć, ile dodatkowego czasu ona wymaga. Pokażemy, że symulacja jednego kroku maszyny wielotaśmowej wymaga najwyżej O(t(n)) kroków maszyny jednotaśmowej. Tak więc całkowity czas symulacji wynosi O(t^2(n)).

Dowód wg Sipser'a. Niech M będzie k-taśmową *Maszyną Turinga*, która działa w czasie t(n). Skonstruujemy jednotaśmową *Maszynę Turinga* S, która działa w czasie O(t^2(n)). Maszyna S

działa, symulując maszynę M w sposób opisany w twierdzeniu 3.8.4.42. Przypomnijmy, że symulacja ta polega na reprezentowaniu zawartości k - taśm maszyny M na jednej taśmie maszyny S. Komórki z takich samych pozycji na taśmach M są zapamiętywane koło siebie, a pozycje głowic maszyny M są zaznaczane kropkami nad odpowiednimi symbolami.

Maszyna S rozpoczyna pracę od zapisania zawartości swojej taśmy w postaci umożliwiającej reprezentowanie wszystkich taśm maszyny M i następnie przystępuje do symulacji działania M. Aby wykonać jeden krok symulacji, maszyna S przegląda całą taśmę, by określić, jakie symbole widzą głowice maszyny M. Potem maszyna S ponownie przegląda całą taśmę, by zaktualizować zawartość taśm i oznaczenia położenia głowic. Jeśli jakaś głowica maszyny M przesuwa się w prawo nad wcześniej nieużywany fragment taśmy, to maszyna S musi zwiększyć ilość miejsca przeznaczonego na tę taśmę. Robi to, przesuwając w prawo o jedną komórkę całą zawartość swojej taśmy. Przeanalizujmy symulację. Dla każdego kroku maszyny M maszyna S wykonuje dwa przejścia nad używaną częścią swojej taśmy. W pierwszym przebiegu zbiera informację konieczną do określenia kolejnego ruchu, a w drugim wykonuje ten ruch. To od długości używanej części taśmy maszyny S zależy, ile czasu zajmie jej przeczytanie tej części, więc powinniśmy oszacować tę wielkość. Możemy to zrobić, sumując długości używanych części k taśm maszyny M. Każda z tych części ma długość najwyżej $t(n)$, ponieważ maszyna M może wykorzystać $t(n)$ komórek pamięci w $t(n)$ krokach, jeśli będzie w każdym kroku przesuwać głowicę o jeden w prawo; natomiast jeśli kiedykolwiek przesunie głowicę w lewo, to na pewno wykorzysta mniej komórek. Widzimy więc, że przejrzenie przez maszynę S używanej części jej taśmy wymaga czasu $O(t(n))$.

Aby zasymulować każdy z kroków maszyny M, maszyna S dwukrotnie przegląda taśmę i najwyżej k razy przesuwa w prawo jej zawartość. Każda z tych operacji jest wykonywana w czasie $O(t(n))$, zatem całkowity czas potrzebny maszynie S na symulację jednego kroku maszyny M wynosi $O(t(n))$. Możemy teraz oszacować całkowity czas potrzebny na symulację. Początkowa faza, w trakcie której maszyna S zapisuje zawartość swojej taśmy we właściwej postaci, wymaga $O(n)$ kroków. Następnie maszyna S symuluje każdy z $t(n)$ kroków maszyna M za pomocą $O(t(n))$ kroków więc ta część symulacji wymaga $t(n) \times O(t(n)) = O(t^2(n)))$ kroków. Widać więc, że cała symulacja jest wykonywana w $O(n) + O(t^2(n))$ krokach.

Założyliśmy że $t(n) \geq n$ (jest to sensowne założenie, ponieważ gdyby maszyna M działa krócej, to nie zdążyłaby nawet przeczytać słowa wejściowego). Stąd czas działania maszyny S wynosi $O(t^2(n))$, co kończy dowód. cbdo.

3.8.5.25. **Wyjaśnienie**. Jako następne rozważmy analogiczne twierdzenie dla nie-deterministycznych, jednotaśmowych *Maszyn Turinga*. Pokażemy, że każdy język rozstrzygalny przez taką maszynę jest także rozstrzygalny przez deterministyczną jednotaśmową *Maszynę Turinga*, która potrzebuje znacznie więcej czasu. Zanim jednak do tego przystąpimy, musimy zdefiniować czas działania niedeterministycznej *Maszyny Turinga*, która jest maszyną rozstrzygającą, jeśli kończy obliczenia na wszystkich ścieżkach dla wszystkich słów wejściowych.

Definicja czasu działania niedeterministycznej *Maszyny Turinga* nie ma odpowiadać realiom spotykanym w jakimkolwiek rzeczywistym urządzeniu liczącym. Jest to raczej tylko definicja matematyczna, która przydaje się przy analizie złożoności dla istotnej klasy problemów obliczeniowych, o czym przekonamy się wkrótce.

3.8.5.26. **Definicja**. Niech N będzie niedeterministyczną *Maszyną Turinga*, która rozstrzyga język. Czas działania maszyny N to funkcja $f : N \to N$, gdzie f(n) jest maksymalną liczbą kroków, które maszyna N wykonuje na dowolnej ścieżce obliczeń dla dowolnego słowa wejściowego długości n, jak to pokazano na rys. 3.8.5.00.

3.8.5.27. **Twierdzenie**. Niech t(n) będzie funkcją dla której zachodzi $t(n) \geq n$. Wówczas dla każdej niedeterministycznej, jednotaśmowej *Maszyny Turinga* działającej w czasie r(n) istnieje równoważna jej deterministyczna, jednotaśmowa *Maszyna Turinga* działająca w czasie $2^{O(t(n))}$.

Dowód (wg Sipser'a). Niech N będzie niedeterministyczną *Maszyną Turinga* działającą w czasie t(n). Zbudujemy deterministyczną *Maszynę Turinga* D, która symuluje działanie N, przeszukując niedeterministyczne drzewo obliczeń N, jak w dowodzie twierdzenia 3.8.5.24. Przeanalizujmy tę symulację. Dla słowa wejściowego długości n każda niedeterministyczna ścieżka w drzewie obliczeń maszyny N ma długość najwyżej t(n). Każdy węzeł w drzewie ma najwyżej b dzieci, gdzie b jest maksymalną liczbą wyborów dopuszczalnych przez funkcję przejścia maszyny N. Stąd maksymalna liczba liści w drzewie wynosi najwyżej $b^{t(n)}$.

Proces symulacji polega na przeglądaniu drzewa wszerz. Inaczej mówiąc, polega na odwiedzeniu wszystkich węzłów na głębokości d przed, przejściem do jakiegokolwiek węzła na głębokości d + 1. Algorytm przedstawiony w dowodzie twierdzenia 3.8.5.24 działa nieefektywnie; za każdym razem aby odwiedzić węzeł, zaczyna drogę od korzenia i schodzi w dół drzewa. Jednak poprawienie tego braku efektywności nie ma wpływu na tezę obecnie wykazywanego twierdzenia, nie będziemy więc tego zmieniać. Całkowita liczba węzłów w drzewie jest mniejsza niż podwojona liczba liści, więc liczbę węzłów w drzewie możemy oszacować przez $O(b^{t(n)})$. Czas potrzebny na dojście z korzenia do węzła wynosi O(t(n)). Stąd czas działania maszyny D wynosi $O(t(n)b^{t(n)}) = 2^{O(t(n))}$.

Maszyna Turinga D przedstawiona w twierdzeniu 3.8.4.42 ma trzy taśmy. Przekształcenie jej w jednotaśmową maszynę Turinga zwiększa czas jej działania najwyżej kwadratowo, co pokazaliśmy w twierdzeniu 3.8.5.24. Czyli czas działania jednotaśmowej maszyny symulującej wynosi $(2^{O(t(n))})^2 = 2O^{(2t(n))} = 2^{O(t(n))}$, co kończy dowód twierdzenia. cbdo.

3.8.5.30. **Wyjaśnienie**. Zanim wprowadzono precyzyjną definicję *funkcji obliczalnych*, matematycy bardzo często używali nieformalnego terminu "*funkcji efektywnych*". Od tego czasu to określenie zaczęto utożsamiać z funkcjami obliczalnymi. Dokładniej można dla niektórych *funkcji efektywnych* wykazać, że każdy algorytm je obliczający będzie niewydajny w takim sensie, że *każdy taki algorytm będzie potrzebował czasu rosnącego wykładniczo* w zależności od długości wprowadzanych doń danych. Teoria obliczalności i teoria złożoności zajmują się zagadnieniami obliczalności oraz złożoności obliczeń *funkcji obliczalnych wydajnie*.

Funkcje obliczalne są podstawowym obiektem badań teorii obliczalności. *Zbiór funkcji obliczalnych jest równoważny zbiorowi funkcji obliczalnych w sensie Turinga oraz funkcji częściowo rekurencyjnych. Funkcje obliczalne* stanowią analogon *intuicyjnego pojęcia algorytmu*. Tego pojęcia używa się do dyskusji obliczalności bez odniesienia do określonego modelu obliczalności takiego jak maszyna Turinga lub maszyna von Neumana. Jednak ich definicja musi mieć odniesienie do określonego modelu obliczalności.

Istnieje wiele równoważnych sposobów określenia klasy funkcji obliczalnych. Najczęściej przyjmuje się, że *funkcje obliczalne* zostały określone, jako skończone funkcje częściowe na liczbach naturalnych, które dają się obliczyć za pomocą maszyny Turinga. Istnieje wiele równoznacznych modeli obliczalności, określających tą samą kategorię funkcji obliczalnych, np. takie jak:

1. Maszyny Turinga – opracowana przez *Alana M. Turinga*
2. Rachunek lambda – opracowany przez *Alonso Churcha*
3. Maszyna RAM (patrz przypis w 1.8.6).

Ze względu na skończone rozmiary pamięci RAM współczesnych komputerów (tzw. złożoność pamięciowa), projektując algorytmy dla rozwiązywania postawionych zadań, bardzo chętnie korzystamy ze struktury programów wykorzystujących schemat funkcji rekursywnej.

Warto w tym miejscu podkreślić, że *funkcje rekursywne* są podstawowym obiektem *badań teorii obliczalności*. *Zbiór funkcji rekurencyjnych jest równoważny zbiorowi funkcji obliczalnych w sensie Turinga oraz funkcji częściowo rekurencyjnych*. Tak rozumiana teoria obliczalności, nie dotyczy jednak rzeczywistych komputerów, ale ich uogólnionych modeli logicznych, takich jak Maszyna Turinga lub Maszyna RAM.

3.8.5.40. **Wyjaśnienie**. *Hipoteza Churcha-Turinga* (zwana również *Tezą Churcha-Turinga*) jest hipotezą określającą możliwości komputerów i innych maszyn obliczeniowych. Mówi ona, że każdy problem, dla którego przy nieograniczonej pamięci oraz zasobach istnieje efektywny *algorytm* jego rozwiązywania, da się rozwiązać na *Uniwersalnej Maszynie Turinga*. Hipoteza jest niemożliwa do sprawdzenia matematycznie, ponieważ łączy w sobie zarówno ścisłe, jak i nieprecyzyjne sformułowania, których interpretacja może zależeć od konkretnej osoby. Dlatego traktowana jest bardziej jako *aksjomat* lub *swoiste prawo*.

Mniej formalnie, hipoteza ta pozwala dokładniej sformułować pojęcie samego algorytmu, mówiąc jednocześnie, że komputery są w stanie go wykonać. Co więcej, wszystkie komputery mają jednakowe możliwości, jeśli chodzi o zdolność do wykonywania algorytmów.

Piśmiennictwo: *Banachowski L.* B.1.1., *Cormen T.* C.5.1., *Kisielewicz A.* K.1.1., *Sipser M.* S.7.1.

3.8.6. Granice możliwości obliczeniowych

3.8.6.10. **Wyjaśnienie**. Gdyby współczesne komputery miały nieograniczoną pojemność pamięci i byłyby nieskończenie szybkie, (czyli - komputery działałyby natychmiast) oraz operowałyby na liczbach całkowitych dowolnej długości oraz wykonywałyby operacje arytmetyczne na liczbach rzeczywistych, to granica możliwości obliczeniowych zależałyby tylko od faktu istnienia algorytmu - pozwalającego na uzyskanie rozwiązania danego zadania. Innymi słowy, o możliwościach obliczeniowych decydowałyby wnioski wynikające z twierdzenia Kurta Gödla o nierozstrzygalności złożonych systemów logicznych - mówiące, że brak rozstrzygalności, czyli prawdziwości lub fałszywości danego twierdzenia, jest równoznaczny z faktem istnienia lub nieistnienia algorytmu dotyczącego rozwiązywania zadania - równoważnego danemu twierdzeniu. Niestety nasze współczesne komputery mają ograniczoną pojemność pamięci, mają skończoną, chociaż bardzo wysoką prędkość działania oraz reprezentowane w komputerach - liczby całkowite mają swoje górne ograniczenie (wynikające z architektury komputerów), a komputery wykonują operacje arytmetyczne na liczbach zmiennopozycyjnych - będących jedynie przybliżeniem liczb rzeczywistych.

Dlatego też, fakt istnienia algorytmu służącego rozwiązaniu danego zadania jest jedynie warunkiem koniecznym, a niewystarczającym - do stwierdzenia, że dane zadanie jest rozwiązywalne. O warunku dostatecznym decydują, zatem czynniki dotyczące:

1. Czasu potrzebnego na wykonywanie poszczególnych ciągów operacji (szybkość komputera) przez dany komputer, czyli tzw. złożoność czasowa;
2. Dostępnej dla danego zadania pojemności pamięci, czyli tzw. złożoność pamięciowa;
3. Niezbędna dokładność prowadzenia obliczeń na danym komputerze, czyli stabilność numeryczna użytego algorytmu (innymi słowy szybkość kumulacji błędu, ze względu na działania na liczbach przybliżonych).

Ten ostatni czynnik dotyczy jedynie algorytmów obliczeniowych, prowadzonych na liczbach zmienno-przecinkowych, natomiast nie dotyczy np. algorytmu sortowania.

3.8.6.20. **Wyjaśnienie**. Czas działania algorytmu można oszacować od góry, (czyli - ocenić tzw. czas pesymistyczny wykonania algorytmu) w zależności od liczby danych wejściowych n. W praktyce dla dostatecznie dużych n - szacując czas działania algorytmu, liczymy jedynie rząd wielkości pesymistycznego czasu działania algorytmu, czyli zajmujemy się *asymptotyczną* funkcją czasu złożoności algorytmu (patrz podrozdział 3.8.5). Oznacza to, że interesuje nas, jak szybko wzrasta czas działania algorytmu, gdy rozmiar danych n dąży do nieskończoności, gdzie n jest liczbą naturalną. Korzystamy, więc z funkcji, których zbiór argumentów $n \in \mathbb{N}$. Przykładowo pesymistyczna ocena czasu działania dla sortowania przez scalanie, jest funkcją czasu $O(n \times \log n)$, gdzie n jest liczbą porządkowanych elementów. Natomiast, pesymistyczna ocena czasu działania dla sortowania przez wstawianie, jest funkcją czasu $O(n^2)$. Jak widać z powyższego, dla dużych n należy stosować algorytm sortowania przez scalanie, ponieważ funkcja $f_1(n) = n \times \log n$; rośnie znacznie wolniej od funkcji $f_2(n) = n^2$. Obok szacunku pesymistycznego czasu wykonywania danego algorytmu, można również dokonać oszacowania czasu optymistycznego w funkcji zmiennej n. Jednak z punktu widzenia wyznaczania granic możliwości obliczeniowych, szacunek optymistyczny jest mało przydatny. Ograniczenie asymptotyczne lub notacja z dużym O umożliwia opisanie rzędu wielkości tej funkcji, jako $f(n) = O(g(n))$. Należy pamiętać, że każda funkcja wykładnicza o podstawie większej od 1 rośnie szybciej niż dowolny wielomian.

3.8.6.30. **Wyjaśnienie**. Z funkcją wykładniczą dla rosnących wartości zmiennej n – wykładnika, wiąże się pojęcie *kombinatorycznej eksplozji*. W związku z powyższą zależnością, jak pisze Andrzej Kisielewicz „Najstarsza prawdopodobnie wzmianka dotycząca kombinatorycznej eksplozji wiąże się z legendą o przywiezieniu szachów do Persji. Jednemu z władców Persji tak miała spodobać się królewska gra (zwana wówczas *szatrangiem*), że chciał sowicie wynagrodzić człowieka, który grę zaprezentował. Ten poprosił o wynagrodzenie w ziarnach pszenicy: na pierwszym polu szachownicy władca miał położyć jedno ziarnko pszenicy, na drugim dwa, na trzecim cztery i tak dalej, na każdym następnym dwa razy więcej niż na poprzednim. Okazuje się, że liczba ziaren przypadająca na ostatnie sześćdziesiąte czwarte pole, jest wielokrotnie większa niż ilość wszystkich ziaren pszenicy na Ziemi (nawet, jeśli założyć, że wszystkie lądy, włącznie z Antarktydą, są w całości obsiane pszenicą). Ten na pierwszy rzut oka, bardzo zaskakujący rezultat wiąże się z faktem, że mamy tu do czynienia z *funkcją wykładniczą* $f(n) = 2^n$. Liczba ziaren przypadająca na $(n+1)$-pole wynosi 2^n. Funkcje wykładnicze mają tak szybki wzrost, że już dla stosunkowo niewielkich n wartości $f(n)$ przekraczają jakiekolwiek wyobrażalne wielkości i, co gorsza, w dalszym ciągu rosną jeszcze szybciej. Ten ostatni fakt sprawia, że jak zobaczymy, nie można liczyć na pokonanie eksplozji kombinatorycznej na drodze

zwiększenia możliwości obliczeniowych komputerów." Dalej Andrzeja Kisielewicz pokazuje (patrz tabela 3.8.6.00), przykład - eksplozji kombinatorycznej funkcji wykładniczej $f(x) = 10^{3n}$, opisującej liczbę ruchów do przodu n - do przeanalizowania na szachownicy dla kolejnych wartości n.

3.8.6.40. **Wyjaśnienie**. Warto zauważyć, że szybciej od funkcji wykładniczych rosną wartości funkcji silnia $f(n) = n!$ (przykłady wartości silni: 0!=1, 1!=1, 2!=2, 3!=6, 4!=24, 5!=120, itd.)

Tabela 3.8.6.00 - Przykład wzrostu wartości funkcji wykładniczej – liczba kombinacji do analizowania (założona szybkość komputera - miliard operacji na 1s).					
n	3	4	5	6	7
$f(x) = 10^{3n}$	1s	2,78 godz.	115 dni	316 lat	>3000 wieków

Jak pokażemy dalej, granica możliwości obliczeniowych, nie jest czymś jednoznacznie określonym. W rzeczywistości, granica ta zależy od możliwości intelektualnych ludzi, mówiąc precyzyjniej od ludzkiej kreatywności. To kreatywność pozwala niejednokrotnie wykorzystać, w wyniku eksploracji danych, ukryte zależności pomiędzy badanymi danymi i zastąpienie algorytmu o asymptocie wykładniczej, algorytmem o asymptocie wielomianowej, którego zastosowanie jest technicznie wykonalne.

3.8.6.50. **Wyjaśnienie**. Kolejnym pojęciem, które wprowadzimy jest tzw *Klasa P* złożoności. W twierdzeniach 3.8.5.24 i 3.8.5.25 pokazaliśmy istotny aspekt złożoności. Z jednej strony wykazaliśmy, że różnica w pomiarze złożoności czasowej problemów na modelu jednotaśmowym i wielotaśmowym deterministycznej *Maszyny Turinga* jest najwyżej kwadratowa, czyli wielomianowa. Z drugiej strony pokazaliśmy, że złożoność czasowa problemów mierzona na deterministycznych i niedeterministycznych *Maszynach Turinga* może różnić się wykładniczo.

3.8.6.51. **Wyjaśnienie**. Kolejnym potrzebnym pojęciem jest tzw *Czas wielomianowy*. Z naszego punktu widzenia wielomianowe różnice w czasie działania są małe, natomiast różnice wykładnicze są duże. Zastanówmy się, dlaczego postawiliśmy granicę właśnie między wielomianami i funkcjami wykładniczymi, a nie między innymi klasami funkcji. Pierwszy powód to znacząca różnica w tempie wzrostu typowych funkcji wielomianowych, jak n^3, i typowych funkcji wykładniczych, jak 2^n. Rozważmy n = 1000, co jest sensownym rozmiarem słowa wejściowego dla algorytmu. W takim przypadku wartość n^3 wynosi miliard, co jest wartością dużą ale można sobie z nią dać radę. Natomiast 2^n jest w tym przypadku liczbą znacznie przekraczającą liczbę atomów we wszechświecie. Algorytmy wielomianowe są więc zazwyczaj wystarczająco szybkie, a algorytmy wykładnicze rzadko kiedy są użyteczne. *Algorytmy wykładnicze powstają zazwyczaj wówczas, gdy przeszukujemy całą przestrzeń możliwych rozwiązań, co możemy nazwać przeszukiwaniem siłowym.* Na przykład, jednym ze sposobów rozłożenia liczby na czynniki pierwsze jest sprawdzenie wszystkich możliwych dzielników. Rozmiar przeszukiwanej przestrzeni jest wykładniczy, więc przeszukiwanie także zajmuje czas wykładniczy. *Czasami można uniknąć przeszukiwania siłowego, wniknąwszy głębiej w problem, co może pozwolić odkryć bardzo przydatny algorytm wielomianowy*.

3.8.6.51. **Wyjaśnienie**. Sensowne deterministyczne modele obliczeń są wielomianowo równoważne. To oznacza, że każdy z nich może symulować innym, jedynie w czasie wielomianowo większym. *Gdy mówimy, że wszystkie sensowne deterministyczne modele obliczeń są wielomianowo równoważne, nie staramy się zdefiniować, co rozumiemy przez modele sensowne*.

Mamy jednak na myśli pojęcie wystarczająco szerokie, by objęło modele dobrze przybliżające czas działania rzeczywistych komputerów. Na przykład, twierdzenie 3.8.5.24 mówi, że modele deterministycznych, jednotaśmowych i wielotaśmowych maszyn Turinga są wielomianowo równoważne.

3.8.6.52. **Wyjaśnienie**. Teraz skupimy się na tych *aspektach teorii złożoności obliczeniowej*, na *które nie mają wpływu wielomianowe różnice w czasie działania*. Różnice takie uznamy za nieistotne i będziemy je pomijać. pozwoli to nam budować teorię niezależną od wyboru konkretnego modelu obliczeń. Przypomnijmy, że naszym celem jest pokazanie zasadniczych własności obliczeń, a nie własności *Maszyn Turinga* ani innego, wybranego modelu obliczeń. Można mieć wrażenie, że pomijanie wielomianowych różnic w czasie działania jest absurdalne. Programiści oczywiście, że zwracają uwagę na takie różnice i często męczą się, by chociaż dwukrotnie przyspieszyć działanie swojego programu. My jednak pomijamy stałe współczynniki, wprowadzając notację asymptotyczną. Teraz, z kolei, proponujemy pominąć także znacznie większe różnicę wielomianowe, jak między czasem n i czasem n^3.

3.8.6.52. **Wyjaśnienie**. Podjęta decyzja o pomijaniu różnic wielomianowych nie oznacza, że różnice te uważamy za nieważne. Przeciwnie, różnicę między czasem n i n^3 oczywiście uważamy za bardzo ważną. Jednak pewne pytania, jak na przykład, wielomianowość czy nie-wielomianowość problemu aproksymacji, nie zależą od różnic wielomianowych i są bardzo ważne. Teraz powinniśmy się zająć właśnie takimi problemami. Pomijanie drzew, by zobaczyć cały las, nie oznacza, że jedno jest ważniejsze niż drugie - są po prostu oglądane z różnej perspektywy.

Przejdźmy teraz do definicji bardzo ważnej dla teorii złożoności.

3.8.6.53. **Definicja**. P jest klasą języków rozstrzygalnych w czasie wielomianowym na deterministycznych, jednotaśmowych maszynach Turinga. Innymi słowy $P = \bigcup_k \mathrm{TIME}(n^k)$. Klasa P odgrywa główną rolę w opisywanej przez nas teorii, a jej znaczenie wynika z następujących faktów:

(1) Definicja klasy P nie zależy od wyboru modelu obliczeniowego, jeśli tylko dany model jest wielomianowo równoważny deterministycznej, jednotaśmowej *Maszynie Turinga*.
(2) Klasa P w przybliżeniu odpowiada klasie tych problemów, które można rozwiązać na współczesnych komputerach.

3.8.6.54. **Wyjaśnienie**. Pierwsza własność powyżej definicji wskazuje, że klasa P jest klasą odporną z matematycznego punktu widzenia. Nie ma na nią wpływu to, jakiego konkretnie modelu obliczeń będziemy używać. Z drugiej własności wynika, że klasa P jest związana z praktyką. Problemy, które należą dla klasy P mogą być rozwiązane za pomocą metody działających w czasie n^k, dla pewnej stałej k. Czy jest to czas do zaakceptowania w praktyce, zależy od wartości k i od zastosowania. Oczywiście jest mało prawdopodobne, by metoda działająca w czasie n^{100} miała zastosowanie praktyczne. Jednak nazwanie czasu wielomianowego granicą od której zaczyna się praktyczna rozwiązywalność problemu, ma chyba sens.

3.8.6.55. **Wyjaśnienie**. Gdy dla problemu, który jak się dotychczas wydawało, wymagał czasu wykładniczego, znajdzie się algorytm wielomianowy, wówczas pojawia się istotnie inny sposób rozumienia tego problemu, za którym zazwyczaj podążają kolejne redukcję złożoności, dające zazwyczaj w rezultacie algorytm przydatny w praktyce.

Pokazując algorytm działający w czasie wielomianowym, podajemy zazwyczaj jego opis na wysokim poziomie, nie skupiając się na cechach charakterystycznych dla konkretnego modelu obliczeniowego. Pozwala to uniknąć zawiłych szczegółów dotyczących zapisu na taśmie i ruchu głowicy. Musimy jednak przestrzegać pewnych zasad opisu algorytmów, by móc uzasadnić ich wielomianowość. Algorytmy opisujemy, numerując kolejne fazy ich działania. Pojęcie fazy algorytmu jest analogiczne do fazy *Maszyny Turinga*, chociaż implementacja jednej fazy algorytmu na *Maszynie Turinga* będzie zazwyczaj wymagała wielu kroków tejże maszyny.

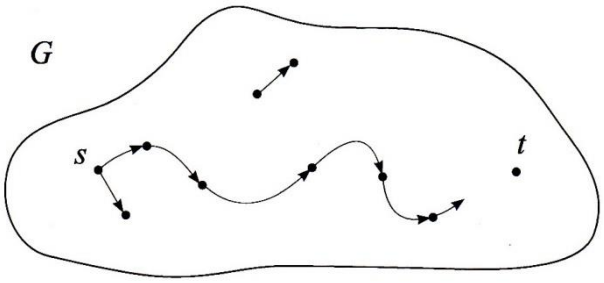
Wykazując, że algorytm działa w czasie wielomianowym, musimy zrobić dwie rzeczy. Po pierwsze, musimy pokazać wielomianowe, górne ograniczenie (zazwyczaj stosując notację wielkie O) liczby faz, które algorytm wykonuje w trakcie obliczeń dla słowa wejściowego długości n. Następnie musimy przeanalizować poszczególne fazy zapisu algorytmu, by upewnić się, ze każdą z nich można wykonać w czasie wielomianowym na sensownym deterministycznym modelu obliczeń. Opisując algorytm, będziemy zwracać uwagę, by tak dzielić algorytm na fazy, aby druga część analizy była łatwa do przeprowadzenia. Po wykonaniu obu części możemy podsumować czas działania całego algorytmu jako wielomianowy, ponieważ wykazaliśmy, że wykonuje on wielomianową liczbę faz, z których każda może być wykonana w czasie wielomianowym, a iloczyn dwóch wielomianów jest wielomianem.

Jeden z elementów, na który musimy jeszcze zwrócić uwagę, to sposób reprezentacji wykorzystywany przy zapisie problemów. Nadal będziemy używać notacji z nawiasami trójkątnymi $< \cdots >$, do oznaczenia sensownej reprezentacji jednego lub większej liczby obiektów w postaci słowa, bez dokładnego wskazywania sposobu zapisu. Tym razem za sensowną reprezentację będziemy uznawali taką którą można wykonać w czasie wielomianowym, zapisując obiekty w naturalnej postaci lub w innej sensownej reprezentacji. Znane metody reprezentowania grafów, automatów i innych obiektów są zazwyczaj sensowne. Zauważmy jednak, że notacja unarna dla liczb (np. w notacji tej - liczbę 17 zapisujemy jako ciąg unarny 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1) nie jest reprezentacją sensowną, ponieważ jest wykładniczo dłuższa niż faktycznie sensowne reprezentacje, jak na przykład zapis przy podstawie k dla k > 2.

Wiele z omawianych dotychczas problemów może być reprezentowanych w postaci grafu. Jedną z sensownych reprezentacji grafu jest lista wierzchołków i krawędzi tego grafu. Druga to macierz sąsiedztwa, w której element (i, j) macierzy jest równy 1, gdy istnieje krawędź z wierzchołka i do wierzchołka j; w przeciwnym przypadku element ten jest równy 0. Gdy analizujemy algorytmy dotyczące grafów, czas działania można wyrazić jako funkcję liczby wierzchołków, a nie jako funkcję długości reprezentacji grafu. W sensownej reprezentacji grafu rozmiar tej reprezentacji jest wielomianowy w stosunku do liczby wierzchołków. Jeśli więc przeanalizujemy algorytm i pokażemy, że jego czas działania jest wielomianowy (lub wykładniczy) względem liczby wierzchołków, to wiemy także, że jest on wielomianowy (lub wykładniczy) względem rozmiaru reprezentacji.

Problem 1. Pierwszy problem, który zaprezentujemy, dotyczy grafów skierowanych. Niech s i r będą wybranymi wierzchołkami w grafie G, jak to widać na rys. 3.8.6.01. Problem PATH polega na określeniu, czy istnieje ścieżka skierowana z wierzchołka s do wierzchołka t. Niech:

PATH = {(G, s, r) |G jest grafem skierowanym, w którym istnieje ścieżka - skierowana z s do t}.



Rysunek 3.8.6.01 – Problem PATH, czy istnieje ścieżka z s do t?

3.8.6.60. **Twierdzenie**. PATH ∈ P.

Schemat dowodu wg Sipsera. Udowodnimy twierdzenie, pokazując algorytm wielomianowy, który rozstrzyga problem PATH. Zanim opiszemy ten algorytm, zauważmy, że algorytm siłowy nie jest wystarczająco szybki. Algorytm siłowy dla problemu PATH polega na zbadaniu wszystkich możliwych ścieżek w grafie G, by sprawdzić, czy jest wśród nich ścieżka skierowana z s do t. Taka potencjalna ścieżka to ciąg wierzchołków grafu G o długości co najwyżej m, gdzie m jest liczbą wierzchołków grafu G (jeżeli istnieje ścieżka skierowana z s do t, to istnieje także taka ścieżka długości co najwyżej m, gdyż w ścieżce nie muszą powtarzać się wierzchołki). Jednak liczba takich potencjalnych ścieżek wynosi iloczyn m×m, co jest wykładnicze względem liczby wierzchołków grafu G. Wynika stąd, że algorytm siłowy wymagałby czasu wykładniczego. Aby uzyskać algorytm wielomianowy dla problemu PATH, musimy w jakiś sposób uniknąć przeszukiwania siłowego. Jeden ze sposobów polega na wykorzystaniu metody przeszukiwania grafu, takiej jak *przeszukiwanie wszerz*. W metodzie tej odwiedzamy kolejno wszystkie wierzchołki grafu G osiągalne z wierzchołka s za pomocą ścieżki skierowanej najpierw długości 1, potem długości 2, potem 3 itd., aż do ścieżek długości m. Oszacowanie czasu działania tej metody przez wielomian jest proste.

Dowód wg Sipsera. Wielomianowy algorytm M dla problemu PATH działa następująco:

M = „Dla słowa wejściowego <G, s, t>;

gdzie G jest grafem skierowanym, a s i t są wierzchołkami w tym grafie:

(1) Zaznacz wierzchołek s.
(2) Powtarzaj następny krok tak długo, jak długo pojawiają się nowo zaznaczone wierzchołki.
(3) Przejrzyj wszystkie krawędzie grafu G. Jeśli znajdzie się krawędź (a, b) prowadząca z wierzchołka zaznaczonego a do wierzchołka niezaznaczonego b, to zaznacz wierzchołek b.
(4) Jeśli wierzchołek t został zaznaczony, to zaakceptuj. W przeciwnym przypadku odrzuć."

Przeanalizujmy powyższy algorytm, by pokazać, że działa w czasie wielomianowym. Oczywiście fazy (1) i (4) są wykonywane jednokrotnie. Faza (3) jest wykonywana m razy, ponieważ za każdym razem, z wyjątkiem ostatniego, w fazie tej jest zaznaczany nowy wierzchołek grafu G. Stąd sumaryczna liczba wykonanych faz wynosi 1 + 1 + m, czyli jest wielomianowa względem rozmiaru grafu G. Fazy (1) i (4) algorytmu M można łatwo zaimplementować w czasie wielomianowym w każdym sensownym modelu deterministycznym. Faza (3) wymaga przejrzenia całego słowa wejściowego i sprawdzenia, czy są tam odpowiednie zaznaczone wierzchołki, co także można łatwo zaimplementować w czasie wielomianowym. Pokazaliśmy więc, że M jest algorytmem wielomianowym dla problemu PATH. cbdo.

Problem 2. Przejdźmy do kolejnego przykładu algorytmu wielomianowego. Powiemy, że dwie liczby są względnie pierwsze, jeśli największą liczbą dzielącą je obie bez reszty jest 1. Na przykład, liczby 10 i 21 są względnie pierwsze, chociaż żadna z nich sama nie jest pierwsza. Natomiast liczby 10 i 22 nie są względnie pierwsze, gdyż obie dzielą się przez 2. Niech RELPRIME to problem sprawdzania, czy dwie liczby są względnie pierwsze. Oznacza to, że

RELPRIME = {(x, y)|x oraz y są względnie pierwsze}.

3.8.6.61. **Twierdzenie.** RELPRIME ∈ P.

Schemat dowodu wg Sipsera. Jeden algorytm, który rozwiązuje problem, polega na przejrzeniu wszystkich możliwych dzielników obu liczb i zaakceptowaniu, jeśli żaden z nich nie jest większy od 1. Jednak wartość liczby reprezentowanej w postaci binarnej lub przy innej podstawie k dla k ≥ 2 jest wykładnicza w stosunku do jej długości. To oznacza, że algorytm siłowy przegląda wykładniczą liczbę potencjalnych dzielników i tym samym działa w czasie wykładniczym. Zamiast powyższej metody zastosujemy starożytny pomysł nazywany algorytmem *Euklidesa*, umożliwiający wyznaczenie największego wspólnego dzielnika liczb. Największy wspólny dzielnik liczb naturalnych x i y, oznaczany gcd(x, y), to największa liczba całkowita, która dzieli zarówno x, jak i y. Na przykład, gcd(18, 24) = 6. Oczywiście liczby x i y są względnie pierwsze wtedy i tylko wtedy, gdy gcd(x, y) = 1. Algorytm Euklidesa opiszemy jako algorytm E w dowodzie twierdzenia. Wykorzystywana jest w nim funkcja x mod y, która dla liczb x i y oznacza resztę z dzielenia x przez y.

Dowód wg Sipsera. Algorytm *Euklidesa* E wygląda następująco:

E = „Dla słowa wejściowego <x, y>, gdzie x i y są liczbami całkowitymi zapisanymi binarnie:

(1) Powtarzaj, aż y = 0:
(2) Podstaw x ← x mod y.
(3) Zamień x i y.
(4) Wypisz w wyniku x."

Algorytm R rozwiązuje problem RELPRIME, wykorzystując algorytm E jako podprogram.

R = „Dla słowa wejściowego <x, y>, gdzie x i y są liczbami całkowitymi zapisanymi binarne:

(1) Uruchom algorytm E dla słowa wejściowego <x, y>.
(2) Jeśli wynikiem powyższego algorytmu jest 1, to zaakceptuj. W przeciwnym przypadku odrzuć."

Jeśli tylko algorytm E będzie działał poprawnie w czasie wielomianowym, to także algorytm R będzie poprawny i wielomianowy. Musimy więc przeanalizować tylko algorytm E. Jego poprawność jest doskonale znana, więc nie będziemy jej tutaj badać. Aby przeanalizować złożoność czasową algorytmu E, pokażemy najpierw, że każde wykonanie fazy (2) (z wyjątkiem, być może, jej pierwszego wykonania) powoduje zmniejszenie wartości liczby x co najmniej o połowę. Po wykonaniu fazy (2) zachodzi nierówność x < y, co wynika z własności funkcji modulo. Po wykonaniu fazy (3) zachodzi nierówność x > y, gdyż wartości x i y zostały zamienione. Jeśli, x/2 ≥ y}, to x mod y = x - y ≤ x/2 i wartość x maleje co najmniej o połowę. Jeśli x/2 < y, to x mod y = x - y < x/2 i wartość x także maleje co najmniej o połowę. Wartości x i y są zamieniane w każdym wykonaniu fazy (3), więc wartość początkowa każdej ze zmiennych x i y jest zmniejszana co najmniej o połowę w co drugim wykonaniu pętli. Stąd maksymalna liczba wykonań faz (2) i (3) jest mniejsza niż mniejsza z liczb $2 \log_2 x$ i $2 \log_2 y$. Wartości tych logarytmów są proporcjonalne do długości ich reprezentacji, co pozwala stwierdzić, że liczba wykonanych faz algorytmu wynosi O(n). Wykonanie każdej fazy algorytmu E wymaga czasu wielomianowego, zatem całkowity czas działania także jest wielomianowy. cbdo.

Przykład 3. Ostatni przykład algorytmu wielomianowego pokazuje, że każdy język bezkontekstowy można rozstrzygnąć w czasie wielomianowym.

3.8.6.62. **Twierdzenie**. Każdy język bezkontekstowy jest elementem klasy P.

Schemat dowodu wg Sipsera. Udowodniono, że każdy język bezkontekstowy jest rozstrzygalny. W tym celu należy pokazać algorytm, który dla każdego języka bezkontekstowego rozstrzyga ten język. Gdyby algorytm ten działał w czasie wielomianowym, aktualnie rozważane twierdzenie byłoby wnioskiem z poprzedniego wyniku. Przedstawmy więc algorytm i sprawdźmy, czy jest on wystarczająco szybki.

Niech L będzie językiem bezkontekstowym generowanym przez gramatykę bezkontekstową G w postaci normalnej Chomsky'ego. Można pokazać, że dla gramatyki w tej postaci dowolne wyprowadzenie słowa *w* przebiega w $(2n-1)$ krokach, gdzie *n* jest długością słowa *w*. Procedura rozstrzygająca dla L działa, testując wszystkie możliwe wyprowadzenia długości $(2n-1)$ dla słowa wejściowego długości *n*. Jeśli któreś z tych wyprowadzeń jest wyprowadzeniem słowa *w*, to procedura akceptuje; w przeciwnym przypadku - odrzuca. Wykonując pobieżną analizę powyższego algorytmu, stwierdzimy, że nie działa on w czasie wielomianowym. Liczba wyprowadzeń długości *k* może być wykładnicza względem *k*, zatem cały algorytm może potrzebować czasu wykładniczego. Aby otrzymać algorytm wielomianowy, wprowadzimy bardzo przydatną metodę nazywaną *programowaniem dynamicznym*. W tej metodzie gromadzi się informacje o rozwiązaniach mniejszych *podproblemów*, by rozwiązać większe *problemy*. Rozwiązanie każdego *podproblemu* zapisujemy, by nie trzeba było go rozwiązywać ponownie. Wyniki dla *podproblemów* przechowujemy w specjalnej tablicy, którą wypełniamy sukcesywnie w czasie działania algorytmu. W tym przypadku będziemy rozważać *podproblemy* polegające na określeniu, czy z wybranej zmiennej można wygenerować określone podsłowo *w*. Wynik pod problemu umieszczamy w tablicy rozmiaru $n \times n$. Dla $i < j$ w komórce (i, j) tablicy umieścimy zbiór zmiennych, z których można wygenerować słowo $w_i w_{i+1} \ldots w_j$. Dla $i > j$ komórki tablicy są nieużywane. Algorytm wypełnia tablicę dla każdego podsłowa *w*. Najpierw wypełnia komórki odpowiadające słowom długości 1, potem odpowiadające słowom długości 2 itd. Wyznaczając wartości dla dłuższych podsłów wykorzystuje wartości dla krótszych pod słów. Przypuśćmy, na przykład, że algorytm znalazł już zmienne generujące dla wszystkich podsłów o długości nieprzekraczającej *k*. Aby sprawdzić, czy ze zmiennej A da się wygenerować określone pod słowo długości $k+1$, rozbija to słowo na dwie niepuste części na *k* możliwych sposobów. Dla każdego rozbicia sprawdza każdą produkcję $A \to BC$, by określić, czy ze zmiennej B można wygenerować pierwszą część pod słowa, a ze zmiennej C - drugą część. Sprawdzenia te wykonuje, odwołując się do wcześniej obliczonych wartości zapisanych w tablicy. Jeżeli okażę się, że ze zmiennych B i C można wygenerować odpowiednie fragmenty, to ze zmiennej A można wygenerować całe podsłowo i zmienną tę można dopisać do odpowiedniej komórki tablicy. Cały algorytm rozpoczyna się od słów długości 1. Poszukując zmiennych, które je wyprowadzają analizuje produkcje postaci $A \to b$.

Dowód wg Sipsera. Podamy algorytm D, w którym jest zaimplementowany podany pomysł. Niech G będzie gramatyką bezkontekstową w postaci normalnej Chomsky'ego, generującą język bezkontekstowy Z. Niech S będzie zmienną początkową (przypomnijmy, że słowo puste jest traktowane w sposób specjalny w gramatyce w normalnej postaci Chomsky'ego. W algorytmie także traktujemy ten przypadek oddzielnie, sprawdzając w pierwszym kroku, czy $w = \varepsilon$.) W algorytmie, w nawiasach kwadratowych, zamieściliśmy wyjaśniające komentarze.

D = „Dla słowa wejściowego <w> = $w_1 \ldots w_n$;

(1) Dla $w = \varepsilon$, jeśli S → ε jest produkcją gramatyki, zaakceptuj, w przeciwnym przypadku odrzuć; [rozważamy przypadek $w = \varepsilon$]

(2) Dla i =1 do n: [analizujemy wszystkie pod słowa długości 1]

(3) Dla każdej zmiennej A:

(4) Sprawdź, czy istnieje produkcja A → b, gdzie b = w;

(5) Jeśli tak, to umieść symbol A w komórce (i, i) tablicy.

(6) Dla l = 2 do n: [l jest długością pod słowa]

(7) Dla i = 1 do $n - l + 1$: [l jest początkiem pod słowa]

(8) Niech $j = i + l - 1$. [j jest końcem pod słowa]

(9) Dla $k = i$ do $j - 1$: [k jest miejscem podziału pod słowa]

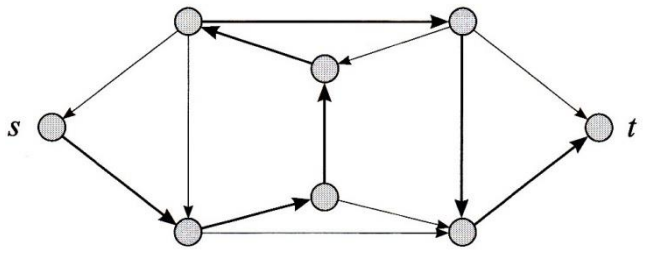
(10) Dla każdej produkcji A → BC:

(11) Jeśli w komórce (i, k) tablicy znajduje się zmienna B i w komórce (k+1, j) tablicy znajduje się zmienna C, to dołącz do zmienną A do komórki (i, j) tablicy.

(12) Jeśli do komórki (1, n) należy zmienna S, to zaakceptuj. W przeciwnym odrzuć ."

Przeanalizujmy algorytm D. Każdą jego fazę można łatwo tak zaimplementować, by działała w czasie wielomianowym. Fazy (4) i (5) są wykonywane najwyżej nv razy, gdzie v jest liczbą zmiennych gramatyki G, czyli stałą niezależną od n. Fazy te działają więc w czasie O(n). Faza (6) jest wykonywana najwyżej n razy. Z każdym jej wykonaniem wiąże się najwyżej n-krotne wykonanie fazy (7). Każde wykonanie fazy (7) powoduje z kolei najwyżej n-krotne wykonanie faz (8) i (9), a każde wykonanie fazy (9) powoduje r-krotne wykonanie fazy (10), gdzie r jest liczbą produkcji gramatyki G i także jest pewną stałą. Faza(11) algorytmu, jego najbardziej wewnętrzną pętla, jest wykonywana najwyżej O(n^3) razy. Z podsumowania złożoności wszystkich faz wynika, że algorytm D działa w czasie O(n^3). cbdo.

3.8.6.70. **Wyjaśnienie**. Kolejnym wprowadzanym pojęciem jest klasa NP. Pokazaliśmy już, że w wielu przypadkach możemy uniknąć przeszukiwania siłowego i uzyskać rozwiązania wielomianowe. Jednak w innych przypadkach, w tym wielu ciekawych przypadkach, próby uniknięcia siłowego przeszukiwania nie powiodło się i nie wiadomo, czy istnieją algorytmy wielomianowe, które te problemy rozwiązują. Dlaczego nie udało się nam znalezienie algorytmów wielomianowych dla tych problemów? *Nie znamy odpowiedzi na to istotne pytanie*. Być może dla tych problemów istnieją jeszcze nieodkryte, algorytmy wielomianowe oparte na nieznanych nam zasadach. A może niektórych z tych problemów po prostu nie da się rozwiązać w czasie wielomianowym. Może są z natury trudne. Jedno ze znaczących odkryć związanych z powyższym pytaniem pokazuje, że złożoności wielu problemów mogą być ze sobą powiązane. Algorytm wielomianowy dla jednego z takich problemów może zostać wykorzystany do rozwiązania całej klasy problemów. Aby wyjaśnić ten fenomen, zacznijmy od przykładu.



Rysunek 3.8.6.02 – Ścieżka *Hamiltona* przechodzi przez każdy wierzchołek dokładnie jeden raz

3.8.6.71. **Wyjaśnienie**. Ścieżka *Hamiltona* w grafie skierowanym G jest ścieżka skierowana w tym grafie, która przechodzi przez każdy wierzchołek dokładnie raz. Rozwiążmy problem sprawdzenia, czy graf skierowany ma ścieżkę *Hamiltona* łączącą dwa wyznaczone wierzchołki, jak jest to pokazane na rys. 3.8.7.02. Niech

HAMPATH = { (G, s, t,)|G jest grafem skierowanym, w którym istnieje ścieżka *Hamiltona* z wierzchołka s do wierzchołka t}.

Można łatwo zbudować algorytm wykładniczy dla problemu HAMPATH, modyfikując algorytm siłowy dla problemu PATH przedstawiony w twierdzeniu 3.8.6.62. Musimy tylko dodać sprawdzenie, czy ewentualna ścieżka jest ścieżką *Hamiltona*. Nie wiadomo, czy problem HAMPATH jest rozwiązywany w czasie wielomianowym. Problem HAMPATH ma własność, którą nazywamy *weryfikowalnością wielomianową* i która jest bardzo istotna w zrozumieniu złożoności. Nie potrafimy szybko (czyli w *czasie wielomianowym*) określić, czy graf ma ścieżkę *Hamiltona*. Jeśli jednak w jakiś sposób taką ścieżkę odkryjemy (być może wykorzystując algorytm wykładniczy), to możemy udowodnić, że ona istnieje, po prostu ją pokazując. Innymi słowy, weryfikacja istnienia ścieżki *Hamiltona* może być znacznie łatwiejsza niż rozstrzygnięcie, czy ona istnieje.

Innym problemem wielomianowo weryfikowalnym jest problem złożoności liczby. Przypomnijmy, że liczba naturalna jest złożona, gdy jest iloczynem dwu liczb większych od 1 (czyli liczba jest złożona, gdy nie jest liczbą pierwszą). Niech

COMPOSITES = { x | x = pq dla liczb całkowitych p, q > 1}.

Potrafimy łatwo zweryfikować, czy liczba jest złożona, Wystarczy tylko mieć dzielnik tej liczby. Ostatnio został odkryty algorytm wielomianowy sprawdzający, czy liczba jest pierwsza, czy złożona, ale jest on zdecydowanie bardziej skomplikowany niż wspomniana metoda weryfikacji złożoności liczby.

3.8.6.72. **Wyjaśnienie**. Pewne problemy być może nie są wielomianowo weryfikowalne. Rozważmy jako przykład $\overline{HAMPATH}$ - dopełnienie problemu HAMPATH. Jeśli nawet potrafilibyśmy (jakkolwiek) określić, że graf nie ma ścieżki *Hamiltona*, to nie wiadomo, jak ktoś inny mógłby się o tym przekonać, nie używając tego samego algorytmu wykładniczego, który można zastosować do siłowego wyznaczenia ścieżki. Poniżej podajemy formalną definicję.

3.8.6.73. **Definicja**. Weryfikator dla języka A to taki algorytm V, że A = {w|V akceptuje słowa <w, c> dla pewnego słowa c}.

Czas działania weryfikatora wyznaczmy jako funkcję długości słowa w, zatem wielomianowy weryfikator działa w czasie wielomianowym względem długości w. Język A jest wielomianowo weryfikowalny, jeśli istnieje dla niego wielomianowy weryfikator. Weryfikator wykorzystuje dodatkową informację, oznaczoną symbolem c w definicji 3.8.6.73, by sprawdzić, że słowo w jest elementem języka A. Informację tę nazywamy certyfikatem lub dowodem przynależności u do A. Zauważmy że dla wielomianowych weryfikatorów certyfikat ma długość wielomianową (względem długości w), bo tylko tej długości informację może odczytać weryfikator w czasie ograniczonym wielomianowo. Zastosujmy powyższą definicję w przypadku języków HAMPATH i COMPOSITES.

Dla problemu HAMPATH certyfikatem dla słowa (G, *s*, *t*) ∈ HAMPATH jest po prostu ścieżka *Hamiltona* prowadząca z *s* do *t*. Dla problemu, COMPOSITES certyfikatem dla liczby złożonej *x* jest jeden z jej dzielników. W obu przypadkach weryfikator może sprawdzić w czasie wielomianowym, że słowo wejściowe należy do języka, jeśli ma do dyspozycji certyfikat.

3.8.6.74. **Definicja**. NP to klasa języków, dla których istnieją wielomianowe weryfikatory. Klasa NP jest ważna, ponieważ dostarcza wiele problemów o znaczeniu praktycznym. Z przeprowadzonych wcześniej rozważań wynika, że zarówno HAMPATH, jak i COMPOSITES są elementami klasy NP. Jak już wspomnieliśmy, problem COMPOSITES jest także elementem klasy P która jest podklasą NP jednak wykazanie tego jest znacznie trudniejsze niż w przypadku poprzednich spostrzeżeń. Oznaczenie NP pochodzi od określenia *niedeterministyczny czas wielomianowy* (*nondeterministic polynomial time*), które wynika z alternatywnego sposobu charakteryzacji omawianej klasy za pomocą niedeterministycznych *Maszyn Turinga* działających w czasie wielomianowym. Problemy z klasy NP są czasami nazywane NP- problemami. Poniżej pokazujemy niedeterministyczną *Maszynę Turinga*, która rozstrzyga problem HAMPATH w niedeterministycznym czasie wielomianowym. Przypomnijmy, że w definicji 3.8.6.73. określiliśmy czas działania maszyny niedeterministycznej jako czas wykorzystany na najdłuższej ścieżce obliczeń.
N_1 = „Dla słowa wejściowego (G, *s*, *t*), gdzie G jest grafem skierowanym, a *s* i *t* są jego wierzchołkami:

(1) Utwórz listę m liczb $p_1,..., p_m$, gdzie *m* jest liczbą wierzchołków grafu G. Każda liczba na liście jest wybierana niedeterministycznie z przedziału od 1 do *m*.
(2) Sprawdź, czy na liście nie występują powtórzenia. Jeśli jakieś występuje, to odrzuć.
(3) Sprawdź, czy $s = p_1$ i $t = p_m$. Jeśli nie, to odrzuć.
(4) Dla każdego *i* od 1 do (*m* -1) sprawdź, czy (p_i, p_{i+1}) jest krawędzią grafu G. Jeśli nie, to odrzuć. W przeciwnym razie wszystkie testy zakończyły się sukcesem, więc zaakceptuj."

Aby przeprowadzić analizę algorytmu i sprawdzić, że działa on w niedeterministycznym czasie wielomianowym, zbadamy każdy jego krok. W kroku (1), wybór niedeterministyczny oczywiście jest wykonywany w czasie wielomianowym. W krokach (2) i (3) w każdej części wykonujemy proste sprawdzenie, więc całość działa w czasie wielomianowym. Ostatni, (4) krok, także działa w czasie wielomianowym. Cały algorytm działa więc także w niedeterministycznym czasie wielomianowym. cbdo.

3.8.6.75. **Twierdzenie**. Język należy do klasy NP wtedy i tylko wtedy, gdy jest rozstrzygany przez pewną niedeterministyczną *Maszynę Turinga* działającą w czasie wielomianowym.

Schemat dowodu wg Sipsera. Pokażemy, jak przekształcić wielomianowy weryfikator *w* równoważną niedeterministyczną *Maszyną Turinga* działającą w czasie wielomianowym i na odwrót. Niedeterministyczna *Maszyna Turinga* symuluje działanie weryfikatora, zgadując certyfikat. Weryfikator symuluje niedeterministyczną *Maszynę Turinga*, wykorzystując ścieżkę akceptującą jej obliczeń w roli certyfikatu.

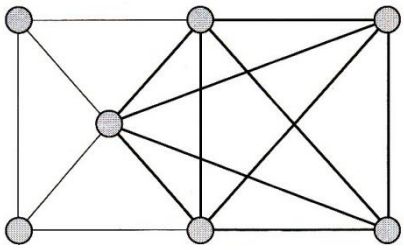
Dowód wg Sipsera. Aby udowodnić twierdzenie w jedną stronę, przyjmijmy, że A ϵ NP i pokażmy, że język A jest wówczas rozstrzygany przez niedeterministyczną wielomianową *Maszynę Turinga* N. Niech V będzie wielomianowym weryfikatorem dla A, którego istnienie wynika z definicji klasy NP. Załóżmy, że V jest *Maszyną Turinga*, która działa w czasie n^k i skonstruujmy maszynę N w następujący sposób:

N = „Dla słowa wejściowego w długości n:
(1) Niedeterministycznie wybierz słowo c długości nie większej niż n^k.
(2) Uruchom maszynę V dla słowa wejściowego <w, c>.
(3) Jeśli maszyna V akceptuje, to zaakceptuj; w przeciwnym przypadku odrzuć."
Aby udowodnić twierdzenie w drugą stronę, załóżmy, że język A jest rozstrzygany przez niedeterministyczną wielomianową *Maszynę Turinga* N i skonstruujmy wielomianowy weryfikator V w następujący sposób.
V = „Dla słowa wejściowego <w, c>, gdzie w i c są słowami:
(1) Zasymuluj działanie maszyny N dla słowa wejściowego w, traktując każdy znak w słowie c jak opis niedeterministycznego wyboru dokonywanego w kolejnych krokach (podobnie jak w dowodzie twierdzenia 3.8.4.45).
(2) Jeśli analizowana ścieżka obliczeń maszyny N prowadzi do akceptacji, to zaakceptuj; w przeciwnym przypadku odrzuć." cbdo.

Zdefiniujemy niedeterministyczną klasę złożoności czasowej NTIME(t(n)) analogicznie do deterministycznej klasy złożoności czasowej TIME (t (n)).

3.8.6.76. **Definicja**. NTIME (t (n)) = { L | L jest językiem rozstrzyganym przez niedeterministyczną *Maszynę Turinga* działającą w czasie t (n)}.

3.8.6.77. **Wniosek**. NP = $\bigcup_k$NTIME(n^k).



Rysunek 3.8.6.03 – Graf z 5 - CLIQUE

3.8.6.78. **Wniosek**. Klasa NP jest niezależna od wyboru modelu obliczeniowego, jeśli tylko jest to jeden z sensownych, niedeterministycznych modeli obliczeniowych z klasy modeli wielomianowo równoważnych. Opisując i analizując wielomianowe algorytmy niedeterministyczne, będziemy zachowywać ten sam sposób zapisu co w przypadku deterministycznych algorytmów wielomianowych. Każdy krok niedeterministycznego algorytmu wielomianowego musi być łatwo implementowany w niedeterministycznym czasie wielomianowym przez sensowny, niedeterministyczny model obliczeń. Podczas analizy algorytmu będziemy wykazywać, że na każdej ścieżce obliczeń algorytm wykonuje najwyżej wielomianowo wiele kroków.

3.8.6.80. **Przykłady problemów należących do Klasy NP**. CLIQUE (klika) w grafie nieskierowanym to podgraf, w którym każde dwa wierzchołki są połączone krawędzią a k- CLIQUE to klika złożona z k wierzchołków. Na rysunku 3.8.7.03 jest pokazany graf zawierający 5 - CLIQUE. Problem CLIQUE (kliki) polega na rozstrzygnięciu, czy graf zawiera CLIQUE (klikę) określonego rozmiaru. Niech

CLIQUE = {<G, k> | G jest grafem nieskierowanym, który ma klikę rozmiaru k}.

3.8.6.81. **Twierdzenie**. Problem CLIQUE należy do klasy NP.

Schemat dowodu wg Sipsera. Certyfikatem przynależności grafu do języka jest odpowiednia CLIQUE (klika).

Dowód . Poniższa maszyna V jest weryfikatorem dla języka CLIQUE.
V = „Dla słowa wejściowego postaci (< G, *k* >, *c*):
(1) Sprawdź, czy *c* jest zbiorem *k* wierzchołków grafu G.
(2) Sprawdź, czy graf G zawiera wszystkie krawędzie łączące wierzchołki ze zbioru *c*.
(3) Jeśli oba testy zakończyły się pozytywnie, to zaakceptuj: w przeciwnym przypadka odrzuć."

Dowód alternatywny wg Sipsera. Można także patrzeć na klasę NP przez pryzmat nie-deterministycznych, wielomianowych *Maszyn Turinga* i udowodnić twierdzenie, podając taką maszynę rozstrzygającą problem CLIQUE. Warto zauważyć podobieństwo tego dowodu do poprzedniego.
N = „Dla słowa wejściowego postaci < G , *k* > , gdzie G jest grafem:
(1) Niedeterministycznie wybierz podzbiór *c* złożony z *k* wierzchołków grafu G.
(2) Sprawdź, czy G zawiera wszystkie krawędzie łączące wierzchołki ze zbioru *c*,
(3) Jeśli tak, to zaakceptuj; w przeciwnym przypadku odrzuć." cbdo.

Przykład. Jako kolejny rozważymy problem SUBSET-SUM dotyczący arytmetyki liczb całkowitych. W problemie tym mamy zbiór liczb $x_1,.., x_k$ i liczbę (wartość docelową) *t*. Chcemy sprawdzić, czy w zbiorze istnieje podzbiór, którego suma wynosi *t*. Tak więc:

SUBSET-SUM = {<S, t> |S : $\{x_1,... , x_k\}$ oraz dla pewnego podzbioru $\{y_1, ..., y_i\} \subseteq \{x_1,..., x_k\}$ mamy $\sum y_i = t$}.

Przykładowo, zachodzi <{4, 11, 16, 21, 27 }, 25> ∈ SUBSETI-SUM, ponieważ 4 + 21 = 25. Zauważmy, że $\{x_1,..., x_k\}$ i $\{y_1,.., y_i\}$ są multi-zbiorami, dozwolone są więc powtórzenia elementów.

3.8.6.82. **Twierdzenie**. Problem SUBSET-SUM naIeży do klasy NP.
Schemat dowodu wg Sipsera. Odpowiedni podzbiór jest certyfikatem.
Dowód wg Sipsera. Następująca maszyna V jest weryfikatorem dla języka SUBSET- SUM.
V = „Dla słowa wejściowego postaci <<S, *t*>, *c*>:
(1) Sprawdź, czy *c* jest zbiorem liczb dających w sumie *t*.
(2) Sprawdź, czy zbiór S zawiera wszystkie elementy ze zbioru *c*.
(3) Jeśli oba testy zakończyły się pozytywnie, to zaakceptuj; w przeciwnym przypadku odrzuć ."
cbdo.

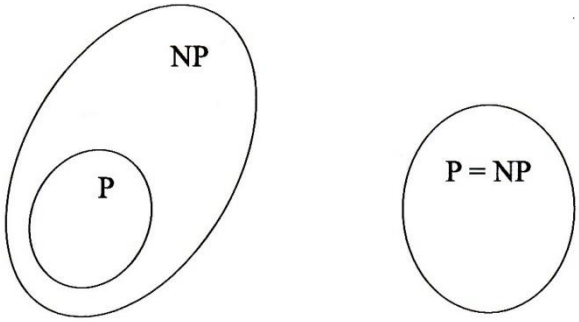
Dowód alternatywny wg Sipsera. Można także patrzeć na klasę NP przez pryzmat nie-deterministycznych, wielomianowych *Maszyn Turinga* i udowodnić twierdzenie, podając taką maszynę rozstrzygającą problem SUBSET-SUM. Warto zauważyć podobieństwo tego dowodu do poprzedniego.
N = „Dla słowa wejściowego postaci <S, *r*>:
(1) Niedeterministycznie wybierz podzbiór *c* złożony z liczb ze zbioru S.
(2) Sprawdź, czy *c* jest zbiorem, którego suma wynosi *t*.
(3) Jeśli tak, to zaakceptuj; w przeciwnym przypadku odrzuć." cbdo.

Zauważmy, że przynależność zbiorów CLIQUE i SUBSET-SUM do NP nie jest oczywista. Sprawdzenie, ze problem nie należy do określonej klasy, wydaje się trudniejsze niż sprawdzenie,

że problem do niej należy. Utworzymy oddzielną klasę złożoności nazywaną coNP, która zawiera języki, których dopełnienia należą do NP. Nie wiadomo, czy klasy coNP i NP są różne.

3.8.6.90. **Wyjaśnienie**. Problem klasy P versus NP. Powiedzieliśmy już, że klasa NP to klasa języków które są rozstrzygalne w czasie wielomianowym przez niedeterministyczne *Maszyny Turinga* lub też równoważnie klasy języków, dla których przynależność do języka może być zweryfikowana w czasie wielomianowym. Klasa P to klasa języków, dla której przynależność do języka może być sprawdzona w czasie wielomianowym. Możemy to podsumować następująco (za obliczenie „szybkie" przyjmiemy przy tym obliczenie w czasie wielomianowym).

P = klasa języków, dla których przynależność może być szybko rozstrzygnięta.
NP = klasa języków dla których przynależność może być szybko zweryfikowana.



Rysunek 3.8.6.04 – Jedna z przedstawionych sytuacji jest poprawna

Pokazaliśmy przykłady języków takich jak HAMPATH i CLIQUE, które należą do klasy NP ale nie wiemy, czy należą do klasy P. Zasięg wielomianowej weryfikowalności wydaje się być znacznie większy, niż zasięg wielomianowej rozstrzygalności. Jednak, choć trudno to sobie wyobrazić, klasy P i NP mogą być sobie równe. Nie potrafimy udowodnić, że istnieje jakikolwiek język należący do klasy NP i nienależący do klasy P.

Pytanie, czy P = NP jest jednym z największych nierozwiązanych problemów informatyki teoretycznej i współczesnej matematyki. Gdyby te klasy były równe, wówczas każdy problem wielomianowo weryfikowalny byłby wielomianowo rozstrzygalny. Większość naukowców uważa, że klasy te są różne, ponieważ poświęcono tak wiele czasu na poszukiwanie wielomianowych algorytmów dla niektórych problemów z klasy NP nie odnosząc sukcesu. Badacze próbowali także wykazać, że klasy te są różne, ale to wymagałoby udowodnienia, że nie istnieje szybki algorytm zastępujący przeszukiwanie siłowe. Niestety, to także jest aktualnie poza możliwościami nauki. Na rysunku 3.8.6.04 pokazane są dwie możliwe sytuacje.

Najlepsze, znane sposoby rozstrzygania problemów z klasy NP w sposób deterministyczny wymagają czasu wykładniczego. Innymi słowy, można udowodnić, że:

$$NP \subseteq EXPTIME = \bigcup_k TIME(2^{n^k}).$$

ale nie wiemy, czy klasa NP jest zawarta w mniejszej, deterministycznej klasie złożoności czasowej.

Istotny postęp w kwestii pytania P versus NP osiągnięto na początku lat siedemdziesiątych dwudziestego wieku. *Stephen Cook* i *Leonid Levin* odkryli w klasie NP pewne problemy, których złożoność jest związana ze złożonością całej klasy.

3.8.6.91. **Wyjaśnienie**. Klasa NP-zupełna w skrócie *NPC* (czyli *NP-Complete*). Jeśli dla któregokolwiek z tych problemów NPC istnieje algorytm wielomianowy, to wszystkie problemy z klasy NP są rozwiązywalne w czasie wielomianowym. Problemy te narywamy NPC – czyli NP. zupełnymi. Zjawisko problemów NPC jest bardzo ważne zarówno z teoretycznego, jak i praktycznego punktu widzenia. Od strony teoretycznej badacz próbujący wykazać, że klasy P i NP są różne, może skoncentrować się na wybranym problemie NPC. Jeśli jakikolwiek problem z klasy NP wymaga czasu dłuższego niż wielomianowy, to także problem NPC wymaga takiego czasu. Co więcej, badacze próbujący wykazać, że klasy P i NP są równe, muszą tylko znaleźć algorytm wielomianowy dla problemu NPC, by osiągnąć ten cel. Od strony praktycznej zjawisko problemów NPC może uchronić nas od straty czasu przy poszukiwaniu nieistniejącego algorytmu wielomianowego rozwiązującego pewien określony problem. Chociaż możemy nie mieć wystarczającego aparatu matematycznego, by wykazać, że problem nie jest rozwiązywalny w czasie wielomianowym, to jednak wierzymy, że klasy P i NP są różne, zatem udowodnienie, że problem jest NPC jest mocną przesłanką, że może nie być wielomianowy.

Piśmiennictwo: Cormen T. C.5.1., Kisielewicz A. K.1.1., SipserM. S.7.1.

Część 4.
Informatyka – przegląd tematyczny

4.1. PROGRAMOWANIE LOGICZNE – JĘZYK PROLOG

4.1.0. UWAGI WSTĘPNE

Prolog powstał jako język programowania służący do automatycznej analizy języków naturalnych, jest jednak językiem ogólnego zastosowania, szczególnie dobrze sprawdzającym się w programach związanych z formalnym wnioskowaniem. Prolog w przeciwieństwie do większości popularnych języków jest językiem deklaratywnym. Program w Prologu składa się z faktów oraz reguł wnioskowania. Aby go uruchomić, należy wprowadzić odpowiednie zapytanie. Prolog został stworzony w 1971 roku przez *Alaina Colmeraurera i Phillipe'a Roussela*. Prolog opiera się o rachunek predykatów pierwszego rzędu. Należy podkreślić, że idee działania - leżące u podstaw języka Prolog, są znacznie starsze od innowacji Johna von Neumanna – komputera z adresowaną pamięcią wewnętrzną służącą do przechowywania zarówno przetwarzanych danych, jak również kodu przetwarzającego programu.

4.1.0.10. **Wyjaśnienie**. Język którym operuje Prolog składa się zgodnie ze standardem ISO (1995) operuje następującymi klasami symboli:

- Zmienne (*variables*) składające się ze znaków alfanumerycznych i zaczynają się od dużej litery, np. X, Xs, Y, Zeta ,...;
- Stałych (*constants*) składające się z cyfr lub znaków alfanumerycznych zaczynających się od małej litery lub cyfry, np. 45, x, alfa, none, 17,...;
- Funktorów (*functors*) składające się ze znaków alfanumerycznych i zaczynają się od małej litery alfabetu, np. *f*
- Predykaty (*predicate symbols*) składające się z ciągów małych liter;
- Funkcje logiczne (*logical connectives*): koniunkcja $\wedge$, negacja $\neg$, równoważność $\leftrightarrow$, implikacja $\rightarrow$, alternatywa $\vee$;
- Kwantyfikatorów (*quantifiers*): ogólny $\exists$ oraz szczegółowy $\forall$;
- Symbole pomocnicze (*auxiliary*), takie jak: nawiasy, przecinek, itp.

4.1.0.11. **Wyjaśnienie**. Programy napisane w języku Prologu zawierają dwie części składowe, nazwane odpowiednio:
(1) *bazą faktów* oraz
(2) *regułami postępowania*.
Podstawową jednostką w programach pisanych w języku Prologu jest *predykat*. Predykat składa się z nagłówka i argumentów, na przykład: ojciec(tomasz, agata), gdzie ojciec to nagłówek a tomasz i agata to argumenty; czytany Tomasz jest ojcem Agaty. Predykat może zostać użyty do wyrażenia pewnych faktów o świecie, które są znane programowi i mogą być składowane w bazie faktów. W oparciu o bazę faktów – stosując zdefiniowane reguły, można wyszukiwać odpowiedzi na zapytania formułowane w języku programowania Prolog.

Piśmiennictwo: *Ben-Ari M*. B.2.1., *Nilson U.* N.2.1.

4.1.1. Reguły i Baza faktów programu Prolog

4.1.1.10. **Wyjaśnienie**. Baza danych Prologu zawiera *reguły* i *fakty*.

4.1.1.20. **Wyjaśnienie**. Przykład reguły to:

```
jest(światło) :- włączony(przycisk).
```

Zapis „:-" oznacza „wtedy, gdy" lub „jeśli". Ta reguła oznacza, że zdanie jest (światło) jest prawdziwe wtedy, gdy prawdziwe jest zdanie włączony (przycisk). Reguły mogą używać zmiennych. Na przykład:

```
ojciec(X, Y) :- rodzic(X, Y), jest_rodzaju_męskiego(X).
```

To oznacza: „dla każdych X i Y, jeśli rodzic (X, Y) i jest rodzaju męskiego (X) to ojciec (X, Y). Przesłanka i wniosek są zapisane w odwrotnej kolejności niż to do czego przyzwyczailiśmy się w typowym zapisie implikacji. Co więcej, reguły muszą mieć predykat jako wniosek.

4.1.1.30. **Wyjaśnienie**. Można umieścić wiele predykatów we wniosku, połączonych koniunkcją, na przykład:

```
a,b,c :- d.
```

ale oznacza to tyle samo, co trzy oddzielne reguły:

```
a :- d.
b :- d.
c :- d.
```

Nie można napisać reguły „`a ∨ b :- c`", czyli „jeśli c to (a lub b)". Wynika to z ograniczenia zapisu do *klauzul Horna*[67].

4.1.1.40. **Wyjaśnienie**. Najprostszym przykładem faktów przechowywanych w bazie danych programu Prolog, są klauzule definiujące predykat wyrażający trójczłonową relację między ojcem, matką i ich dzieckiem. Na przykład:

```
1. rodzice(uranus, gaia, rhea).
2. rodzice(uranus, gaia, cronus).
3. rodzice(cronus, rhea, zeus).
4. rodzice(cronus, rhea, hera).
5. rodzice(cronus, rhea, demeter).
6. rodzice(zeus, leto, artemis).
7. rodzice(zeus, leto, apollo).
8. rodzice(zeus, demeter, persephone).
```

4.1.1.50. **Wyjaśnienie**. Relację między ojcem i dzieckiem oraz między matką i dzieckiem można zapisać w postaci reguł:

- `ojciec(X, Y) :- rodzice(X,_, Y).`
- `matka(X, Y) :- rodzice(_, X, Y).`

W powyższych regułach symbol :- czytamy jako "jeśli" i oznacza wynikanie warunku po lewej jego stronie z warunku po stronie prawej. Zatem reguły te możemy przeczytać następująco:

- jeśli X jest pierwszym rodzicem dla Y, to X jest ojcem Y;
- jeśli X jest drugim rodzicem dla Y, to X jest matką Y.

Relację bycia rodzicem możemy zdefiniować następująco:

- `rodzic(X, Y) :- ojciec(X, Y).`

---

[67] Klauzula Horna (*Horn clause*) to klauzula, w której co najwyżej jeden element jest niezanegowany. Przykładami takich klauzul są {p, ¬r ,¬q} i {¬r , ¬q}. Klauzule Horna zapisuje się zwykle w postaci implikacyjnej:

- $p \leftarrow r \wedge q$ albo wręcz w konwencji języka Prolog: p :- r, q
- $\leftarrow r \wedge q$:- r, q

- `rodzic(X, Y) :- matka(X, Y).`

Ten sam predykat dwuargumentowy rodzic możemy zapisać za pomocą jednej klauzuli używając spójnika alternatywy:

```
rodzic(X, Y) :- ojciec(X, Y); matka(X, Y).
```

Piśmiennictwo: *Ben-Ari M.* B.2.1., *Nilson U.* N.2.1.

4.1.2. Reguły wyszukiwania programu Prolog

4.1.2.10. **Wyjaśnienie**. Reguły wyszukiwania polegają na wybieraniu kolejnych klauzul od góry do dołu z listy klauzul danej procedury. Pamiętając, że notacja Prologu różni się od notacji matematycznej do której używania jesteśmy przyzwyczajeni: (a) nazwa zmiennej zaczyna się od dużej litery, (b) nazwa predykatu zaczyna się od małej litery, (c) para symboli „:-" jest używana zamiast symbol „←".

```
przodek (X,Y) :- rodzic (X,Y).
przodek (X,Y) :- rodzic (X,Z), przodek (Z,Y).
```

4.1.2.20. **Wyjaśnienie**. Baza danych zawiera fakty, o których zakładamy, że są prawdziwe, takie jak catherine jest matką allen'a. Przyjmując, że procedura przodek określa deklaratywnie znaczenie pojęcia relacji rodzicielskiej rodzic, wykonujemy obliczenia i stosujemy reguły wnioskowania w stosunku do klauzuli wynikowej

```
:- przodek(Y, bob), przodek(bob, Z).
```

otrzymujemy poprawną odpowiedz, w wyniku użycia podstawienia `Y = dave`, `Z = allen` oznaczają, że `dave` jest przodkiem `bob`'a, który z kolei jest przodkiem `allen`'a. Poniżej pokazany jest schemat wnioskowania:

```
1. :- przodek (Y, bob), przodek(bob, Z).
2. :- rodzic (Y, bob), przodek(bob, Z).   {podstawienie Y ⇐ dave}
3. :- przodek (bob, Z).
4. :- rodzic (bob, Z).                    {podstawienie Z ⇐ allen}
5. :-
```

Piśmiennictwo: *Ben-Ari M.* B.2.1., *Nilson U.* N.2.1.

4.1.4. Inne predykaty i arytmetyka

Dotychczas zajmowaliśmy się predykatami logicznymi programowania w notacji Prolog. Dla napisania programów uniwersalnych obsługujących urządzenia wejścia / wyjścia komputera oraz wykonywujące instrukcje arytmetyczne – rozszerzono listę predykatów z czynności będących operacjami logicznymi

4.1.3.10. **Wyjaśnienie**. Przykładem predykatów nie realizujących funkcjonalności logicznej są predykaty obsługujące czynność wejścia / wyjścia, takie `read` i `write`. Na przykład - użycie predykatu `write('Jak się macie')`, wypisuje na ekranie tekst 'Jak się macie'.

4.1.2.20. **Wyjaśnienie**. Prolog odchodzi od podejścia typowego dla teorii logiki - traktowania danych typu numerycznego, ze względu na dwa problemy dotyczące formalizacji: (a) wykonanie zapytania np. dotyczącego liczby zatrudnionych w organizacji, może prowadzić do uzyskania termu w rodzaju $f(f(f(f(f(a)))))$ - zamiast liczby 5; (b) nieefektywność zastosowania metod wnioskowania logicznego do obliczeń numerycznych.

4.1.2.30. **Wyjaśnienie**. Prolog wspiera standardową arytmetykę obliczeń komputerowych. Składnia opiera się o wyrażenia arytmetyczne, na podstawie których wyznaczany jest wynik działania. Poniższa klauzula pobiera z bazy danych – dla poszczególnych pozycji towarów

(`Item`), cennik (`Price`) oraz od obliczonej sumarycznej wartości sprzedaży (`List`) odejmuję obliczone na podstawie pobranego z bazy danych upustu (`Discount`) i wartości sprzedaży – należny upust, po czym oblicza cenę netto (`Selling_price`), jak niżej:

```
Selling_price (Item, Price) : -
      list_price (Item, List) ,
      discount_percent (Item, Discount),
      Price is List - List * Discount / 100.
```

Wynik wyrażenia arytmetycznego jest zawsze wartością numeryczną (liczbą). Predykaty arytmetyczne nie są wyrażeniami przypisującymi jedną z dwu wartości: *`True, False`*.

4.1.2.40. **Wyjaśnienie**. Najbardziej kontrowersyjną modyfikacją zasad programowania logicznego wprowadzoną w Prologu jest operator „*cut*". Rozważmy poniższy program obliczania N! (`factorial`):

```
factorial(0,1).
factorial(N,F): -
      N1 is N - 1,
      factorial (N1, F1) ,
      F is N * F1.
```

Jest to napisanie w Prologu poniższej formuły rekurencyjnej:

$$f(0) = 1,\ f(N) = N \cdot f(N-1).$$

Załóżmy z kolei, że procedura obliczania silni wywołuję następną procedurę, np. sprawdzającą (`check`) właściwości (`property`) kolejnych liczb silni:

```
check(N) :- factorial(N, F), property(F).
```

Jeśli procedura check jest wywołana dla N = 0, czemu odpowiada `factorial(0,F)`, to mamy do czynienia z wywołaniem property (`1`). Załóżmy, że to wywołanie nie może być obsłużone przez tą procedurę, którą to wywołanie w konsekwencji doprowadzi do zapętlenia programu *ad infinitum*. Aby uniknąć takiej katastrofy, twórcy Prologu wprowadzili operator *cut*, zapisany za pomocą wykrzyknika umieszczonego za znakiem podstawienia pierwszej klauzuli procedury:

```
factorial(0, 1) :- !.
```

Używając operatora *cut* przeciwdziałamy zapętleniu *ad infinitum* procedury rekurencyjnej.

Piśmiennictwo: *Ben-Ari M*. B.2.1., *Nilson U.* N.2.1.

4.2. Zastosowanie algebry Boole'a do układów elektronicznych

4.2.0. Uwagi wstępne

W dalszym ciągu będziemy mieli do czynienia z dwoma klasami układów podstawowych, zwanych odpowiednio układami kombinacyjnymi i układami i układami sekwencyjnymi. Układy kombinacyjne działają w ciągu jednostki czasu, czyli są układami pozbawionymi pamięci (o układach kombinacyjnych wspominaliśmy już w podrozdziale 3.8.2 – omawiając przykład takiego układu). Układy sekwencyjne są układami działającymi w czasie, dla systemów synchronicznych – synchronizowanych impulsami zegarowymi systemu, natomiast dla systemów asynchronicznych synchronizowanych zdarzeniami systemowymi (np. osiąganiem określonych stanów).

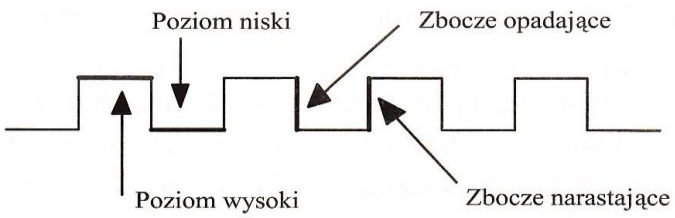
Kombinacyjne układy logiczne są budowane z elementów zwanych bramkami – połączonymi przewodami. Kombinacyjny układ logiczny ma stałą liczbę wejść i stałą liczbę wyjść, realizujących określone funkcje Boole'a. Wejścia i wyjścia przyjmują wartości należące do zbioru

dwu-elementowego {0, 1}, które oznaczają odpowiednio wartości *false* (*fałsz*) oraz *true* (*prawda*).

Sekwencyjne układy logiczne zbudowane są z kombinacji układów zwanych przerzutnikami oraz układów kombinacyjny. Z kolei przerzutniki również zbudowane są z układów bramkowych. Zasadnicza różnica w budowie układów kombinacyjnych i przerzutników polega na tym, że wchodzące w skład układów kombinacyjnych bramki połączone są między sobą szeregowo lub równolegle, natomiast w przerzutnikach - wykorzystane są sprzężenia zwrotne pomiędzy bramkami, co w konsekwencji pozwala na uzyskanie efektu zapamiętania stanów w kolejnych jednostkach kwantach czasu.

Taktowanie pracy elektronicznych układów cyfrowych odbywa się na jednej z dwóch dróg:

1. W przypadku układów asynchronicznych, role taktowania pełnią kolejne zdarzenia (np. uzyskanie określonego stanu).



Rysunek 4.2.0.01- Cyfrowy przebieg zegarowy.

2. W przypadku układów synchronicznych, za synchronizację odpowiada urządzenie taktujące zwane zegarem lub generatorem impulsów zegarowych. Cyfrowy przebieg zegarowy jest pokazany na rysunku 3.5.0.01.

Piśmiennictwo: *Mano M.* M.1.1., *Shannon C.* S.1.1., *Stallings W.* S.11.1.

4.2.1. Bramki – realizacja funkcji algebry Boole'a

4.2.1.01. **Wyjaśnienie**. Bramką (*gate*) nazywa się układ elektroniczny realizujący funkcję boolowską, posiadający określoną liczbę wejść i jedno wyjście. Jak każdy układ elektroniczny, tak i bramki opisywane są wieloma parametrami zarówno funkcjonalnymi (liczba wejść, liczba wyjść, realizowana funkcja, przeznaczenie i in.), jak i elektrycznymi (pobierana moc zasilania, obciążalność prądem układów sterujących wejściami, możliwość wysterowania wejść innych układów itp.) oraz dynamicznymi (czasy zmiany sygnału na wyjściu układu, wnoszone opóźnienia i inne).

4.2.1.02. **Wyjaśnienie**. Bramki produkowane są, jako układy scalone. Do produkcji układów scalonych były i są stosowane różne technologie. Pierwsza z nich to technologia TTL (*transistor-transistor logic*), jedną z następnych CMOS (*complementary MOS*). W jednym układzie scalonym znajdowało się początkowo kilka bramek. Natomiast współczesne obwody scalone wielkiej skali integracji - VLSI (czyli Very Large Scale Integration) zawierają już miliony bramek.

4.2.1.03. **Wyjaśnienie**. W roku 1938 *Claude Shannon*, wówczas asystent na wydziale elektrycznym MIT (późniejszy twórca teorii informacji), zasugerował zastosowanie algebry Boole'a do rozwiązywania problemów projektowania układów przekaźnikowych. Metody *Shannona* zostały następnie użyte do analizowania oraz projektowania elektronicznych układów cyfrowych.

4.2.1.04. **Wyjaśnienie**. Bramkę, można zdefiniować na jeden z trzech sposobów:

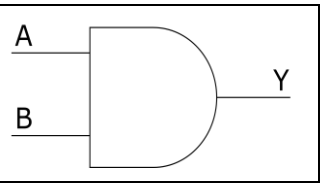
1. Tablicą zero-jedynkową. Dla każdej z 2 możliwych kombinacji sygnałów wejściowych podana jest wartość binarna każdego z m sygnałów wyjściowych.
2. Schematem graficznym – czyli schematem połączeń bramek.
3. Funkcjami Boole'owskimi – gdzie każdy sygnał wyjściowy jest wyrażony jako funkcja sygnałów wejściowych.

4.2.1.05. **Wyjaśnienie**. Jednym z parametrów charakterystycznych każdej bramki, jest czas propagacji sygnału, czyli opóźnienie, jakie wprowadza układ, transmitując dany sygnał oddziaływujący na wejście lub wejścia bramki na wyjście bramki. Uwaga: w dalszym ciągu, zmienne Booleo'wskie będziemy oznaczali dużymi literami alfabetu łacińskiego, zgodnie z przyjętymi konwencjami w elektronice, dotyczących zapisywania tych zmiennych.

4.2.1.11. **Definicja**. Bramka AND realizuje iloczyn logiczny. Jeżeli na jej wejściach podane są jedynki to na wyjściu jest jedynka, w każdym innym przypadku na wyjściu jest zero: Y = A ∧ B Bramka AND może być bramką wielowejściową ≥ 2. Tabliczka 3.5.1.11 opisuje bramkę AND o dwóch wejściach i jednym wyjściu.

Tabliczka 4.2.1.11

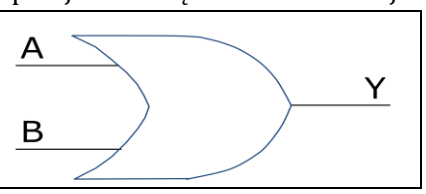
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1



4.2.1.12. **Definicja**. Bramka OR realizuje sumę logiczną. Jeżeli przynajmniej na jednym wejściu podana jest jedynka, to na wyjściu jest jedynka: $Y = A \vee B$. Bramka OR może być bramką wielowejściową ≥ 2. Tabliczka 3.5.1.11 opisuje bramkę OR o dwóch wejściach i jednym wyjściu.

Tabliczka 4.2.1.12

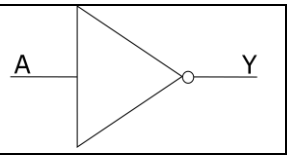
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1



4.2.1.13. **Definicja**. Bramka NOT realizuje operację zaprzeczenia. Jeżeli na wejściu podana jest jedynka, to na wyjściu będzie zero i odwrotnie: $Y = \neg A$. Bramka NOT, opisana tabliczką 4.2.1.13, jest zawsze bramką jednowejściową z jednym wyjściem.

Tabliczka 4.2.1.13

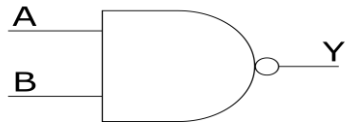
A	Y
0	1
1	0



4.2.1.14. **Definicja**. Bramka NAND jest złożona z bramek NOT i AND. Zasada działania jest taka sama jak bramki AND z tą różnicą, że sygnał wyjściowy jest jeszcze negowany: $Y = \neg(A \wedge B)$ Bramka NAND, może być bramką wielowejściową ≥ 2. Tabliczka 4.2.1.14 opisuje bramkę NAND o dwóch wejściach i jednym wyjściu.

Tabliczka 4.2.1.14

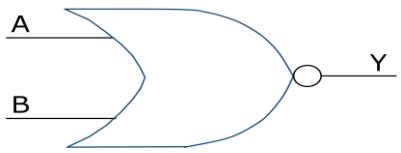
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0



4.2.1.15. **Definicja**. Bramka NOR jest złożona z bramek: NOT i OR. Zasada działania jest taka sama jak bramki OR z tą różnicą, że sygnał wyjściowy jest jeszcze negowany: $Y = \sim (A \vee B)$ Bramka NOR, może być bramką wielowejściową ≥ 2. Tabliczka 4.2.1.15 opisuje bramkę NOR o dwóch wejściach i jednym wyjściu.

Tabliczka 4.2.1.15

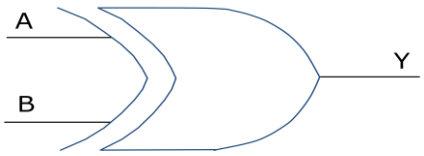
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0



4.2.1.16. **Definicja**. Bramka XOR, jeżeli sygnały wejściowe są sobie równe (A=B=0 lub A=B=1), to na wyjściu (Y) jest zero: $Y = (A \wedge B) \vee [(\neg A) \wedge (\neg B)]$. Bramka NOR, opisana tabliczką 4.2.1.15 jest bramką dwuwejściową o jednym wyjściu.

Tabliczka 4.2.1.16

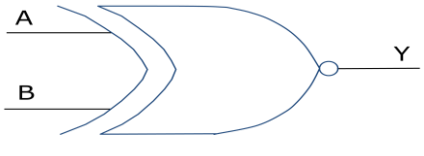
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1



4.2.1.17. **Definicja**. Bramka XNOR, jeżeli sygnały wejściowe są sobie równe (A=B=0 lub A=B=1), to na wyjściu (Y) jest jedynka: $Y = [(\neg A) \wedge B] \vee [A \wedge (\neg B)]$. Bramka NOR, opisana tabliczką 4.2.1.15 jest bramką dwuwejściową o jednym wyjściu.

Tabliczka 4.2.1.17

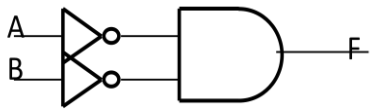
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

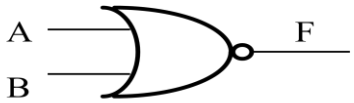


4.2.1.21. **Wyjaśnienie**. Z pierwszego twierdzenia DeMorga'na (patrz 4.2.4.11.) - wynika równoważność funkcjonalna sieci trzech bramek i bramki NOR – pokazana w tabliczce 4.2.1.21:

Tabliczka 3.5.1.21

A	B	F
0	0	1
0	1	1
1	0	1
1	1	0

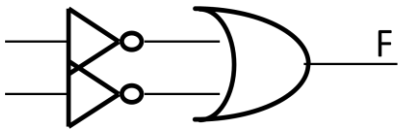


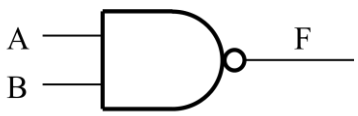


4.2.1.22. **Wyjaśnienie**. Z drugiego twierdzenia DeMorga'na (patrz 4.2.4.12.) - wynika równoważność funkcjonalna sieci trzech bramek i bramki NAND – pokazana w tabliczce 4.2.1.22:

Tabliczka 3.5.1.22

A	B	F
0	0	1
0	1	0
1	0	0
1	1	0





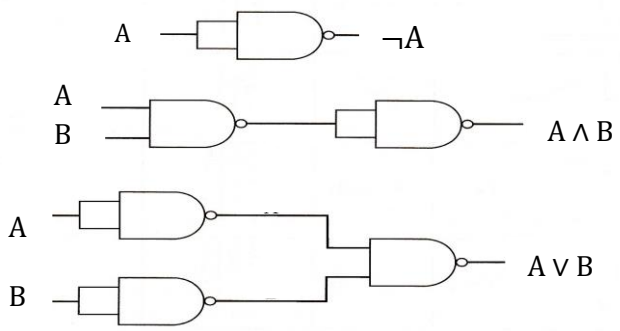
4.2.1.41. **Wyjaśnienie**. Z bramek budowane są układy kombinacyjne, takie jak: Multiplexery, Kodery i Dekodery.

4.2.1.51. **Wyjaśnienie**. Z bramek budowane są przerzutniki – składowe układów sekwencyjnych, takie jak: Przerzutnik S/R, Przerzutnik synchroniczny S/R, Przerzutnik D i Przerzutnik J/K.

Piśmiennictwo: *Hyde R.* H.5.1., *Mano M.* M.1.1., *Stallings W.* S.11.1.

4.2.2. UKŁADY KOMBINACYJNE

4.2.2.11. **Wyjaśnienie**. Układ kombinacyjny jest zbiorem wzajemnie połączonych bramek, którego stan wyjść w dowolnej chwili (kwancie czasowym – określonym taktem zegara) jest wyłącznie funkcją stanu wejść w tej samej chwili. Podobnie jak w przypadku pojedynczej bramki, po ustaleniu stanu na wejściu – prawie natychmiast pojawia się sygnał na wyjściu, przy czym występuje tylko opóźnienie bramkowe, odpowiadające czasowi propagacji sygnału elektrycznego w bramce. Ogólnie rzecz biorąc, układ kombinacyjny zawiera n wejść i m wyjść binarnych i realizuje określone funkcje boolowskie.



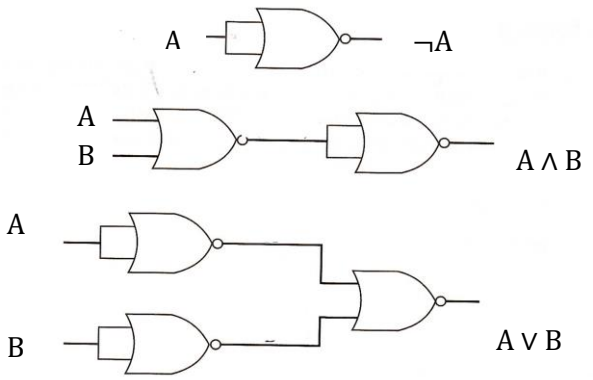
Rysunek 4.2.2.12 Bramka NAND umożliwia realizację funkcji Boole'a NOT, OR i AND

Układy kombinacyjne budowane są z bramek. Złożoność funkcji jest ograniczoną związkiem pomiędzy długością sumy czasu propagacji najdłuższej ścieżki danych binarnych układu kombinacyjnego – a długością sygnału pojedynczego taktu zegara. Suma czasu propagacji sygnału w układzie kombinacyjnym nie może przekraczać czasu trwania pojedynczego sygnału taktującego zegara.

4.2.2.12. **Wyjaśnienie**. Zastosowanie bramki NAND. Bramka NAND umożliwia zastąpienie standardowych bramek NOT, OR i AND – tak jak pokazano na rysunku 4.2.2.12:

4.2.2.13. **Wyjaśnienie**. Zastosowanie bramki NOR. Bramka NOR umożliwia zastąpienie standardowych bramek NOT, OR i AND – tak jak pokazano na rysunku 3.5.2.13.

Współczesne obwody scalone wielkiej skali integracji - VLSI (czyli Very Large Scale Integration) zawierające już miliony bramek są zbudowane z jednego typu bramek np. NAND, co w konsekwencji pozwala w istotny sposób zwiększyć upakowanie układu scalonego, czyli umieścić na płytce krzemu większą liczbę bramek składających się na bardziej złożony układ scalony, np. mikroprocesor. Jak już powiedzieliśmy, układ kombinacyjny zawiera n wejść i m wyjść binarnych. Dowolna funkcja Boole'a może być zrealizowana w postaci elektronicznej jako sieć bramek realizujących układ kombinacyjny.

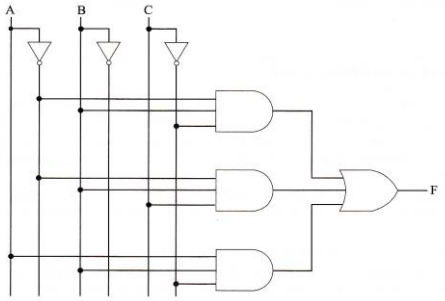


Rysunek 4.2.2.13 Bramka NOR umożliwia realizację funkcji Boole'a NOT, OR i AND

Dla każdej funkcji istnieje pewna liczba alternatywnych postaci funkcji. Jako przykład rozważmy funkcję Boole'a realizującą tabelkę pokazaną na rysunku 4.2.2.14. Jedną z postaci alternatywnych funkcji F, możemy wyrazić po prostu przez wyszczególnienie kombinacji wartości zmiennych A, B i C, które powodują, że F jest równe 1. W tym przypadku są to trzy kombinacje. W dolnej części rysunku 4.2.2.14, podana jest zbudowana na powyższej zasadzie funkcja Boole'owska.

Trzy równorzędne sposoby prezentacji funkcji **F** – tabelka,
schemat (zbudowany z bramek NOT, AND i OR) i funkcja Boole'a.

A	**B**	**C**	**F**
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0



$$\mathbf{F} = [(\neg\mathbf{A})\wedge\mathbf{B}\wedge(\neg\mathbf{C})] \vee [(\neg\mathbf{A})\wedge\mathbf{B}\wedge\mathbf{C}] \vee [\mathbf{A}\wedge\mathbf{B}\wedge(\neg\mathbf{C})]$$

Rysunek 4.2.2.14

4.2.2.21. **Wyjaśnienie**. Uproszczenia algebraiczne wyrażeń Boole'a. Uproszczenie algebraiczne polega na stosowaniu tożsamości przedstawionych w rozdziale 2.2 do redukowania liczby elementów i ich złożoności w wyrażeniach Boole;a. Rozważmy jako przykład wyrażenie:

4.2.2.22. $$\mathbf{F}=[(\neg\mathbf{A})\wedge\mathbf{B}]\vee[\mathbf{B}\wedge(\neg\mathbf{C})];$$

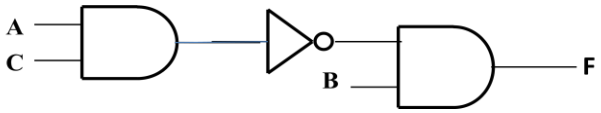
równoważne wcześniej omawianemu wyrażeniu równoważnemu tabelce z rysunku 4.2.2.14. Wyrażenie powyższe możemy przekształcić dalej do postaci:

4.2.2.23. $$\mathbf{F} = \mathbf{B} \wedge [(\neg\, \mathbf{A}) \vee (\neg\mathbf{C})].$$

Wykorzystując następnie jedno z twierdzeń De Morgana, otrzymamy koleją uproszczoną postać:

4.2.2.24. $$\mathbf{F} = \mathbf{B} \wedge [\neg\, (\mathbf{A} \wedge \mathbf{C})].$$

Sieć bramek odpowiadająca powyższemu wyrażeniu zawiera jedną bramkę NOT i dwie bramki AND, patrz rysunek 4.2.2.25.



Rysunek 4.2.2.25 Sieć bramkowa równoważna sieci pokazanej na rysunku 4.2.2.14.

Kolejnym problemem realizacji funkcji Boole'a jest typ użytych bramek. Nie jednokrotnie (o czym już wspominaliśmy wcześniej) układ kombinacyjny jest zbudowany wyłącznie za pomocą bramek NAND lub wyłącznie za pomocą bramek NOR. Na ogół nie jest to realizacja o minimalnej liczbie bramek, ma ono zaletę regularności – powtarzalności, co istotnie upraszcza proces projektowania układów scalonych VLSI.

4.2.1.31. **Wyjaśnienie**. Funkcje Boole'a częściowo określone. W pewnych przypadkach układ kombinacyjny może dotyczyć sytuacji, w której pewne kombinacje stanów wejściowych układu nigdy nie wystąpią. Mamy wówczas do czynienia z sytuacją istnienia pewnej dowolności wyboru rozwiązania układowego, ponieważ istnieje kilka funkcji Boole'a, które dla wybranych stanów wejściowych – tych, które mogą występować przyjmują zadane wartości, natomiast dla pozostałych stanów wejściowych – tych, które nie mogą wystąpić, przyjmują różne wartości. Z pośród tych funkcji Boole'a – wybieramy tę, która jest najprostsza w realizacji bramkowej. Wybór funkcji, może np. zależeć od zbioru bramek wybranych dla naszego projektu. Dla potrzeb wyboru funkcji Boole'a o umownie najprostszej postaci, ze względu na przyjęte kryterium wyboru, opracowano szereg metod. Jedną z pierwszych była metoda zwana *Mapą Karnaugha* (wygodna do ręcznego stosowania w przypadku małej liczby zmiennych 4 ÷ 6). Bardziej uniwersalną jest metoda *Quine'a-McKluskeya*, która nadaje się do oprogramowania i tym samym jest dzisiaj powszechnie stosowana przez projektantów cyfrowych obwodów scalonych.

Piśmiennictwo: *Hyde R.* H.5.1., *Mano M.* M.1.1., *Stallings W.* S.11.1.

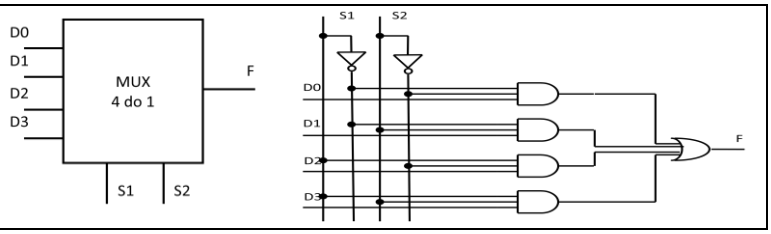
4.2.3. MULTIPLEKSERY, DEKODERY, PAMIĘĆ STAŁA (ROM)

4.2.3.10. **Wyjaśnienie**. Multiplekser jest układem kombinacyjnym, który łączy wiele wejść, (np. D0, D1, D2 i D3) z jednym wyjściem F, a o którego działaniu decydują sygnały sterujące (np. S1 i S2). Na wyjście F multipleksera dostarczany jest sygnał z jednego wybranego wejścia. Sygnały sterujące multiplekserem decydują o wyborze wejścia. Multipleksery są używane w układach cyfrowych do sterowania przepływem sygnałów i danych. Przykładem jest ładowanie licznika programu (PC). Liczba wprowadzana do licznika programu, może pochodzić z jednego z kilku źródeł:

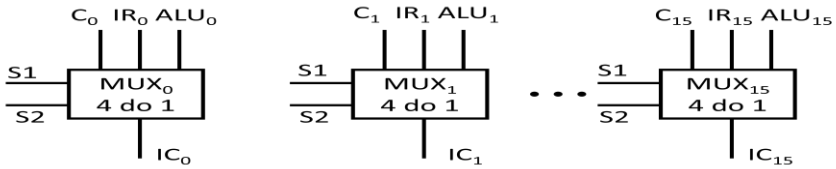
- Układu pół-sumatora, jeśli stan PC ma być inkrementowany w celu określenia adresu następnego rozkazu
- Rejestru rozkazu (IR), jeśli został właśnie wykonany rozkaz skokowy używający adresu bezpośredniego
- Wyjścia ALU, jeśli rozkaz skokowy określa adres, używając trybu indeksowego.

Tabliczka 4.2.3.11

S2	S1	F
0	0	D0
0	1	D1
1	0	D2
1	1	D3



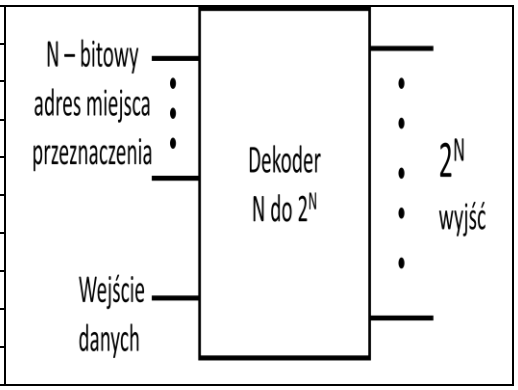
Tabliczka 4.2.3.11 pokazująca, jakim kombinacjom sygnałów sterującym S1, S2 odpowiada przełączenie poszczególnych wejść na wyjście oraz symbol graficzny multipleksera i sieć bramek składających się na jego układ kombinacyjny. Przykład pokazany na rysunku 4.2.3.12, dotyczący 16 bitowego adresu (IC_0, IC_1, ... , IC_{15}) ładowanego równolegle do rejestru IC (*Instruction Counter*). Źródła zasilania rejestr C - (C_0, C_1, ... , C_{15}), rejestr instrukcji IR - (IR_0, IR_1, ... , IR_{15}) oraz jednostka arytmetyczno-logiczna ALU – są połączone z liniami wejściowymi poszczególnych multiplekserów, zaś wyjścia tych ostatnich – połączone są do poszczególnych bitów rejestru IC.



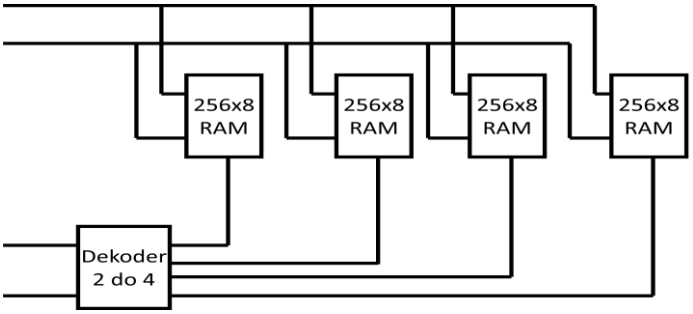
Rysunek 4.2.3.12. Wejście multiplekserowe do rejestru PC

Tabliczka 4.2.3.21

A	B	C	D0	D1	D2	D3	D4	D5	D6	D7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1



4.2.3.20. **Wyjaśnienie**. Dekoder jest układem kombinacyjnym o pewnej liczbie linii wyjściowych, z których w określonej chwili sygnał może być wysłany na jedynie jedną linię, zależnie od kombinacji sygnałów na liniach wejściowych. Ogólnie mówiąc, dekoder ma N wejść i 2^N wyjść.



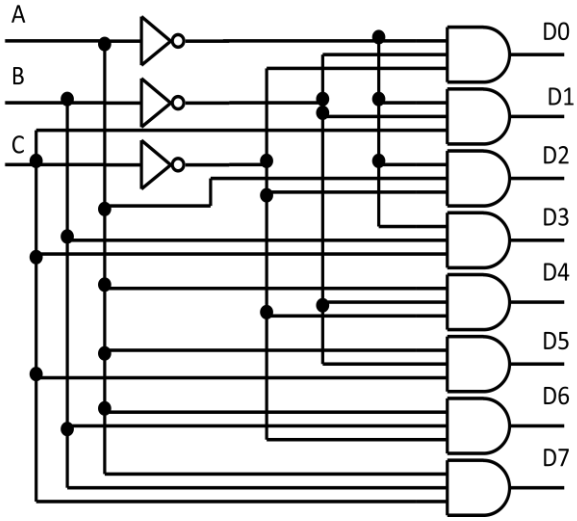
Rysunek 4.2.3.22 Przykład schematyczny układu kombinacyjnego dekodowania adresu

Dekodery znajdują wiele zastosowań w komputerach. Jednym z przykładów zastosowań jest dekodowanie adresu. Załóżmy dalej, że chcemy zbudować pamięć 1-kilobajtową przy użyciu 4 układów RAM o pojemności 256x8 bitów każdy. Przy czym, chcemy mieć jedną zunifikowaną przestrzeń adresową, którą możemy podzielić następująco:

- Adresy szesnastkowe od **0000** do **00FF** układ pamięciowy RAM **0.**
- Adresy szesnastkowe od **0100** do **01FF** układ pamięciowy RAM **1.**
- Adresy szesnastkowe od **0200** do **02FF** układ pamięciowy RAM **2.**

- Adresy szesnastkowe od **0300** do **03FF** układ pamięciowy RAM **3.**

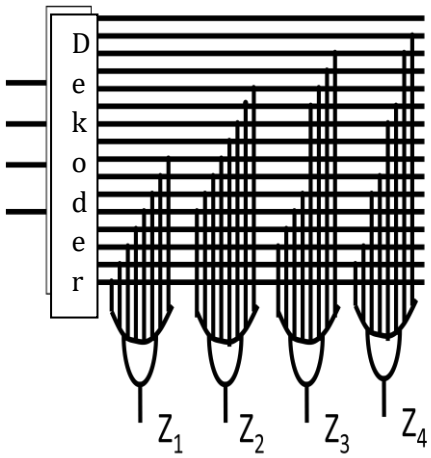
Tabliczka 4.2.3.21 - pokazuje translacje adresów 3 – bitowych dla potrzeb wybrania jednej z ośmiu linii wyjściowych, natomiast rysunki 4.2.3.22 i 4.2.3.23 - przykład układu kombinacyjnego dekodowania adresu.



Rysunek 4.2.3.22 Schemat bramkowy dekodera.

Tabelka 3.5.3.31 Przykład pamięci stałej ROM

Linie wejściowe				Linie wyjściowe			
X_1	X_2	X_3	X_4	Z_1	Z_2	Z_3	Z_4
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0



4.2.3.30. **Wyjaśnienie**. Pamięć stała ROM. Układy kombinacyjne są często określane jako układy „bez pamięci", ponieważ ich wyjścia zależą jedynie od bieżącego stanu ich wejść, a żadne informacje historyczne o poprzednich stanach wejść nie są zachowywane. Istniej jednak pewien rodzaj pamięci, który jest realizowany za pomocą układów kombinacyjnych, a mianowicie pamięć stała ROM. ROM jest pamięcią, która umożliwia wykonanie wyłącznie operacji odczytu. Informacja binarna zawarta w ROM jest w związku z tym trwała. Jest ona zapisywana w pamięci ROM w toku procesu wytwarzania modułu pamięci. Wobec tego określona kombinacja sygnałów wejściowych ROM – podana na linie adresowe, prowadzi zawsze do tej samej kombinacja sygnałów na liniach wyjściowych. W tabelce 4.2.3.31 przedstawiony jest przykład pamięci stałej

ROM o czterech liniach wejściowych oraz czterech liniach wyjściowych oraz schemat bramkowy tej przykładowej pamięci ROM.

Piśmiennictwo: *Hyde R.* H.5.1., *Mano M.* M.1.1., *Stallings W.* S.11.1.

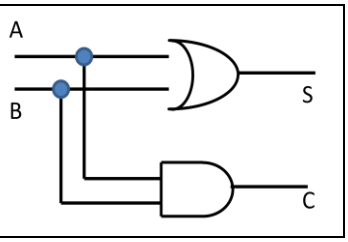
4.2.4. SUMATORY

Obszarem o zasadniczym znaczeniu, którego jeszcze nie rozpatrywaliśmy, są operacje arytmetyczne. Obecnie omówimy, jako przykład funkcję dodawania binarnego liczb. Dodawanie binarne można wyrazić za pomocą wyrażeń algebry Boolea. Suma binarna różni się tym od sumy w algebrze Boole'a, że wynik obejmuje przeniesienie pomiędzy pozycjami binarnymi dodawanych liczb binarnych.

4.2.4.00. **Wyjaśnienie**. Półsumator binarny jest najprostszym układem dodającym liczby binarne, a dokładniej mówiąc dwie cyfry binarne A i B, dając w wyniku liczbę binarną dwucyfrową CS; gdzie cyfra binarna S jest sumą na pozycji 2^0, zaś cyfra binarna C jest przeniesieniem wyniku sumowania na pozycję 2^1 (patrz tabliczka 3.5.4.01. Nazwa półsumator pochodzi od możliwości złożenia szeregowego sumatora binarnego z pary półsumatorów.

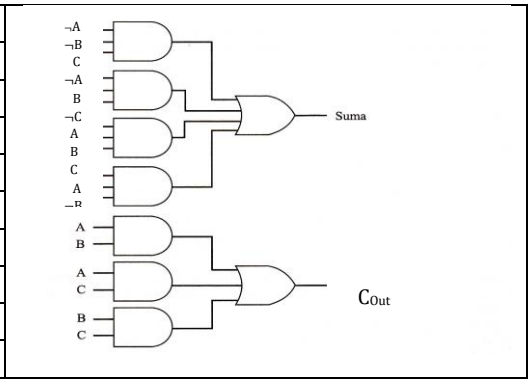
Tabliczka 4.2.4.01

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Tabliczka 4.2.4.11

C_{in}	A	B	Suma	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



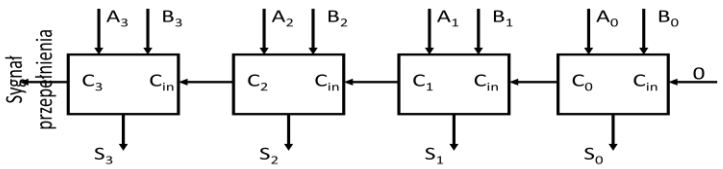
4.2.4.10. **Wyjaśnienie**. Sumator binarny szeregowy realizuję funkcję dodawania binarnego pary liczb binarnych całkowitych, zawierających po n pozycji, począwszy od 2^0, skończywszy na pozycji $2^{(n+1)}$. Tablica dodawania binarnego pary cyfr binarnych i przeniesienia z niższej pozycji potęgowej na wyższą, przedstawia tabliczka 4.2.4.11. Sumator binarny szeregowy działa w n krokach zegarowych. Dodanie, więc 16 – cyfrowych liczb binarnych całkowitych, wymaga wykonania 16 cykli zegara taktującego pracę procesora. Wzory na stany dwóch wyjść sumatora, nazwanych w tabelce 4.2.4.11, Suma i C_{out} odpowiednio jest określona parą wzorów, dla k-tej pozycji binarnej sumowania (gdzie k =0, 1, …, n-1):

4.2.4.12. $$\text{Suma} = [(\neg A)\wedge(\neg B)\wedge C_{in}] \vee [(\neg A)\wedge B\wedge(\neg C_{in})] \vee (A\wedge B\wedge C_{in}) \vee [A\wedge(\neg B)\wedge(\neg C_{in})].$$

4.2.4.13. $$C_{out} = (A\wedge B) \vee (A\wedge C_{in}) \vee (B\wedge C_{in}).$$

4.2.4.20. **Wyjaśnienie**. Sumator czterobitowy równoległy. Dysponujemy już, zgodnie z powyższymi wyjaśnieniami, układami logicznymi niezbędnymi do budowy wielobitowego

sumatora. Zauważmy, że gdybyśmy chcieli złożyć sumator wielobitowy z modułów sumatora jednobitowego, to ponieważ wartość wyjściowa każdego modułu - kroku działania sumatora zależy od przeniesienia z poprzedniego kroku, istnieje opóźnienie narastające od najmniej znaczącego do najbardziej znaczącego bitu, ponieważ w każdym sumatorze jednobitowym następują pewne opóźnienia bramkowe, które się kumulują. Dlatego sumator wielobitowy, zbudowany z sumatorów jednio-bitowych, tak jak to pokazuje rysunek 4.2.4.21, nie będzie działał poprawnie.



Rysunek 4.2.4.21 Ideowy schemat sumatora czterobitowego równoległego.

Gdyby wartości przeniesienia mogły być określone bez przechodzenia przez wszystkie pośrednie szczeble, każdy sumator jednobitowy mógłby działać niezależnie i opóźnienia by się nie kumulowały. Można to osiągnąć, stosując rozwiązanie nazywane układem przeniesienia na bardziej znaczące pozycje (*carry look-a-head*).

4.2.4.30. **Wyjaśnienie**. Mechanizm *carry look-a-head.* Żeby otrzymać wyrażenia, które określają wartości na wejściu dowolnego szczebla sumatora, bez odnoszenia się do poprzednich wartości przeniesienia, należy przyjąć:

4.2.4.31. $$C_0 = A_0 \wedge B_0;$$

4.2.4.32. $$C_1 = (A_1 \wedge B_1) \vee [(A_1 \vee B_1) \wedge C_0];$$

Podstawiając równości (3.5.4.31.) do (3.5.4.32.) otrzymamy:

4.2.4.33. $$C_1 = (A_1 \wedge B_1) \vee (A_1 \wedge B_1 \wedge B_0) \vee (B_1 \wedge A_0 \wedge B_0);$$

Powtarzając tę samą procedurę, otrzymujemy:

4.2.4.34. $$C_2 = (A_2 \wedge B_2) \vee (A_2 \wedge A_1 \wedge B_1) \vee (A_2 \wedge B_1 \wedge A_0 \wedge A_1) \vee (B_2 \wedge A_1 \wedge B_1) \vee (B_2 \wedge A_1 \wedge A_0 \wedge B_0) \vee (B_2 \wedge B_1 \wedge A_0 \wedge B_0);$$

Postępowanie to możemy powtarzać dla dowolnie długich sumatorów. Każdy składnik przeniesienia może być wyrażony w postaci sumy logicznej iloczynów logicznych, jako funkcja wyłącznie oryginalnych danych wejściowych, bez zależności od przeniesień. Warto podkreślić, że w przypadku długich liczb binarnych rozwiązanie powyższe może prowadzić do bardzo rozbudowanych układów kombinacyjnych. Praktycznie układ przeniesień *carry look-a-head* jest zwykle realizowany dla sumatorów 4 ÷ 8 bitowych.

4.2.4.40. **Wyjaśnienie**. Sumator binarno-dziesiętny. Dla omówienia tego rozwiązania ponownie rozważmy sumator 4-bitowy. Współczesne komputery wykorzystują czterobitowe kombinacje do przechowywania liczb dziesiętnych. W tym celu, wykorzystuje się pierwszych dziesięć kombinacji z ogółu szesnastu dostępnych. Jest to tzw. metoda BCD (*Binary Coded Decimal*), w której kombinacja 0000 odpowiada 0 dziesiętnemu, 0001 odpowiada 1 dziesiętnej, 0010 odpowiada dwójce dziesiętnej, 0011 – trójce, itd., aż do 1001 odpowiadającej cyfrze 9 dziesiętnej. Kombinacje od 1010 do 1111 nie są wykorzystane. Wykonywanie operacji

dodawania liczb w kodzie BCD wymaga zmodyfikowania metody tworzenia przeniesień. Tabela 3.5.4.41, pokazuje wartości przeniesień dla sumy pary liczb dziesiętnych, począwszy od pary (0, 0), a skończywszy na parze (9, 9).

Tabela 4.2.4.41 Sumy pary cyfr dziesiętnych z przeniesieniem 0 lub 1 na wyższą pozycję										
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0+0=00	0+1=01	0+2=02	0+3=03	0+4=04	0+5=05	0+6=06	0+7=07	0+8=08	0+9=09
1	1+0=01	1+1=02	1+2=03	1+3=04	1+4=05	1+5=06	1+6=07	1+7=08	1+8=09	1+9=10
2	2+0=02	2+1=03	2+2=04	2+3=05	2+4=06	2+5=07	2+6=08	2+7=09	2+8=10	2+9=11
3	3+0=03	3+1=04	3+2=05	3+3=06	3+4=07	3+5=08	3+6=09	3+7=10	3+8=11	3+9=12
4	4+0=04	4+1=05	4+2=06	4+3=07	4+4=08	4+5=09	4+6=10	4+7=11	4+8=12	4+9=13
5	5+0=05	5+1=06	5+2=07	5+3=08	5+4=09	5+5=10	5+6=11	5+7=12	5+8=13	5+9=14
6	6+0=06	6+1=07	6+2=08	6+3=09	6+4=10	6+5=11	6+6=12	6+7=13	6+8=14	6+9=15
7	7+0=07	7+1=08	7+2=09	7+3=10	7+4=11	7+5=12	7+6=13	7+7=14	7+8=15	7+9-16
8	8+0=08	8+1=09	8+2=10	8+3=11	8+4=12	8+5=13	8+6=14	8+7=15	8+8=16	8+9=17
9	9+0=09	9+1=10	9+2=11	9+3=12	9+4=13	9+5=14	9+6=15	9+7=16	9+8=17	9+9=18

Jak widać przeniesienie może przyjmować jedną z dwóch wartości 0 lub 1. Jeżeli wynik dodawania jest mniejszy lub równy 9, to jak widać z tabeli 4.2.4.41, działanie sumatora czterobitowego daje poprawny wynik dla działań w systemie dziesiętnym. Natomiast wynik 10 ÷ 15 wymaga wykonania działania korekcyjnego wyniku i przeniesienia. Tabliczka 4.2.4.42 zawiera cztery bity wyniku sumowania binarnego X, Y, W, Z oraz przeniesienie C' i skorygowane bity wyniku X', Y', W', Z'.

Tabliczka 4.2.4.42 – Korekty wyniku sumowania									
Wynik	X	Y	W	Z	C'	X'	Y'	W'	Z'
10	1	0	1	0	1	0	0	0	0
11	1	0	1	1	1	0	0	0	1
12	1	1	0	0	1	0	0	1	0
13	1	1	0	1	1	0	0	1	1
14	1	1	1	0	1	0	1	0	0
15	1	1	1	1	1	0	1	1	0

Jak widać, układ kombinacyjny korekty, jest funkcją czterech zmiennych X, Y, W, Z – wyników działania sumatora czterobitowego i daje w wyniku pięć wartości korygujących wynik sumatora binarnego: C', X', Y', W', Z'. Poniżej podajemy wzory na funkcje Boole'a korekty:

4.2.4.43. $$C' = X \wedge [(\neg Y \wedge W) \vee Y];$$

4.2.4.44. $$X' = 0 ;$$

4.2.4.45. $$Y' = X \wedge Y \wedge W ;$$

4.2.4.46. $$W' = X \wedge Y \wedge [(\neg W) \wedge Z \vee W];$$

4.2.4.47. $$Z' = X \wedge Z \wedge [(\neg Y) \wedge W \vee Y \wedge (\neg W)];$$

4.2.4.50. **Wyjaśnienie**. Sumator 64 – bitowy, typowy dla współczesnych mikroprocesorów składa się z ośmiu 8 – bitowych sumatorów, z których każdy z osobna działa w oparciu o mechanizm *carry look-a-head* i z kolei zestaw sumatora 64 – bitowego - działa łącznie w oparciu o mechanizm *carry look-a-head* dla przeniesień z ośmiu 8 – bitowych sumatorów.

Piśmiennictwo: *Hyde R.* H.5.1., *Mano M.* M.1.1., *Stallings W.* S.11.1.

4.2.5. UKŁADY SEKWENCYJNE, PRZERZUTNIKI

Układy kombinacyjne służą do wdrażania podstawowych funkcji komputera. Jednak, poza szczególnym przypadkiem pamięci ROM, nie umożliwiają one zrealizowania układów pamięci lub przechowywania informacji o stanie, które również mają zasadnicze znaczenie dla działania komputera. Bieżący stan wyjścia układu sekwencyjnego zależy nie tylko od bieżącego stanu wejścia, ale również od historii stanu wejścia. Inaczej mówiąc, bieżący stan wyjścia układu sekwencyjnego zależy od bieżącego stanu wejścia oraz od bieżącego stanu samego układu sekwencyjnego.

W podrozdziale 2.3.6, wprowadzone zostały, między innymi, pojęcia trzech operatorów:

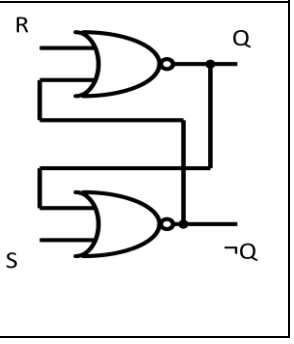
- Operator O następstwa czasu („the next time operator" O), który w przypadku czasu dyskretnego mówi o następnej jednostce (kwancie) czasu.
- S (p, q) - "p ma wartość prawda, tak długo jak q ma wartość prawda".
- U (p, q) - "p będzie miało wartość prawda, gdy q będzie miało wartość prawda".

Powyższe operatory są wzorcem logicznym dla opisania działania układów sekwencyjnych, czyli układów pamiętających stany i przenoszące te stany (wynik działania układów kombinacyjnych) pomiędzy kolejnymi quantami czasu – odpowiadającym kolejnym sygnałom synchronizującym.

4.2.5.01. **Wyjaśnienie**. Najprostszą formą układu sekwencyjnego jest przerzutnik, czyli układ dwustanowy i dwu wejściowy. Jedno wejście „set" decyduje ustawieniu przerzutnika w stanie 1, drugie wejście „reset" decyduje o ustawieniu przerzutnika w stan 0. Przerzutniki budowane są z bramek, ale w odróżnieniu od układów kombinacyjnych w połączeniu bramek wykorzystane są sprzężenia zwrotne pomiędzy wyjściami i wejściami bramek wchodzących w skład układu.

Tabliczka 4.2.5.12 – prosty przerzutnik SR

S	R	Q(t)	Q(t+1)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	?
1	1	1	?



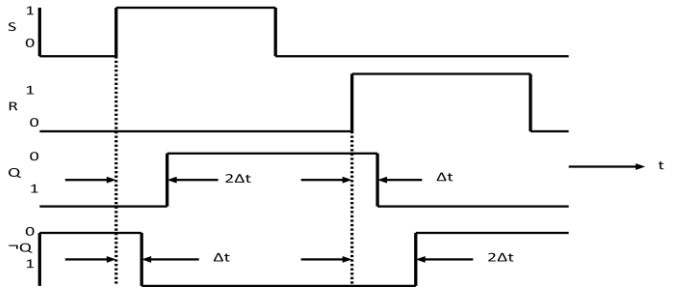
Istnieją dwa rodzaje układów sekwencyjnych, przy czym ich klasyfikacja zależy od taktujących je sygnałów:

4.2.5.02. **Wyjaśnienie**. Asynchroniczny układ sekwencyjny jest systemem, w którym stany wewnętrzne i stany wyjść zależą od kolejności, w jakiej zmieniają się jego stany wejściowe, w konsekwencji stany wewnętrzne i stany wyjść mogą się zmieniać w dowolnej chwili.

4.2.5.03. **Wyjaśnienie**. Synchroniczny układ sekwencyjny może zmieniać stany wewnętrzne i stany wyjść, jedynie w określonych chwilach czasu, w których stany wejść układu mogą zmieniać stany wewnętrzne i stany wyjść. Synchronizację uzyskuje się za pomocą urządzenia taktującego zwanego zegarem lub generatorem impulsów zegarowych.

4.2.5.11. **Wyjaśnienie**. Przerzutnik prosty asynchroniczny RS (zwany również przerzutnikiem zatrzaskowym) jest zbudowany z połączonych sprzężeniami zwrotnymi dwóch bramek

dwuwejściowych typu NAND lub NOR. Sprzężenie polega w tym przypadku na połączeniu wyjścia każdej z bramek z wejściem drugiej. Tabliczka 4.2.5.12 - zawiera zestawienie tzw. stanów wzbudzenia przerzutnika asynchronicznego SR oraz schemat bramkowy. Wyjście Q powiela stan wewnętrzny przerzutnika asynchronicznego RS, natomiast wyjście ¬Q zawiera negację stanu wewnętrznego, wejście S – „set", zaś wejście R – „reset". Uwaga: stan w chwili Q(t+1) jest nieokreślony, w przypadku, gdy równocześnie oba wejścia S i R przyjmują wartość 1. Rysunek 4.5.5.13 - pokazuje przebiegi czasowe wynikające ze sprzężenia zwrotnego przerzutnika asynchronicznego SR.

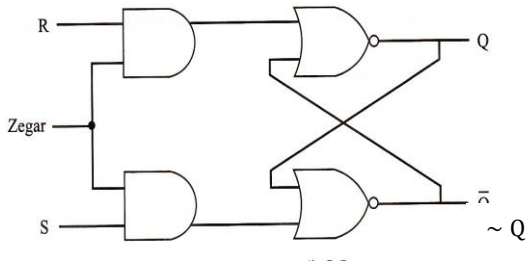


Rysunek 4.2.5.13 - Przebiegi czasowe przerzutnika asynchronicznego SR.

4.2.5.21. **Wyjaśnienie**. Przerzutnik synchroniczny RS jest zbudowany z połączonych sprzężeniami zwrotnymi dwóch bramek dwuwejściowych typu NAND lub NOR oraz dwóch bramek typu AND – zapewniających synchronizację. Tabliczka 3.5.5.22 - zawiera zestawienie tzw. stanów wzbudzenia przerzutnika synchronicznego SR oraz schemat bramkowy. Wyjście Q powiela stan wewnętrzny przerzutnika synchronicznego RS, natomiast wyjście ¬Q zawiera negację stanu wewnętrznego, wejście S – „*set*", zaś wejście R – „*reset*". Uwaga: stan w chwili Q(t+1) jest nieokreślony, w przypadku, gdy równocześnie oba wejścia S i R przyjmują wartość 1.

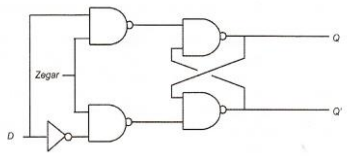
Tabliczka 4.2.5.22 stany przerzutnika synchronicznego RS

S	R	Q(t)	Q(t+1)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	?
1	1	1	?

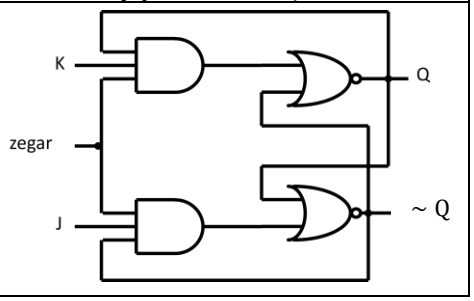


W przypadku przerzutnika SR problem jest to, że należy zapobiegać powstawaniu sytuacji, w której równocześnie R = 1, S = 1. Jednym ze sposobów osiągnięcia tego jest pozostawienie tylko jednego wejścia.

4.2.5.31. **Wyjaśnienie**. Jednowejściowo jest rozwiązany przerzutnik D, gdzie za pomocą inwertora (funkcji negacji) zagwarantowano, że nie-zegarowe wejścia do dwóch bramek NAND dają stany, które są swoją negacją. Stan na wyjściu przerzutnika D jest zawsze równy ostatniej wartości stanu wejścia. Dlatego przerzutnik D jest często określany, jako linia opóźniająca, ponieważ pojawienie się na wejściu 0 lub 1 – pojawia się na wyjściu z opóźnieniem jednego cyklu zegara. Tabliczka 4.2.5.32 - zawiera zestawienie tzw. stanów wzbudzenia przerzutnika D oraz schemat bramkowy.

Tabliczka 4.2.5.32 stany przerzutnika D			
D	Q(t)	Q(t+1)	
0	0	0	
0	1	1	
1	0	1	
1	1	0	

4.2.5.41. **Wyjaśnienie**. Przerzutnik JK, Podobnie jak przerzutnik SR ma dwa wejścia. W odróżnieniu jednak od przerzutnika SR wszystkie możliwe kombinacje stanów wejść przerzutnika JK są dopuszczalne i dają określone wartości wyjścia. Zauważmy, że pierwsze sześć kombinacje wartości wejść i wyjść przerzutnika JK są identyczne jak dla przerzutnika SR. Samo wejście J umożliwia realizację ustawienia na wejściu wartości 1, czyli *„set"*, zaś samo wejście K realizuje kasowanie czyli *„reset"*. Gdy stany obu wejścia (J i K) są równe 1, stan wyjścia ulega zanegowaniu. Jeśli więc Q(t) jest równe 1, to ustawienie stanów wejść J = 1 i K = 1 powoduje zmianę stanu wewnętrznego Q(t=1) na 0. Jeśli zaś Q(t) jest równe 0, to ustawienie stanów wejść J = 1 i K = 1 powoduje zmianę stanu wewnętrznego Q(t+1) na 1. Stan wyjścia Q jest równy stanowi wewnętrznemu Q(t+1) przerzutnika JK, zaś stan wyjścia ¬Q jest zawsze negacją wyjścia Q. Tabliczka 4.2.5.42 - zawiera zestawienie tzw. stanów wzbudzenia przerzutnika asynchronicznego SR oraz schemat bramkowy.

Tabliczka 4.5.5.42 stany przerzutnika JK				
S	R	Q(t)	Q(t+1)	
0	0	0	0	
0	0	1	1	
0	1	0	0	
0	1	1	0	
1	0	0	1	
1	0	1	1	
1	1	0	1	
1	1	1	0	

4.2.5.42. **Wyjaśnienie**. Przerzutnik T jest szczególnym przypadkiem przerzutnika JK. Przerzutnik T otrzymujemy przez połączenie obu wejść J i K. Stan wewnętrzny przerzutnika T jest równy 0, wówczas gdy - oba wejścia przerzutnika J i K mają wartość 0. Stan wewnętrzny przerzutnika T jest równy 1, wówczas gdy oba wejścia przerzutnika J i K mają wartość 1.

Piśmiennictwo: *Hyde R.* H.5.1., *Mano M.* M.1.1., *Stallings W.* S.12.1.

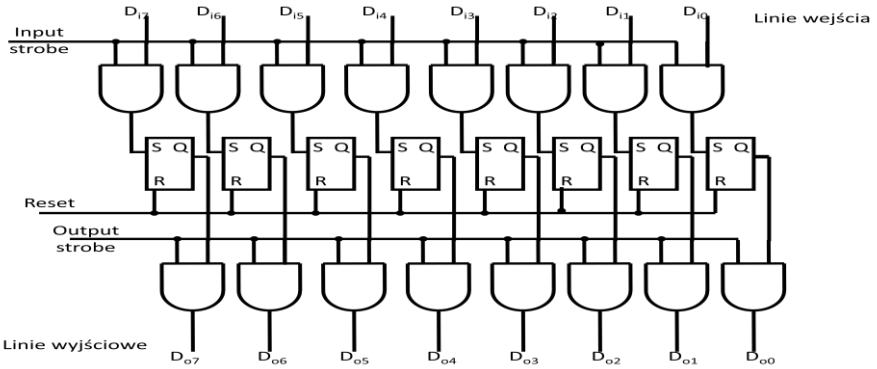
4.2.6. REJESTRY, LICZNIKI I SEKWENSERY

Jako przykład zastosowania przerzutników, przeanalizujemy jeden z zasadniczych elementów procesora - czyli rejestr. Jak wiemy, rejestr jest układem cyfrowym używanym między innymi wewnątrz procesora do przechowywania jednego lub wielu bitów danych. Powszechnie używane są dwa rodzaje rejestrów:

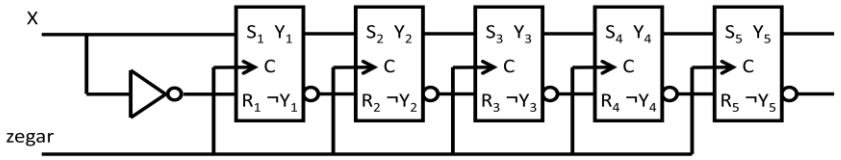
- Rejestry równoległe (patrz przykład na rysunku 4.2.6.01)
- Rejestry przesuwne (patrz przykład na rysunku 4.2.6.02).

4.2.6.11. **Wyjaśnienie**. Rejestr równoległy jest typowym rejestrem do przechowywania jednostek danych, na których operuje komputer (np. pojedynczych bajtów – rejestr 8 bitowy, jak na rysunku 4.2.6.01; połówek słów – rejestr 16 bitowy; słów – rejestr 32 bitowy i słów podwójnej długości – rejestr 64 bitowy).

4.2.6.12. **Wyjaśnienie**. Rejestr przesuwny jest typowym rejestrem dla wykonywania działań na jednostkach danych, na których operuje komputer wykonując operacje przesuwania zawartości bitowej rejestru w lewo (binarne mnożenie przez dwa) albo w prawo (dzielenie binarne przez dwa) - przypadki tzw. przesunięć arytmetycznych, albo przesuwania typu krążenia zawartości rejestru, gdzie ostatni n - ty bit rejestru jest równocześnie poprzednikiem zerowego bitu rejestru – przypadek tzw. przesuwania logicznego.



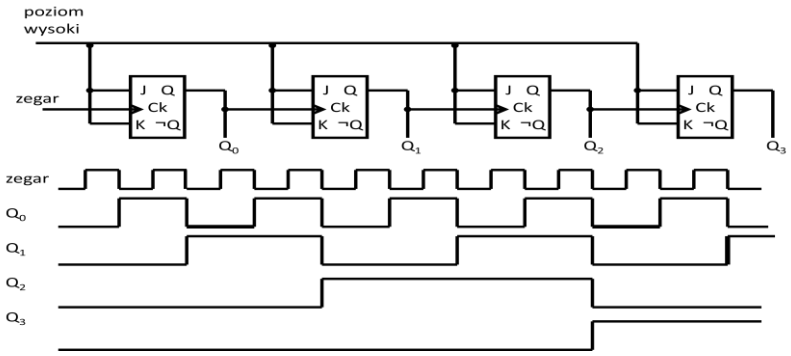
Rysunek 4.2.6.01 Rejestr równoległy – przerzutniki SR



Rysunek 4.2.6.02 Rejestr przesuwny – przerzutniki SR

4.2.6.13. **Wyjaśnienie**. W praktyce rejestr przesuwny jest z punktu widzenia wprowadzania (*input*) oraz wyprowadzania (*output*) danych rejestrem równoległym. Co oznacza, że we współczesnych komputerach mamy do czynienie z rozwiązaniem mieszanym – równoległo-przesuwnym.

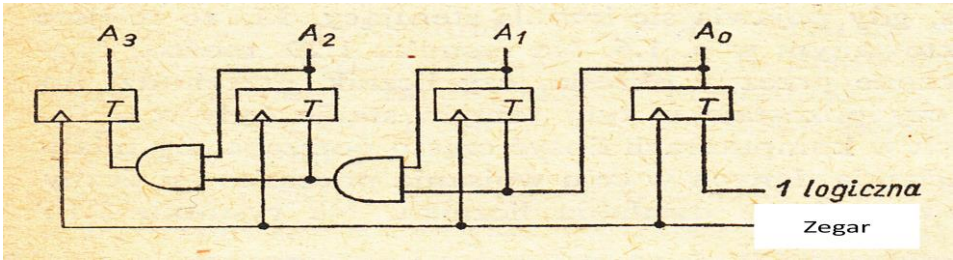
4.2.6.21. **Wyjaśnienie**. Inną użyteczną kategorią układów sekwencyjnych są liczniki. Licznik jest rejestrem, którego zawartość może być z łatwością inkrementowana 1 modulo pojemność rejestru. Rejestr wykonany z n przerzutników może liczyć od 0 do $2^n - 1$. Gdy zawartość licznika jest zwiększana poza jego wartość maksymalną, jest on ustawiony na 0. Przykładem licznika występującego w procesorze jest licznik programu IC.



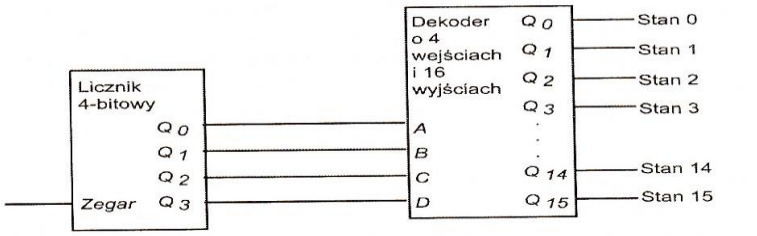
Rysunek 4.2.6.22. Licznik asynchroniczny (szeregowy).

Liczniki mogą być projektowane, jako asynchroniczne lub synchroniczne. Liczniki asynchroniczne są stosunkowo powolne, ponieważ przerzutnik wyzwala zmianę stanu następnego przerzutnika. Rysunek 4.2.6.22 przedstawia licznik asynchroniczny (szeregowy), zbudowany z przerzutników JK. W liczniku synchronicznym wszystkie przerzutniki zmieniają stan jednocześnie i dlatego są o wiele szybsze od liczników asynchronicznych. Dlatego też, są one powszechnie stosowane w procesorach.

Liczniki synchroniczne są znacznie szybsze od liczników asynchronicznych i dlatego są powszechnie stosowane w układach elektronicznych komputerów. Rysunek 4.2.6.22 przedstawia licznik trzybitowy synchroniczny (szeregowy), zbudowany z przerzutników typu T.



Rysunek 4.2.6.23 Licznik synchroniczny (szeregowy), zbudowany z przerzutników T.



Rysunek 4.2.6.24 Prosty sekwenser

Typowym przykładem zastosowania licznika synchronicznego w powiązaniu z dekoderem jest układ generowania serii kroków, np. wykorzystywanych przy sterowania mikroprogramowym. Układ taki umożliwia uruchamianie wykonywania kolejnych kroków mikroprogramu. Na rysunku 4.2.6.24 – pokazany jest prosty sekwenser generujący szesnaście kroków sterowania, np. uruchamiania wykonywania kolejnych kroków mikroprogramu.

Piśmiennictwo: *Mano M.* M.1.1., *Stallings W.* S.11.1.

4.2.7. Uwagi końcowe

Rozdział poświęcony zastosowaniu algebry Boole'a i logiki temporalnej do projektowania i opisu układów elektronicznych – zawiera jedynie stosunkowo proste przykłady układów cyfrowych realizujących wybrane funkcjonalności. Pominęliśmy w tych rozważaniach układy wykonywania takich operacji arytmetycznych jak: zmiana znaku liczby całkowitej, mnożenie liczb całkowitych, dzielenie liczb całkowitych, nie mówiąc o operacjach – na liczbach zmiennopozycyjnych. Większość złożonych operacji arytmetycznych (zarówno na liczbach naturalnych, jak i na zmiennopozycyjnych) jest realizowanych przez wbudowane mikroprogramy. Zainteresowanych odsyłamy do piśmiennictwa.

Piśmiennictwo: *Hyde R.* H.5.1., *Mano M.* M.1.1., *Stallings W.* S.11.1.

4.3.TABLICE DECYZYJNE

4.3.0. UWAGI WSTĘPNE

W pierwszych latach istnienia komputerów cyfrowych, począwszy od 1951 roku (komputer EDSAC opracowany i uruchomiony Wielkiej Brytanii przez zespól kierowany przez *Marcela Wilkesa*), sieci działań programów komputerowych opisywane były w sposób graficzny, z pomocą tzw. schematów blokowych. Zresztą używanych powszechnie do dzisiaj. W połowie lat pięćdziesiątych pojawiła się metoda reprezentacji sieci działań programów komputerowych – w formie tablicowej. Metodę tą nazwano przyjęto pod nazwą tablica decyzyjna programu komputerowego lub krótko tablica decyzyjna (*DecisionTable*). Trudno dzisiaj ustalić, kto był autorem tej metody, niemniej jednak rozpowszechniła się ona szybko. W roku 2000 zostało opublikowane opracowanie zawierające przegląd literatury dotyczącej tablic decyzyjnych za lata 1982 – 2000[68]. Autorzy zestawienia literatury - doliczyli się łącznie za lata 1960 – 2000 łącznie blisko tysiąca – różnych publikacji (artykułów i książek) poświęconych zarówno podstawom realizacji programowej tablic decyzyjnych, jak również zastosowaniu tablic decyzyjnych w różnych aplikacjach informatycznych. Najstarszą publikacją wymienioną w zestawieniu było opracowanie z 1962 roku autorstwa S. L. Pollacka, memorandum opublikowane przez Rand Corp. Wymieniono również najstarsze opracowania książkowe[69] z 1966 roku.

Piśmiennictwo: *Greniewski M.* G.3.1., *Pollack S.* P.5.1.

4.3.1. STRUKTURA I ZAWARTOŚĆ TABLICY DECYZYJNEJ

Tablica decyzyjna składa się z czterech wzajemnie powiązanych części, zwanych kwadrantami (*Quadrant*). Są to odpowiednio (patrz rys. 4.3.1.01):

1) Warunki sterujące wyborem czynności (*Conditions*);
2) Reguły sterujące wyborem czynności(*Rules of conditionalternatives*) – stanowiące matrycę złożoną z liter F (skrót od słowa *false* - fałsz) oraz T (skrót od słowa *true* - prawda);
3) Czynności realizowane np. przez program komputerowy opisany przez daną tablicę (*Actions*);
4) Reguły wywołań czynności (*Rules of actionentries*) – stanowiące matrycę złożoną z liter x (symbolizującej wywołanie czynności wymienionej w danym wierszu) oraz miejsc pustych (symbolizujących brak wywołania czynności).

Warunki sterujące wyborem czynności	**Reguły sterujące wyborem czynności**							
Światło czerwone	**F**	**F**	**F**	**F**	**T**	**T**	**T**	**T**
Światło żółte	**F**	**F**	**T**	**T**	**F**	**F**	**T**	**T**
Światło zielone	**F**	**T**	**F**	**T**	**F**	**T**	**F**	**T**
Czynności realizowane przez kierowcę	**Reguły wywołań czynności**							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Zatrzymaj się					**x**			
Jeśli możesz już nie jedź			**x**					
Przygotuj się do jazdy				**x**				
Można jechać		**x**						
Postępuj wg zasad ruchu	**x**							

Rysunek 4.3.1.01 Prosty przykład tablicy decyzyjnej

Rysunek 4.3.1.01 pokazuje przykład tablicy decyzyjnej, opisującej postępowanie kierowcy na skrzyżowaniu z regulacją świetlną ruchu. Jak widać reguły oznaczone cyframi 6, 7 i 8 dotyczą sytuacji awarii świateł i jako takie mogą zostać usunięte z tablicy decyzyjnej. Tak więc,

[68]A.M. Moreno Garcia, M. Verhelle& J. Vanthienen,*An Overview of decision table literature 1982-2000*, Katholieke Universiteit Leuven Department of Applied Economics, Naamsestraat 69, B-3000 Leuven (Belgium) Jan.Vanthienen@econ.kuleuven.ac.be & Ana.Moreno@econ.kuleuven.ac.be

[69]L. T. Reinvald, R. M. Soland, *Conversion of limited Entry Decision Table to optimal computer programming*, Prentice-Hall 1966.

pokazaliśmy pierwszą zasadę upraszczania tablic decyzyjnych. Rysunek 4.3.1.02 przedstawia równoważną tablicę decyzyjną zawierającą jedynie pięć reguł postępowania.

Warunki sterujące wyborem czynności	**Regułysterującewyboremczynności**				
Światło czerwone	**F**	**F**	**F**	**F**	**T**
Światło żółte	**F**	**F**	**T**	**T**	**F**
Światło zielone	**F**	**T**	**F**	**T**	**F**
Czynności realizowane przez użytkownika	**Reguły wywołań czynności**				
	1	**2**	**3**	**4**	**5**
Zatrzymaj się					**x**
Jeśli możesz już nie jedź			**x**		
Przygotuj się do jazdy				**x**	
Można jechać		**x**			
Postępuj wg zasad ruchu	**x**				

Rysunek 4.3.1.02 Przykład tablicy decyzyjnej po skreśleniu trzech zbędnych reguł

Piśmiennictwo: *Greniewski M.* G.3.1., *Pawlak Z.* P.1.1., P.1.2., P.1.3., *Pollack S.* P.5.1.

4.3.2. KOLEJNE ZASADY UPRASZCZANIA TABLICY DECYZYJNEJ

Rozważmy kolejny przykład tablicy decyzyjnej, dotyczący zachowania drukarki komputerowej. Rysunek 2.3.2.01, zawiera taką przykładową tablicę decyzyjną.

Warunki sterujące wyborem czynności	**Reguły sterujące wyborem czynności**							
Drukarka działa i komputer widzi drukarkę	**F**	**F**	**F**	**F**	**T**	**T**	**T**	**T**
Czerwona lampka nie świeci	**F**	**F**	**T**	**T**	**F**	**F**	**T**	**T**
Sprawdzenia / wymiany nie pomogły	**F**	**T**	**F**	**T**	**F**	**T**	**F**	**T**
Czynności realizowane przez użytkownika	**Reguły wywołań czynności**							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Korzystaj z drukarki							**x**	**x**
Sprawdź połączenie drukarki z komputerem	**x**		**x**		**x**			
Sprawdź / uzupełnij papier w pojemniku	**x**		**x**		**x**			
Sprawdź czy papier nie blokuje podajnika	**x**		**x**		**x**			
Sprawdź / wymień pojemnik z tuszem	**x**		**x**		**x**			
Sprawdź czy driver drukarki jest zainstalowany	**x**		**x**					
Wezwij konserwatora		**x**		**x**		**x**		

Rysunek 4.3.2.01 Przykład tablicy decyzyjnej korzystania z drukarki komputerowej

Zwróćmy na początek uwagę na dwie reguły wywołania czynności, a mianowicie 7 i 8. Warunek ostatni z pośród sterujących wyborem czynności zawiera odpowiednio w kolumnie 7 wartość F, a w kolumnie 8 wartość T. Oznacza to, że ostatni z warunków sterujących nie ma wpływu na wybór czynności „Korzystaj z drukarki". Możemy, więc usunąć jedną z tych dwu reguł wywołań czynności oraz wpisać w nowo utworzonej kolumnie 7 zamiast symboli „F" albo „T" symbol „-", czyli wartość neutralną[70] (patrz rysunek 4.3.2.02). Podobnie możemy postąpić jeszcze z dwoma parami reguł, a mianowicie 1 i 3 oraz 2 i 4. Oczywiście nie jest to koniec możliwości upraszczania naszej tablicy decyzyjnej, ale na razie poprzestaniemy na tym. Popatrzmy, co różni reguły sterujące tablic decyzyjnych przedstawionych odpowiednio na rysunkach 4.3.2.01 oraz 4.3.2.02. Tablica decyzyjna z rysunku 4.3.2.01 ma dwuwartościowe reguły sterujące wyborem czynności – zbiór {F, T}, natomiast tablica decyzyjna 4.3.2.02 ma trójwartościowe reguły sterujące wyborem czynności – zbiór {F, -, T}. Jeżeli - dokonamy przypisania symbolowi „F" wartości 0, symbolowi „-" wartości ½, zaś symbolowi „T" wartości 1, to mamy odpowiednio zbiory wartości {0, 1} oraz {0, ½, 1}. Z dotychczasowych rozważań dotyczących upraszczania tablic decyzyjnych możemy wyciągnąć następujący wniosek:

4.3.2.10. **Wniosek**. W wyniku upraszczania tablicy decyzyjnej o dwuwartościowym zbiorze reguł sterujących wyborem czynności – na drodze eliminacji jednej z pary kolumn o przeciwstawnych regułach sterowania dla jednego warunku („F" i „T") i braku różnic wartości

[70] Patrz podrozdział 2.4.1. – logika trójwartościowa *Sobocińskiego*.

pozostałych warunkach, otrzymujemy tablicę decyzyjną o zmniejszoną o jeden liczbą kolumn reguł (sterujących wyborem czynności oraz wywołań czynności) z wartością „-" (czyli neutralny ze względu na warunek) w miejsce pary wartości „F" oraz „T".

Innymi słowy, operacja upraszczania powoduje przejście od rachunku dwuwartościowego do rachunku trójwartościowego. Należy podkreślić, że powyższa procedura ma czysto formalny charakter, co nasuwa kolejny wniosek:

Warunki sterujące wyborem czynności	**Regułysterującewyboremczynności**				
Drukarka działa i komputer widzi drukarkę	**F**	**F**	**T**	**T**	**T**
Czerwona lampka nie świeci	**-**	**-**	**F**	**F**	**T**
Sprawdzenia / wymiany nie pomogły	**F**	**T**	**F**	**T**	**-**
Czynności realizowane przez użytkownika	**Reguły wywołań czynności**				
	1	**2**	**5**	**6**	**7**
Korzystaj z drukarki					**x**
Sprawdź połączenie drukarki z komputerem	**x**		**x**		
Sprawdź / uzupełnij papier w pojemniku	**x**		**x**		
Sprawdź czy papier nie blokuje podajnika	**x**		**x**		
Sprawdź / wymień pojemnik z tuszem	**x**		**x**		
Sprawdź czy driver drukarki jest zainstalowany	**x**				
Wezwij konserwatora		**x**		**x**	

Rysunek 4.3.2.02 Przykład tablicy decyzyjnej po częściowych uproszczeniach.

4.3.2.11.**Wniosek**. Pomiędzy dwuwartościowym rachunkiem zdań i trójwartościowym rachunkiem zdań - istnieje czysto formalny związek, umożliwiający przejście od wyrażeń rachunku dwuwartościowego do rachunku trójwartościowego i odwrotnie.

Piśmiennictwo: *Greniewski M.* G.3.1., *Pawlak Z.* P.1.1., P.1.2., P.1.3., *Pollack S.* P.5.1.4.4.

4.3.3. SYSTEM INFORMACYJNY PAWLAKA

Jak już powiedzieliśmy (w paragrafie 2.8.2), teoria zbiorów przybliżonych, wraz z teorią systemów informacyjnych została sformułowana przez *Zdzisława Pawlaka* w 1982 roku. Istnieje szereg struktur, które mogą być wykorzystane do przechowywania danych. Sposób reprezentacji danych powinien jednak posiadać dwie podstawowe cechy:

1. uniwersalność (czyli powinien pozwalać na gromadzenie i przechowywanie danych opisujących badane zjawiska i procesy), oraz
2. efektywność (powinien umożliwiać w łatwy sposób na komputerową analizę tak zapisanych danych).

Obie te cechy posiada znany i często wykorzystywany w praktyce tablicowy sposób reprezentacji danych. W tym podejściu zbiór danych przedstawiany jest w postaci tablicy, której kolumny są etykietowane przez atrybuty (parametry, własności, cechy), wiersze odpowiadają zaś elementom (sytuacjom, stanom), a na przecięciu wierszy i kolumn znajdują się wartości odpowiednich atrybutów dla poszczególnych elementów. Tak określoną strukturę Pawlak nazywał *systemem informacyjnym SI* (*information system*).

2.3.3.10. **Definicja**. Formalnie systemem informacyjnym nazywamy czwórką uporządkowaną *SI = (U , A, V , f); gdzie:*

(1) U jest niepustym, skończonym zbiorem zwanym uniwersum, przy czym składowe zbioru U nazywamy elementami;
(2) A jest niepustym, skończonym zbiorem atrybutów;
(3) $V = \{V_a \circ \forall a \in A V_a$ jest dziedziną atrybutu a$\}$
(4) $f : U \times A \rightarrow V$jest funkcją informacji taką, że dla $[\forall(x \in U \wedge a \in A)\ f(x, a) \in V_a]$.

Typowym dla systemu informacyjnego Pawlaka, jest pewna niepełna rozróżnialność elementów, co pokażemy na prostym przykładzie.

Tabela 4.3.3.01. Prosty przykład systemu informacyjnego Pawlaka				
Pacjent	Ból głowy (g)	Ból mięśni (m)	Temperatura (t)	Grypa (c)
1	nie	tak	wysoka	tak
2	tak	nie	wysoka	tak
3	tak	tak	bardzo wysoka	tak
4	nie	tak	bardzo wysoka	tak
5	tak	nie	wysoka	nie
6	nie	tak	normalna	nie

Tabela 4.3.3.01 przedstawia przykładowy system informacyjny Pawlaka, zawierający wyniki badań przeprowadzonych dla grupy pacjentów. System ten składa się z sześciu elementów (1, 2, ..,6) oraz czterech atrybutów (Ból głowy, Ból mięśni, Temperatura, Grypa). Dalej kolejne atrybutu będą oznaczane za pomocą liter: g, m, t, c. Zgodnie z definicją 2.3.3.10. rozpatrywany system informacyjny może zostać zapisany w postaci, jak w definicji 2.3.3.01, gdzie:
(1) V_g={nie, tak};
(2) V_m={nie, tak};
(3) V_t={normalna, wysoka, bardzo wysoka};
(4) V_c={nie, tak}; $f: U \times A \rightarrow V$ (*np.* f(1,Ból głowy)=nie; f(3, Grypa) = tak).

4.3.3.20. **Wyjaśnienie.** Przez relacja nierozróżnialności, na przykładzie tabeli 4.3.3.01, można zauważyć, że elementy o numerach 1, 4 i 6 mają te same wartości atrybutów: *ból głowy* oraz *ból mięśni* zaś elementy o numerach 1 i 5 mają tę samą wartość atrybutu *temperatura*. O elementach numer 1, 4 i 6 powiemy, że są nierozróżnialne ze względu na atrybuty: *ból głowy* oraz *ból mięśni*, zaś elementy o numerach 1 i 5 są nierozróżnialne ze względu na atrybut: *temperatura*. Tę obserwację można uogólnić i wyrazić w sposób formalny stosując odpowiednio zdefiniowaną relację.

4.3.3.30. **Definicja**. Niech $SI = (U, A, V, f)$gdzie systemem informacyjnym i niech $B \subseteq A$ *Relację nierozróżnialności* (*indiscernibility relation*) na zbiorze elementów U generowaną przez zbiór atrybutów B określamy jako:

$$IND(B) = \{(x, y) \in U \times U : \forall a \in B f(x, a) = f(y, a)\}$$

Poszczególne pary obiektów należą do relacji wtedy, gdy posiadają te same wartości dla wszystkich atrybutów ze zbioru B. Można pokazać, że relacja nierozróżnialności, jest relacją równoważności, gdyż jest relacją zwrotną, symetryczną i przechodnią. Jedną z praktycznych metod eliminacji z systemu informatycznego Pawlaka nierozróżnialności, jest wprowadzenie atrybutu prawdopodobieństwa wystąpienia poszczególnych nierozróżnialnych elementów systemu informacyjnego. W wyniku usunięcia nierozróżnialności, system informacyjny Pawlaka – przedstawiony w formie tablicy, staje się szczególnym przypadkiem tablicy decyzyjnej.

Piśmiennictwo: *Pawlak Z.* P.1.1., P.1.2., P.1.3.

4.4. NOTACJA Z

4.4.0. CZYM JEST NOTACJA Z

J. M. Spivey opublikował w 1989 roku wstępną wersję raportu „*The Z Notation: a Reference Manual*". Ten raport zawierał wyniki prac zespołu nazwanego „*Program Research Group, University of Oxford*". W pracach zespołu wzięli udział: *J. R. Arial, I. J. Hades, C. A. R. Hoar, He Jifeng, C. C. Morgan, J. W. Anders, I. H. Sørensen, J. M. Spivey* oraz *B. A. Sufrin*. Końcowa wersja raportu została opublikowana w 1998 roku. W 2001 roku notacja Z otrzymała standard ISO[71].

Notacja Z jest językiem formalnej specyfikacji systemów informatycznych wzorowanym na języku logiki pierwszego rzędu, dodatkowo umożliwiający formalne przekształcanie i upraszczanie specyfikacji. Tego rodzaju czynności określone zostały wspólną nazwą „*Refinaments*". Notacja Z wykorzystuje własny zbiór znaków i symboli (patrz tablica 4.4.0.01).

Tablica 4.4.0.01 – zbiór znaków i symboli używanych w notacji Z		
1.	Paragraphs	┼, ┽, ┾, ┿, ::=, ==, [ ], ?, !
2.	Predicates	∀, ∃, ∧, ∨, ¬, ⇒, ⇔, ≠, ∈, ∉, ⊆, ⊂
3.	Sets	ℙ, 𝔽, {, |, •, }, ∅, ∪, ⋃, ∩, ⋂, \, ⊖
4.	Relations	↔, ↦, ×, ◁, ⩤, ▷, ⩥, ⊕, ⨾, ~, ⦇ ⦈, ↗+↙
5.	Functions	→, ⇸, ↣, ⤖, ⤀, ⤔, ⤖, ∘, λ, μ
6.	Sequences	⟨ ⟩ , ⌢ , ⌢ /, 1, ↾, #
7.	Arithmetic	ℝ, ℤ, ℕ, 𝔸, +, −, -, *, div, mod, ≤, ≥
8.	Schemas	Δ, Ξ, θ, ⦇, NAME == EXPR ⦈, ╲, ↾, ⨾, ≫, ′, ↘1↖

1. Notacja Z umożliwia opisanie specyfikacji w kategoriach – „co system ma robić", a nie w kategoriach – „jak to system ma zrobić".
2. Kolejną możliwością oferowaną przez notację Z, jest dekompozycja specyfikacji systemu na bloki zwane schematami (*schema*).
3. Każdy taki blok – schemat może być powiązany ze słownym komentarzem objaśniający zawartość bloku zapisaną metodą formalną.
4. Schematy notacji Z są używane zarówno do opisu statycznych, jak i dynamicznych aspektów specyfikacji systemu.
5. Aspekty opisu statycznego obejmują: (a) stany, jakie system może przyjmować; (b) utrzymywane przez system niezmienniki związków utrzymywanych przez system przy przejściu od dowolnie wybranego stanu – do każdego innego stanu, jeśli takie przejście jest dopuszczalne.
6. Aspekty opisu dynamicznego obejmują: (a) działania systemu, które są możliwe oraz dopuszczalne; (b) związki pomiędzy stanami wejść oraz stanami wyjść; (c) zmiany, jakie zachodzą w stanach wewnętrznych systemu.
7. Posługiwanie się schematami umożliwia dokonywania niezależnego opisu działania systemu z wielu perspektyw, a następnie powiązanie tych opisów w całość.
8. Notacja Z wykorzystuje trzy znaczniki (*decorations*) dla wyróżnienia abstrakcyjnych danych: (a) znak „**'** " dla oznaczenia stanu końcowego danych po zakończeniu wykonania operacji (*final state of operation*); (b) znak „**?**" dla oznaczenia danych wejściowych systemu (*input*); (c) znak „**!**" dla oznaczenia danych wyjściowych systemu (*output*).

Piśmiennictwo: *Greniewski M*. G.3.2., G.3.3., *Spivey J.* S.10.1., *Woodcock J.* W.7.1.

[71] "ISO/IEC 13568:2002(E) International Standard Information technology – *Z formal specification notation – Syntax, type system and semantics*".

4.4.1. SCHEMATY NOTACJI Z - PODSTAWOWE BLOKI KONSTRUKCYJNE

Jako przykład prostych wymagań na system aplikacyjny, omówimy poniżej specyfikację fragmentu wymagań na system MRP II, który dotyczy podsystemu opisu elementów technologicznych (*item*) komponentów produktu używanych w procesie planowania produkcji. APICS Dictionary[72] w następujący sposób objaśnia termin: "*The master item - Typically it contains identifying and descriptive data and control values (lead time, lot size, etc.) and may contain data and inventory status, requirements, planned orders, and costs.*"[73] Jak już wzmiankowaliśmy, zapis wymagań systemowych sporządzony w notacji Z – składa się z bloków zwanych schematami.

4.4.1.01. **Wyjaśnienie**. Schematy i ich właściwości. Omówimy z kolei zasady i przykłady pisania schematów, jak również i rachunek schematów. Schemat to wyrażenia matematyczne umieszczone wewnątrz ramki, z załączoną nazwą.

```
┌─NazwaSchematu──────────────────────────────────────────────
│ Deklaracje zmiennych lokalnych schematu, jak również zmiany stanów
├────────────────────────────────────────────────────────────
│ Związki (zależności) pomiędzy wartościami przyjmowanymi przez zmienne lokalne schematu
└────────────────────────────────────────────────────────────
```

Podane rozumienie schematu jest pojęciem unikalnym dla notacji Z. Rachunek schematów umożliwia budowanie dużych schematów na drodze łączenia mniejszych schematów. Ponieważ schemat modeluje stany i operacje, to schemat może być użyty jako deklaracja, predykaty, wyrażenia, typy, zbiory …

Uwaga: schemat może zawierać w swoich czynnościach wywołanie innego schematu.

4.4.1.02. **Definicja**. Określamy `ItemMaster`, jako plik (zbiór) iloczynów kartezjańskich domen, identyfikowanych przez unikalną wartość `ItemCode`. Każda pozycja zbioru `ItemMaster`, w notacji Z ma postać:

ItemCode ↦ (Description, Measuring-Unit, On-Hand Balance, Lot-Size, Order-Policies, Lead-Time, Safety-Stock, Shrinkage-Factor, Indicator-MPS-Item, Low-Level-Code, Statistic-Code, Item-Type);

4.4.1.10. **Wyjaśnienie**. Opis `ItemMaster` zrealizowany jest z pomocą kilku schematów, z których przedstawiamy cztery, przedstawionych poniżej. Są to odpowiednio: (1) główny schemat opisu; (2) początkowa zawartość pliku; (3) dodawanie nowej pozycji do pliku; (4) wyszukiwanie pozycji w pliku.

4.4.1.11. **Definicja**. Schemat głównego opisu pliku (zbioru) `ItemMaster` ma postać następującą:

```
┌──ItemMaster──────────────────────────────────────
│ ItemCode : ℙ
│ Description : ℙ TEXT
│ Measuring-Unit : {cm, m, pice, pack, g, kg, m2 , …}
│ On-Hand Balance, Lot-Size, Lead-Time, Safety-Stock,
│ Shrinkage-Factor, Low-Level-Code, Statistic-Code : ℙℤ
│ Order-Policies : {lot-for-lot, fixed-lot-size, EOQ, dynamic-lot-size, special}
│ Indicator-MPS-Item : Boolean
│ Item-Type : { manufactured, purchase }
├──────────────────────────────────────────────────
│ Items = dom (ItemCode, Description, Measuring-Unit, On-Hand Balance, Lot-Size,
│ Order-Policies, Lead-Time, Safety-Stock, Shrinkage-Factor, Indicator-MPS-Item,
│ Low-Level-Code, Statistic-Code, Item-Type)
└──────────────────────────────────────────────────
```

[72] APICS® The Education Society for Resource Management, "APICS Dictionary", 8th Edition, APICS, USA VA - 1995.

[73] Master Item – typowo zawiera dane identyfikujące, opisujące i sterujące dla poszczególnych materiałów kupowanych, części wytwarzanych i wyrobów gotowych (np. *lead time* – czas wyprzedzenia, wielkość partii, modelu kształtowania zapasu, itp.) oraz zawiera dane dotyczące zapasu, zapotrzebowania, planowanych ilości oraz kosztu.

Uwaga: linia 1 schematu zawiera nazwę schematu, linia 2 deklarację typu domeny `ItemCode`, itd. Linie 1 i 2 tworzą wspólnie tzw. sygnaturę schematu. Linie 10, 11 i 12 określają czynność tworzenia pozycji ItemMaster.

4.4.1.12. **Definicja**. Schemat określający początkową zawartość `ItemMaster`, jako pliku (zbioru) pustego:

```
┌──InitItemMaster──────────────────────────
│ ItemMaster
├──────────────────────────────────────────
│ items = ∅
└──────────────────────────────────────────
```

Uwaga: Linia 2 schematu zawiera odwołanie do deklaracji podanej w schemacie `ItemMaster`.

4.4.1.13. **Definicja**. Schemat określający dodanie nowej pozycji do pliku (zbioru) `ItemMaster` z wartością `ItemCode` jaka do tych czas nie występuje w pliku, ma postać następującą:

```
┌──AddNewItem──────────────────────────────────────────────────────────
│ ΔItemMaster
│ ItemCode?, Description?, Measuring-Unit?, On-Hand Balance?, Lot-Size?, Lead-Time?,
│ Safety-Stock?, Shrinkage-Factor?, Low-Level-Code?, Statistic-Code?,
│ Order-Policies?, Indicator-MPS-Item?, Item-Type?
├──────────────────────────────────────────────────────────────────────
│ ItemCode? ∉ ItemMaster
│ items′ = items ∪ {ItemCode? ↦ (Description?, Measuring-Unit?,
│ On-Hand Balance?, Lot-Size?, Order-Policies?, Lead-Time?, Safety-Stock?,
│ Shrinkage-Factor?, Indicator-MPS-Item?, Low-Level-Code?, Statistic-Code?, Item-Type?)
└──────────────────────────────────────────────────────────────────────
```

Uwaga: Δ - oznacza operator dołączania pozycji do pliku, którego nazwa odwołuje się do schematu 4.4.1.11. Predykat ItemCode? ∉ ItemMaster – warunkuje wykonanie operacji dodawania, w przypadku unikalności identyfikatora nowo wprowadzanej pozycji; zaś znacznik „ ’ ” został użyty dla oznaczenia stanu końcowego danych po zakończeniu wykonania operacji; natomiast znacznik „?”oznacza dane nowej - wprowadzanej pozycji dołączanej do pliku.

4.4.1.14. **Definicja**. Schemat określający wyszukanie pozycji pliku (zbioru) o zadanej wartości ItemCode?, która występuje w pliku, ma postać następującą:

```
┌──FindItem────────────────────────────────
│ ΞItemMaster
│ ItemCode? : ℙ ItemMaster
│ item!
├──────────────────────────────────────────
│ ItemCode? ∈ {ItemCode}
│ item! ∈ dom ItemMaster
└──────────────────────────────────────────
```

Uwaga: Ξ - oznacza operator wyszukania pozycji z pliku, której przypisany jest identyfikator równy zadanemu ItemCode?: zaś znacznik „!”oznacza dane wyszukanej pozycji pliku.

4.4.1.21. **Wyjaśnienie**. Poniżej pokazany jest schemat operacji zbiorczej uzyskany w wyniku zastosowania rachunku schematów:

$$T\_ItemMaster \triangleq InitItemMaster \vee ItemMaster \vee AddNewItem \vee FindItem$$

4.4.1.21. **Wyjaśnienie**. Włączanie i rozszerzanie schematów. Schematy mogą być traktowane podobnie jak makra w językach programowania. Każda występująca w definicyjnej części schematu nazwa (lub referencja schematu) może posłużyć do rozszerzenia schematu zastępując nazwę zawartością schematu.

Piśmiennictwo: *Greniewski M.* G.3.2., G.3.3., *Spivey J.* S.10.1. *Woodcock J.* W.7.1.

4.4.2. ZBIORY, TYPY, ZMIENNE

Notacja Z wykorzystuje teorię zbiorów z typami. Każdy element wykorzystywany w notacji Z - posiada jednoznacznie określony typ. W przypadku elementów posiadających interpretację fizyczną, typem jest jednostka miary służącej do mierzenia danego elementu lub zbiorowości elementów. Zbiór w sensie teorii *Zermelo–Frenkela*, jest dobrze zdefiniowaną kolekcją elementów jednego typu. „Dobrze zdefiniowana" oznacza w tym przypadku, że istnieje możliwość stwierdzenia, czy dany element należy do danego zbioru, czy też nie. Zwykle przyjmuje się, że zbiory oznaczamy dużymi literami alfabetu lub nazwami zaczynającymi się od dużej litery alfabetu. Przykładowe nazwy zbiorów: A, B, Z, DYSK, LAMPA. Elementy zbiorów oznaczamy małymi literami lub nazwami zaczynającymi się od małych liter. Przykładowe nazwy elementów: a, b, c, x, y, z, alfa, delta. Zbiór może być również kolekcją zbiorów, pod warunkiem jednak, że wszystkie zbiory wchodzące w skład kolekcji są tego samego typu.

4.4.2.01. **Wyjaśnienie**. Zbiory mogą być zbiorami skończonymi, czyli zbiorami o skończonej liczbie elementów, albo zbiorami nieskończonymi, czyli zbiorami złożonych z nieskończonej liczby elementów. Przykłady zbiorów skończonych: {1, 2, 3, 4, 5, 6}, {4, 5, 5, 6, 1, 1, 1, 2, 3}, {czerwony, żółty, zielony}. Przykład błędnego zbioru, z punktu widzenia teorii zbiorów Zermelo – Frenkla {żółty, czerwony, zielony, 1, 2} – błąd niejednorodności typu.

Zarówno zbiory skończone jak i nieskończone, można opisywać z pomocą wyrażeń. Przykład opisu zbioru z pomocą wyrażeń: {1 ... 6}, { i: Z | 1≤ i ≤ 6 }. Pierwszy z wymienionych przypadków dotyczy zbiorów licznych o skończonej liczbie elementów, Przy takim zapisie podajemy zawsze pierwszy i ostatni element zbioru. Gdybyśmy chcieli zapisać w podobny sposób zbiór nieskończony, np. zbiór wszystkich liczb całkowitych, to zapis wyglądałby następująco ℕ = {1, 2, 3, ...}. Notacja Z wykorzystuje dodatkowo analityczną metodę definicji zborów, która dla zbioru liczb jednocyfrowych może mieć postać A = {x : ℕ | dla x jednocyfrowych}. Uogólniony opis analityczny zbioru w notacji Z wygląda następująco: A = {x : X | P(x) • jawna postać elementu zbioru} gdzie P(x) predykat narzucający ograniczenia na zmienną, do którego podstawiono element x.

4.4.2.02. **Wyjaśnienie**. Mówimy, że zbiór B jest podzbiorem właściwym zbioru A, wtedy i tylko wtedy, kiedy każdy element zboru B jest również elementem zbioru A, ale istnieje również, co najmniej jeden element należący do zbioru A, który nie jest elementem zbioru B. Jeżeli zbiór B jest podzbiorem zbioru A, to oznacza, że zbiór B jest zawarty w zbiorze A. Notacja B ⊂ A oznacza, że zbiór B jest podzbiorem właściwym zbioru A. Natomiast notacja B ⊆ A oznacza, że zbiór B jest podzbiorem zbioru A lub jest identyczny ze zbiorem A, czyli niej jest podzbiorem właściwym zbioru A. Dodatkowo przyjmuje się, że zbiór pusty (pisany jako Ø) jest podzbiorem każdego zbioru.

4.4.2.03. **Wyjaśnienie**. Definiowanie typów. Notacja Z, jak zostało to wcześniej zauważone wykorzystuje zbiory z typami, czyli zbiory do których mogą należeć elementy tego samego typu. Nawiązując do omawianych wcześniej przykładów zbiorów, należy stwierdzić, że każdy element należący do danego zbioru ma swój typ. Np. zbiór złożony z liczb całkowitych {1, 2, ...}, którego wszystkie elementy są typu ℤ (gdzie przez ten symbol oznaczony typ wszystkich liczb), jest również typu ℤ. Innym przykładem jest zbiór kolorów typu COLOR. Natomiast elementy zbioru kolorów, np.: czerwony, żółty, zielony mają typ COLOR.

4.4.2.04. **Wyjaśnienie**. Typ może być deklarowany na dwóch drogach:

1. Pierwsza - deklarowania typu polega na tworzenia tzw. *Free Type* jak np.: COLOR ::= red | green | blue | yellow | cyan | magenta | white | black

2. Druga - posługiwanie zdefiniowanymi w notacji Z typami, tak zwanymi *Basic Types* jak: $\mathbb{Z}$, [NAMES], [PROCESS, FILE].

4.4.2.11. **Wyjaśnienie**. Zmienne, jak wiadomo zmienna jest nazwą elementu posiadającego określoną wartość. W notacji Z zmienne są wprowadzane w deklaracjach. Np: x: S zmienna x przyjmuje wartości ze zbioru S. W notacji Z, podobnie jak w językach programowania, rozróżniamy zmienne globalne i zmienne lokalne. Zmienne globalne są dostępne z każdego miejsca skryptu napisanego w notacji Z, natomiast zmienne lokalne są dostępne wewnątrz wyrażenia, w którym zostały zdefiniowane. Tzw. definicje aksjomatyczne (*Axiomatic definitions*) w notacji Z, służą do deklarowania zmiennych globalnych i mogą zawierać opcjonalnie ograniczenia. Przykład poniższy pokazuje zasadę zapisu definicji aksjomatycznej:

```
│ d1, d2 : DISCE
├──────────────────────────────────────
│ d1 + d2 = 7
│ d1 < d2
```

W notacji Z, podobnie jak w teorii zbiorów, zapis $\mathbb{P}$S oznacza zbiór wszystkich podzbiorów zbioru S.

```
│DISCE : ℙℤ
├──────────────────────────────────────────
│DISCE = 1 ... 6
```

4.4.2.11. **Wyjaśnienie**. Stałymi nazywamy zmienne, które przyjmują tylko jedną wartość, z pomocą tzw. skróconych definicji (*Abbreviation definitions*) można również definiować stałe. Poniżej pokazany jest przykład definicji:

DISCE == 1

4.4.2.12. **Wyjaśnienie**. Typy, zbiory i normalizacja. Typy zbiorów mogą tworzyć również zbiory, ale oczywiście nie wszystkie zbiory są zbiorami typów typami. Wiele różnych zborów może składać się z elementów tego samego typu. Przykładowo, jeżeli ODD, EVEN, PRIME są zbiorami odpowiednio liczb parzystych, nieparzystych i pierwszych, to $\mathbb{Z}$ jest typem ich elementów. Dowolny zbiór może pojawić się w każdej deklaracji, na przykład zmienne przyjmują wartości ze zbiorów e : EVEN, o : ODD, p : PRIME. Notacja Z przyjmuje, że jeżeli zapisujemy deklarację zmiennych w tak zwanej deklaracji znormalizowanej, to zadeklarowane zmienne tworzą tzw. sygnaturę (*signature*) zbioru, która między innymi służy dla pokazania typu zmiennych:

```
│ e, o, p : ℤ
├──────────────────────────────────────
│ e ∈ EVEN
│ o ∈  ODD
│ p ∈  PRIME
```

4.4.2.21. **Wyjaśnienie**. Przez zmienne globalne rozumie się w językach programowania, takie zmienne, których deklaracje obowiązują w całym programie – począwszy od miejsca zadeklarowania, a skończywszy na końcu skryptu programu. Analogiczna zasada obowiązuje w notacji Z.

4.4.2.22. **Wyjaśnienie**. Podstawowe sposoby deklarowania zmiennych globalnych pokazane są poniżej:

[NAZWA] – Deklarowanie globalnej nazwy pojedynczego zbioru, którą mogą przebiegać zmienne.

[NAZWA, DATA] – Deklarowanie dwu globalnych nazw zbiorów, które mogą przebiegać zmienne.

Podobnie można zadeklarować trzy i więcej globalnych nazw zbiorów, jak poniżej:

ID == $\mathbb{N}$ - Deklarowanie, że zbiór o nazwie ID jest zbiorem liczb naturalnych.

WYDZIAŁ ::= admin | produkcja | badania – Deklarowanie zbioru, którego elementami są wyliczone wartości.

PRACOWNICY == ID x NAZWA x WYDZIAŁ - Deklarowanie produktu typu kartezjańskiego, którego elementami są krotki – w tym przypadku złożone z trzech składowych: (a, b, c) - gdzie a ∈ ID, b ∈ NAZWA, c ∈ WYDZIAŁ.

4.4.2.23. **Wyjaśnienie**. Bezpośrednie deklarowanie krotek (*tuples*) które są składnikami zadeklarowanego typu można dokonać w sposób następujący:

```
│ Franek, Adam : PRACOWNICY
├──────────────────────────────────────────
│ Franek = (0019, Franek, admin)
│ Adam = (7408, Adam, badania)
```

4.4.2.24. **Wyjaśnienie**. Relacje są zbiorami krotek. Relacje odpowiadają tabelom składowym baz danych.

ID	NAZWA	WYDZIAŁ
0019	Franek	Admin
0308	Filip	Badania
7408	Adam	Badania
…	…	…

1.4.2.25. **Wyjaśnienie**. W notacji Z tabelę deklarujemy następująco:

```
1.4.2.26.  │ PRACOWNICY : ℙ OSOBY
├──────────────────────────────────────────
│ PRACOWNICY : {(0019, Franek, admin),
│ (0308, Filip, badania), (0708, Adam, badania) … }
```

Piśmiennictwo: *Spivey J.* S.9.1., *Woodcock J.* W.7.1.

4.4.3. WYRAŻENIE I OPERACJE ARYTMETYCZNE ORAZ OPERACJE NA ZBIORACH

4.4.3.00. **Wyjaśnienie**. Wyrażenia mają wartości. Najprostszymi wyrażeniami są stałe i zmienne: 1, 2, red, x, d1, DISCE, ℝ, … Natomiast operatory arytmetyczne tworzą złożone wyrażenia z prostszych. Operatory arytmetyczne są dobrze znanymi przykładami:

4.4.3.01. Operacja m + n dodawanie;

4.4.3.02. Operacja m – n odejmowanie;

4.4.3.03. Operacja m * n mnożenie;

4.4.3.04. Operacja m div n dzielenie;

4.4.3.05. Operacja m mod n pozostałość z dzielenia (modulo) m;

4.4.3.06. Operacja n ≤ m mniejsze lub równe;

4.4.3.07. Operacja max A maksymalna (największa) z danego zbioru liczb;

4.4.3.08. Operacja min A minimum (najmniejsza) z danego zbioru liczb.

4.4.3.10. **Wyjaśnienie**. Z kolei operacje na zbiorach to operacje takie jak: wyznaczania liczebności zbioru, konkatenacji (łączenia) zbiorów, operacje mnogościowe na zbiorach

4.4.3.11. Operator liczebności zbioru # zlicza różne elementy danego zbioru. Np. #{red, yellow, blue, green, red} = 4

4.4.3.12. Operator łączenia zbiorów ∪ łączy zbiory. Np. {1, 2, 3} ∪ {2, 3, 4} = {1, 2, 3, 4}

4.4.3.13. Operator różnicy zbiorów \ usuwa elementy jednego zbioru z drugiego. Np. {1, 2, 3, 4} \ {2, 3} = {1, 4}

4.4.3.14. Operator przecięcia zbiorów znajduje elementy wspólne dla obu zbiorów. Np. {1, 2, 3} ∩ {2, 3, 4} = {2, 3}

4.4.3.20. **Wyjaśnienie**. Operator łączenia działa ze zbiorami dowolnego typu, pod warunkiem jednak, że typy obu zbiorów argumentów operatora są identyczne. Np. {1, 2, 3} ∪ {red, greek} – wynik nieokreślony, niezgodność typów argumentów.

Piśmiennictwo: *Spivey J.* S.10.1., *Woodcock J.* W.7.1.

4.4.4. PREDYKATY, KWANTYFIKATORY I STANY

4.4.4.01. **Wyjaśnienie**. Przez predykaty rozumiemy tzw. funktory zdaniotwórcze, czyli wyrażenia stające się zdaniami logicznymi (w rozumieniu rachunku zdań) po podstawieniu do nich określonych wartości na ich zmienne (tzw. zmienne zdaniowe). Predykaty służą do ograniczenia przyjmowanych przez zmienne wartości. Wiele predykatów ma postać e1Pe2, gdzie P jest predykatem, zaś e1 oraz e2 są wyrażeniami zmiennych predykatu. Predykat równości, zmienne x oraz y mają tę samą wartość, wówczas predykat (x = y) ma wartość prawda (*true*), w pozostałych przypadkach ma wartość fałsz (*false*).

4.4.4.02. **Wyjaśnienie**. Predykat relacji arytmetycznej, jeśli zmienna n ma mniejszą wartość od zmiennej m (n < m), wówczas predykat ma wartość prawda (*true*), w pozostałych przypadkach ma wartość fałsz (*false*). Predykat przynależności do zbioru (x ∈ S), jeśli element x należy do zbioru S, wówczas predykat ma wartość prawda (*true*), w pozostałych przypadkach ma wartość fałsz (*false*). Predykat sprawdzenia przynależności podzbioru S do danego zbioru T (S ⊆ T), jeśli S jest podzbiorem zbioru T, wówczas predykat ma wartość prawda (*true*), w pozostałych przypadkach ma wartość fałsz (*false*).

4.4.4.03. **Wyjaśnienie**. Predykaty nie są wyrażeniami, są jak już zostało powiedziane, funktorami zdaniotwórczymi, mogą przyjmować po podstawieniu wartości: prawda (*true*) albo fałsz (*false*). Przykład łączenia wyrażeń prostych w wrażenia złożone, jest opisane niżej pokazanym schematem. Przypuśćmy, że pociąg porusza się ze stałą szybkością sześćdziesięciu kilometrów na godzinę, przez cztery godziny, jaki dystans pociąg przejechał?

```
│ dystans, prędkość, czas : ℕ
├──────────────────────────────
│ dystans = prędkość * czas
│ prędkość = 60
│ czas = 4
```

Żeby wyznaczyć dystans należy podstawić drugie i trzecie równanie do pierwszego. Z zależności, że jeśli a = b oraz b = c wynika, że a = c.

dystans = prędkość * czas – z definicji

dystans = 60 * czas – z drugiego równania

dystans = 60 * 4 – z trzeciego równania

dystans = 240 - z zasad arytmetyki.

Innym przykładem jest pracownik Filip zatrudniony w zespole zajmującym się adhezją w grupie badań materiałów, która z kolei jest częścią ośrodka badawczego. Czy Filip pracuje w ośrodku badawczym?:

```
│ Filip : OSOBY
│ inżynieria_przylepców, inżynieria_materiałowa, badania : ℙ JednOrg
├──────────────────────────────────────────
│ inżynieria_przylepców ⊆ inżynieria_materiałowa
│ inżynieria_materiałowa ⊆ badania
│ Filip ∈ inżynieria przylepców
```

Pokażemy, że Filip ∈ badania

1. Filip ∈ inżynieria przylepców – z definicji
2. inżynieria przylepców ⊆ materiały
3. materiały ⊆ badania

Ponieważ relacja bycia podzbiorem jest przechodnia S ⊆ T ⊆ U, to S ⊆ U, czyli jeśli Filip ∈ badania to Filip ∈ reaserch.

4.4.4.11. **Wyjaśnienie**. Aparat logiki matematycznej, a w szczególności rachunek zdań i rachunek funktorów, może z powodzeniem być użyty zarówno do opisu jak i do dowodzenia. Notacja Z podkreśla opisowe użytkowanie logiki. Przypomnijmy kilka podstawowych pojęć:

1. Funktory logiczne są nazywane predykatami. Predykaty po podstawieniu odpowiednich wartości na swoje zmienne przyjmują jedną z dwu wartości *true* albo *false*.
2. Używając logikę do opisu, powodujemy klasyfikowanie sytuacji do jednej z dwu kategorii: akceptowalnej (predykat przyjmuje wówczas wartość *true*) albo nieakceptowanej (predykat przyjmuje wówczas wartość *false*).
3. Specyfikacje pisane w notacji Z opisują sytuacje odpowiadające przyjęciu przez predykaty wartości *true*.
4. Użycie predykatów w powyższy sposób, pozwala na traktowanie ich jako testu akceptacji.

Zajmijmy się z kolei użycie predykatów dla celów opisowych. Na działanie każdego predykatu mają wpływ ograniczenia, które na dany predykat zostały nałożone. Np. zmienne d1, d2: 1 … 6, mają 36 możliwych kombinacji, jak pokazano w tabeli 4.4.4.12.

Tabela 4.4.4.12. Lista możliwych kombinacji pary zmiennych d1, d2.

1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3	3	4	4	4	4	4	4	5	5	5	5	5	5	6	6	6	6	6	6
1	2	3	4	5	6	1	2	3	4	5	6	1	2	3	4	5	6	1	2	3	4	5	6	1	2	3	4	5	6	1	2	3	4	5	6

Te same zmienne z dodatkowym predykatem d1 + d2 = 7 dają możliwość użycia zaledwie siedmiu kombinacji zmiennych d1, d2.

Przykłady elementarnych predykatów:

4.4.4.21. Stałe (Constant) : true, false

4.4.4.22. Równanie (Equations) : e1 = e2

4.4.4.23. Przynależność (Membership): x ∈ S

4.4.4.30. **Wyjaśnienie**. Wiele innych złożonych predykatów – zbudowanych jest z elementarnych predykatów. Możemy zbiory i relacje wykorzystywać, jako predykaty. Używanie składni prefiksowej, deklarowanej z podkreśleniem. Np. predykaty przynależności:

4.4.4.31. odd(x) czyli x ∈ (odd_); 5 < 12 czyli 5 ∈ (_ < _):

4.4.4.32. k divides 12 czyli (k, 12) ∈ divides;

4.4.4.33. matka(ismail, x) czyli (ismail, x) ∈ matka;

4.4.4.34. leap year czyli year ∈ (leap_)

4.4.4.40. **Wyjaśnienie**. Dla użycia składni typu infix wystarczy wypełnić podkreślone pole:

```
│ odd_, ODD : ℙS
├──────────────────────────────────────────────
│ odd(k) k ∈ ODD
```

Inny przykład to:

```
│ divides : S↔ℙ
├──────────────────────────────────────
│ (k, 12) ∈ divides
```

4.4.4.50. **Wyjaśnienie**. Złożone predykaty składają się z zasady z kombinacji prostszych predykatów. Zasady tworzenie złożonych predykatów z podstawowych można przedstawić następująco:

4.4.4.51. Iloczyn logiczny (*conjunction*) „∧" inaczej and używany do łączenia wymagań, ograniczeń i wzmocnień ograniczeń (podobnie działa do && w języku C).

4.4.4.52. Suma logiczna (*disjunction*) „∨" inaczej or używany do pokazania alternatywy, analizy przypadków i osłabienia ograniczeń (podobnie działa jak || w języku C).

4.4.4.53. Negacja logiczna (*negation*) „not" lub „!" (podobnie działa jak ! w języku C).

4.4.4.54. Równoważność (*eqivalence*) „⇔" inaczej „if and only if" (podobnie jak == w języku C).

4.4.4.50. Implikacja (*implication*) „⇒" inaczej „if ... then ..." używany do opisania niezmienników warunkowych.

4.4.4.60. **Wyjaśnienie**. Analiza przypadków. Operatory iloczynu logicznego i sumy logicznej są wspólnie używane przy analizie przypadków. Iloczyn logiczny łączy warunki, które występują w danym przypadku. Z kolei suma logiczna separuje poszczególne przypadki pomiędzy sobą. Np.:

```
TEMP == ℤ
PHASE :: = solid | liquid | gas

│ temp : TEMP
│ phase : PHASE
├──────────────────────────────────────
│ (temp < 0 ∧ phase = solid) ∨
│ (0 ≤ temp ≤ 100 ∧ phase = liquid) ∨
│ (temp > 100 ∧ phase = gas)
```

Powyższy przykład jest typowym przykładem opisu analizy przypadków w notacji Z.

4.4.4.70. **Wyjaśnienie**. Implikacje (wynikanie) zwykle oznaczone ⇒ lub if pokazują kolejność oddziaływania. Np. silniejszy predykat ⇒ związany z nim słabszy predykat. Poniżej tabelka 4.4.4.71. przedstawia zero – jedynkowa implikacji:

Tabelka 4.4.4.71. Zero – jedynkowa implikacja

p	false	false	true	true
q	false	true	false	true
⇒	true	true	false	true

Z pomocą implikacji można na przykład opisać wymagania bezpieczeństwa:

```
BEAM ::= off | on
DOOR ::= closed | open

│ beam : BEAM
│ door : DOOR
├──────────────────────────────────────
│ beam = on ⇒ door = closed
```

Powyższy przykład opisuje trzy sytuacje dotyczące wiązki światła i stanu drzwi. Poniżej tabelka 4.4.4.72.

Tabelka 4.4.4.71. Związki stanów			
BEAM	off	off	on
DOOR	closed	open	closed

4.4.4.80. **Wyjaśnienie**. Kwantyfikatory. Obok funktorów zdaniotwórczych (predykatów), ważną rolę w wysławianiu twierdzeń logicznych odgrywają słowa istnieje i każdy. Słowa te nazywamy kwantyfikatorami. Kwantyfikatory wprowadzają lokalne (*bound*) zmienne do predykatów identyfikujących liczny zestaw elementów.

4.4.4.81. **Wyjaśnienie**. ∀ uniwersalny kwantyfikator każdy (*all*), który można traktować jako uogólnienie iloczynu logicznego. Używany jest do opisania zbioru (funkcji, …) od lokalnej (lokalnych) zmiennej (zmiennych), których wartości tworzą zbiór rangi danej funkcji.

4.4.4.82. **Wyjaśnienie**. ∃ egzystencjalny kwantyfikator istnieje (*some*), który można traktować jako uogólnienie sumy logicznej. Używany jest od opisu istnienia co najmniej jednej wartości zmiennej lokalnej spełniającej predykat.

4.4.4.83. **Wyjaśnienie**. Zmienne lokalne (*bound*) kwantyfikatora istnieją tylko wewnątrz wyrażenia poprzedzonego kwantyfikatorem.

4.4.4.84. **Wyjaśnienie**. Kwantyfikator uniwersalny - jeżeli dana jest zmienna swobodna ns = {n1, n2, n3, ...}, to zapis:

∀ i : ns • i ≤ nmax jest równorzędny zapisowi n1 ≤ nmax ∧ n2 ≤ nmax ∧ n3 ≤ nmax ...

Kwantyfikator uniwersalny umożliwia całkowite opisanie (definiowanie) z pomocą jednego predykatu zbioru (relacji, funkcji, sekwencji, …). Wartości lokalnej zmiennej należą do zbioru elementów i działają integracyjnie z definicji.

```
│ divides : ℤ ⇔ ℤ
├──────────────────────────────────────
│ ∀ d, n : ℤ • d divides n ⇔ n mod d = 0
```

4.4.4.85. **Wyjaśnienie**. Kwantyfikator egzystencjonalny - jeżeli dana jest zmienną swobodną ns = {n1, n2, n3, ...} to zapis ∃ i : ns • i ≤ nmax jest równorzędny zapisowi n1 ≤ nmax ∨ n2 ≤ nmax ∨ n3 ≤ nmax ...

Kwantyfikator egzystencjonalny umożliwia w definicjach użycie lokalnych (w zasadzie ukrytych pod kwantyfikatorem) zmiennych, które mogą być więzami dla poszczególnych wartości.

```
│ _mod_ : ℤ → ℤ ( \ {0}) → ℤ
├──────────────────────────────────────
│ ∀ n, r : ℤ ; q : ℤ \ {0} •
│ n mod q = r ⇔ ( ∃ d • r < d ∧ n = q * d + r )
```

4.4.4.86. **Wyjaśnienie**. Notacja Z obok kwantyfikatora egzystencjonalnego korzysta z kwantyfikatora oznaczonego symbolem $\exists_1$ – istnieje jeden i tylko jeden.

4.4.4.90. **Wyjaśnienie**. Funkcjonalny opis zbioru (*Set Comprehension*). Analityczny opis zbioru mnogościowego pozwala na jednoznacznym opisywaniu zarówno zbiorów skończonych jak i poza-skończonych (nieskończonych). Pełny opis zbioru łączy deklaracje, predykat i wyrażenie postaci elementu zbioru: { deklaracja | predykat • wyrażenie } i powinno być rozumiane jako { źródło | filtr • wzorzec}. Trzeba pamiętać jednak, że wyrażenie i predykat występują opcjonalnie. Przykład

bez bezpośredniego podania postaci wyrażenia: $\mathbb{N} == \{i : \mathbb{Z} \mid i \geq 0\}$. Przykład bez bezpośredniego podania predykatu: $ODD == \{i : \mathbb{Z} \bullet 2 * i + 1\}$ Jeśli wyrażenie jest charakterystyczną krotką (*characteristic tuple*), to w pełnym opisie zbioru może zostać pominięte. Przykładowo:

4.4.4.91. $line == \{x, y : \mathbb{R} \mid y = m * x + b\}$;

znaczy to samo co: $line == \{x, y : \mathbb{R} \mid y = m * x + b \bullet (x, y)\}$.

Należy zauważyć, że pełne opisy zbiorów są specyfikacjami w miniaturze. Przykładem niech będzie definicja zbioru liczb pierwszych. Opis nieformalny zbioru liczb pierwszych wygląda jak niżej:

4.4.4.92 $PRIME == \{2, 3, 5, 7, 11, 13, 17, \ldots\}$

Jest to opis zbioru liczb pierwszych zawierający domyślne wartości liczb pierwszych większych od 17 i nie nadaje się do formalnego przekształcania i wnioskowania. Kolejną próbą opisania zbioru liczb pierwszych jest próba zapisania definicji zbioru liczb pierwszych.

Parafrazując definicję zbioru liczb pierwszych w języku potocznym, możemy przedstawić sformalizowaną pełny opis zbioru liczb pierwszych, jak poniżej:

4.4.4.93 $PRIME == \{n : \mathbb{N} \mid n > 1 \wedge \neg (\exists\, m : 2 \ldots n\text{-}1 \bullet n \bmod m = 0)\}$

Czy można to zapisać prościej? Pamiętamy, że liczba pierwsza nie jest iloczynem liczb większych niż jeden, czyli:

4.4.4.94 $\mathbb{N}_2 == \mathbb{N} \setminus \{0, 1, 2\}$ $PRIME == \mathbb{N}_2 \setminus \{n, m : \mathbb{N}_2 \bullet n * m\}$.

Prostszą postać opisu zbioru liczb pierwszych uzyskaliśmy stosując operatory zbiorów w miejsce posługiwania się negacją i kwantyfikatorem. Jest to zapis znacznie prostszy od formalizacji potocznej definicji liczb pierwszych.

4.4.4.95. **Wyjaśnienie**. Ograniczenia kwantyfikatorów. Ograniczenia kwantyfikatorów narzucają limity oddziaływania predykatom. Np.:

$\forall\, i : ns \mid odd(i) \bullet i \leq nmax$ oznacza

$\forall\, i : ns \bullet odd(i) \Rightarrow i \leq nmax$

Podobnie

$\exists\, i : ns \mid odd(i) \bullet i \leq nmax$ oznacza

$\exists\, i : ns \bullet odd(i) \Rightarrow i \leq nmax$

4.4.4.96. **Wyjaśnienie**. Ogólnie można powiedzieć, że

$(\forall\, d \mid p \bullet q) \Leftrightarrow (\forall\, d \bullet p \Rightarrow q)$

$(\exists\, d \mid p \bullet q) \Leftrightarrow (\exists\, d \bullet p \wedge q)$

4.4.4.97. **Wyjaśnienie**. Poniżej pokażemy konwencje i zapisy ze skrótami. Przykładowo - lokalna definicja czynnika w wyrażeniach:

$(let\ r == iroot(a) \bullet r * r \leq a < (r + 1) * (r + 1))$ oznacza

$iroot(a) * iroot(a) \leq a < (iroot(a) + 1) * (iroot(a) +1)$

4.4.4.98. **Wyjaśnienie**. Podobnie, wyrażenia warunkowe upraszczają dwu wynikową analizę przypadków:

$| x | = if\ x \neq 0\ then\ x\ else\ -x$ oznacza

$(x \geq 0 \wedge | x | = x) \vee (x < 0 \wedge | x | = -x)$

4.4.4.99. **Wyjaśnienie**. Abstrakcyjny typ danych składa się ze zbioru stanów, nazwanych przestrzenią stanów (*state space*), niepustego zbioru stanów początkowych i pewnej liczby operacji. W notacji Z zbiór stanów abstrakcyjnego typu danych jest specyfikowany schematem,

na ogół z tą samą nazwą jak opisywany typ danych. Powracając do pojęcia stanu, możemy powiedzieć, że:

1. Stan jest opisem sytuacji lub wynikiem zdarzenia.
2. Stan jest modelowany, jako przypisanie wartości do danej kolekcji zmiennych.
3. Stan może reprezentować założenie (warunki wstępne czyli *preconditions*), wynik (czyli *postconditions*) lub niezmienniczość wymagań (*invariants*).
4. Stany mogą być dzielone na te o stałej konfiguracji (*constans*) i o zmiennych składnikach (*state variables*).
5. Języki programowania nie opisują bezpośrednio stanów, mogą jedynie opisywać przejścia (*transition*) od jednego stanu do innego.
6. Natomiast język specyfikacji opisuje stany bezpośrednio.

Piśmiennictwo: *Spivey J.* S.10.1., *Woodcock J.* W.7.1.

4.4.5. MODELE STRUKTURY DANYCH

4.4.5.01. **Wyjaśnienie**. Wyrażenia w notacji Z, podobnie jak matematyka dyskretna umożliwiają modelowanie różnorodnych struktur danych. Przyjmuje, że interesujące są następujące konwencje dotyczą wyrażeń, zwanych odpowiednio:

- Krotkami (*Tuples*),
- Relacjami (*Relations*),
- Funkcjami (*Functions*) i
- Ciągami (*Sequences*).

4.4.5.02. **Wyjaśnienie**. Przez krotkę (*tuple*) – rozumiemy skończony uporządkowany ciąg zmiennych lub stałych rozdzielonych przecinkami i ujęte w nawiasy, np. (x, y, z, 2). Krotki są przykładem elementów typu produktu karteziańskiego (*cartesian product type*), gdzie poszczególne składowe odpowiadają elementom poszczególnych zbiorów składowych iloczynu kartezjańskiego. Krotki - służą do opisywania rekordów. Krotki (*tuples*) przypominają struktury z języka C lub rekordy z języka Pascal.

4.4.5.03. **Wyjaśnienie**. Relacja (*Relation*) binarna – są ważna konsekwencja teorii zbiorów dotyczy właściwości iloczynu karteziańskiego pary zbiorów zwanych odpowiednio domeną (*domain*) i rangą (*range*) . Relacje czyli tabele służą zarówno do opisania tabeli oraz do opisywania struktur danych powiązanych referencjami - tzw. *Linked data structures*.

4.4.5.04. **Wyjaśnienie**. Funkcja (*Function*) dyskretna – która też jest zbiorem uporządkowanych par elementów, jest szczególnym przypadkiem relacji. Funkcje spełniają pewien istotny warunek, mianowicie, że są jednowartościowe. Co oznacza, że jeśli pierwszy składnik z pary elementów składających się na argument funkcji jest identyczny to wówczas drugi składnik pary jest również równy dla obu par. Funkcje generują tzw. *Lookup tables*, drzewa i listy.

4.4.5.05. **Wyjaśnienie**. Ciąg (*Sequence*) – czyli skończona sekwencja elementów rozdzielonych przecinkami i ujętych w nawiasy trójkątne, np. robocze dnia tygodnia ⟨ monday, tuesday, wednesday, thursday, friday⟩ , służy do opisania list (*lists*) i tablic (*arrays*). Ciągi są definiowane z pomocą funkcji, co pokażemy dalej. Sekwencje są funkcjami, a funkcje są zbiorami. Sekwencje można zapisać następująco:

workday = {1↦monday, 2↦tuesday, 3↦wednesday, 4↦thursday, 5↦friday},

gdzie znak ↦ jest symbolem przyporządkowania pierwszego elementu pary – drugiemu elementowi.

Piśmiennictwo: *Spivey J.* S.10.1., *Woodcock J.* W.7.1.

4.4.6. OPERACJE NA RELACJACH

4.4.6.01. **Wyjaśnienie**. Pary i relacje binarne. Dotychczas zajmowaliśmy się krotkami o liczbie składowych trzy lub więcej, teraz zajmiemy się krotkami o dwu składowych. Pary to krotki o dwóch składowych: np. (Adam, 417). Użycie strzałki mapowania ↦ dostarcza alternatywnej bez-nawiasowej składni. Adam ↦ 4117. Na parach określona są operatory. Operatory rzutowania pary *first* oraz *second* wybierają odpowiednio pierwszy oraz drugi komponent pary.

first (Adam, 4117) = Adam
second (Adam, 4117) = 4117

4.4.6.02. **Wyjaśnienie**. Relacje binarne są zbiorem par. Relacje binarne zapisujemy na jeden z dwóch sposobów:

ℙ (NAME x PHONE) lub NAME ↔ PHONE.

Poniżej przykład tabelki 4.4.6.71. opisu relacji binarnej:

Tabelka 4.4.6.71. Relacja binarna								
NAME	Adam	Filip	Danek	Danek	Filip	Franek	Franek	...
PHONE	4019	4107	4107	4136	0113	0110	6190	...

Relacje binarne służą do modelowania mechanizmu zwanego *lookup tables*. W notacji Z relacja binarna może być zadeklarowana następująco:

```
│ phone : NAME ↔ PHONE
├──────────────────────────────────────────────
│ phone = { ..., Adam↦4019, Filip↦4107, Danek↦4107, Danek↦4136,
│       Filip↦0113, Franek↦0110, Franek↦6190, ... }
```

4.4.6.10. **Wyjaśnienie**. Rachunek relacji. *Domain* oraz *Range* są zbiorami odpowiednio pierwszego i drugiego komponentu binarnej relacji. Wyobraźmy sobie, że dysponujemy parą zbiorów na których określona jest relacja binarna, jak poniżej:

dom phone = { ... , Adam, Filip, Danek, Franek, ... }
ran phone = { ... , 4019, 4107, 4136, 0113, ... }

Autorzy notacji Z dostarczyli w tzw. *tool kit,* czyli definicję szeregu operacji wykonywanych na relacjach binarnych.

4.4.6.11. **Wyjaśnienie**. Pierwszą operacją na relacji binarnej, o której będziemy mówili, jest tzw. *Relational image,* jest modelem funkcjonalności zwanym *table lookup*:

phone ⦇ { Danek, Filip } ⦈ = {4107, 4136, 0113}.

4.4.6.12. **Wyjaśnienie**. Kolejną omawianą operacją są tzw. operatory restrykcji, które mogą np. służyć do modelowania kwerend bazy danych. Operator restrykcji domeny selekcjonuje pary w oparciu o wartości pierwszego komponentu:

{Danek, Filip}◁ phone={Filip↦4107,Danek↦4107,Danek↦4136, Filip↦0113}

4.4.6.13. **Wyjaśnienie**. Operator restrykcji rangi (*range*) selekcjonuje pary w oparciu o wartość drugiego komponentu:

phone ▷ (4000 ... 4999) = {Adam↦4019, Filip↦4107, Danek↦4107, Danek↦4136}

4.4.6.14. **Wyjaśnienie**. Operator *Overriding* modeluje aktualizację bazy danych. Np.:

phone ⊕ {heater↦4026, Adam↦4026} = { … , Adam↦4026, Filip↦4107, Danek↦4107, Danek↦4136, Filip↦0113, Franek↦0110, Franek↦6190, heater↦4026, … }

4.4.6.15. **Wyjaśnienie**. Operator *Inverse* zamienia miejscami domenę i rangę – wymieniając komponenty każdej pary. Np.:

phone˜ = { … , 4019↦Adam, 4107↦Filip, 4107↦Danek, 4136↦Danek, 0013↦Filip, 0110↦Franek, 6190↦Franek, … }

4.4.6.16. **Wyjaśnienie**. Operator kompozycji ⨾ łączy dwie relacje w relacje binarną, na drodze dobierania par z kolejnych składników obu relacji. Np. komponując NAME ↔ PHONE nazwaną phone, z poniższą relację:

PHONE ↔ DEPT nazwaną dept.

```
│ dept : PHONE ↔ DEPT
├──────────────────────────────────────────
│ dept = { 0000↦admin, … , 0999↦admin, 4000↦badania,
│ … , 4999↦badania, 6000↦Franek, … }
```

W wyniku powstaje relacja binarna:

phone ⨾ dept = { … , Adam↦badania, Filip↦badania, Danek↦badania, Filip↦admin, Franek↦admin, Franek↦Franek}

Piśmiennictwo: *Spivey J.* S.10.1., *Woodcock J.* W.7.1.

4.4.7. FUNKCJE I FORMUŁY

4.4.7.01. **Wyjaśnienie**. Funkcje są binarnymi relacjami, w których każdy element domeny pojawia się dokładnie jeden raz. Każdy element domeny jest unikalnym kluczem.

```
│ phonef : NAME ⇸ PHONE
├──────────────────────────────────────────
│ phonef = { … , Adam↦4019, Filip↦4107,
│ Danek↦4107, Franek↦6109, … }
```

4.4.7.02. **Wyjaśnienie**. Jednym z zastosowań funkcji, jest szczególny przypadek *Relational image*. Wskazany element domeny jest związany z jedną wartością rangi. Np.

phonef ⦇ { Danek } ⦈ = 4107 – ponieważ phonef (Danek) = 4107,
czyli phonet Danek = 4107

4.4.7.03. **Wyjaśnienie**. Relacje binarne i funkcje – mają typ zbioru ℙ (X x Y)

4.4.7.11 X ↔ Y Relacje binarne typu wiele-do-wielu.
4.4.7.12 X ⇸ Y Tzw. *partial functions*: wiele-do-jednego. Niektóre funkcje nie są zdefiniowane.
4.4.7.13 X → Y Tzw. *total functions*. Wszystkie funkcje aplikacji są zdefiniowane.
4.4.7.14 X ⤔ Y Tzw. *partial injections*: jeden-do-jednego. Odwrotność jest również funkcją.
4.4.7.15 X ↣ Y Tzw. total injections.
4.4.7.16 X ⤖ Y Tzw. *Bijections*: funkcje, które pokrywają całkowity – niepodzielny zbiór wartości (*range*).

4.4.7.21. **Wyjaśnienie**. Format operatorów. Większość operatorów to funkcje z tzw. *infix syntax*. Np. 2 + 3 = 5 to funkcja aplikacyjna, która może być zapisana jako: (_ + _) (2, 3) = 5. Ta funkcja może być zapisana jako:

```
│ _ + _ : ℤ×ℤ→ℤ
├──────────────────────────────────────
│ …
```

4.4.7.22. **Wyjaśnienie**. W notacji Z znany nam dobrze znak plus „+” jest nazwą zbioru: (_ + _) = { … , (1,1)↦2, (1,2)↦3, … }

Wybrane symbole operatorów i składnia. Symbole operatorów są definiowane poniżej:

4.4.7.31	$X \lhd R$	Ograniczenie domeny
4.4.7.32	$X \lhd R$	Anty-ograniczenie domeny
4.4.7.33	$R \rhd Y$	Ograniczenie rangi
4.4.7.34	$R \lhd Y$	Anty-ograniczenie rangi

4.4.7.41. **Wyjaśnienie**. Symbole operacji (*pictorials*) przypomina nam kolejność występowania operatorów oraz ułatwiają zapamiętanie funkcjonalności poszczególnych operatorów. Np.

{Danek, Filip} ◁ phone ▷ {4000, …, 4999} =
{Filip↦4107, Danek↦4107, Danek↦4136}

4.4.7.42. **Wyjaśnienie**. Operacje notacji Z są definiowane z pomocą formuł. W opisie formuł wykorzystywane są z pomocą symboli. Te symbole zawierają w większości przypadków, to samo znaczenie, jakie mają w znanych nam działach matematyki.

```
┌=[X,Y]=====================================
│ _◁_, _⩤_ : ℙ X×(X↔Y)→X↔Y
│ _◁_, _⩤_ : (X↔Y)×ℙY→X↔Y
│    ˜ : (X↔Y)→(Y↔X)
│ _ (| _ |) : (X↔Y)×ℙX→ℙY
│ _ ⊕ _ : (X↔Y)× (X↔Y)→( X↔Y)
├──────────────────────────────
│ ∀ x : X; y : Y; S : ℙX; T : ℙY; Q, R : X↔Y •
│ S ◁ R = {x : X; y : Y | x ∈ S ∧ x R y } ∧
│ X ⩤ R = {x : X; y : Y | x ∉ S ∧ x R y } ∧
│ R ▷ T = {x : X; y : Y | x R y ∧ y ∈ T} ∧
│ R ⩥ T = {x : X; y : Y | x R y ∧ y ∉ T} ∧
│ R ˜ = {x : X; y : Y | x R y • y ↦ x} ∧
│ R (| S |) = {x : X; y : Y | x ∈ S ∧ x R y • y} ∧
│ Q ⊕ R = ((dom R) ⩤ Q) ∪ R
└──────────────────────────────
```

Piśmiennictwo: *Spivey J.* S.10.1., *Woodcock J.* W.7.1.

4.4.8. Definicje uogólnione i typy schematów

4.4.8.01. **Wyjaśnienie**. Definicja konkatenacji sekwencji (_⌢_), zastosowana tylko do elementów jednego typu nie będzie w praktyce zbyt przydatna. Np.:

```
│ _⌢_ : TEXT × TEXT → TEXT
├──────────────────────────
│ …
```

4.4.8.02. **Wyjaśnienie**. Dla umożliwienia wprowadzenia definicji dotyczących wielu typów, w notacji Z wprowadzono pojęcie definicji uogólnionej:

```
┌=NAME[TYPE]=============================
│ DECLS
├──────────────────────────
│ PREDS
└──────────────────────────
```

W przypadku konkatenacji sekwencji, przyjmie ona postać

```
┌=[X]=====================================
│ _⌢_ : seqX × seqX → seqX
├──────────────────────────
│ …
└──────────────────────────
```

gdzie X jest parametrem definicji uogólnionej, który można zastępować potrzebną wartością parametru. Definicja uogólniona korzysta z wzorca: $X \leftrightarrow Y == \mathbb{P}(X \times Y)$

Definicje uogólnione pozwalają na rozszerzanie notacji Z.

4.4.8.11. **Wyjaśnienie**. Typy swobodne (*Free Types*). Typ swobodny umożliwia modelowanie danych typu rekursywnego używanego do tworzenia struktur z linkami. Np.: definiowanie składni prostych wyrażeń arytmetycznych na liczbach naturalnych, tak jak

2 + 3 oraz (12 div (2 + 3)) – 7:
OP ::= plus | minus | times | divie
EXP ::= const ⟨⟨ ℕ ⟩⟩ | binop ⟨⟨ OP × EXP × EXP ⟩⟩

Zajmiemy się z kolei pozostałymi własnościami rachunku schematów. Np. istnieje operator określony na schematach dla każdej operacji logicznej oraz kwantyfikatorów. Podkreślić należy, że najbardziej przydatne są operacje iloczynu logicznego i sumy logicznej schematów. Natomiast negacja logiczna schematu nie jest intuicyjna i niezbyt przydatna.

4.4.8.21. **Wyjaśnienie**. Dodatkowo dwa istniejące operatory nie bazują na rachunku logicznym, Są to: S ⨾ T gdzie S i T są dwoma schematami, a operacja nazywa się kompozycją schematów (*schema composition*) oraz S >> T operacja nazywa się potokiem schematów (*schema piping*). Operacja kompozycji jest podobna operacji kompozycji dla relacji, zaś potok jest podobny do koniunkcji, a nie do sekwencji wyrażeń. Z nie jest językiem programowania.

4.4.8.22. **Wyjaśnienie**. Schematy mogą być używane, jako deklaracje lub predykaty. Przy czym, schematy mogą być używane podobnie jak makra do pisania zwartych formuł. Każdy schemat definiuje własny typ którego wystąpienia są podobnymi do rekordów elementami wiążącymi (*bindings*). Notacja Z posiada jedynie pięć rodzajów typów:

- Typy podstawowe [X] – wystąpienia są indywidualne
- Typy zbiorowe ℙX – wystąpieniami są zbiory
- Typy produktów kartezjańskich X × X wystąpienia są krotkami (*tuples*)
- Typy schematów ⦃ x: X; y: Y ⦄ wystąpienia są elementami wiążącymi
- Typy schematów oraz elementy wiążące umożliwiają modelowanie w notacji Z wielkich systemów i stosowanie podejścia obiektowego.

4.4.8.23. **Wyjaśnienie**. Typy produktów kartezjańskich. Deklaracja produktu karteziańskiego DATE, ma postać:

DATE == DAY × MONTH × YEAR gdzie wystąpieniami DATA są krotki (*tuples*)

```
│ landing, opening : DATE
├──────────────────────────────
│ landing = (20, 7, 1969)
│ opening = (9, 11, 1998)
```

Nic jednak nie chroni krotek – przed prezentowaniem nieistniejących dat. Np.:

```
┌──────────────────────────────
│ d : DATE
├──────────────────────────────
│ d = (29, 2, 1997)
└──────────────────────────────
```

Typ produkt kartezjański nie może opisywać więzów pomiędzy swoimi składnikami.

4.4.8.24. **Wyjaśnienie**. Typy schematów. Typ schematu może zawierać więzy pomiędzy komponentami. Np.:

days == ⟨31, 28, 31, 30, 31, 30, … , 31, 30, 31 ⟩

```
┌─TypicalDate──────────────────────────────
│ day : 1 … 31
│ month : 1 … 12
│ year : ℤ
├──────────────────────────────────────────
│ day ≤ days month
└──────────────────────────────────────────
```

Innym przykładem może być rok przestępny:

(leap _) == {y : ℤ • 4 * y} \ ({y : ℤ • 100 * y} \ {y : ℤ • 400 * y})

```
┌─Feb29────────────────────────────────────
│ month, day, Lear : ℤ
├──────────────────────────────────────────
│ month = 2
│ day = 29
│ leap year
└──────────────────────────────────────────
```

Date ≙ TypicalDate ∨ Feb29

4.4.8.31. **Wyjaśnienie**. Sygnatura typu schemat. Typ schematu wyznaczany jest tzw. sygnaturę (*signature*) schematu. Na sygnaturę składają się nazwy i typy zmiennych stanu schematu. Kolejność występowania zmiennych stanu schematu nie ma wpływu na typ danego schematu. Po normalizacji, omawiany wcześniej schemat TypicalDate, przyjmie postać:

```
┌─TypicalDate──────────────────────────────
│ days, month, year : ℤ
├──────────────────────────────────────────
│ day ∈ 1 … 31
│ month ∈ 1 … 12
│ day ≤ days month
└──────────────────────────────────────────
```

Typem schematu jest wyrażenie: ⦇ day, month, year : ℤ ⦈

4.4.8.32. **Wyjaśnienie**. Używanie typu schemat. Referencje schematu mogą pojawić się w deklaracji innego schematu. Selektor kropka (*dot*) wskazuje określoną zmienną stanu. Np.:

```
┌─Landing──────────────────────────────────
│ landing : Date
├──────────────────────────────────────────
│ landing.day = 20
│ landing.month = 7
│ landing.year = 1969
└──────────────────────────────────────────
```

Innym przykładem referencji jest:

```
┌─Multi.Editor─────────────────────────────
│ active : NAME
│ buffet : NAME ⇸ Editor
├──────────────────────────────────────────
│ active ∈ dom buffer
└──────────────────────────────────────────
```

Referencje schematu mogą być użyte, jako zbiór wartości wyrażenia.

4.4.8.33. **Wyjaśnienie**. Notacja typu schematu. Zgodnie z zasadami notacji Z, notacja dotycząca typu schematu i elementów wiążących nie są częścią notacji Z. Dlatego też poniższy zapis nie jest zapisem w notacji Z:

```
┌─Landing──────────────────────────────────
│ landing : ⦇ day, month, year : ℤ ⦈
├──────────────────────────────────────────
│ landing : ⦇ day:>20, month:>7, year:>1969 : ℤ ⦈
└──────────────────────────────────────────
```

Zamiast powyższego zapisu musimy użyć zapisu:

```
┌─ Landing──────────────────────────────────
│ landing : Date
├───────────────────────────────────────────
│ landing.day = 20
│ landing.month = 7
│ landing.year = 1969
└───────────────────────────────────────────
```

4.4.8.34. **Wyjaśnienie**. Operator formacji wiązań. Operator θS (theta S) jest nienazwanym wystąpieniem schematu typu S wchodzącego w zakres. Zamiast więc pisać:

```
┌─Calendar──────────────────────────────────
│ weekday : Date 0 … 6
│ d : Date
├───────────────────────────────────────────
│ weekday(d) = (d.day + …) mod 7
└───────────────────────────────────────────
```

Możemy napisać:

```
┌─ Calendar─────────────────────────────────
│ weekday: Date 0 … 6
├───────────────────────────────────────────
│ ∀ Date • weekday(θDate) = (day + … ) mod 7
└───────────────────────────────────────────
```

4.4.8.31. **Wyjaśnienie**. Schematy, jako relacje. Można zdefiniować binarną relację odpowiednik pewnego schematu operacji. Np.:

```
┌─Precedes──────────────────────────────────
│ ΔDate
├───────────────────────────────────────────
│ (year < year′) ∨
│ (year < year′ ∧ month = month′) ∨
│ (year < year′ ∧ month = month′ ∧ day < day′)
└───────────────────────────────────────────
```

Precedes == {Precedens • (θDate, θDate′)}

Teraz Precedes może być użyty, jako rodzaj binarnej relacji na danych:

```
┌─Biography─────────────────────────────────
│ name : NAME
│ birth, death : Date
│ …
├───────────────────────────────────────────
│ birth precedes death
│ …
└───────────────────────────────────────────
```

Piśmiennictwo: *Spivey J.* S.10.1., *Woodcock J.* W.7.1.

4.4.9. Podstawy tzw. Refinement'u

4.4.9.01. **Wyjaśnienie**. Podstawą tak zwanego *refinement'u* jest wnioskowanie formalne. Należy pamiętać, że notacja Z służy, w pierwszej kolejności, do opisywania i przekształcania - abstrakcyjnego modelu wymagań oraz powstającego projektu dla systemu informatycznego. Dotyczy to więc tego, co w tradycyjnych metodach projektowania systemów informatycznych, nazywamy wymaganiami funkcjonalnymi i tej części projektu systemu która odpowiada za realizację wymagań funkcjonalnych. Tak zwane wymagania poza-funkcjonalne dotyczą np. liczby równoczesnych użytkowników przyszłego systemu, liczby równocześnie obsługiwanych transakcji, czasu reakcji systemu, pojemności bazy danych, itd. W wyniku wieloetapowego procesu - tzw. *refinement* - następuje włączanie do opisu systemu kolejnych wymagań poza-funkcjonalnych i przechodzenie tym samym od modelu abstrakcyjnego systemu do modelu

rzeczywistego. Włączanie kolejnych ograniczeń poza-funkcjonalnych do modelu systemu, powoduje zmniejszanie ogólności przyjętych kolejno wersji modelu systemu.

Tym samym poprzedzający dany wariant modelu sytemu – model systemu jest bardziej ogólny od danego wariantu modelu systemu. Można to powiedzieć inaczej, że poprzedzający wariant modelu systemu powstaje przez rozszerzenie ogólności (skali abstrakcji) danego modelu systemu.

4.4.9.02. **Wyjaśnienie**. Pamiętajmy, że:

1. Nie wszystkie wymagania funkcjonalne mogą być w pełni określone w czasie formułowania pierwszej wersji wymagań na system.
2. Z pewnych wymagań formalnych i ich związków, możemy wyprowadzić wiele wniosków na drodze rozumowania formalnego.
3. Traktujmy model formalny jako nie wykonywalny prototyp.
4. Badamy właściwości kolejnych wariantów modelu systemu w hipotetycznych sytuacjach.
5. W szczególności, oceniamy jak ma się opisana w notacji Z specyfikacja - do wymagań funkcjonalnych.
6. Sprawdźmy spójność i kompletność.
7. Przeliczmy wszystkie warunki początkowe.
8. Sprawdźmy udoskonalenia (*refinement*) specyfikacji postępując w kierunku projektu szczegółowego.

4.4.9.11. **Wyjaśnienie**. Zadanie wnioskowania formalnego, w przypadku zależności arytmetycznych, jest rodzajem kalkulacji. Np.: „Pociąg porusza się ze stałą szybkością sześć mil na godzinę. jaki dystans pokona pociąg w cztery godziny?”

4.4.9.12. **Wyjaśnienie**. Wnioskowanie na temat przynależności do zbiorów. W tym przypadku mamy do czynienia z wnioskowaniem formalnym, niebędącym procesem obliczeń arytmetycznych. Np.: „Filip pracuje w zespole ds. przylepców, zespól ten wchodzi w skład grupy badań materiałowych, która z kolei wchodzi w skład dywizji badawczej. Czy Filip pracuje w obszarze badań?”

4.4.9.13. **Wyjaśnienie**. Wnioskowanie z wykorzystaniem równoważności i wynikania (implikacji). Przyjmiemy założenie, że każdy krok jest predykatem, a kolejne kroki między sobą połączone są przechodnie logiczne spójniki ⇔ oraz ⇒

```
┌─Example──────────────────────────
│ x : ℤ
├──────────────────────────────────
│ 2 * x + 7 = 13
└──────────────────────────────────
```

Poszukamy wartości niewiadomej x metodą wnioskowania z wykorzystaniem równoważności i wynikania:

1. $2 * x + 7 = 13$ - to wiemy z definicji
2. $\Leftrightarrow 2 * x = 13 - 7$ - przez odjęcie od każdej ze stron liczby 7
3. $\Leftrightarrow 2 * x = 6$ - wynika z arytmetyki
4. $\Rightarrow (2 * x) \text{ div } 2 = 6 \text{ div } 2$ - przez dzielenie obu stron przez 2
5. $\Leftrightarrow x = 3$ - wynika z arytmetyki.

Tak więc dowiedliśmy, że $2 * x + 7 = 13 \Rightarrow x = 3$

Prawa logiki funktorów (takie jak: prawo *Leibnica*, prawo pochłaniania fałszu, prawo pochłaniania prawdy, prawo pochłaniania fałszu, prawo wyłączonego środka, prawo kontradykcji, prawa *DeMorgana*, itd.) są wykorzystywane we wnioskowaniu, w ramach czynności zwanych refinment'em.

4.4.9.21. **Wyjaśnienie**. Badanie warunku wstępnego (*Precondition by Inspection*). Warunek wstępny jest predykatem, który musi przyjmować wartość true dla operacji określonej na wejściach oraz wartościach zmiennych nie oznaczonych prymem.

4.4.9.22. **Wyjaśnienie**. Przypadek domyślnego warunku wstępnego. Predykat domyślny ma miejsce wtedy i tylko wtedy, gdy operacja określa interakcję z nie-zmiennikiem stanu. Np.: warunek wstępny operacji Op dotyczy stanu S, jeśli $\exists S' \bullet Op$, co można odczytać „Istnieje stan końcowy S′ spełniający predykat operacji Op".

4.4.9.31. **Wyjaśnienie**. Przekształcanie projektu od wymagań do systemu aplikacyjnego (*refinement*). Przekształcanie odbywa się na drodze kolejnych kroków projektowych (*refinement steps*). Przejścia pomiędzy kolejnymi fazami projektowymi wykorzystują dla sprawdzenia poprawności funkcję logiczną implikację. Wyrażenie $p \Rightarrow q$ oznacza, że p ma wszystkie właściwości q oraz pewne dodatkowe. Badając q powinniśmy być zadowoleni z własności p:

stronger ⇒ weaker
concrete ⇒ abstract
implementation ⇒ specification
implementation ⇒ design_n ⇒ … ⇒design_1⇒ specification

4.4.9.31. **Wyjaśnienie**. Powyższy kierunek jest odwrotny do kierunku kolejnych kroków projektowych.
Przykład:

4.4.9.32	$x' > x$	- specyfikacja
4.4.9.33	$x' = x + 1$	- implementacja
4.4.9.34	$x' = x + 1$ to przecież spełnia $x' > x$	- dowód.

4.4.9.41. **Wyjaśnienie**. Modelowanie dużych systemów z wykorzystaniem notacji Z wymaga:

1. Wydzielenie procesów obsługi kolejnych poziomów od procesów głównych systemu aplikacyjnego.
2. Opisanie hierarchii procesów systemu aplikacyjnego, traktując je, jako przypadki użycia (*use casus*).

Na tym zakończymy rozważania dotyczące charakterystyki notacji Z. Więcej informacji dostarcza piśmiennictwo.

Piśmiennictwo: *Spivey J.* S.10.1., *Woodcock J.* W.7.1.

4.5. Język SQL

4.5.0. Uwagi wstępne

Język SQL – to strukturalny język zapytań używany do tworzenia, modyfikowania baz danych (zarówno transakcyjnych, jak i analitycznych) oraz do umieszczania i pobierania danych z baz danych. SQL jest językiem deklaratywnym. Decyzję o sposobie przechowywania i pobrania danych pozostawia się systemowi zarządzania bazą danych (DBMS). Język SQL *(Structured Query Language)* pojawił się w roku 1974, pochodne pierwotnej wersji języka to kolejno: SQL-86, SQL-

89, SQL-92, SQL:1999, SQL:2003, SQL:2006, SQL:2008. Aktualna stabilna wersja języka to SQL:2008/2011. Twórcami języka SQL są *Donald D. Chamberlin* i *Raymond F. Boyce*. Język SQL jest dostępny na większości platform systemowych.

Język SQL, jak to już zostało powiedziane, był opracowany w latach 70. w firmie IBM (pierwotną nazwą języka miał być *SEQUEL*, jednakże okazało się, że nazwa ta była już zastrzeżona przez brytyjską wytwórnię lotniczą Hawker Siddeley). Stał się standardem w komunikacji z serwerami relacyjnych baz danych. Wiele współczesnych systemów relacyjnych baz danych używa do komunikacji z użytkownikiem SQL, dlatego potocznie mówi się, że korzystanie z relacyjnych baz danych to korzystanie z SQL-a. Pierwszą firmą, która włączyła SQL do swojego produktu komercyjnego, był Oracle. Dalsze wprowadzanie SQL-a, w produktach innych firm, wiązało się nierozłącznie z wprowadzaniem modyfikacji pierwotnego języka. Wkrótce utrzymanie dalszej jednolitości języka wymagało wprowadzenia standardu.

Piśmiennictwo: *Gruber M.* G.4.1., *Jakubowski A.* J.2.1.

4.5.1. Standardy SQL

W 1986 SQL stał się oficjalnym standardem, wspieranym przez Międzynarodową Organizację Normalizacyjną (ISO) i jej członka, Amerykański Narodowy Instytut Normalizacji (ANSI). Wczesne wersje specyfikacji (SQL86 i SQL89) były w dużej mierze jedynie określeniem wspólnej płaszczyzny łączącej różne istniejące wówczas produkty i pozostawiały wiele swobody twórcom implementacji. Z czasem jednak systemy komputerowe uległy integracji i rynek zaczął domagać się aplikacji oraz ich funkcji umożliwiających współpracę z wieloma różnymi bazami danych. Pojawiła się potrzeba określenia ściślejszego standardu, który mógłby jednocześnie obejmować nowe elementy, nieujęte do tej pory w języku. Tak powstał standard SQL92, obowiązujący w produktach komercyjnych do dziś.

W 2003 przedstawiono SQL:2003 – nowy standard języka SQL. Został on opublikowany w Sigmod Record Vol. 33 No. 1 marca 2004. Jest to w zasadzie poprawione SQL:1999 z wyjątkiem części SQL/XML oraz kilku dodatkowych właściwości.

Zmiany wprowadzone w SQL:2003:

1. Dodano nowe typy danych (`BIGINT, MULTISET` oraz `XML`).
2. Usunięto typy `BIT` oraz `BIT VARYING`.
3. Wprowadzono rozszerzenia w sposobie wywoływania procedur.
4. Poszerzono instrukcję `CREATE TABLE (CREATE TABLE { LIKE | AS }).`
5. Wprowadzono instrukcję `MERGE`.
6. Wprowadzono nowy obiekt schematu – generator sekwencji.
7. Wprowadzono dwa nowe typy kolumn – identyfikatory oraz generowane.
8. Wprowadzono retrospektywne sprawdzanie więzów integralności.
9. Wprowadzono rozszerzenia dla `OLAP` w formie wbudowanych funkcji (skalarnych i agregujących).
10. Wprowadzono klauzulę `WINDOW`.

Piśmiennictwo: *Gruber M.* G.4.1., *Jakubowski A.* J.2.1.

4.5.2. FUNKCJE SILNIKA I OPROGRAMOWANIA POŚREDNICZĄCEGO

Produkty związane z relacyjnymi bazami danych to nie tylko serwery. Sam serwer określa się często takimi nazwami jak „*back end*", „*engine*", czy też „*motor/silnik bazy danych*". Przechowuje on dane oraz zapewnia ich pobieranie i aktualizacje w odpowiedzi na pobierane instrukcje w SQL-u. Uzupełnieniem serwera jest zazwyczaj „*front end*", „*oprogramowanie pośredniczące*" czy też „*fronton*" – narzędzia upraszczające komunikację z serwerem i wyposażone w mechanizmy pozwalające wykorzystać pobrane dane. Należą do nich mechanizmy generowania i obsługi formularzy oraz raportów, języki czwartej generacji (4GL), graficzne języki zapytań, narzędzia konstrukcyjne użytkownika, oprogramowanie do prezentacji multimedialnych, systemy tworzenia hipertekstu, systemy CAD/CAM, arkusze kalkulacyjne, jak również interfejsy dostępu bezpośredniego. Wszystkie one wykorzystują do komunikacji z serwerem i wykonywania za jego pośrednictwem różnych operacji język SQL. Serwer odpowiada za przechowywanie, porządkowanie i pobieranie danych, zapewnia ich integralność, bezpieczeństwo oraz zabezpiecza przed ewentualnymi konfliktami między użytkownikami.

Piśmiennictwo: *Gruber M.* G.4.1., *Jakubowski A.* J.2.1.

4.5.3. FORMY SQL

Z technicznego punktu widzenia, SQL jest pod-językiem danych. Oznacza to, że jest on wykorzystywany wyłącznie do komunikacji z bazą danych. Nie posiada on cech pozwalających na tworzenie kompletnych programów. Jego wykorzystanie może być trojakie i z tego względu wyróżnia się trzy formy SQL-a:

4.5.3.10. **Interakcyjny** kod SQL (autonomiczny) wykorzystywany jest przez użytkowników w celu bezpośredniego pobierania lub wprowadzania informacji do bazy. Przykładem może być zapytanie prowadzące do uzyskania zestawienia aktywności kont w miesiącu. Wynik jest wówczas przekazywany na ekran, z ewentualną opcją przekierowania go do pliku lub drukarki.

4.5.3.20. **Statyczny** kod SQL (*Static SQL*) nie ulega zmianom i pisany jest wraz z całą aplikacją, podczas której pracy jest wykorzystywany. Nie ulega zmianom w sensie zachowania niezmiennej treści instrukcji, które jednak zawierać mogą odwołania do zmiennych lub parametrów przekazujących wartości z lub do aplikacji. Statyczny SQL występuje w dwóch odmianach:

1. **Embedded SQL** (*Osadzony SQL*) oznacza włączenie kodu SQL do kodu źródłowego innego języka. Większość aplikacji pisana jest w takich językach jak C++ czy Java, jedynie odwołania do bazy danych realizowane są w SQL. W tej odmianie statycznego SQL-a do przenoszenia wartości wykorzystywane są zmienne.
2. **Język modułów.** W tym podejściu moduły SQL łączone są z modułami kodu w innym języku. Moduły kodu SQL przenoszą wartości do i z parametrów, podobnie jak to się dzieje przy wywoływaniu podprogramów w większości języków proceduralnych. Jest to pierwotne podejście, zaproponowane w standardzie SQL. Embedded SQL został do oficjalnej specyfikacji włączony nieco później.

4.5.3.30. **Dynamiczny** kod SQL (*Dynamic SQL*) generowany jest w trakcie pracy aplikacji. Wykorzystuje się go w miejsce podejścia statycznego, jeżeli w chwili pisania aplikacji nie jest możliwe określenie treści potrzebnych zapytań – powstaje ona w oparciu o decyzje

użytkownika. Tę formę SQL generują przede wszystkim takie narzędzia jak graficzne języki zapytań. Utworzenie odpowiedniego zapytania jest tu odpowiedzią na działania użytkownika.

4.5.3.40. **Wyjaśnienie**. Wymagania tych trzech form różnią się i znajduje to odbicie w wykorzystywanych przez nie konstrukcjach językowych. Zarówno statyczny, jak i dynamiczny SQL uzupełniają formę autonomiczną cechami odpowiednimi tylko w określonych sytuacjach. Zasadnicza część języka pozostaje jednak dla wszystkich form identyczna.

Piśmiennictwo: *Gruber M.* G.4.1., *Jakubowski A.* J.2.1.

4.5.4. Składnia SQL

Użycie SQL, zgodnie z jego nazwą, polega na zadawaniu zapytań do bazy danych. Zapytania można zaliczyć do jednego z czterech głównych podzbiorów:

- SQL DML (*Data Manipulation Language* – „język manipulacji danymi”),
- SQL DDL (*Data Definition Language* – „język definicji danych”),
- SQL DCL (*Data Control Language* – „język kontroli nad danymi”).
- SQL DQL (*Data Query Language* – „język definiowania zapytań”).

Instrukcje SQL w obrębie zapytań tradycyjnie zapisywane są wielkimi literami, jednak nie jest to wymóg. Każde zapytanie w SQL-u musi kończyć się znakiem średnika (;).

Dodatkowo, niektóre programy do łączenia się z silnikiem bazy danych (np. *psql* w przypadku *PostgreSQL*), używają swoich własnych instrukcji, spoza standardu SQL, które służą np. do połączenia się z bazą, wyświetlenia dokumentacji itp.

4.5.4.01. DML (*Data Manipulation Language*) służy do wykonywania operacji na danych – do ich umieszczania w bazie, kasowania, przeglądania oraz dokonywania zmian. Najważniejsze polecenia z tego zbioru to:

- `INSERT` – umieszczenie danych w bazie,
- `UPDATE` – zmiana danych,
- `DELETE` – usunięcie danych z bazy.

Dane tekstowe muszą być zawsze ujęte w znaki pojedynczego cudzysłowu (').

4.5.4.02. DDL (*Data Definition Language*) umożliwia operowanie na strukturach, w których dane są przechowywane – czyli np. dodawać, zmieniać i kasować tabele lub bazy. Najważniejsze polecenia tej grupy to:

- `CREATE` (np. `CREATE TABLE`, `CREATE DATABASE`, ...) – utworzenie struktury (bazy, tabeli, indeksu itp.),
- `DROP` (np. `DROP TABLE`, `DROP DATABASE`, ...) – usunięcie struktury,
- `ALTER` (np. `ALTER TABLE ADD COLUMN` ...) – zmiana struktury (dodanie kolumny do tabeli, zmiana typu danych w kolumnie tabeli).

4.5.4.03. DCL (*Data Control Language*) ma zastosowanie do nadawania uprawnień do obiektów bazodanowych. Najważniejsze polecenia w tej grupie to:

- `GRANT` - służące do nadawania uprawnień do pojedynczych obiektów lub globalnie konkretnemu użytkownikowi (np. `GRANT ALL PRIVILEGES ON EMPLOYEE TO PIOTR`

WITH GRANT OPTION – przyznanie wszystkich praw do tabeli EMPLOYEE użytkownikowi PIOTR z opcją pozwalającą mu nadawać prawa do tej tabeli).

- REVOKE – służące do odbierania wskazanych uprawnień konkretnemu użytkownikowi (np. REVOKE ALL PRIVILEGES ON EMPLOYEE FROM PIOTR - odebranie użytkownikowi wszystkich praw do tabeli EMPLOYEE).
- DENY.

4.5.4.04. DQL (*Data Query Language*) to język formułowania zapytań do bazy danych. W zakres tego języka wchodzi jedno polecenie - SELECT. Często SELECT traktuje się jako część języka DML, ale to podejście nie wydaje się właściwe, ponieważ DML z definicji służy do manipulowania danymi - ich tworzenia, usuwania i uaktualniania.

Na pograniczu obu języków DQL i DML- znajduje się polecenie SELECT INTO, które dodatkowo modyfikuje (przepisuje, tworzy) dane.

Piśmiennictwo: *Gruber M.* G.4.1., *Jakubowski A.* J.2.1.

4.5.5. Przykłady typów danych

Wszystkie kolumny tabel w bazie danych SQL muszą mieć określony typ danych. SQL operuje siedmioma podstawowymi typami danych, jednak większość z nich ma po kilka różnie nazywanych odmian. Kategorie te są następujące:

4.5.5.01. CHARACTER STRING – Wszystkie typy danych służące do reprezentacji tekstu, bez względu na zestaw znaków (np. CHARACTER, CHAR, VARCHAR).

4.5.5.02. NATIONAL CHARACTER – Oznacza w praktyce to samo, co CHARACTER STRING, ale zestaw znaków jest tu określony przez implementację i odpowiada najczęściej stosowanemu w bazie językowi narodowemu.

4.5.5.03. BIT STRING – Proste dane binarne (np. BIT, BITVARYING, BLOB).

4.5.5.04. EXACT NUMERIC – Liczby reprezentujące dane dokładne (np. NUMERIC, DECIMAL, INTEGER, INT, SMALLINT).

4.5.5.05. APROXIMATE NUMERIC – Liczby reprezentujące wartości przybliżone wyrażone w postaci notacji mantysa i wykładnik (np. FLOAT, REAL, DOUBLE PRECISION).

4.5.5.06. DATETIME – Daty, godziny oraz połączenie dat i godzin (np. DATE, TIME, TIMESTAMP).

4.5.5.07. INTERVAL – Liczba jednostek czasu pomiędzy dwoma datami lub godzinami.

Piśmiennictwo: *Gruber M.* G.4.1., *Jakubowski A.* J.2.1.

4.5.6. Przykładowe instrukcje języka

Opis każdej instrukcji standardu języka zawiera następujące informacje: przeznaczenie instrukcji, składnię, reguły stosowania, wymagania na różnych poziomach zgodności i odwołania do tematów związanych. Podane poniżej przykładowe instrukcje pozwalają na zorientowanie się w charakterze języka SQL.

4.5.6.01. `ALTER DOMAIN` – instrukcja pozwala zmienić definicję domeny, a w szczególności – dodać lub usunąć wartość domyślną lub ograniczenia.

4.5.6.02. `ALTER TABLE` – instrukcja zmienia definicję istniejącej tabeli bazy danych.

4.5.6.11. `COMMIT WORK` – instrukcja zatwierdza transakcję i trwale zapisuje wprowadzone zmiany do tabel bazy danych.

4.5.6.21. `CREATE CHARACTER SET` – instrukcja definiuje zestaw znaków używany w danej bazie danych.

4.5.6.22. `CREATE COLLATION` – instrukcja definiuje uporządkowanie (kolejność wystąpienia) znaków.

4.5.6.23. `CREATE DOMAIN` – instrukcja definiuje domenę, czyli dopuszczalny zbiór wartości wskazanego atrybutu (zmiennej).

4.5.6.24. `CREATE SCHEMA` – instrukcja definiuje (nazywa) schemat bazy danych.

4.5.6.25. `CREATE TABLE` – instrukcja tworzy stałą lub tymczasową tabelę (relacje) bazy danych, czyli określa nazwę i strukturę tabeli oraz właściwości wraz z kontrolą dostępu.

4.5.6.26. `CREATE VIEW` – instrukcja definiuje widok, czyli wtórną tabelę bazy danych, której zawartość powstaje na podstawie zawartości tabel bazy danych. Widoki są np. stosowane w celu ułatwienia kontrolowania sposobów, w jaki użytkownicy korzystają z bazy danych. Przyznanie uprawnień dotyczących widoków w miejsce uprawnień dotyczących tabel pozwala na ukrycie różnych wierszy i kolumn tabel.

4.5.6.31. `DELETE` – instrukcja usuwa wskazane wiersze z określonej tabeli.

4.5.6.41. `DROP CHARAKTER SET` – instrukcja usuwa definicję zestawu znaków.

4.5.6.42. `DROP COLLATION` – instrukcja usuwa uporządkowanie znaków.

4.5.6.43. `DROP DOMAIN` – instrukcja usuwa domenę.

4.5.6.44. `DROP SCHEMA` – instrukcja usuwa schemat bazy danych.

4.5.6.45. `DROP TABLE` – instrukcja usuwa tabelę z bazy danych.

4.5.6.46. `DROP VIEW` – instrukcja usuwa widok z bazy danych.

4.5.6.51. `GRANT` – instrukcja nadaje użytkownikowi uprawnienia do korzystania z określonych części składowych i określonej funkcjonalności bazy danych.

4.5.6.61. `INSERT` – instrukcja wstawia wskazane wiersze do określonej tabeli.

4.5.6.71. `REVOK` – instrukcja usuwa uprawnienia do korzystania z określonych części składowych i określonej funkcjonalności bazy danych.

4.5.6.81. `SELECT` – instrukcja pobiera wiersze z jednej lub wielu tabel bazy danych. Przy formułowaniu zapytań – instrukcja wykorzystuje szereg klauzul. Kolejność przetwarzania

poszczególnych klauzul instrukcji SELECT jest następująca: (1) FROM, (2) WHERE, (3) GROUP BY, (4) HAVING, (5) UNION lub EXCEPT, (6) INTERSECT, (7) ORDER BY, (8) INTO.

4.5.6.91. SET CONNECTION – instrukcja określa aktywne połączenia, np. dołącza tabelę do schematu bazy danych.

4.5.6.92. SET TRANSACTION – instrukcja definiuje transakcję i ustala kolejność wykonywania czynności składowych.

4.5.6.90. UPDATE – instrukcja zmienia zawartość wskazanego wiersza określonej tabeli.

Piśmiennictwo: *Gruber M.* G.4.1., *Jakubowski A.* J.2.1.

4.5.7. Przykładowe elementy języka

Z kolei zajmiemy się elementami języka SQL, które dotyczą, na ogół, co najmniej kilku instrukcji (przykłady instrukcji podane zostały w 4.5.6.).

4.5.7.10. Funkcje agregujące wyznaczają z różnych wierszy tabeli: AVG – średnią wartość; SUM – sumaryczną wartość; MAX – wartość maksymalną; MIN – wartość minimalną; COUNT – liczbę zliczonych pozycji.

4.5.7.20. Funkcje liczbowe operują na nie-liczbowych typach danych, a zwracają wartości liczbowe argumentów: POSITTION – operuje na parze łańcuchów znaków; EXTRACT – wydziela określone pola wartości typu DATETIME lub INTERVAL; CHARACTER_LENGHT – operuje na pojedynczym łańcuchu znaków, a zwraca liczbę znaków (liczbą znaków pola NULL, jest 0); OCTET_LENGHT – operuje na pojedynczym łańcuchu bitów, traktując jako pozycję łańcucha kolejne trójki bitów; BIT_LENGHT – operuje na pojedynczym łańcuchu bitów, traktując jako pozycję łańcucha kolejne bity.

4.5.7.30. Funkcje łańcuchowe: SUBSTRING – wydziela podciąg z łańcucha znaków lub bitów, pierwszą pozycję podciągu określa klauzula FROM, wybiera tyle kolejnych elementów ile podano w klauzuli FOR; UPPER/LOWER – przekształca łańcuch znaków na, odpowiednio, na wielkie lub małe litery; CONVERT – zmienia format zapisu łańcucha znaków; TRIM – usuwa wszystkie wystąpienia na początku i/lub końcu łańcucha znaków po klauzuli FROM, w zależności od słowa sterującego LEADING (początkowe), TRAILING (końcowe), BOTH (początkowe i końcowe).

4.5.7.40. Funkcje typu data/godzina: CURRENT_DATE – data podawana przez zegar systemu; CURRENT_TIME – godzina podawana przez zegar systemu; CURRENT_TIMESTAMP – połączona data i godzina podawana przez zegar systemu.

4.5.7.50. Konstruktory wartości wierszy i tabel: wartości NULL i DEFAULT – dopuszczalne są jedynie wówczas, kiedy konstruktor wartości wiersza zawarty jest w zapytaniu generującym wiersze dla instrukcji INSERT.

4.5.7.60. Ograniczenia nakładające wymagania dotyczące danych umieszczanych w tabelach: CONSTRINT – nazwa ograniczenia; PRIMARY KEY – klucz podstawowy tabeli danych, zawsze NOT NULL; FOREIN KEY – klucz obcy, odwołujący się do PRIMARY KEY innej tabeli bazy danych; CHECK – określa predykat odnoszący się do jednej lub większej liczby wartości z jednej lub kilku tabel bazy danych.

4.5.7.70. Predykaty dzielą się na trzy grupy:

- predykaty trójwartościowe (np. NOT TRUE - przypisuje wartość – FALSE; NOT FALSE - przypisuje wartość – TRUE; NOTUNKNOW - przypisuje wartość – UNKNOWN);
- predykaty porównania (predykat BETWEEN – czyli pomiędzy dwoma wartościami; predykat IN – czyli należy do zbioru wartości; predykat LIKE – umożliwia porównanie pary łańcuchów znaków, w łańcuchu wzorcu można korzystać z dwóch znaków wieloznacznych
- „ _” oraz „%”. Podkreślenie pozwala zastąpić dowolny znak innym, zaś procent pozwala zastąpić dowolną sekwencję znaków inną podaną.);
- predykaty kwantyfikowane (ANY, ALL, SOME, EXISTS, UNIQUE i MATCHOVERLAPS).

4.5.7.80. Wyrażenia CASE – opiera się na jednej lub więcej par klauzul WHEN i THEN uzupełnionych opcjonalną klauzulą ELSE.

4.5.7.90. Wyrażenia CAST – umożliwia konwersje zmacanej wartości na inny typ danych lub dane z innej domeny.

Piśmiennictwo: Gruber M. G.4.1., Jakubowski A. J.2.1.

4.5.8. Przykłady wyrażeń napisanych w SQL

Przykłady użycia kilku wybranych rodzajów wyrażeń, w tym zapytań:

4.5.8.01. Przykład użycia instrukcji:

```
SELECT *
    FROM pracownicy
    WHERE pensja > 2000
    ORDER BY staz DESC;
```

Zwraca tabelę (listę) utworzoną ze wszystkich kolumn (*) tabeli „pracownicy” (FROM pracownicy) zawierającą pracowników, których pensja jest większa niż 2000 (WHERE pensja > 2000) i sortuje wynik malejąco według parametru staz (ORDER BY staz DESC).

4.5.8.02. Przykład użycia instrukcji:

```
INSERT INTO pracownicy
    (imie, nazwisko, pensja, staz)
VALUES
    ('Jan', 'Kowalski', 5500, 1);
```

Dodaje do tabeli „pracownicy” (INTO pracownicy) wiersz (rekord) zawierający dane pojedynczego pracownika.

4.5.8.03. Przykład użycia instrukcji:

```
SELECT imie, nazwisko, staz
    FROM pracownicy
    ORDER BY nazwisko, imie ASC;
```

Zwraca tabelę (listę) utworzoną z kolumn imie, nazwisko, staz tabeli „pracownicy” (FROM pracownicy) zawierającą wszystkich pracowników i sortuje wynik alfabetycznie rosnąco (ORDER BY nazwisko, imie ASC).

4.5.8.04. Przykład użycia instrukcji:

```
UPDATE pracownicy
    SET pensja = pensja * 1.1
    WHERE staz > 2;
```

Podwyższa o 10% pensję (SET pensja = pensja * 1.1) pracownikom, których staż jest większy niż 2 (np. lata).

4.5.8.05. Przykład użycia instrukcji:

```
DELETE FROM pracownicy
    WHERE imie = 'Jan' AND nazwisko = 'Kowalski';
```

Usuwa z tabeli „pracownicy" wszystkie wiersze (rekordy) dotyczące pracownika o imieniu „Jan" i nazwisku „Kowalski" (czyli takie, w których pole "imię" ma wartość Jan, a pole "nazwisko" – Kowalski).

4.5.8.06. Przykład użycia instrukcji:

```
CREATE TABLE pracownicy
(
    ident CHAR(4) NOT NULL,
    imie VARCHAR(255) NOT NULL,
    nazwisko VARCHAR(255) NOT NULL,
    pensja FLOAT NOT NULL,
    staz INT NOT NULL,
    nr_miejsca SMALLINT
PRIMARY KEY ident);
```

Tworzy tabelę „pracownicy" zawierającą: unikalny identyfikator czteroznakowy (ident CHAR(4)), pola tekstowe zmiennej długości (varchar) o nazwach „imie" (imię) i „nazwisko", o maksymalnej długości 255 znaków, zapisaną za pomocą liczby rzeczywistej (float lub floating point), pensję zapisany za pomocą liczby całkowitej (int lub integer) „staz"staż oraz „nr_miejsca" numer miejsca pracy za pomocą krótkiej liczby całkowitej (smallint). Wszystkie pola, za wyjątkiem nr_miejsce, należy wypełnić tworząc pozycję – wiersz tabeli dla nowego pracownika.

4.5.8.07. Przykład użycia instrukcji:

```
CREATE TABLE miejsca
(   nr_miejsca SMALLINT NOT NULL,
    adres VARCHAR(255) NOT NULL,
PRIMARY KEY nr_miejsca);
```

Tworzy tabelę „miejsca" zawierającą: unikalny identyfikator „nr_miejsca" typu (smallint) , pole tekstowe zmiennej długości (varchar) o nazwie „adres" i maksymalnej długości 255 znaków. Wszystkie pola, należy wypełnić tworząc pozycję dla miejsca.

4.5.8.08. Przykład użycia instrukcji:

```
SELECT pracownicy.nazwisko, pracownicy.nr_miejsca,
miejsca.nr_miejsca, miejsca.adres
FROM pracownicy, miejsca
WHERE pracownicy.nr_miejsca = miejsca.nr_miejsca
ORDER BY pracownicy.nazwisko ASC;
```

Tworzy zestawienie – listę uporządkowaną alfabetycznie, zawierającą nazwiska i adres miejsca pracy zatrudnionego.

4.5.8.09. Przykład użycia instrukcji:

```
SELECT SUM(pensja) AS pensje, AVG(pensja) AS srednia,
MIN(pensja) minimalna, MAX(pensja) AS maksymalna, COUNT(*) liczba
FROM pracownicy;
```

Zastosowanie funkcji kolumnowych umożliwia obliczenie dla pracowników zarejestrowanych w tabeli pracownicy następujących informacji: sumę płac „pensje", średnią płacę „srednia",

minimalną płacę „minimalna”, maksymalną płacę „maksymalna” oraz liczbę pracowników zarejestrowanych „liczba”.

4.5.8.10. Przykład użycia instrukcji:

```
DROP TABLE pracownicy;
```

Usuwa z bazy tabelę „pracownicy”.

4.5.8.11. Przykład użycia instrukcji:

```
ALTER TABLE pracownicy
    ADD dzial VARCHAR(255);
```

Dodaje do struktury tabeli „pracownicy” kolumnę „dzial” (dział), jako pole tekstowe o długości maks. 255 bajtów.

Piśmiennictwo: *Gruber M.* G.4.1., *Jakubowski A.* J.2.1.

4.5.9. Bezpieczeństwo

Jako że SQL jest językiem interpretowanym, istnieje możliwość nadużyć w przypadku konstruowania zapytań z wykorzystaniem parametrów pochodzących z zewnątrz aplikacji. Szczególnie podatne na ten typ ataku są tworzone dynamicznie w oparciu o SQL-ową bazę danych serwisy internetowe. Jeśli twórca aplikacji nie zadba o sprawdzenie poprawności (tzw. walidację) danych wejściowych stanowiących część zapytania, atakujący może być w stanie dopisać do zapytania („wstrzyknąć”) dodatkowe komendy lub zmienić ich sposób działania.

4.5.9.01. **Wyjaśnienie**. *SQL injection* (dosłownie *zastrzyk SQL*) – luka w zabezpieczeniach aplikacji internetowych polegająca na nieodpowiednim filtrowaniu lub niedostatecznym typowaniu i późniejszym wykonaniu danych przesyłanych w postaci zapytań SQL do bazy danych.

4.5.9.02. **Wyjaśnienie**. Zabezpieczanie na poziomie aplikacji. Podstawowym sposobem zabezpieczania przed *SQL injection* jest niedopuszczenie do nieuprawnionej zmiany wykonywanego zapytania.

4.5.9.03. **Wyjaśnienie**. Bezpieczniejszą techniką, niż wyżej wymienione odpowiednie przygotowanie parametrów jest użycie mechanizmu tzw. „zaślepek”, gdzie zmienne nie są używane bezpośrednio do tworzenia zapytania, a odpowiednie dane dołączane są do zapytania w momencie jego wykonania (czy to poprzez wykorzystanie API danego silnika, czy też w ramach warstwy aplikacji poprzez odpowiednie zacytowanie wszystkich parametrów).

4.5.9.04. **Wyjaśnienie**. Techniką utrudniającą wykorzystanie istniejących luk jest zastosowanie paradygmatu *security through obscurity* poprzez wyłączenie wyświetlania komunikatów o błędach. Nie uniemożliwi to ataku, lecz może spowodować, że trudniejsze będzie wykrycie i późniejsze wykorzystanie luki.

4.5.9.05. **Wyjaśnienie**. Zabezpieczenie na poziomie bazy danych. Istnieją również metody zabezpieczenia przed skutkami wykonania błędnych zapytań, które mimo wszystko dostaną się do bazy. Udostępnienie użytkownikowi bazy tylko niezbędnych uprawnień nie da całkowitej ochrony, jednak pozwoli na minimalizację szkód. Przykładowo – niewiele aplikacji potrzebuje uprawnienia do kasowania tabel z bazy danych. Podobny efekt – czyli zmniejszenie możliwości

atakującego – uzyskać można wyłączając niepotrzebną funkcjonalność na poziomie samego silnika.

4.5.9.06. **Wyjaśnienie**. Pomocą mogą służyć również *procedury składowane*, dzięki którym zapytanie budowane jest po stronie bazy danych i aplikacja nie ma bezpośredniego wpływu na jego postać. Chociaż i w tym przypadku skonstruowanie ataku nie jest niemożliwe.

4.5.9.07. **Wyjaśnienie**. Eliminację możliwości wstrzyknięcia kodu SQL, można uzyskać poprzez całkowite wyłączenie możliwości podawania parametrów, jako części zapytania. Jest to działanie analogiczne do mechanizmu zaślepek po stronie aplikacji, jednak tym razem zastosowane w samym silniku. Nie jest to jednak ogólnie dostępna cecha systemów zarządzania bazami.

4.5.9.08. **Wyjaśnienie**. Zabezpieczanie na poziomie serwera aplikacji/www. Jest możliwość instalacji dodatkowych modułów do serwera warstwy aplikacyjnej (np. *mod_security* do serwera Apache) filtrujących wg zdefiniowanych reguł przychodzące żądania i blokujących te potencjalnie groźne. Reguły pozwalają na wychwycenie typowych uniwersalnych ataków lub znanych luk w popularnych aplikacjach. Wadą rozwiązania jest możliwość zablokowania także pożądanych wywołań (np. w aplikacji do zarządzania bazą danych).

Piśmiennictwo: *Gruber M.* G.4.1., *Jakubowski A.* J.2.1.

4.6. STRUKTURY DANYCH - RELACYJNE A HIERARCHICZNE

4.6.0. UWAGI WSTĘPNE

Model relacyjny był dominującym podejściem do przechowywania i przetwarzania danych przez ponad 30 laty. Starsze systemy, które organizowały dane w sposób hierarchiczny, takie jak IMS z IBM, zostały odrzucone na rzecz bardziej elastycznego podejścia związanego z wstawianiem danych do tabeli. Dzisiaj dominują relacyjne bazy danych. Jednak idea hierarchicznych baz danych ciągle powraca. W latach dziewięćdziesiątych ubiegłego wieku - osoby związane z obiektowymi bazami danych próbowały zainteresować szerzej użytkowników hierarchicznymi bazami danych, rozumując, że skoro obiekty w oprogramowaniu były często organizowane w drzewa, to organizowanie danych w ten sam sposób powinno wiązać się z lepszą wydajnością i lepszym dopasowaniem pomiędzy kodem a danymi. Takie spojrzenie było interesujące, jednak ponieważ informacje w większości baz danych są współdzielone przez więcej aplikacji, żadna pojedyncza hierarchia nie będzie działała sprawnie dla wszystkich użytkowników. Bardziej ogólne niż hierarchiczne - podejście relacyjne w większości przypadków było nadal lepsze, ze względu na swoją uniwersalność.

XML jest najnowszą technologią, która wykorzystuje przydatność danych hierarchicznych. W przeciwieństwie do specjalistów od obiektowych baz danych, orędownicy XML nie starali się wyprzeć ugruntowanego podejścia relacyjnego. Zamiast tego podjęli temat wykorzystania relacyjnych baz danych do składowania danych o strukturze drzewiastej, co w konsekwencji okazało się lepszą strategią. Prawda jest taka, ze niektóre rodzaje informacji, są naprawdę modelowane lepiej w postaci drzewa niż tabeli. A biorąc pod uwagę, niesamowitą przydatność XML jako wspólnego formatu wymiany pomiędzy aplikacjami, oprogramowanie, które będzie obsługiwało dokumenty XML, jest niewątpliwie potrzebne. XML dobrze zadomowiło się w

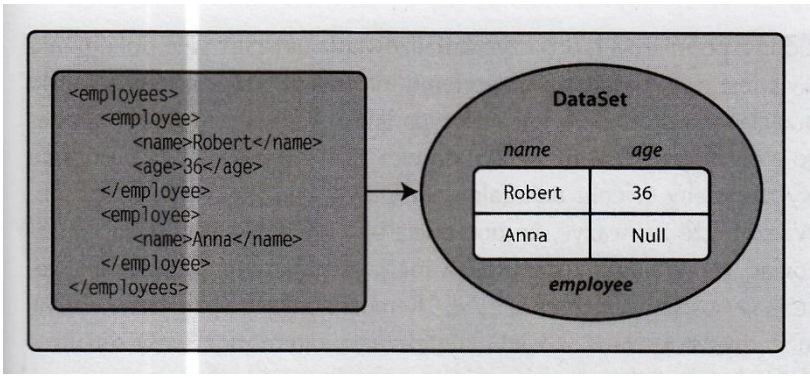
świecie oprogramowania, połączenie hierarchicznej struktury, którą wykorzystuje XML, z dostępem do danych relacyjnych jest bardzo przydatnym pomysłem. Microsoft uwierzył w XML kilkanaście lat temu, definiując funkcjonalność jednego ze swoich flagowych produktów MS Office w XML'u. Jednym z efektów wiary w uniwersalność XML, było połączenie w Redmond grup odpowiedzialnych za dostęp do danych relacyjnych z grupami odpowiedzialnymi za dostęp do danych XML. Rezultatem tej zmiany był, między innymi ADO.NET, a także korzystanie z przestrzeni nazw System.Xml. Systemy operacyjne Windows posiadają API strumieniowe - odpowiednio DataReader i XmlReader - oraz APl nawigacyjne powiązane z DataSet, a także Xml.Document. Co ważniejsze, integracja tych dwóch modeli w .NET Framework, pozwala na dostęp do tych samych danych - albo w formie tabeli relacyjnej, albo jako hierarchii XML.

Piśmiennictwo: *Chappel D*. C.1.1.

4.6.1. ODWZOROWANIA STRUKTUR

Pojawia się więc - oczywisty problem. Zarówno XML, jak i model relacyjny posiadają koncepcję schematu opisywania danych. Jednak dwie technologie postrzegają ową koncepcję zupełnie inaczej. XML definiuje hierarchię z wykorzystaniem języka XSD (XML Schema Definition). Model relacyjny definiuje schemat jako zbiór tabel, z których każda posiada daną grupę kolumn w określonym porządku. Obiekty DataSet i DataTable, znajdują się zdecydowanie w obozie relacyjnym. Jak powinien wyglądać schemat tabeli, kiedy dokument XML jest ładowany do DataSet? Jak w przypadku większości pytań, odpowiedź może być tylko jedna: Metoda DataSet.ReadXml pozwala na przekazywanie parametru Xml ReadMode decydującego, o sposobie w jaki schemat dokumentu XML jest odwzorowywany na schemat dla DataTable, w którym będą następnie przechowywane informacje z tego dokumentu. Jedną z opcji jest nakazanie DataSet, by odczytywał schemat XSD, który pokazuję się w czytanym dokumencie XML, podczas gdy inna opcja instruuje DataSet, by wywnioskował ze schematu XML, jaki będzie schemat relacyjny.

Odwzorowywanie hierarchicznej formy dokumentu XML w DataSet jest niebanalnym przedsięwzięciem. W ogólności reguły służące do analizowania relacyjnej struktury i przetwarzanie jej na schemat - nie są zbyt proste. Niektóre przypadki są jednak łatwe. Odwzorowanie hierarchii na tabele jest jednak bardziej interesujące. Kiedy tworzy się schemat relacyjny na podstawie schematu XML, DataSet zakłada, że element XML z atrybutami - powinien stać się tabelą, w której atrybuty i wartości elementów są reprezentowane jako kolumny tej tabeli. Element z elementami dzieci także stanie się tabelą zawierającą jakieś dane.



Rysunek 4.6.1.00. Odwzorowanie struktury hierarchicznej dokumentu XML w strukturę relacyjną.

Na rysunku 4.6.1.00. pokazano, w jaki sposób prosty dokument XML, który prezentujemy, mógłby zostać odwzorowany na DataSet, gdyby został odczytany za pomocą DataSet.Read Xml. Elementy

<employee> są umieszczane w pojedynczym obiekcie DataTable o nazwie employee, a informacje z każdego z elementów dzieci są odwzorowane na wiersze tej tabeli. Warto zauważyć, że ponieważ nie ma podanego wieku (age) dla Anny, ta wartość jest typu null (a konkretnie DBNull, zdefiniowanego w przestrzeni nazw System).

Language INtegrated Query (LINQ) – część technologii Microsoft.NET (firmy Microsoft Corporation), która została opracowana przez *Andersa Hejlsberga* – znanego z zaprojektowania języka Delphi i języka C#. Technologia LINQ umożliwia zadawanie pytań na obiektach. Składnia języka LINQ jest prosta i przypomina SQL.

Piśmiennictwo: *Chappel D*. C.1.1.

4.6.2. JAK DZIAŁA LINQ?

LINQ stanowi warstwę abstrakcji nad różnymi źródłami danych. Baza danych i jej elementy traktowane są również jak obiekty. Przestrzenie, które obsługuje LINQ, to:

1. obiekty implementujące interfejs IEnumerable <T>
2. bazy danych
3. język XML

LINQ posiada pełne wsparcie dla transakcji, widoków oraz procedur przechowanych. Zapytania stają się częścią języka .NET wspierającego .NET 3.5 (C#, Visual Basic, Delphi Prism itd.). Jeśli LINQ ma działać, musi znać mapę całej bazy danych. Zapytanie LINQ zwraca kolekcję z przestrzeni nazw typów ogólnych. Kolekcja ta może być modyfikowana, a następnie zwrócona do źródła. Dzięki temu zachowywana jest pełna kontrola typów danych i ich konwersji w poszczególnych mechanizmach pośredniczących w pobieraniu danych.

Standardowe operatory zapytań LINQ:

1. `Select` – operator ten jest używany, by wybrać z kolekcji odpowiedniego rodzaju dane (zbiory/podzbiory danego obiektu)
2. `SelectMany` – jest używany, by dokonać wyświetlenia w przypadku relacji jeden-do-wielu, czyli jeśli obiekt w kolekcji zawiera inną kolekcję jako część danych, SelectMany będzie użyte do wybrania całej pod-kolekcji.
3. `Where` – operator Where pozwala zdefiniować zbiór zasad dla każdego obiektu w kolekcji. Wszystkie te obiekty, które nie pasują do wybranej reguły, są odfiltrowywane.
4. `Sum` – używany do otrzymywania sumy
5. `Min` – używany do otrzymania minimalnej wartości
6. `Max` – używany do otrzymania maksymalnej wartości
7. `Average` – używany do otrzymania średniej
8. `Aggregate` – używany do stworzenia wyrażenia agregującego wszystkie elementy w kolekcji
9. `Join` – operator Join to operator bazujący na dwóch kolekcjach i tworzy z nich obiekt wynikowy.
10. `GroupJoin` – operator dla dokonania połączenia grupowego. Tak jak w przypadku operatora Select, wynik łączenia jest instancją nowej klasy z wszystkimi członkami obiektów źródłowych lub ich podzbiorem.
11. `Take` – operator Take jest używany do wybrania pierwszych n obiektów z kolekcji

12. `TakeWhile`
13. `Skip`
14. `SkipWhile`
15. `OfType` – operator używany do wybrania elementów konkretnego typu
16. `Concat` – operator tworzący konkatenacje dwóch kolekcji
17. `OrderBy` – sortuje wyniki po wybranym elemencie kolekcji – według określonego klucza
18. `OrderByDescending` – sortuje w odwrotnej kolejności niż standardowo
19. `ThenBy` – określa, po czym sortować w następnym etapie
20. `ThenByDescending`
21. `Reverse` – odwraca kolekcje
22. `GroupBy`
23. `Distinct` – usuwa duplikacje wartości kluczy w kolekcji
24. `Union` – operacja 'suma' – zwraca połączone ze sobą dwie kolekcje
25. `Intersect` – operacja 'iloczyn' – zwraca część wspólną dwóch kolekcji
26. `Except`
27. `EqualAll` – sprawdza, czy wszystkie elementy w kolekcji są takie same
28. `First` – zwraca pierwszy element kolekcji
29. `FirstOrDefault` – zwraca pierwszy element kolekcji lub NULL, gdy kolekcja jest pusta
30. `Last` – zwraca ostatni element kolekcji
31. `LastOrDefault` – zwraca ostatni element kolekcji lub NULL, gdy kolekcja jest pusta
32. `Single`
33. `ElementAt` – zwraca element o wybranym indeksie (index) z kolekcji
34. `Any` – sprawdza, czy którykolwiek element kolekcji spełnia warunek podany w nawiasach
35. `All` – sprawdza, czy wszystkie elementy kolekcji spełniają warunek podany w nawiasach
36. `Contains` – sprawdza, czy kolekcja zawiera element podany w argumencie
37. Count – zlicza elementy kolekcji.

Piśmiennictwo: *Chappel D*. C.1.1.

4.6.3. Przykład zapytania LINQ poza bazami danych

Przykład wybrania w zapytaniu wszystkich obiektów z właściwością SomeProperty mniejszą niż 10:

```
int someValue = 5;
var results =  from c in someCollection
               let x = someValue * 2
               where c.SomeProperty < x
               select new {c.SomeProperty, c.OtherProperty};

foreach (var result in results)
{
        Console.WriteLine(result);
}
```

Ostatecznie kompilator wygeneruje klasę:

```
IEnumerable<SomeOtherClass> results =
     SomeCollection.Where
     (
          c => c.SomeProperty < (SomeValue * 2)
     )
```

```
     .Select
     (
          c => new {c.SomeProperty, c.OtherProperty}
     );
foreach (SomeOtherClass result in results)
{
     Console.WriteLine(result.ToString());
}
```

Piśmiennictwo: *Chappel D*. C.1.1.

4.7. PODEJŚCIE OBIEKTOWE DO SYSTEMÓW INFORMATYCZNYCH

4.7.0. UWAGI WSTĘPNE

Modelowanie działalności biznesowych (ogólniej mówiąc, modelowanie systemów) oparte o podejście obiektowe odgrywa dzisiaj istotną rolę we współczesnej inżynierii biznesu, projektowaniu systemów informatycznych wspomagania zarządzania, itp. Historia opracowania tego podejścia jest w rzeczywistości bardzo krótka. Wczesne modele, pochodzące między innymi, od Jay W. Forrester'a (*Industrial Dynamics – a major breakthrough for decision Makers*, Harvard Business Review, Vol. 36 No 4 pp 37 – 66.), były budowane, z tzw. skrzynek funkcjonalnych (*boxes*) oraz powiązań między nimi. Stosowane były dwie kategorie skrzynek funkcjonalnych, zwane odpowiednio: (1) czarnymi skrzynkami (*black boxes*) oraz (2) białymi skrzynkami (*white box*). Działanie funkcjonalności czarnych skrzynek było znane, natomiast ich wewnętrzna struktura była niezdefiniowana. Nazwa białe skrzynki dotyczyła przypadku, w którym znane jest zarówno działanie funkcjonalności jak i formalny opis struktury wewnętrzne.

W 1956 roku Henryk Greniewski[74] (*Logique et cybernetique*, *Actes du 1-er Congres International de Cybernetique, Namur 1956*.) przedstawił na Pierwszym Międzynarodowym Kongresie Cybernetyki – oryginalną koncepcję nazwaną, przez autora, „*systemem względnie odosobnionym*". Słabą stroną tej koncepcji, był brak sformalizowanego opisu takiego systemu. Autor ograniczył się do opisu słownego oraz graficznej prezentacji.

Mniej więcej w tym samym czasie, w Danii opracowany został język programowania ALGOL 60, będący w rzeczywistości, znacznie doskonalszym w sensie logiki, konkurentem opracowanego w USA języka programowania FORTRAN 2. Warto zauważyć, że po latach konkurencji, oba wyżej wymienione języki programowania, połączyły się we wspólny twór, nazwany FORTRANEM 96. W naszych rozważaniach istotniejsze jest, opracowanie czegoś w rodzaju makro-wyrażeń dla ALGOL 60 w Norwegii, które nazwane zostało językiem programowania symulacji SYMULA 67[1] (authorami języka Simula 67[75]- extension of Algol 60, first simulation language proposing concepts of primitive classes and objects: Ole-Johan Dahl & Kristen Nygaard of Norwegian Computing Center) . Podstawowym pojęciem języka SIMULA 67 jest obiekt. Pojęcie obiektu wprowadzona odwołując się do charakteru dynamicznie tworzonych jednostek programu napisanych w języku ALGOK 60. Każda dynamicznie tworzona jednostka programu, czyli obiekt - ma własną lokalną strukturę danych jak i listę zadań (funkcji) do wykonania. Dodatkowo w języku ALGOL 60 istnieją mechanizmy pozwalające na przerwanie wykonywania

[74] Henryk Greniewski, *Logique et cybernetique*, Actes du 1-er Congres International de Cybernetique, Namur 1956.
[75] Ole-Johan Dahl and Kristen Nygaard, *Class and subclass declarations*. In Proceedings from IFIP TC2 Conference on Simulation Programming Languages, Lysebu, Oslo, ed.: J. N. Buxton, pages 158-174. North Holland, May 1967.

poszczególnych zadań obiektu oraz ponowne ich wznawianie. W ten sposób powstały podstawy pod sformalizowany opis systemu względnie odosobnionego, który jest obiektem w rozumieniu pojęcia wprowadzonego przez twórców SYMULI 67.

Dalszy rozwój formalizacji systemu względnie odosobnionego wiąże się z dalszym rozwojem obiektowych języków programowania. Pierwszym całkowicie obiektowym językiem programowania był język Smalltalk (*Adele Goldberg & Alan Kay, ed. (March 1976). Smalltalk-72 Instruction Manual. Palo Alto, California: Xerox Palo Alto Research Center*). Obok języków programowania, rozpoczął się rozwój obiektowych języków modelowania systemów biznesowych. W świetle dostępnej literatury, można przyjąć, że idea ta rozwijała się początkowo całkiem niezależnie w Szwecji i USA. Ostatecznie proces ten zaowocował powstaniem kolejnych wersji języka UML (*Grady Booch, Ivar Jacobson & James Rumbaugh (2000) OMG Unified Modeling Language Specification, Version 1.3 First Edition: March 2000)*.

Piśmiennictwo: *Booch* G. B.6.1., *Dahl O.* D.1.1., *D'Souza D*. D.4.1., *Jacobson I.* J.1.1., *Guttag J.* G.5.1.

4.7.1. SYSTEM WZGLĘDNIE ODOSOBNIONY

Sama idea „systemu względnie odosobnionego (zwanego również w polskojęzycznym piśmiennictwie *układem względnie odosobnionym*)", nie jest w gruncie rzeczy czymś nowym. Uczeni europejscy od setek lat posługiwali się nim milcząco w nauce, przynajmniej od czasów *Korpusu Hipokratejskiego*. Potrzeba jasnego i ścisłego wprowadzenia tego pojęcia odbyło się jednak dopiero w ramach nauki i sterowaniu maszyn i istot żywych, nazwanej przez Norberta Winnera w 1947 roku – *Cybernetyką*. Jak już powiedzieliśmy, koncepcja „*systemu względnie odosobnionego*", została po raz pierwszy sformułowana przez Henryka Greniewskiego w referacie wygłoszonym na Pierwszym Światowym Zjeździe Cybernetyki w Namur (Belgia) – w 1956 roku.

4.7.1.00. **Wyjaśnienie**. Nasze rozważania rozpoczniemy od wprowadzenia pojęcia „*systemu bezwzględnie odosobnionego*", przez który rozumiemy układ (nie zajmując się zagadnieniem fikcyjności, czy też nie-fikcyjności takiego systemu) spełniający następujące dwa warunki:
1. Nie pozostaje pod wpływem reszty Wszechświata;
2. Nie wywiera żadnego wpływu na resztę Wszechświata.

4.7.1.01. **Wyjaśnienie**. Mówiąc natomiast „*system względnie odosobniony*" mamy na myśli wszelki tylko taki system, który posiada obie poniższe własności:
1. Reszta Wszechświata działa na nasz system, ale oddziaływanie odbywa się tylko na pewnych, że tak powiemy, „*drogach*" - zwanych „*wejściami systemu*";
2. Nasz system wywiera wpływ na resztę Wszechświata, ale oddziaływanie to realizowane jest tylko na pewnych, że tak powiemy, „*drogach*" – zwanych „*wyjściami systemu*".

Rozpoczęliśmy nasz przegląd aparatury pojęciowej od pojęcia systemu względnie odosobnionego. Pojęcie takiego systemu, jak również pojęcia wejścia i wyjścia takiego systemu, są tworami abstrakcyjnymi. Stosując typowe podejście stosowane w logice formalnej, można uznać je za tzw. *pojęcia pierwotne*, a następnie przyjąć pewien zbiór postulatów nadających sens tym pojęciom.

4.7.1.10. **Postulat**. *Kalendarz, repertuar, trajektoria*. Każdemu *wejściu* i każdemu *wyjściu* danego systemu względnie odosobnionemu przyporządkujemy:
1. *Kalendarz*, tj. pewien - co najmniej dwuelementowy zbiór chwil albo przedziałów czasu;

2. *Repertuar*, tj. pewien zbiór stanów wyróżnionych, przy czym każde wejście i każde wyjście przyjmuje dokładnie jeden stan wyróżniony w obrębie dowolnego elementu swego kalendarza.

Funkcja przyporządkują ca poszczególnym elementom kalendarza danego wejścia (wyjścia) poszczególne stany wyróżnione należące do repertuaru tegoż wejścia (wyjścia) – to *trajektoria* danego wejścia (wyjścia).

4.7.1.11. **Postulat**. *Bodziec i reakcja.* Zamiast mówić „*stan wyróżniony wejścia*" mówimy krótko „*bodziec*", zamiast mówić „*stan wyróżniony wyjścia*" mówimy krótko „*reakcja*".

4.7.1.20. **Wyjaśnienie**. *Cztery warianty.* Pojęcie systemu względnie odosobnionego wprowadzimy w czterech wariantach; rozróżniać będziemy z jednej strony systemy względnie odosobnione: *zawodne* i *niezawodne*, a drugiej strony, niezależnie od dwupodziału powyższego, systemy *prospektywne* i *retrospektywne*.

4.7.1.21. **Wyjaśnienie**. *Prospektywne systemy niezawodne.* Charakterystyka tych systemów sprowadza się do dwu punktów:

1. Repertuar każdego z wejść składa się, z co najmniej dwu stanów wyróżnionych.
2. Aktualny stan wyróżniony dowolnego wejścia jest zawsze jednoznacznie wyznaczany przez aktualne i minione stany wyróżnione wszystkich wejść.

Warunek 2 nazywamy postulatem determinizmu lokalnego.

4.7.1.22. **Wyjaśnienie**. *Prospektywne systemy zawodne*. Charakterystyka tych systemów sprowadza się do dwu punktów:

1. Repertuar każdego z wejść składa się, z co najmniej dwu stanów wyróżnionych.
2. Aktualny stan wyróżniony dowolnego wejścia jest zawsze wyznaczony ze stałym prawdopodobieństwem, większym niż 50%, przez aktualne i minione stany wyróżnione wszystkich wejść.

Warunek 2 nazywamy postulatem pseudo-determinizmu lokalnego.

4.7.1.23. **Uwaga**. Determinizm filozoficzny zdaje się twierdzić, że każdy system prospektywny, jest niezawodny. Praktyka codzienna przemawia zaś, jak by na pierwszy rzut oka zdawało, za tym, że każdy układ prospektywny jest zawodny. Pozorna ta kontrawersja daje się łatwo wyjaśnić na gruncie determinizmu filozoficznego: W praktyce nie znamy wszystkich wejść danego systemu prospektywnego i niezawodnego; tak „okaleczony" przez naszą niewiedzę system niezawodny staje się, rzecz prosta, systemem zawodnym.

4.7.1.24. **Wyjaśnienie**. *Czas reakcji.* Zwykle mamy do czynienia z takimi systemami prospektywnymi (obojętnie: *zawodnymi*, czy *niezawodnymi*), w których opisane wyżej oddziaływania przeszłości i teraźniejszości wejść na teraźniejszość wyjść ograniczone jest w czasie w sposób następujący: każdej parze złożonej z wejścia i wyjścia przypisujemy liczbę nieujemną jednostek czasu zwaną *czasem reakcji* (*time-lag*). Zakładamy, że na teraźniejszość, tj. na aktualny stan wyróżniony dowolnego wejścia, nie działa przeszłość oddalona w czasie więcej niż o *lag* przypisany danej parze złożonej z wejścia i wyjścia.

4.7.1.25. **Wyjaśnienie**. *Retrospektywne systemy niezawodne.* Charakterystyka tych systemów sprowadza się do dwu punktów:

1. Repertuar każdego z wyjść składa się, z co najmniej dwu stanów wyróżnionych.

2. Miniony (dostatecznie odległy od chwili bieżącej) stan wyróżniony dowolnego wejścia jest zawsze jednoznacznie wyznaczony przez aktualne i minione (jednak nie wcześniejsze niż wchodzący w grę stan wejścia) stan wyróżniony wszystkich wejść.

Warunek 2 nazywamy *postulatem pseudo-determinizmu lokalnego*.

4.7.1.26. **Wyjaśnienie**. *Retrospektywne systemy zawodne.* Charakterystyka tych systemów sprowadza się do dwu punktów:

1. Repertuar każdego z wyjść składa się, z co najmniej dwu stanów wyróżnionych.
2. Miniony (dostatecznie odległy od chwili bieżącej) stan wyróżniony dowolnego wejścia jest zawsze wyznaczony z prawdopodobieństwem większym niż 50% przez aktualne i minione (jednak nie wcześniejsze niż wchodzący w grę stan wejścia) stan wyróżniony wszystkich wejść.

Warunek 2 nazywamy *postulatem pseudo-determinizmu lokalnego*.

4.7.1.27. Uwaga. Przez „*determinator*" dowolnego wyjścia danego systemu prospektywnego rozumiemy funkcję, która przyporządkuje *bodźcom - reakcję*.

4.7.1.28. Uwaga. Przez „*para-determinator*" dowolnego wyjścia danego systemu prospektywnego rozumiemy funkcję, która przyporządkuje *reakcjom - bodziec*.

4.7.1.31. **Wyjaśnienie**. *Dwoistość.* Odwróćmy bieg czasu w dowolnym systemie prospektywnym, zastąpmy wejścia - wyjściami i odwrotnie wyjścia – wejściami; a wówczas z systemu prospektywnego otrzymamy system retrospektywny. W systemach prospektywnych zawsze teraźniejszość wyjść jest wyznaczona (logicznie czy tylko probabilistycznie) przez teraźniejsze i przeszłe wejść, natomiast w układach retrospektywnych sprawa przedstawia się odwrotnie – tj. przeszłość wejść jest zawsze wyznaczona przez nie-wcześniejszą przeszłość i teraźniejszość wyjść.

W teorii systemów względnie odosobnionych występuje dwoistość: każdemu twierdzeniu o systemach prospektywnych – odpowiada dokładnie jedno twierdzenie dwoiste, o systemach retrospektywnych; i odwrotnie (wystarczy przy tym podanie dowodu jednego z tych twierdzeń – drugi dowód otrzymuje się automatycznie z pierwszego.

4.7.1.32. **Wyjaśnienie**. *Hipoteza względnego odosobnienia.* Nie wiemy skąd się bierze przekonanie, iż pewien twór może być traktowany, jako *system względnie odosobniony prospektywnym* (*retrospektywnym*), przyjmujemy jednak hipotezę, że tak jest. Doświadczenie ludzkości kształtowane od tysięcy lat mówi, że jest to płodna hipoteza, pozwalająca prowadzić badania empiryczne, eksperymentować, a przy tym posiada zasadniczą metodologiczną zaletę: nie można tej hipotezy wprawdzie w pełni dowieść, ale można ją doświadczalnie obalić.

4.7.1.41. **Definicja**. *Sprzężenie szeregowe systemów względnie odosobnionych.* Weźmy pod uwagę dwa systemy względnie odosobnione, np. oznaczone I oraz II. Załóżmy, że jedno z wyjść systemu I jest połączone z jednym z wejść systemu II. Jest to tzw. bezpośrednie sprzężenie szeregowe pary systemów I oraz II (patrz rys. 4.7.1.40a). Warunkiem dostatecznym dla zbudowania bezpośredniego sprzężenia szeregowe pary systemów względnie odosobnionych, jest następujący: repertuar stanów wyjścia systemu I sprzęganego z wejściem systemu II, musi być podzbiorem repertuaru stanów wejścia systemu II, do które przyłączane jest wzmiankowane wyżej wyjście.

4.7.1.42. **Definicja**. *Sprzężenie zwrotne systemów względnie odosobnionych.* Weźmy pod uwagę dwa systemy względnie odosobnione, np. oznaczone I oraz II. Załóżmy, że istnieją dwa sprzężenia szeregowe pomiędzy tymi systemami:

1. jedno z wyjść systemu I jest połączone z jednym z wejść systemu II, a równocześnie
2. jedno z wyjść systemu II jest połączone z jednym z wejść systemu I.

Jest to tzw. bezpośrednie sprzężenie zwrotne pary systemów I oraz II (patrz rys. 4.7.1.40b). Warunkiem dostatecznym dla zbudowania bezpośredniego sprzężenia zwrotnego pary systemów względnie odosobnionych, jest następujący: repertuar stanów wyjścia systemu I sprzęganego z wejściem systemu II, musi być podzbiorem repertuaru stanów wejścia systemu II, do które przyłączane jest wzmiankowane wyżej wyjście oraz repertuar stanów wyjścia systemu II sprzęganego z wejściem systemu I, musi być podzbiorem repertuaru stanów wejścia systemu I, do które przyłączane jest wzmiankowane wyżej wyjście.

4.7.1.43. **Definicja**. *Sprzężenie równoległe systemów względnie odosobnionych.* Weźmy pod uwagę dwa systemy względnie odosobnione, np. oznaczone I oraz II. Załóżmy, że jedno z wejść systemu I łączymy z jednym wejściem systemu II, jest to bezpośrednie sprzężenie równoległe wejść dwóch systemów względnie odosobnionych (patrz rys. 4.6.1.40c). Warunkiem dostatecznym dla zbudowania bezpośredniego sprzężenia pary systemów względnie odosobnionych jest identyczność repertuaru stanów pary wejść należących odpowiednio do systemów I oraz II.

4.7.1.44. **Definicja**. *Stan wewnętrzny systemu względnie odosobnionego.* Jedną z przydatnych metod wprowadzenia pojęcia stanu wewnętrznego systemu względnie odosobnionego, jest wprowadzenie pojęcia *samo sprzężenia* systemu. Przez samo sprzężenie systemu rozumiemy sprzężenie jednego wyjścia danego systemu z jego jednym z wejść. Warunkiem dostatecznym dla możliwości zbudowania samo sprzężenia systemu, jest następujący: repertuar stanów wyjścia systemu sprzęganego z wejściem systemu, musi być podzbiorem repertuaru stanów wejścia systemu, do które przyłączane jest wzmiankowane wyżej wyjście.

4.7.1.45. **Wyjaśnienie**. Stan wewnętrzny systemu względnie odosobnionego, z reguły wpływa podobnie jak stany jego wejść, na stany wyjść systemu. Z drugiej strony stan wejść systemu i jego aktualny stan wejść ma wpływ na przyszły stan wewnętrzny systemu.

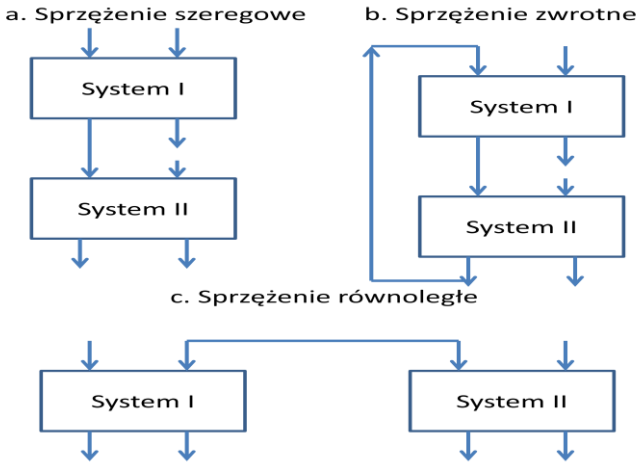
4.7.1.56. **Definicja**. *Złożony system względnie odosobniony.* Przez złożony system względnie odosobniony rozumiemy system, który powstał w wyniku sprzężeń (szeregowych lub zwrotnych lub równoległych) dwóch lub więcej systemów względnie odosobnionych.

4.7.1.61. **Definicja**. *Analiza złożonego systemu względnie odosobnionego.* Analiza istniejącego złożonego systemu względnie odosobnionego (organizmu żywego, maszyny, organizacji), polega na rozróżnieniu w obrębie danego złożonego systemu względnie odosobnionego jego części składowych, będących również systemami względnie odosobnionymi wzajemnie sprzężonych (sprzężeniami: szeregowymi lub zwrotnymi lub równoległymi).

4.7.1.62. **Definicja**. *Synteza złożonego systemu względnie odosobnionego.* Typowa synteza złożonego systemu względnie odosobnionego przebiega w następujących krokach:

1. Opracowanie specyfikacji funkcjonalnej złożonego systemu względnie odosobnionego, tzn. określenie wymagań funkcjonalnych i poza-funkcjonalnych, jakie ma spełniać zsyntezowany złożony system względnie odosobniony.

2. Wybranie pewnego zbioru prostych systemów względnie odosobnionych – elementów konstrukcyjnych syntezowanego złożonego systemu względnie odosobnionego.
3. Konstruowanie (synteza) złożonego systemu względnie odosobnionego zgodnie z przyjętą specyfikacją (patrz pkt. 1) z elementów konstrukcyjnych (patrz pkt.2).
4. Może się okazać, że nie ma możliwości syntezy złożonego systemu względnie odosobnionego zgodnie z przyjętą specyfikacją (patrz pkt. 1) z elementów należących do zbioru elementów konstrukcyjnych (patrz pkt.2). Poszukujemy wówczas kompromisu, polegającego np. na uproszczenia przyjętej specyfikacji lub modyfikacji (np. rozszerzenia) zbioru elementów konstrukcyjnych. Po czym powtarzamy czynności pkt. 3.
5. Może się okazać, że istnieje więcej niż jedno rozwiązanie, spełniające opracowaną specyfikację, wówczas możemy przyjąć kryterium optymalizacji i wybrać ten ze złożonych systemów względnie odosobnionych, który spełnia kryterium optymalizacji.



Rysunek 4.7.1.40. Sprzężenia systemów względnie odosobnionych

4.7.1.62. **Wyjaśnienie**. *Modelowanie.* Wyraz „model"- używany jest, jak powszechnie wiadomo, w wielu różnych rozumieniach. Dlatego uważamy za celowe ustalenie rozumienia, w jaki znaczeniu używać będziemy słowa „model". Załóżmy, że dany jest pewien złożony system względnie odosobniony, który nazwiemy oryginałem. Zwykle, jako oryginał dobieramy system już istniejący. Przez „model" będziemy tu rozumieli system możliwie mało złożony, działający podobnie do oryginału. Natomiast przez „modelowanie" będziemy rozumieli syntezowanie modelu oryginału.

Piśmiennictwo: *Greniewski H.* G.2.2., G.2.3.

4.7.2. OBIEKT - JAKO FORMALIZACJA SYSTEMU WZGLĘDNIE ODOSOBNIONEGO

Jak już powiedzieliśmy, pojęcie obiektu wprowadzono w języku SYMULA 67, odwołując się do charakteru dynamicznie tworzonych jednostek programu napisanych w języku ALGOL 60. Każda dynamicznie tworzona jednostka programu, czyli obiekt - ma własną lokalną strukturę danych jak i listę zadań (funkcji) do wykonania. W rzeczywistości, obok pojęcia obiektu, wprowadzone zostało również pojęcie *klasy obiektów* – rozumianej, jako matryca (wyposażona w odpowiedni mechanizm zwany *konstruktorem obiektów*, krócej *konstruktorem*) do tworzenie danego rodzaju obiektów. Postaramy się z kolei, precyzyjnie zdefiniować pojęcie podejścia obiektowego do systemu informatycznego. W tym celu wprowadzimy 17 poniższych definicji:

4.7.2.01. **Definicja**. *Obiekt* jest systemem względnie odosobnionym. Od otoczenie oddziela obiekt tzw. kapsułkowanie (*an capsulated*) i zawiera w swoim „wnętrzu" *atrybuty* – reprezentujące stan wewnętrzny i *metody* (funkcje wbudowane obiektu), których argumentami i wartościami są atrybuty.

4.7.2.02. **Definicja**. *Interfejs obiektu*. Kontakt z otoczeniem obiektu odbywa się za pośrednictwem jednego (lub więcej) tzw. *interfejsu* (*interface*), które odgrywają rolę wejścia albo wyjścia obiektu (systemu względnie odosobnionemu). Za działanie każdego z interfejsów obiektu, jest odpowiedzialna któraś z metod obiektu.

4.7.2.03. **Definicja**. *Klasa*. Obiekty należą do rodzin obiektów o identycznych atrybutach i metodach, zwanych *klasami*. Jednocześnie klasa jest traktowana, jako matryca do tworzenia nowych wystąpień obiektów danej klasy.

4.7.2.04. **Definicja**. *Metoda i metoda abstrakcyjna*. Każda metoda składa się z nazwy i swojej implementacji. Metodę, dla której określono jedynie nazwę (bez określenia implementacji) nazywamy *metodą abstrakcyjną*.

4.7.2.05. **Definicja**. *Klasa abstrakcyjna*. Klasę. której wszystkie metody zostały jedynie nazwane (bez określenia implementacji) nazywamy *klasą abstrakcyjną*. Klasa abstrakcyjna nie może tworzyć obiektów, za to mogą po niej dziedziczyć klasy pochodne.

4.7.2.06. **Definicja**. *Konstruktor*. Mechanizm wchodzący w skład każdej klasy, umożliwiający tworzenie obiektów danej klasy nazywamy konstruktorem obiektów klasy lub krótko konstruktorem.

4.7.2.07. **Definicja**. *Atrybut dynamiczny*. Atrybut, którego wartość jest indywidualnie wyznaczana dla każdego obiektu danej klasy, nazywamy *atrybutem dynamicznym*.

4.7.2.08. **Definicja**. *Atrybut statyczny*. Atrybut, którego wartość jest wyznaczana równocześnie dla wszystkich obiektów danej klasy, nazywamy *atrybutem statycznym*.

4.7.2.09. **Definicja**. *Asocjacje*. Obiekty mogą się między sobą komunikować, tylko wtedy, jeśli są wyposażone w taki sam (tego samego typu) interfejs. Obiekty o identycznym interfejsie mogą być łączone w grupy obiektów. Obiekty tej samej klasy wyposażone są w interfejs wspólny dla obiektów danej klasy. Łączenie obiektów może być tworzone w układzie 1:1, 1:N albo N:M. Połączenia pomiędzy obiektami nazywamy *asocjacjami*.

4.7.2.10. **Definicja**. *Implementacja abstrakcyjnego interfejsu*. Obok interfejsów przypisanych wszystkim wystąpieniom obiektów danej klasy, zgodnie z definiowaną strukturą nadawanej tworzonym przez mechanizm kreacji nowych wystąpień obiektów danej klasy, istnieją tzw. abstrakcyjne interfejsy z nadaną nazwą interfejsu i nazwanymi metodami, ale bez implementowanej funkcjonalności metody, które mogą być dołączane do klas lub pojedynczych wystąpień obiektu, z jednoczesnym przypisaniem temu interfejsowi dołączanemu do indywidualnego obiekty (lub wszystkich obiektów danej klasy) określonej metody lub metod. Tę ostatnią czynność implementacji funkcjonalności metody do dodawanego interfejsu do obiektu - nazywamy *implementacją interfejsu*.
Uwaga. Jak widać z tej definicji abstrakcyjny interfejs może zawierać odwołanie do jednej lub więcej nazw metod bez implementacji funkcjonalności tym metodom. Przykładowo, metoda o

nazwie „*otwieranie*" – może mieć implementowane różne funkcjonalności, takie jak: „*otwieranie drzwi*" albo „*otwieranie słoika*".

4.7.2.11. **Definicja**. *Dziedziczenie klas*. Nowe klasy mogą być tworzone z istniejącej klasy obiektów, przez rozszerzane o dodatkowe atrybuty i metody (z określoną funkcjonalnością), taką operację rozszerzania klasy nazywamy dziedziczeniem. Klasę, do której dodajemy dodatkowe atrybuty oraz metody nazywamy klasą rodzicielską lub krótko rodzicem. Klasę utworzoną w wyniku operacji dziedziczenia, nazywamy *klasą pochodną*.
Uwaga. Klasa pochodna ma tylko jedną klasę rodzicielską.

4.7.2.12. **Definicja**. *Dziedziczenie abstrakcyjnych interfejsów.* Nowy abstrakcyjny interfejs może być tworzone poprzez składanie wcześniej zdefiniowanych abstrakcyjnych interfejsów oraz przez dołączanie dodatkowych nazw metod. Przez analogię do dziedziczenia klas operację tą nazywamy - *dziedziczeniem interfejsów*. Interfejsy abstrakcyjne, składające się na nowo tworzony interfejs abstrakcyjny nazywamy rodzicami, a nowo wykreowany interfejs swobodny nazywamy *interfejsem pochodnym*.
Uwaga. Abstrakcyjny interfejs pochodny, może mieć wiele interfejsów rodzicielskich.

4.7.2.13. **Definicja**. *Prawa dostępu*. W podejściu obiektowym wprowadzono trzy kategorie: (1) *public* (czyli element ogólnie widoczne i dostępny), (2) *private* (czyli elementy widoczne i dostępne jedynie dla ich autora), oraz wreszcie, (3) *protected* (czyli elementy pochodne widoczne i dostępne). Tymi elementami są: klasy, obiekty, swobodne interfejsy, atrybuty i metody. Dodatkowym mechanizmem zabezpieczającym jest opcja *final* – blokowania dziedziczenia klas, swobodnych interfejsów, metod oraz atrybutów.

4.7.2.14. **Definicja**. *White box technique,* czyli technika posługiwania się elementami (klasami, swobodnymi interfejsami, obiektami, metodami), których funkcjonalność jest znana. Taka sytuacja istnieje, gdy tworzymy systemy z obiektów tworzonych w oparciu o znane klasy, swobodne interfejsy oraz operacje dziedziczenia. Posługiwanie się *white box technique,* prowadzi do stosowania struktury statycznej aplikacji.

4.7.2.15. **Definicja**. *Black box technique,* czyli technika posługiwania się obiektami należącymi do różnych klas, na drodze wykorzystywania komunikatów przekazywanych pomiędzy obiektami należącymi do różnych klas, ale wyposażonych we wspólne interfejsy. Posługiwanie się *black box technique,* prowadzi do stosowania struktury dynamicznej aplikacji.

4.7.2.16. **Definicja**. *Polimorfizm*, czyli wielopostaciowość, własność metody pozwalająca na operowanie obiektami różnych klas.

4.7.2.17. **Definicja**. *Przeciążanie* (lub przeładowywanie, *overload*) pozwala nazwać tak samo kilka metod operujących na różnych typach danych i następnie obsługiwać te dane w jednolity sposób.

Piśmiennictwo: *Booch* G. B.6.1., *Dahl O.* D.1.1., *D'Souza D*. D.4.1., *Jacobson I.* J.1.1., *Guttag J.* G.5.1.

4.7.3. Rozwój podejścia obiektowego

Na przełomie wieków, można powiedzieć, że podejście obiektowe zdominowało rozwój oprogramowania oraz rozwój języków projektowania systemów informatycznych. Jak już wspominaliśmy, w drugiej połowie lat sześćdziesiątych XX wieku, koncepcja podejścia obiektowego w programowaniu, pojawiła się w Norwegii w języku SIMULA 67 – nadbudówki

nad opracowanym w Danii językiem programowania ALGOL 60. Prawdopodobnie w tym samym okresie, podejście obiektowe pojawiło się w praktyce projektowania systemów informatycznych w Szwecji (prace projektowe prowadzone na rzecz firmy ERICSON)[76].

Przyczyn rozpowszechnienia się podejścia obiektowego, w ostatnim trzydziestoleciu XX wieku, w językach programowania można upatrywać w:

1. *Obiekty* są dobrym abstrakcyjnym modelem „*istnień*" w świecie realnym, a w szczególności przydatne są do opisywania systemów rozproszonych (np. sieciowych).
2. *Obiekty* wydają się dobrymi komponentami budowania modularnego oprogramowania dostosowanego do wielokrotnego użycia, przy zapewnieniu niezbędnej jakości, w złożonych aplikacjach systemów informatycznych.

Oczywistym jest, że podejście obiektowe nie jest panaceum, które zapewnia poprawność formalną – dodatkowo wolną od wszelkiego rodzaju błędów, w procesach projektowania systemów złożonych z wielokrotnie wykorzystywanych modułów. Dodatkowo, brak jest teorii uzasadniającego podejście obiektowe – porównywalnej, np. z matematyczną teorią relacji, wykorzystaną w systemach relacyjnych baz danych. Istnieją wprawdzie pewne zalążki teorii (*Abadi and Cardelli – Sigma Calculus, 1996; Ciaffaglione et aI, 2006*), ale nie stanowią one podstawy teoretycznej podejścia obiektowego, są niekompletne i ograniczają się jedynie do własności *polimorfizmu* i *nadpisywania*.

Należy stwierdzić, że podejście obiektowe do zadań stawianych przed informatyką, jest czymś więcej niż konstrukcyjnym podejściem do budowy poprawnych rozwiązań. Podejście obiektowe oferuje bardzo szeroki wachlarz możliwości rozwiązywania problemów dotyczących różnorodnych dziedzin, z uwzględnieniem opracowywania elastycznych modeli realnych procesów, projektowania wygodnych interfejsów użytkownika, pisania czytelnych programów, zespołową organizację prac projektowych oraz zespołowego programowania, wreszcie wielokrotnego wykorzystania opracowanych rozwiązań projektowych systemów informatycznych i modułów programów.

Centralną koncepcją w podejściu obiektowym są nie tylko pojęcia klas obiektów i samych obiektów; ale również przekazywanie danych pomiędzy obiektami za pośrednictwem wiążących je asocjacji, do czego wystarczy znajomość (widoczność z zewnątrz) interfejsów obiektów oraz metod, które zamierzamy wykorzystać. Oczywiści ta centralna dla podejścia obiektowego koncepcja powstawała w kolejnych krokach rozwoju podejścia obiektowego. Podstawa tej koncepcji pojawiła się już w języku SIMULA 67 (*Dahl, Nygaard* - 1966), zaprojektowanym dla symulacji równolegle przebiegających procesów. Wprowadzone w języku SIMULA 67 klasy umożliwiły abstrahowanie od szczegółów przyjętych rozwiązań, na drodze ukrycia sposobów implementacji i tworzenie modeli istnień o rosnącej złożoności.

Aspekt abstrakcji od szczegółów dotyczącej wewnętrznej struktury obiektu, umożliwił ukrycie przed użytkownikiem obiektu struktury danych obiektu i został sformalizowany w latach siedemdziesiątych ubiegłego wieku w ramach teorii abstrakcyjnych typów danych (*Liskov and ZiIIes, 1975*; *Guttag et al, 1978*). Dalszy rozwój centralnej koncepcji podejścia obiektowego, to zdefiniowanie zbioru podstawowych metod obiektu (np. działań na stosie: *push*, *pop* i *top*) i

[76] Porównaj "Invar Jacobson, Maria Ericson and Agneta Jacobson, *The Object Advantage – business process reengineering with object technology*, Addison-Wesley Publishing Company, 1994."

określenie formalnej specyfikacji tych metod oraz ich wzajemnych związków w postaci zbioru aksjomatów, nie precyzujących sposobów ich implementacji (*Goguen et al, 1978*).

Równolegle do formalizacji typu abstrakcyjnego danych, opracowany został pierwszy język programowania obiektowego SMALLTALK (*Kay and Goldberg, 1976*); podobnie jak język SIMULA 67 zaimplementował koncepcję obiektowości w postaci klas obiektów, uzupełnioną przekazywaniem komunikatów (*message passing*), pojęciem zapożyczonym z koncepcji aktora – zewnętrznego partnera systemu informatycznego (*Hewitt et al, 1973*). Język programowania obiektowego SMALLTALK użył również uogólnionej koncepcji dziedziczenia klas i obiektów – dla tworzenia struktur hierarchicznej koncepcji klas. Można powiedzieć, że język SMALLTALK stał się źródłem inspiracji dla dalszego rozwoju podejścia obiektowego. Między innymi rozwinięcia pomysłu systemu o wielu oknach (*multi-window system*), z graficznym interfejsem użytkownika – umożliwiającym przeciąganie i porzucanie obiektów graficznych, itd.

Kolejnym krokiem rozwoju podejścia obiektowego, było powstanie kolejnej wersji języka programowania obiektowego SMALLTALK, nazwanego SMALLTALK-80. Trzy klasowy model MVC (*Model/View/Controller*), został użyty do zaimplementowania interfejsu użytkownika w SMALLTALK-80. Pojawienie się kolejnych wersji języka programowania obiektowego SMALLTALK, spowodowało zainteresowanie pisaniem programów – bloków konstrukcyjnych wielokrotnego użycia, bloki te nazwano *wzorcami (patterns*)[77].

4.7.3.00. **Wyjaśnienie**. Wzorzec to coś znacznie więcej niż model. Wzorzec powstaje w wyniku znalezienia typowego rozwiązania danego problemu, wyjaśniający jednocześnie istotę tegoż problemu – z wyjaśnieniem, dlaczego dany wzorzec jest poprawnym rozwiązaniem tegoż problemu. Wzmiankowany wyżej trzy klasowy model MVC, jest przykładem wzorca.

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides – autorzy biblioteki wzorców opublikowanej w 1995 roku zakładali, że liczba niezbędnych standardowych bloków konstrukcyjnych niezbędnych do budowy aplikacji informatycznych jest ograniczona i stosunkowo nieliczna, w opracowanej bibliotece wzorców, opisali oni jedynie 23 wzorce. Badanie przeprowadzone trzynaście lat później przez zespół z Uniwersytetu Nebraska-Lincoln (*Surveying Software Pattern Collections 1-1-2007 – DigitalCommons@University of Nebraska - Lincoln*), wykazały że istnieje 170 rodzin wzorców, zawierających łącznie 2.241 wzorców. Badanie to raczej wskazuje, że nadzieje na powstanie powszechnie akceptowanej biblioteki wzorców, bibliotece zawierającej stosunkowo niewielką liczbę standardowych modułów konstrukcyjnych, okazały się płonne.

Równolegle do prac nad językami programowania obiektowego, prowadzono w latach osiemdziesiątych oraz dziewięćdziesiątych XX wieku – prace nad obiektowymi językami graficznymi wspomagania projektowania systemów informatycznych. Połączenie wysiłków *Grady'ego Booch'a*, *Jamesa Rumbaugh'a* i *Ivara Jacobsona* – doprowadziło do powstania języka graficznego UML (*Unified Modeling Language*), który następnie został przyjęty, jako standard[78].

Piśmiennictwo: *Abadi M*. A.1.1., *Booch G*. B.6.1., *D'Souza D*. D.4.1., *Jacobson I.* J.1.1., *Gamma E.* G.1.1., *Guttag J.* G.5.1., *Liscov B.* L.1.1., *Wikipedia* W.2.1., W.2.27., W.2.28.

---

[77] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, 1st Edition published by 1995.
[78] *Unified Modeling Language* (*UML*) – defined by The Object Management Group (OMG) - *www.omg.org*.

4.7.4. UNIFIED MODELING LANGUAGE

Graficzny język wspomagania obiektowego projektowania i dokumentowania systemów informatycznych UML (*Unified Modeling Language*), jest wynikiem wieloletnich prac zespołowych. Jak pisze Marti Fowler „Najwcześniejsza publikowana wersja pierwowzoru UML-a, to wersja 0.8 Metody Zunifikowanej (*Unified Method*). Została ona przygotowana na konferencję OOPSLA, która odbyła się w październiku 1995 roku. Było to dzieło *Boocha* i *Rumbaugha* - *Jacobson* rozpoczął pracę w firmie *Rational Software* dopiero później. W 1996 roku *Rational Software* wydała wersje 0.9 i 0.91, które zawierają już wkład *Jacobsona*. Po wersji 0.91 *Rational Software* zmieniła nazwę specyfikacji na UML. W styczniu 1997 roku *Rational Software* wraz z grupą partnerów przedstawiła wersję 1.0 UML-a grupie roboczej OMG - *Analysis and Design Task Force*. Następnie we wrześniu 1997 roku, łącząc tę wersję UML-a z dodatkowymi elementarni, *Rational* wraz ze swoimi partnerami i innymi twórcami przedstawiła skonsolidowany tekst, teraz wersji 1.1, jako propozycję standardu OMG, który został przyjęty pod koniec 1997 roku. Niestety, OMG nadała wersji standardowej numer 1.0. Tak więc UML istniał teraz w wersji 1.0 OMG i w wersji 1.1 *Rational* – nie mylić z wersją 1.0 *Rational*, w praktyce, każdy nazywa tę wersję standardową wersją 1.1. Od tego czasu wprowadzono do UML-a wiele dalszych rozszerzeń. UML 1.2 pojawił się w l998 roku, wersja 1.3 w 1999 roku, wersja 1.4 w 2001 roku, a wersja 1.5 w 2002 roku. Większość zmian w wersjach 1.x, oprócz wersji 1.3, miało głęboki wpływ na UML-a i wprowadzało bardzo widoczne różnice, szczególnie w przypadkach użycia i diagramach czynności. Wraz z kontynuowaniem serii UML 1.x, twórcy UML-a zaczęli skupiać się na przygotowywaniu większego wydania UML-a w, postaci UML-a 2.0, Pierwsze dokumenty z propozycjami wersji zostały wydane w 2000 roku, ale wersja UML 2.0, zaczęła się stabilizować dopiero w 2003 roku."[79].

UML nie jest metodą samą w sobie, lecz był projektowany z myślą o zachowaniu zgodności z głównym trendem, rozwoju obiektowych metod oprogramowania (na przykład opracowaniami OMT). Odkąd powstał i rozwijał się UML, niektóre z tych metod zostały uaktualnione tak, by wykorzystywać nową notację. Powstały też nowe podejścia obiektowe programowania, na bazie istniejącego już UML. Najbardziej znana jest metoda *Rational Unified Process* (RUP) firmy Rational.

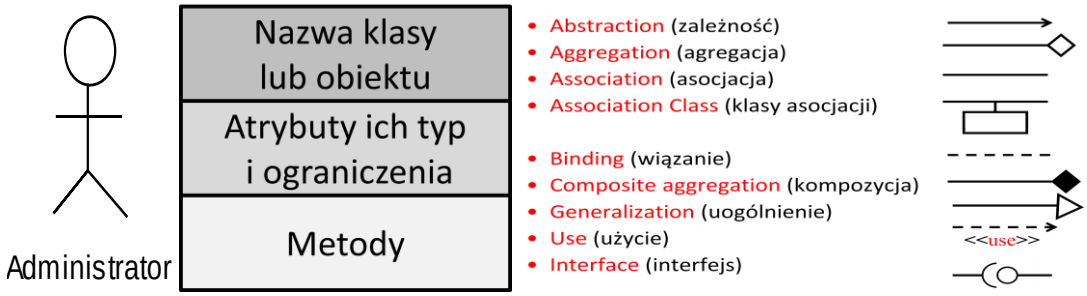
4.7.4.01. **Wyjaśnienie.** Dla UML 2.0, opracowano łącznie 14 diagramów głównych oraz 3 abstrakcyjne (struktur, zachowań i interakcji). Istnieją niestety pewne niejednoznaczności, co do polskiego tłumaczenia nazw diagramów, np. *deployment diagram* jest tłumaczony jako diagram wdrożenia, co nie odpowiada znaczeniu słowa *deployment* (rozwijanie w przypadku wojsk, spadochronu lub rozmieszczanie); albo *timing diagram* jest tłumaczony, jako diagram czasowy, zależności czasowych, harmonogramowania, uwarunkowań czasowych czy diagram przebiegów czasowych.

4.7.4.02. **Uwaga.** Wydaje się, że o ile pierwsze wersje UML były rozsądnym kompromisem pomiędzy różnorodnymi potrzebami analityków i projektantów systemów informatycznych, a różnymi ideami prezentacji graficznej zależności i funkcjonalności, to kolejne wersje zawierają coraz więcej pomysłów prezentacyjnych i stają się nadmiernie ciężkimi narzędziami, daleko przerastającymi potrzeby praktyki. Przykładowo, UML wersja 2.4.1 (ogłoszona w sierpniu

[79] Martin Fowler, UML w kropelce (tłumaczenie z angielskiego), Wydawnictwo LTP, Warszawa 2005 (str. 170).

2011)[80], oferuje mnóstwo dodatkowych możliwości, które chyba nie są niezbędne dla praktyka. W dalszym ciągu ograniczymy nasz wykład do podstawowych funkcjonalności UML wersji 2.0.

4.7.4.03. **Wyjaśnienie.** Jak już wzmiankowaliśmy, UML jest językiem graficznym, którego podstawowymi składnikami są tzw. diagramy. Diagramy budowane są z symboli klas lub obiektów, symboli powiązań oraz użytkowników systemu (patrz rys. 4.6.4.00). Pokazany na wzmiankowanym rysunku symbol klasy lub obiektu, jest również używany do prezentowania szczegółowej struktury interfejsy (w zasadzie abstrakcyjnego), nazwa interfejsu jest wtedy poprzedzona słowem kluczowym ≪interface≫, przedział atrybutów jest pusty (bo interfejsy nie mają atrybutów), a w przedziale metod umieszczana jest lista nazw metod interfejsu.



Rysunek 4.7.4.00. - konwencje graficznej UML prezentacji: użytkownika systemu, klasy lub obiektu oraz związków pomiędzy klasami lub obiektami i interfejsu.

4.7.4.04. **Wyjaśnienie.** Diagramy UML reprezentują zarówno statykę, jak i dynamikę systemu informatycznego, są zwykle klasyfikowane w trzech grupach:

1. *Diagramy struktur*, które objaśniają składniki system informatycznego modelujące zachowanie rzeczywistości. Diagramy te, reprezentują strukturę system i są następnie wykorzystywane w opisie architektury oprogramowania systemu informatycznego. Do grupy diagramów struktury zaliczamy: diagram klas (*class diagram*), diagram komponentów (*component diagram*), diagram struktur złożonych (*composite structure diagram*), diagram wdrożenia (*deployment diagram*), diagram obiektów (*object diagram*), diagram pakietów (*package diagram*) i ewentualnie diagram profili (*profile diagram*).
2. *Diagramy zachowań*, opisują reakcje na zdarzenia, które zachodzą w modelowanym systemie informatycznym. Ponieważ diagramy zachowań opisują zachowanie system, są one przydatne do definiowania funkcjonalności oprogramowania. Do grupy diagramów zachowań zaliczamy: diagram czynności (*activity diagram*), diagram maszyny stanowej (*state machine diagram*), diagram przypadków użycia (*use case diagram*).
3. *Diagramy interakcji*, są w zasadzie podzbiorem zbioru diagramów zachowań, ale wyróżniają się tym od pozostałych diagramów zachowań, że pokazują przepływ sterowania i danych wewnątrz systemu, którego zachowanie modelujemy. Do grupy diagramów interakcji zaliczamy: diagram komunikacji (*communication diagram*), diagram przeglądu interakcji (*interaction overview diagram*), diagram sekwencji (*sequence diagram*), diagram zależności czasowych (*timing diagrams*).

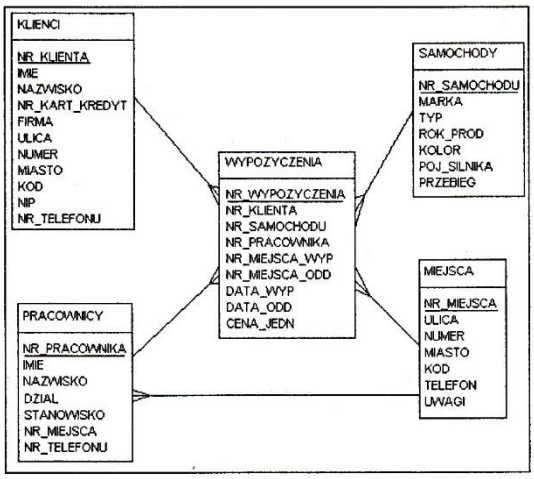
4.7.4.05. **Wyjaśnienie.** W przypadku modelowania biznesowego można korzystać z pewnych modyfikacji wyżej wymienionych diagramów UML, np. diagramu biznesowych przypadków użycia (charakterystyczne powiązanie aktora i przypadku użycia). W praktyce rzadko, kiedy

[80] Patrz http://www.omg.org/spec/UML/2.4.1

trzeba opracowywać wszystkie diagramy i w większości przypadków korzysta się z mniej niż połowy wyżej wymienionych. Nie powinno modelować się tylko dla samego modelowania, dlatego nie zawsze wszystkie rodzaje są potrzebne. Projektując *system informatyczny*, rozpoczyna się przeważnie od tworzenia diagramów w następującej kolejności:

1. Diagramy przypadków użycia
2. Diagramy sekwencji
3. Diagramy klas
4. Diagramy czynności.

Są to najczęściej wykorzystywane diagramy. Pozostałe bywają pomijane, zwłaszcza przy budowaniu niedużych systemów informatycznych.



Rysunek 4.7.4.10. – przykład uproszczonego diagramu klas dla systemu wypożyczalni aut

4.7.4.06. **Wyjaśnienie.** Opracowano, tzw. pięć zasad modelowania w UML, a oto one:

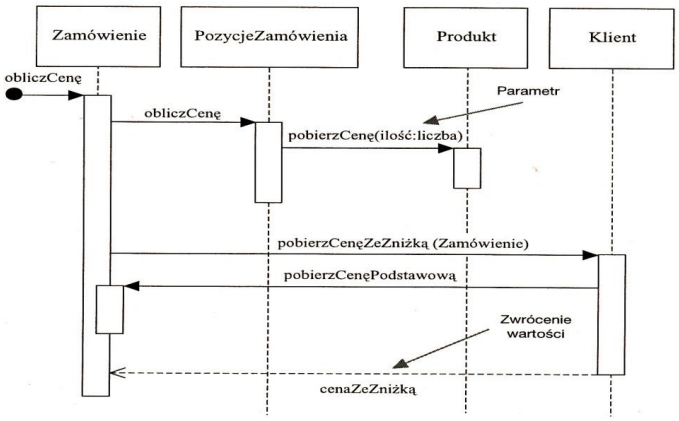
(1) Podjęcie decyzji, jakie modele tworzymy, ma wielki wpływ na to, w jaki sposób jest zaatakowany problem oraz jaki kształt przyjmie rozwiązanie.
(2) Każdy model może być opracowany na różnych poziomach szczegółowości.
(3) Najlepszy model odpowiada rzeczywistości.
(4) Żaden pojedynczy model nie jest wystarczający.
(5) Zrozumienie architektury modelowanego systemu, wymaga przeanalizowania dopełniających się modeli opracowanych z kilku perspektyw:
    a. Przypadków użycia (wg wymagań stawianych systemowi)
    b. Projektowej (uwzględniającej specyfikę zagadnienia oraz dopuszczalne rozwiązania)
    c. Procesowej (modelująca podział systemu na procesy)
    d. Rozwinięcia instalacji (pokazującej umiejscowienie komponentów systemu na instalacji sprzętowej).

4.7.4.11. **Wyjaśnienie.** Diagramy klas opisuje strukturę systemu, pokazując klasy z atrybutami i metodami oraz wzajemne powiązania obiektów tych klas. Rys. 4.7.4.10. zawiera przykładowy diagram klas dla wypożyczalni samochodów z wieloma punktami (lokalizacjami) wypożyczania i zwracania pojazdów. Diagram zawiera 8 klas z atrybutami (i ich typom) i metodami (z prawami dostępu: + *public*; - *private*) oraz powiązania (asocjacje) pomiędzy obiektami tych 8 klas. Krotność asocjacji ile obiektów danej klasy może być podłączone obiektów innej klasy. Występujące oznaczenia krotności to:

- 1 - *związek* (*asocjacja*) dołącza jeden obiekt do obiektu danej klasy;

- 0..1 – *związek* (*asocjacja*) dołącza zero lub jeden obiekt do obiektu danej klasy;
- * – *związek* (*asocjacja* lub *kompozycja*) dołącza jeden lub wiele obiektów do obiektu danej klasy;
- 0..* - *związek* (*asocjacja* lub *kompozycja*) dołącza zero lub wiele obiektów do obiektu danej klasy.

4.7.4.12. **Wyjaśnienie.** Omawianie diagramów UML, odbywa się zwykle począwszy od diagramu klas, w rzeczywistości projektowanie systemu informatycznego rozpoczyna się od przypadków użycia (patrz pkt. 4.7.4.61), które tworzone są na podstawie wymagań użytkownika na użytkowanie systemu przez poszczególnych użytkowników systemu (zwanych w UML aktorami). Proces poszukiwania klas wchodzących w skład systemu - polega na wyborze z opisów wszystkich przypadków użycia rzeczowników i zdań rzeczownikowych, które łącznie tworzą listę kandydatów na klasy systemu. Następnym krokiem jest redukcja listy kandydatów na klasy, na drodze usunięcia synonimów nazw klas (*redundant classes*), klas z poza systemu (*irrelevant classes*), niejednoznacznie określonych tworów przypominających klasy (*vague classes*), atrybuty klas (*attributes*), rzeczownikowe określenia nazw czynności (*operations*), rzeczownikowe określenia roli (*roles*), nazwy sugerowanych konstrukcji systemowych (*implementation constructs*). W wyniku powstaje lista zredukowana lista – kandydatów na klasy. Na podstawie danych wybranych z opisów przypadków użycia można już opracować opisowe charakterystyki każdej z klas, a raczej kandydatów na klasy. Kolejnym krokiem projektowym, jest poszukiwanie asocjacji pomiędzy obiektami poszczególnych klasami, w tym celu wybieramy z opisów wszystkich przypadków użycia czasowniki i zdania czasownikowe łączące dwa lub więcej potencjalne obiekty różnych klas, które po redukcji pozwalają na określenie listy niezbędnych asocjacji dla klas umieszczonych w diagramie klas. Dalsze doskonalenie projektowanego diagramu klas polega na identyfikacji operacji dziedziczenia pomiędzy klasami oraz na dwóch analizach, tzw. „*Top down*" oraz „*Bottom up*", w celu identyfikacja atrybutów dla klas i asocjacji klas (*Identifying Associations between Classes*) i upraszczanie modelu klas.



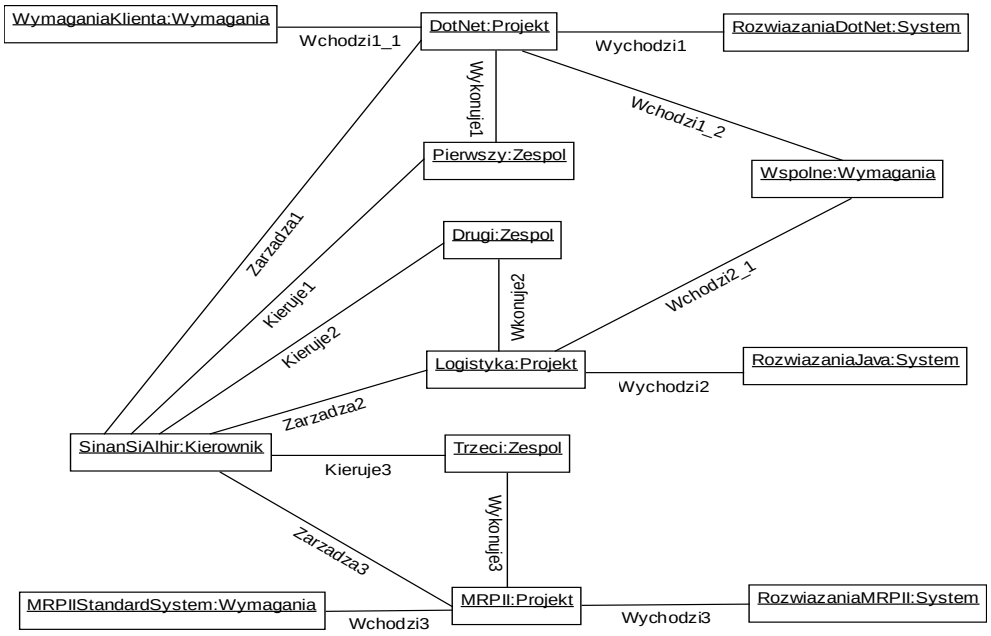
Rysunek 4.7.4.20. – przykład diagramu sekwencji pokazujący proces opracowania przez system pozycji zamówienia.

4.7.4.13. **Wyjaśnienie.** Jak wynika z powyższego, diagram klas powstaje metodą kolejnych przybliżeń (czyli iteracyjnie). Pokazany na rys. 4.7.4.10 diagram klas ma postać końcową, czyli taką postać, jaką uzyskujemy po zakończeniu iteracyjnego procesu budowy tegoż diagramu. Po zakończeniu postępowania opisanego w pkt. 4.7.4.12, mamy pokazany jedynie zbiór klas połączonych asocjacjami z pokazaniem jedynie podstawowych atrybutów wystąpień obiektów tworzonych dla tych klas. Określenie metod dla obiektów poszczególnych klas może nastąpić w

wyniku opracowania diagramów sekwencji (patrz pkt. 4.7.4.21.) dla poszczególnych działań systemu, wynikających z realizacji poszczególnych przypadków użycia (patrz punkty 4.7.4.61 - 4.7.4.64).

4.7.4.21. **Wyjaśnienie.** Diagramy sekwencji pokazuje zasady komunikacji pomiędzy obiektami przesyłającymi sobie komunikaty (sekwencje komunikatów) i tym samym pokazując wzajemne czasowe zależności pojawiające się w wyniku przesyłania tych komunikatów. Pokazany na rys. 4.7.4.20 przykład dotyczy przypadku, gdy sterowanie jest zdecentralizowane pomiędzy współdziałające obiekty, czyli każdy obiekt zarządza przydzielonym fragmentem procesu.

4.7.4.22. **Wyjaśnienie.** Opracowania diagramów sekwencji, odbywa się dla poszczególnych działań systemu, wynikających z realizacji poszczególnych przypadków użycia z pomocą diagramów czynności (patrz punkty 4.7.4.81 - 4.7.4.82). Każdy z diagramów czynności jest przedstawiany, jako sekwencja czynności realizowanych na kolejnych obiektach (klas określonych na diagramie klas, patrz pkt. 4.7.4.11) z wykorzystaniem wcześniej określonych atrybutów, uzupełnianych metodami, jakie muszą dla zrealizowania założonej funkcjonalności posiadać kolejne obiekty sekwencji, a następnie wstępne określanie zawartości komunikatów (*messages*) przekazywanych pomiędzy obiektami.



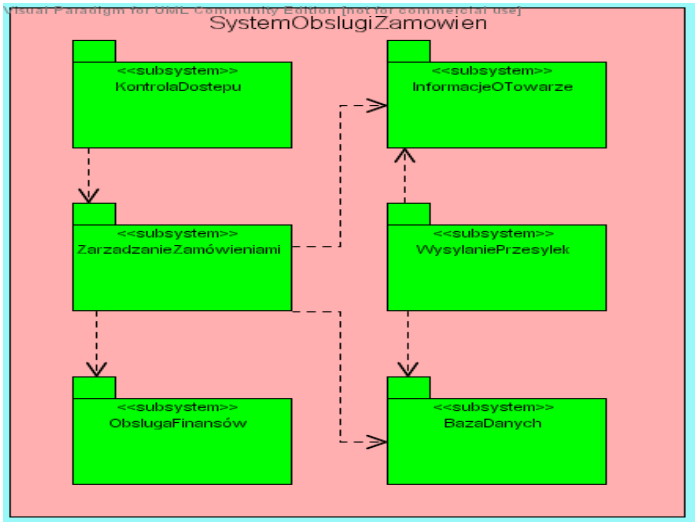
Rysunek 4.7.4.30**.** – przykład diagramu obiektów pokazujący powiązanie obiektów w procesie projektowania systemu informatycznego klasy MRP II.

4.7.4.31. **Wyjaśnienie.** Diagram obiektów pokazuje cząstkowe lub kompletne powiązania pomiędzy obiektami jednej lub wielu klas w danym przedziale czasu. Pokazany na rys. 4.7.4.30 przykład dotyczy przypadku, przykładowego systemu informatycznego powiązanie obiektów w procesie projektowania systemu klasy MRP II. Jest to raczej niezbyt typowy przykład użycia diagramu obiektów.

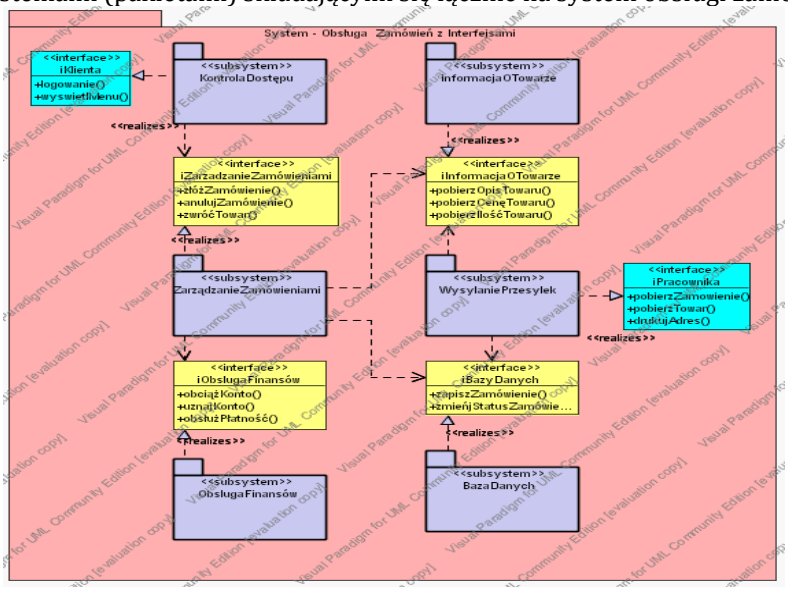
4.7.4.32. **Wyjaśnienie**. Diagramy obiektów, są w pierwszej kolejności używane do zintegrowania wymagań na funkcjonalności metod obiektów i zawartość komunikatów

przekazywanych pomiędzy obiektami, określonych wcześniej w ramach poszczególnych diagramów sekwencji (patrz punkty 4.7.4.21 - 4.7.4.22).

4.7.4.41. **Wyjaśnienie.** Diagram pakietów pokazuje jak system główny (pakiet główny) jest podzielony na podsystemy (pakiety podrzędne) grupujące wyspecjalizowane funkcjonalności oraz równocześnie pokazujący zależności pomiędzy systemem głównym, podsystemami i ewentualnie wzajemne zależności pomiędzy tymi ostatnimi. Pokazany na rys. 4.7.4.40a przykład dotyczy przypadku systemu zbudowanego z sześciu współpracujących ze sobą podsystemów – pakietów. Natomiast rys. 4.7.4.40b pokazuje ten sam schemat zależności uzupełniony interfejsami.



Rysunek 4.7.4.40a. – przykład diagramu pakietów pokazujący powiązania pomiędzy podsystemami (pakietami) składającymi się łącznie na system obsługi zamówień.

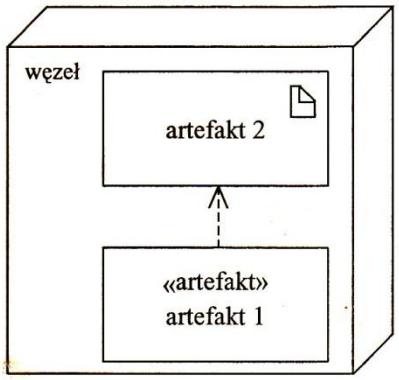


Rysunek 4.7.4.40b. – przykład diagramu pakietów pokazujący powiązania pomiędzy podsystemami (pakietami) składającymi się łącznie na system obsługi zamówień z uwzględnieniem interfejsów.

4.7.4.42. **Wyjaśnienie.** Jak pokazano w pkt. 4.7.4.73, diagram pakietów powstaje w wyniku grupowania fragmenty systemu, w których system przyjmuje ten sam stan. Po dokonaniu grupowania według danego stanu, należy zapewnić możliwość wzajemnej współpracy poszczególnych pakietów na drodze wyposażenia każdego z pakietów w odpowiedni interfejs, umożliwiający założoną współpracę (wywołanie jak podprogramu lub przekazanie sterowania).

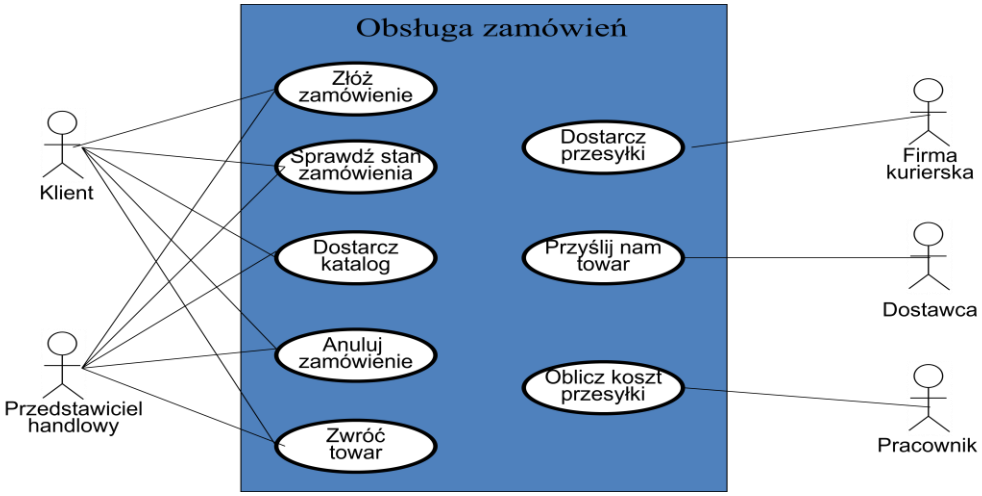
4.7.4.51. **Wyjaśnienie.** Diagram wdrożenia pokazuje jak na sieci komputerowej, czyli na poszczególnych węzłach sieci (urządzeniach i sprzętowych modułach systemu sieciowego), rozmieszczone są poszczególne pakiety składowe oprogramowania systemu informatycznego.

4.7.4.52. **Wyjaśnienie.** Diagramy wdrożenia powstaje w wyniku przydzielenia składowania i wykonywania pakietów wraz z łączącymi je interfejsami poszczególnym węzłom sieci komputerowej systemu informatycznego.



Rysunek 4.7.4.50. – przykład diagramu wdrożenia – pokazujący, które pakiety systemu są zainstalowane na danym węźle sieci komputerowej

4.7.4.61. **Wyjaśnienie.** Diagramy przypadków użycia opisuje funkcjonalność dostarczaną przez system poszczególnym aktorom (co tworzy poszczególne przypadki użycia systemu, przez aktora), określa cele poszczególnych przypadków użytkowania systemu oraz ich wzajemne zależności. Diagram przypadków użycia pokazuje granicę systemu, umieszczając wszystkich aktorów – użytkowników systemu na zewnątrz systemu (poza granicami systemu). Gdzie przez aktora rozumiemy zarówno osobę fizyczną, reprezentanta grupy osób lub inny system, czy też innego systemu lub urządzenia.



Rysunek 4.7.4.60. – przykład diagramu przypadku użycia zamówień składanych do sklepu internetowego, obejmuje 8 przypadków użycia i 5 aktorów.

4.7.4.62. **Wyjaśnienie**. Każdy przypadek użycia ma swojego głównego aktora, który zamawia wykonanie przez system określonej usługi. Specyfikacja systemu powinna zawierać szczegółowy opis wykonywania usługi, wraz z podaniem algorytmu wykonania usługi. Powyższy algorytm jest podstawą do opracowania tzw. scenariusza wykonywania przypadku użycia, scenariusz taki

dzieli się na scenariusz podstawowy (wykonywany przy standardowej realizacji usługi) oraz scenariuszy sytuacji szczególnych, których uruchomienie następuje w przypadku powstania sytuacji nadzwyczajnych, np. w przypadku wystąpienia błędnej czynności aktora. Scenariusz zawiera kolejne kroki jego realizacji przez system oraz każdy scenariusz opisuje jedną ścieżkę przejścia przez przypadek użycia. Czyli każdemu scenariuszowi odpowiada ciąg zdarzeń odpowiadający głównemu ciągowi zdarzeń lub głównemu ciągowi zdarzeń rozszerzonemu o jeden lub więcej nadzwyczajnych ciągów zdarzeń przypadku użycia.

4.7.4.63. **Wyjaśnienie**. Poniżej przedstawiamy zalecenia przy pisaniu scenariuszy przypadku użycia, które wydaje się, że pomogą zrozumieć znaczenie tegoż pojęcia:

(1) Sformułować warunek wstępny, jaki musi być spełniony, żeby scenariusz mógł być realizowany.
(2) Jednoznaczne określić wynik, jaki ma zostać dostarczony, w wyniku wykonania scenariusza.
(3) Określić zdarzenie – wyzwalacz, powodujący uruchomienie wykonywania scenariusza.
(4) Używać tylko prostych form gramatycznych (np. System ... odejmuje ... kwotę ... od salda konta)
(5) Jasno wskazać gdzie jest „piłka". Scenariusz ma strukturę, w której w każdym kroku ktoś (czy coś) ma „piłkę. Piłka to komunikat (*message*) i dane przekazywane pomiędzy uczestnikami interakcji. Z opisu jasno musi wynikać, gdzie jest „piłka" po wykonaniu każdego kroku scenariusza.
(6) Pisać „z lotu ptaka". Np. Klient podaje swoją kartę bankomatową i hasło. System odejmuję kwotę od salda konta. Pokazać postęp zawansowania proces (do przodu). Np. Użytkownik wprowadza nazwisko i adres.
(7) Pokazać interakcję aktora z systemem, a nie elementarne czynności aktora. Np. Użytkownik podaje nazwisko i adres. System wyświetla profil użytkownika.
(8) Przypadek użycia ma zawierać sensowny zbiór czynności (*actions*). Np. transakcje powinny być opisywane, jako cztery połączone razem wykonywane czynności:
    a. Aktor wysyła zlecenie i dane do systemu
    b. System stwierdza poprawność zlecenia i danych
    c. System zmienia swój wewnętrzny stan
    d. System dostarcza aktorowi wynik transakcji.
(9) Stwierdzać, że ..., a nie „Sprawdzaj, czy ...".
(10) Można wspomnieć o synchronizacji. Np. W każdej chwili między krokami 3 i 4, aktor ...; albo Jak tylko aktor ..., system ...
(11) Zamiast pisać, że system A pobrał informację z systemu B, w wyniku działań użytkownika, należy raczej napisać: Np. Użytkownik sprawił, że system A sprowadził w tle dane z systemu B.
(12) Czasem należy zaznaczyć, że pewne kroki mają być powtórzone. Jeśli jest to tylko jeden powtarzalny krok, to należy napisać: Np. Użytkownik wybiera jeden albo więcej produktów.
(13) Zapewnić, aby warunki mówiły o tym, co wykryto. Czyli napisz, co system wykrył, a nie tylko to, co się stało.
(14) Wcinać w tekście obsługę rozszerzeń. Np.:
    a. Za małe saldo:
    b. .1 System informuje klienta i prosi o podanie innej kwoty.
    c. .2 Klient podaje nową kwotę.

4.7.4.64. **Wyjaśnienie**. Jak już powiedzieliśmy wcześniej (patrz pkt. 4.6.4.12.), opisy przypadków użycia są podstawą do budowy diagramu klas.

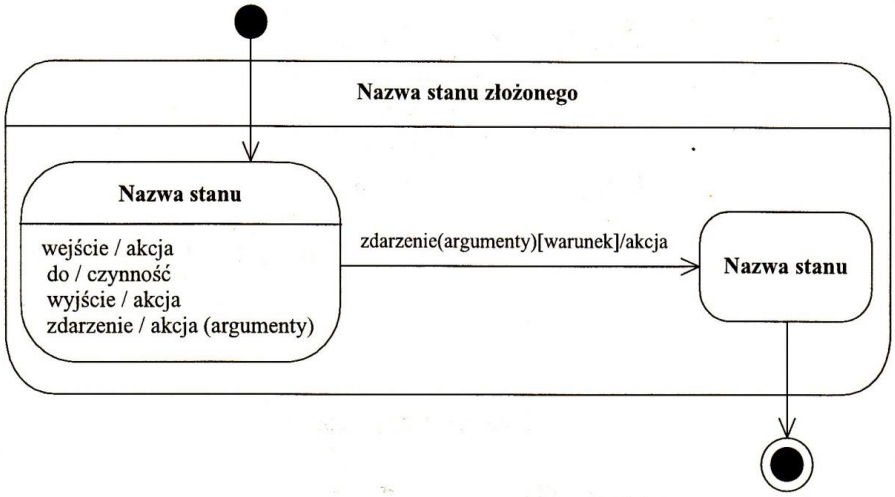
4.7.4.71. **Wyjaśnienie.** Diagram maszyny stanów opisuje stany przyjmowane przez system oraz dopuszczalne przejścia od jednego stanu do innego.

4.7.4.712. **Wyjaśnienie.** Diagram maszyny stanów powstaje w oparciu o diagram obiektów, a raczej przypadek istnienia kilku diagramów obiektów dla poszczególnych grup obiektów uczestniczących w interakcjach pomiędzy obiektami w czasie wykonywania zróżnicowanych funkcjonalności projektowanego systemu informatycznego.

4.7.4.73. **Wyjaśnienie.** Diagram maszyny stanów jest zazwyczaj używany, jako punkt wyjścia do dokonania podziału systemu na pakiety (patrz pkt. 4.7.4.41). Fragmenty systemu, w których system przyjmuje ten sam stan, są następnie grupowane w pakiety.

4.7.4.81. **Wyjaśnienie.** Diagramy czynności pokazuje kolejne kroki wykonywania (obsługiwania) przez system czynności przez poszczególne komponenty systemu. Mówiąc krótko, diagram czynności pokazuje zasady przekazywania sterowania pomiędzy modułami (np. obiektami lub pakietami) w czasie wykonywania poszczególnych funkcjonalności systemu.

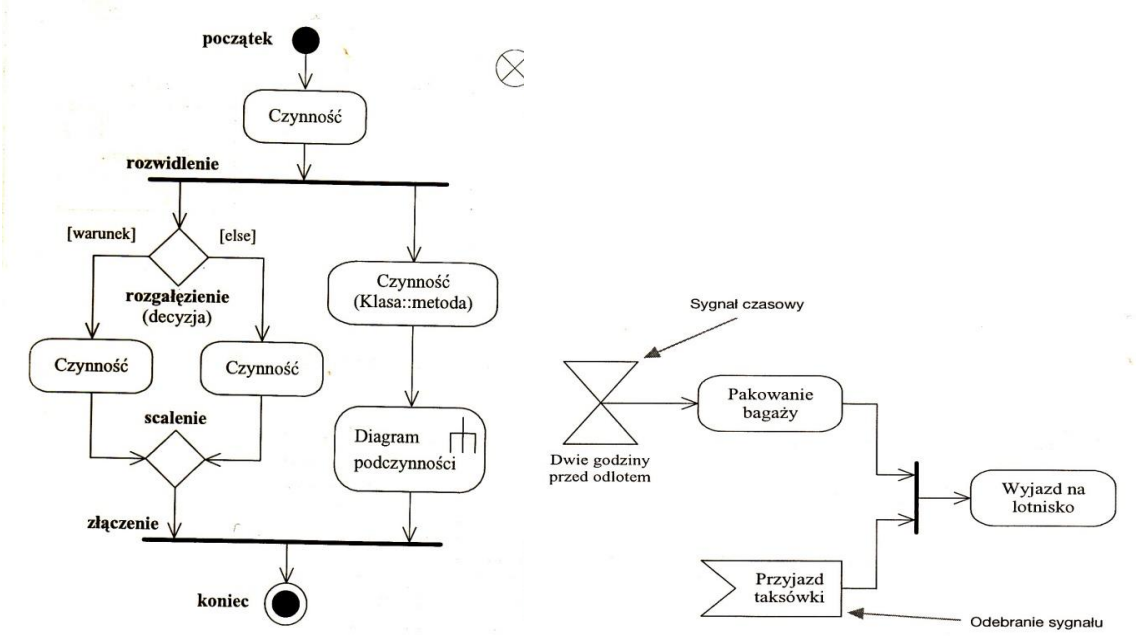
4.7.4.82. **Wyjaśnienie.** Diagramy czynności powstaje z przypisania do kroków scenariusza realizacji poszczególnych przypadków użycia (patrz punkty: 4.7.4.61. i 4.7.4.62.), do poszczególnych klas określonych w diagramie klas (patrz punkty: 4.7.4.11. i 4.7.4.12.) oraz skonfrontowania z tworzeniem lub przekształcaniem obiektów tych klas w poszczególnych krokach scenariusza.



Rysunek 4.7.4.50. – przykład diagramu maszyny stanów, przypisujący zdarzeniom wraz z warunkiem, określony stan systemu

4.7.4.90. **Wyjaśnienie.** Procesory UML (oferowane przez kilka znanych firm) - wspomagają praktyczne stosowanie funkcjonalności UML, w szczególności tworzenie diagramów UML (modułami przykładowego procesora UML są: *Project Manager, Business Analyst, Diagram Elaboration, Software Architect and Simulator, System Analyst, System Designer and System Developer*), wspomagane dodatkowym językiem, takim jak *Business Process Modeling Notation*

(BPMN)[81]. Użycie dodatkowo języka BPMN wynika z faktu, że UML umożliwia opisanie funkcjonalności projektowanego systemu informatycznego, ale nie pozwala na samodzielne opisanie tych czynności procesów biznesowych, które mają miejsce w otoczeniu systemu. Dla stworzenia jednolitego opisu funkcjonalnego całego systemu zarządzania np. organizacją gospodarczą, koniecznym jest opisanie całości procesów biznesowych danej organizacji, a nie jedynie tych fragmentów procesów biznesowych, które mają być wspomaganie systemem informatycznym, opisanym modelem UML.



Rysunek 4.7.4.80. – przykład diagramu czynności, lewa strona rysunku
(a) czynności rozdzielone na dwa równolegle wykonywane wątki, zaś prawa strona rysunku
(b) pokazuje użycie symbole „sygnału czasowego" o zdarzenia „odebranie sygnału",
z których każda rozpoczyna oddzielny wątek.

PIŚMIENNICTWO*: BPMN* B.5.1., *Fowler M.* F.1.1., *UML* U.1.1.

4.8. KRÓTKO O GEOMETRII OBLICZENIOWEJ I GRAFICE KOMPUTEROWEJ

4.8.0. UWAGI WSTĘPNE

Geometria obliczeniowa jest działem informatyki, który wyodrębnił się z dziedziny projektowania i analizy algorytmów w końcu lat siedemdziesiątych dwudziestego wieku. Geometria obliczeniowa znalazła zastosowanie w wielu dziedzinach, w szczególności:

1. Wspomaganie prac inżyniertskich (CAD/CAM)
2. Symulacje i wirtualna rzeczywistość
3. Robotyka i druk 3D
4. Medycyna

Geometria obliczeniowa w pierwszym okresie swojego rozwoju, skoncentrowała się na opracowaniu efektywnych algorytmów dotyczącej zobrazowania na siatce pikseli: figur geometrycznych płaskich oraz rzutów brył trójwymiarowych. Kolejnym krokiem w rozwoju geometrii obliczeniowej było opracowanie algorytmów zobrazowania na przestrzennej siatce

[81] Business Process Modeling Notation (BPMN) - *www.bpmn.org.*

workcell - brył trójwymiarowych, dotyczących zarówno robotyki i druku 3D, jak również medycyny.

Geometria obliczeniowa odgrywa istotną rolę w grafice komputerowej. Z grafiką komputerową spotykamy się dzisiaj prawie na każdym kroku. I najczęściej nie zdajemy sobie z tego sprawy, a przecież coraz większa grupa urządzeń elektronicznych, z których korzystamy jest obsługiwana za pośrednictwem interfejsu graficznego. Być może łatwiej byłoby dzisiaj pokazać dziedzinę, która nie korzysta z grafiki komputerowej.

Wspomaganie prac inżynierskich:

- Projektowanie wspomagane komputerowo (*Computer Aided Design - CAD*).
- Komputerowe wspomaganie kreślenia i projektowania (*Computer Aided Drafting and Design - CADD*).
- Komputerowe wspomaganie procesu wytwarzania (*Computer Aided Manufacturing - CAM*).
- Komputerowo zintegrowana produkcja (*Computer Integrates Manufacturing - CIM*).
- Komputerowe wspomaganie działalności inżynierskiej (*Computer Aided Engineering – CAE*).

Komputer "uczestniczy" w *procesie inżynierskim produkcji wyrobu* (przemysłowego) na każdym etapie jego powstawania. Poczynając od pomysłu (wizji projektanta produktu – mająca postać np. wstępnych szkicy), poprzez szczegółowe modelowanie kształtu, utworzenie dokumentacji i przygotowanie warunków technologicznych, aż do sterowania obrabiarką numeryczną. Najistotniejsze jest to, że kolejne etapy projektowania i wytwarzania, są ze sobą powiązane. Wprowadzenie poprawek i uzupełnień nie stanowi żadnego problemu. Pozwala to znacznie uprościć proces zarówno projektowy jak i wytwarzania. Stosowana jest również tzw. *inżynieria odwrotna*. Na podstawie istniejącego, rzeczywistego obiektu jest tworzona pełna dokumentacja projektowa i technologiczna, która może posłużyć do dalszej obróbki. Na przykład zrobienia wiernej kopii.

Poligrafia i skład komputerowy:

- Jest to zestaw czynności związanych z przygotowaniem publikacji zrealizowany za pomocą systemów komputerowych zamiast tradycyjnymi metodami poligraficznymi (typograficznymi).
- Czynności te obejmują obróbkę tekstu i obrazów, łączenie ich w zamierzoną formę publikacji, a także prace związane z dostosowaniem barw do wykorzystywanych urządzeń drukujących.
- Podstawowy zakres takiego działania wykonują dzisiaj edytory tekstu komputerów osobistych.

Symulacja i wirtualna rzeczywistość:

- Współczesne symulatory lotu dają pełnię wrażeń obsługi rzeczywistego samolotu.
- Dzięki odpowiednio sterowanym podnośnikom hydraulicznym jest możliwość zasymulowania również zmian położenia, wstrząsów i przeciążeń.
- Dodając do tego inne wrażenia odczuwane przez pilota (np. hałas) oraz pełne wyposażenie kabiny można uzyskać wrażenie rzeczywistego lotu.

Oczywiście kluczowym zagadnieniem jest zapewnienie wrażeń wzrokowych. Jest to bardzo trudne zadanie stojące przed grafiką komputerowa, wymaga bowiem przygotowania skomplikowanych realistycznych wizualizacji w czasie rzeczywistym.

Architektura:
- Grafika komputerowa stała się nieocenionym narzędziem w pracowniach urbanistów i architektów. Także architektów krajobrazu i projektantów wnętrz.
- Dzisiaj projektując np. kuchnię swojego mieszkania można skorzystać z prostego oprogramowania, które pozwoli „zobaczyć" jak będzie ona wyglądała.

Medycyna:
- Bez tomografii komputerowej czy rezonansu magnetycznego współczesna medycyna nie byłaby w stanie postawić często właściwej diagnozy.
- Ale również badania USG a ostatnio nawet RTG w gabinecie stomatologicznym dostarcza wyników w postaci obrazu na monitorze zamiast tradycyjnej kliszy fotograficznej.

Tych narzędzi nie byłoby bez przetwarzania obrazów i grafiki komputerowej, ale również bez algorytmów dostarczanych przez geometrię obliczeniową.

Ivan Sutherland doktorantem MIT w latach sześćdziesiątych XX wieku, skonstruował pierwszą *stację graficzną* – kompletny system składający się monitora, urządzenia wskazującego (pióra świetlnego) i oprogramowania obsługi interaktywnej. Niestety zaproponowana przez niego nazwa (*sketch-pad)* nie przyjęła się. Później razem z *Davidem Evansem* założyli pierwszą firmę zajmującą się zastosowaniami grafiki komputerowej. W latach sześćdziesiątych i siedemdziesiątych ubiegłego stulecia na Uniwersytecie w Utah pracował pierwszy zespół naukowy zajmujący się grafiką komputerową. Powstało tam wiele podstawowych algorytmów stosowanych do dzisiaj, które weszły w skład geometrii obliczeniowej. Pracowali tam między innymi: *Ivan Sutherland*, *James Blinn*, *Edwin Catmull* i wiele innych osób których nazwiska są kojarzone jednoznacznie z określonymi algorytmami.

Piśmiennictwo: *Shirley P.* S.5.1., *Zabrodzki J.* Z.1.1.

4.8.1. GRAFIKI RASTROWA I WEKTOROWA

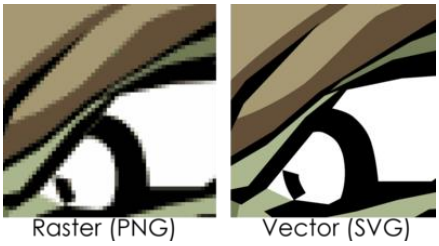
4.8.1.10. **Wyjaśnienie.** Rozróżniamy dwa typy pracy komputerowych urządzeń prezentujących obraz na siatce punktów (*rastrze*), czyli np. ekranie monitora:
1. *Wektorowy*, gdzie obraz przeznaczony do wyświetlenia przechowywany jest w postaci zależności funkcyjnych figury geometrycznej (przypadek 2D) albo bryły geometrycznej (przypadek 3D).
2. *Rastrowy*, gdy obraz przeznaczony do wyświetlania przechowywany jest w postaci tzw. mapy bitowej, będącą zero-jedynkowym zbiorem odpowiadającym siatce punktów urządzenia obrazującego.

4.8.1.20. **Wyjaśnienie.** Podstawową zaletą grafiki wektorowej, jest skalowalność obrazów. Dzięki temu można zdefiniować a następnie przechować rysunek, który nie będzie tracił jakości podczas np. obracania i skalowania. Natomiast po przeliczeniu go do konkretnej rozdzielczości, będzie mógł być wyświetlony na rastrowym monitorze lub wydrukowany na rastrowej drukarce.

4.8.1.30. **Wyjaśnienie.** W przeciwieństwie do grafiki rastrowej *grafika wektorowa* jest grafiką w pełni *skalowalną*, co oznacza, iż obrazy wektorowe można nieograniczenie powiększać oraz zmieniać ich proporcje bez uszczerbku na jakości. Ma to swoje uzasadnienie w matematycznym opisie elementów (prymitywów), dlatego też obraz może być wyświetlony w maksymalnie dostępnej dla ekranu czy wydruku rozdzielczości. Sama jakość obrazu uzależniona jest

wyłącznie od dokładności opisu obrazu przez prymitywy: czarne włosy rysowanej postaci można określić jako zamkniętą krzywą wypełnioną na czarno, choć można też opisać każdy włos krzywą o względnie niewielkiej grubości i czarnym kolorze.

W przypadku grafiki rastrowej obrót obrazu może zniekształcić go powodując utratę jakości (w szczególności, jeśli nie jest to obrót o wielokrotność kąta prostego). Typowe edytory grafiki wektorowej pozwalają oprócz zmiany parametrów i atrybutów prymitywów także na przekształcenia na obiektach, np.: obrót, przesunięcie, odbicie lustrzane, rozciąganie, pochylanie, czy zmiana kolejności obiektów na osi głębokości. Jest to więc kolejny stopień opisu obrazu ideowego, nie zaś literalnego.



Rysunek 4.8.1.0 Porównanie przykładów grafiki rastrowej i wektorowej.

Dwa tryby pracy urządzeń prezentujących obraz:

- *Rastrowy* (patrz rys. 4.8.1.0, część lewa) – możliwe położenia plamki są z góry ustalone i nie można na nie wpłynąć, można sterować, w pewnym zakresie jedynie jasnością (barwą) plamki.
- *Wektorowy* (patrz rys. 4.8.1.0, część prawa) – możliwe jest dowolne sterowanie jasnością (barwą) plamki.

Dowolne położenie, sekwencyjna struktura danych obrazu, zajętość pamięci zależna od skomplikowania struktury obrazu, możliwość skalowania, odbiór rysunku niezależny od urządzenia (linia jest zawsze gładka), możliwość operowania kreską, trudność w operowaniu plamą.

4.8.1.40. **Wyjaśnienie.** Wielkość obrazka rastrowego nie może zostać zwiększona bez zmniejszenia jego ostrości. Jest to przeciwnie niż w grafice wektorowej, którą łatwo można skalować, dostosowując jej wielkość do urządzenia, na którym jest wyświetlany obraz. Grafika rastrowa jest bardziej użyteczna od wektorowej do zapisywania zdjęć i realistycznych obrazów, podczas gdy grafika wektorowa jest częściej używana do obrazów tworzonych z figur geometrycznych oraz prezentacji tekstu (w tym tabel i wzorów).

4.8.1.50. **Wyjaśnienie.** Aktualnie większość komputerowych monitorów wyświetla od 72 do 130 pikseli na cal (ppi), podczas gdy drukarki mogą drukować materiały w rozdzielczości 1200 punktów na cal (dpi) lub wyższej. Ustalenie najbardziej właściwej rozdzielczości obrazka dla danej rozdzielczości drukarki może być bardzo trudne, gdyż dokument drukowany może zawierać większą liczbę detali (może mieć większą rozdzielczość) niż ten, który jest wyświetlany na ekranie monitora.

Piśmiennictwo: *de Berg M.* B.4.1., *Shirley P.* S.5.1., *Wikipedia* W.2.12., W.2.13., W.2.23., W.2.29., *Zabrodzki J.* Z.1.1.

4.8.2.10. **Wyjaśnienie.** Współrzędne jednorodne to sposób reprezentacji punktów *n*-wymiarowych za pomocą (*n+1*) współrzędnych. Współrzędne jednorodne zostały wprowadzone do geometrii w 1827 przez *Augusta Möbiusa* w pracy zatytułowanej *„Der barycentrische Calcul"*. Ze względu na kilka zalet znalazły zastosowanie w grafice komputerowej. Punkt w przestrzeni dwuwymiarowej (na płaszczyźnie) opisuje para liczb (*x, y*), we współrzędnych jednorodnych trójka (*x, y, W*); podobnie punkt trójwymiarowy we współrzędnych jednorodnych reprezentuje czwórka (*x, y, z, W*), itd. Jeśli współrzędna $W \neq 0$, wówczas można podzielić przez nią pozostałe współrzędne, np. (*x/W, y/W*). Liczby *x/W, y/W* (itd.) nazywane są *współrzędnymi kartezjańskimi punktu jednorodnego*. Jeśli natomiast $W = 0$, wówczas takie punkty nazywane są *punktami w nieskończoności* lub *niewłaściwymi*. Dwa punkty jednorodne reprezentują ten sam punkt wtedy, gdy jeden jest wielokrotnością drugiego; istnieje nieskończenie wiele reprezentacji jednego punktu. W przestrzeni jednorodnej punkt reprezentuje prostą przechodzącą przez środek układu współrzędnych, natomiast punkt we współrzędnych kartezjańskich jest rzutem środkowym na płaszczyznę $W = 1$.

4.8.2.20. **Wyjaśnienie.** W roku 1965 *L. Roberts* zauważył, że współrzędne jednorodne znakomicie nadają się do macierzowego opisu przekształceń w przestrzeniach n-wymiarowych. Podstawowymi przekształceniami stosowanymi w grafice komputerowej są: skalowanie, obrót, pochylenie i translacja. Zapis macierzowy wszystkich tych przekształceń przedstawia się następująco (przykład dla dwóch wymiarów):

4.8.2.21
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = A \cdot X + T = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} = \begin{bmatrix} a_{11}x + a_{12}y + t_x \\ a_{21}x + a_{22}y + t_y \end{bmatrix}$$

gdzie macierz *A* zawiera skumulowane informacje o obrocie, skalowaniu i pochyleniu, natomiast wektor *T* przesunięcie. Stosując współrzędne jednorodne można za pomocą macierzy 3x3 przedstawić powyższe przekształcenie:

4.8.2.22
$$\begin{bmatrix} x' \\ y' \\ W' \end{bmatrix} = A \cdot X = \begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ W \end{bmatrix} = \begin{bmatrix} a_{11}x + a_{12}y + t_xW \\ a_{21}x + a_{22}y + t_yW \\ W \end{bmatrix}$$

4.8.2.30. **Wyjaśnienie.** Macierz będąca iloczynem dowolnej liczby macierzy reprezentujących różne przekształcenia $x' = x \cdot M_1$, $x'' = x' \cdot M_2$, ..., $x^{(n)} = x^{(n+1)} \cdot M_n$ zawiera złożenie tych przekształceń. Dzięki temu zamiast osobno wykonywać kolejne przekształcenia, można najpierw wykonać mnożenie odpowiednich macierzy $M = M_1 \cdot M_2 \cdot \cdot \cdot M_n$, później zaś używać wynikowej macierzy $x' = x \cdot M$. Czyli zamiast wykonywać n - mnożeń punktu przez macierze, to samo uzyskuje się jednym mnożeniem punktu przez macierz, dzięki uprzedniemu wykonaniu (*n-1*) - mnożeń macierzy. Za pomocą macierzy przekształceń we współrzędnych jednorodnych można także zwięźle opisać rzut perspektywiczny:

$$M = M_1 \cdot M_2 \cdot \cdot \cdot M_n.$$

Za pomocą macierzy przekształceń we współrzędnych jednorodnych można także zwięźle opisać rzut perspektywiczny:

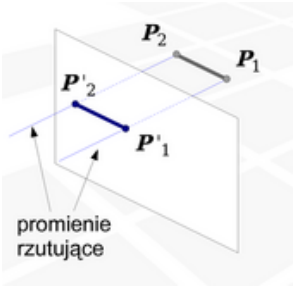
4.8.2.31
$$\begin{bmatrix} x' \\ y' \\ z' \\ W' \end{bmatrix} = M \cdot X = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ W \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ \frac{z}{d} \end{bmatrix}$$

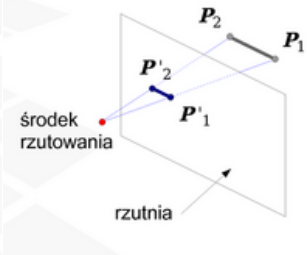
Przy przejściu na współrzędne kartezjańskie otrzymuje się:.

4.8.2.32 $$\left[\frac{x}{z/d}, \frac{y}{z/d}, d\right]^T$$

4.8.2.40. **Wyjaśnienie.** Jak już zostało powiedziane należy zaproponować pewien minimalny zestaw operacji, dla potrzeb grafiki - realizowanych przez macierz *M,* np.:
(1) operację odbicia symetrycznego (względem osi układu współrzędnych i środkowa),
(2) operację obrotu (również względem osi układu współrzędnych i środkowa),
(3) operację przesunięcie (translacja),
(4) operacji skalowanie oraz
(5) operację pochylenia.
Należy zauważyć, że operacja pochylenia jest rzadziej stosowanym przekształceniem, wprawdzie daje możliwość zniekształcenia figury, nie zachowuje odległości punktów, a w wyniku figura i jej obraz po tym przekształceniu nie są wzajemnie podobne.

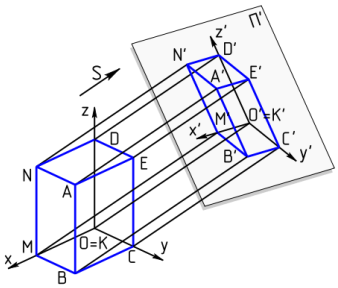
4.8.2.50. **Wyjaśnienie.** Rzut – w geometrii to odwzorowanie trójwymiarowej przestrzeni euklidesowej na daną powierzchnię zwaną rzutnią, które każdemu punktowi *x* przestrzeni przypisuje punkt przecięcia się z rzutnią pewnej prostej z danej rodziny prostych rzutujących przechodzącej przez punkt *x*. Natomiast rzutowanie – to operacja pozwalająca przedstawić obiekty trójwymiarowe na płaszczyźnie.





Rysunek 4.8.2.01. Rzutowanie równoległe Rysunek 4.7.2.02. Rzutowanie perspektywiczne

4.8.2.60. **Wyjaśnienie.** Rzutnia jest najczęściej płaszczyzną, choć stosuje się również rzuty na powierzchnię kuli, walca, stożka i inne. Rzut można także rozumieć jako funkcję odwzorowania płaszczyzny na pewną jej prostą (będącą rzutnią) i ogólniej jako funkcję odwzorowania n-wymiarową przestrzeń euklidesową na pewną jej hiperpłaszczyznę.



Rysunek 4.8.2.03. Rzutowanie bryły trójwymiarowej na płaszczyznę.

4.8.2.70. **Wyjaśnienie.** Rzutowanie jest przekształceniem przestrzeni trójwymiarowej na przestrzeń dwuwymiarową. Rzutowanie polega na poprowadzeniu prostej przez dany punkt obiektu i znalezieniu punktu wspólnego tej prostej z rzutnią. Wyznaczony punkt nazywany jest rzutem a prosta promieniem rzutującym. Powszechna definicja rzutu jako przekształcenia na

płaszczyznę jest pewnym uproszczeniem gdyż rozpatruje się też np. rzuty na powierzchnię walca lub na wycinek sfery. Jednak rzeczywiście z rzutowaniem na płaszczyznę mamy najczęściej do czynienia (grafika komputerowa, fotografia).

4.8.2.80. **Wyjaśnienie.** W zależności od definicji rodziny prostych rzutujących wyróżnia się:

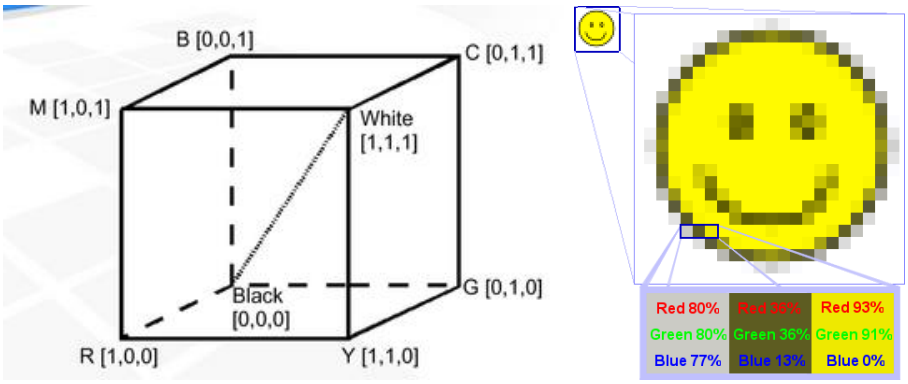
- Rzut równoległy – wszystkie proste rzutujące są równoległe do obranego kierunku:
    1. prostokątny – kierunek rzutowania jest prostopadły do rzutni
    2. ukośny – kierunek rzutowania nie jest prostopadły do rzutni
    3. aksonometryczny – kierunek rzutowania i orientacji rzutni dostosowana do kształtu rzutowanego obiektu.
- Rzut środkowy (perspektywa) – wszystkie proste rzutujące przechodzą przez pewien punkt zwany środkiem rzutu.

4.8.2.90. **Wyjaśnienie.** Rzutowanie równoległe zachowuje równoległość prostych oraz proporcje długości odcinków równoległych. Rzutowanie perspektywiczne pozwala uzyskać obraz zbliżony do postrzeganego przez człowieka. Trzeba jednak pamiętać o tym, że obraz na siatkówce oka powstaje w wyniku rzutu środkowego na wycinek sfery (w przybliżeniu). Zatem wszystkie promienie rzutujące będą w tym przypadku prostopadle padały na rzutnię. Oznacza to, że rzut na płaszczyznę będzie tylko przybliżeniem obrazu powstającego na siatkówce oka.

Piśmiennictwo: *de Berg M.* B.4.1., *Shirley P.* S.5.1., *Zabrodzki J.* Z.1.1.

4.8.3. Modele barw

4.8.3.10. **Wyjaśnienie.** Przestrzeń barw - widma fal elektromagnetycznych z zakresu od 380 do 780 ηm (tj. światło widzialne), których matematyczne modele są przedstawiane w trójwymiarowej przestrzeni barw. Dzięki tym modelom barwę można opisać nie tylko przez podanie jej widma, ale przez modele w różnym stopniu zbliżone do ludzkiej percepcji barwy, związanej z fizjologią oka ludzkiego, a szczególnie z występowaniem w siatkówce trzech rodzajów czopków. Najważniejsze przestrzenie barw ujęto w normach międzynarodowych. Stosuje się je w różnych dziedzinach przemysłu: farbiarskim, tekstylnym, spożywczym, fotografii itd. Utworzono różne modele przestrzeni barw, między innymi, dwa modele istotne z punktu widzenia informatyki: RGB i CMYK.



Rysunek 4.8.3.01. Model RGB

4.8.3.20. **Wyjaśnienie.** Model RGB. Jest to addytywny model barw, odzwierciedlający działanie świateł. Barwa opisywana jest przez intensywności każdej z barw podstawowych (Red, Green, Blue). Barwy opisywane są w sześcianie jednostkowym (patrz rys. 4.8.3.01). Model barw jest ukierunkowany na sprzęt wyświetlający typu: rzutnik, monitor.

Rysunek 4.8.3.02. Model CMY

4.8.3.30. **Wyjaśnienie.** Model CMY. Jest to model subtraktywnego mieszania barw oparty o barwy (Cyan – zielono-niebieska, Magenta – purpurowa, Yellow – żółta). Model ten został opracowany dla potrzeb poligrafii i wszystkich urządzeń wykorzystujących subtraktywne mieszanie barw (patrz rys. 4.8.3.02). CMY jest modelem analogicznym do RGB pod względem właściwości. Ze względu na technologiczne problemy uzyskania barwy czarnej mieszaniny zaproponowano dodanie barwnika czarnego (K). Wtedy można usunąć składową szarą G:

G = min(C,M,Y) - zmieniając wartości barw

Trzeba natomiast podkreślić, że różnice w jakości między np. obrazem wyświetlanym przez monitor, a tym samym obrazem wydrukowanym na drukarce nie wynikają, jak niektórzy sądzą z przejścia z RGB na CMYK, ale z możliwości technologicznych tych urządzeń. Praktycznie zawsze zakres barw dobrego monitora będzie szerszy niż zakres dobrej drukarki. Podobnie jak slajd fotografii tradycyjnej jest zawsze lepszy niż odbitka na papierze.

4.8.3.40. **Wyjaśnienie.** Model CMYK. W modelu CMY szarość jest otrzymywana przez zmieszanie równych ilości trzech barw podstawowych (c=m=y). W modelu CMYK jest ona generowana przez czwartą barwę podstawową K (blacK – czarny).

4.8.3.50. **Wyjaśnienie.** *Pixmap* (czyli *mapa pikseli*) – plik wykorzystujący rastrowy sposób reprezentacji komputerowej grafiki dwuwymiarowej polegający na określeniu położenia każdego piksela obrazu, oraz przypisaniu mu wartości określającej kolor w danym trybie koloru. *Pixmap*'ę, której elementy mogą przyjmować wartości binarne nazywamy bitmapą (mapą bitową). W węższym znaczeniu *pixmap*'ą bywa nazywany jedynie sam obraz (pozbawiony ewentualnej kompresji i innych elementów dołączonych do pliku, jak np. ścieżki, profile koloru, opis tekstowy pliku itp.).

4.8.3.60. **Wyjaśnienie.** W grafice całotonalnej (jednokolorowej) *pixmap* przypisuje jedynie kolejnym pikselom stan zapalony/wygaszony, co jest zapisywane w postaci jednego bitu. Taka grafika zwykle daje się dobrze kompresować metodami bezstratnymi (np. LZW, Huffman, CCITT).

4.8.3.70. **Wyjaśnienie.** W grafice wielotonalnej *pixmap* określa dokładny kolor (np. RGB, CMYK) i położenie każdego piksela, co przy braku kompresji powoduje, że jest ona bardzo dokładna i wyraźna (w porównaniu z np. formatem JPEG), ale pliki takie są bardzo duże. Mimo to nieskompresowana *pixmap* jest dzisiaj wykorzystywana bardzo często, zwłaszcza wewnątrz programów komputerowych, ponieważ nie wymaga przeliczania żadnych algorytmów, najczęściej do prostych tekstur.

4.8.3.80. **Wyjaśnienie.** Kompresja *pixmap* może być bezstratna (odwracalna), gdy po dekompresji uzyskuje się obraz identyczny z oryginałem, lub stratna, gdy z obrazu usuwane są pewne informacje w celu zmniejszenia objętości pliku. Metodami bezstratnej kompresji są np. RLE, używana w plikach BMP, LZW lub ZIP, używana również w plikach TIFF; deflate (LZ77) - w plikach PNG; czy tryb bezstratny JPEG 2000. Kompresję stratną osiąga się zwykle przez

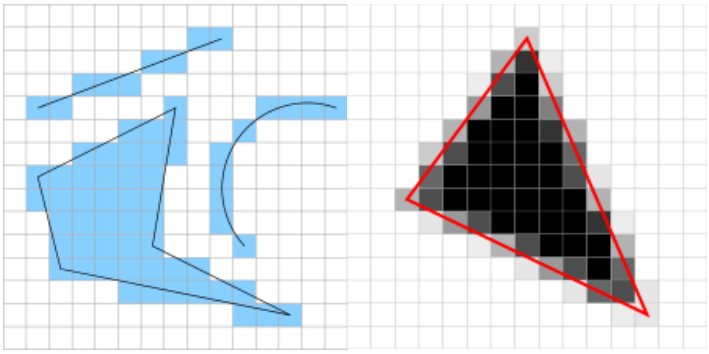
ograniczenie palety kolorów, dzięki czemu pojedynczy piksel da się opisać mniejszą liczbą bitów. Takie rozwiązanie oferują pliki GIF, a także stratny tryb TIFF. Inną formą kompresji stratnej jest usuwanie słabo zauważalnych szczegółów w celu poprawienia możliwości kompresji, jak w przypadku stopniowalnych algorytmów JPEG (DCT) czy JPEG 2000 (kompresja falkowa). Uproszczone dane zwykle są poddane dalszej, bezstratnej już kompresji (w przypadku JPEG algorytmem Huffmana, w przypadku GIF - LZW).

Piśmiennictwo: *Shirley P.* S.5.1., *Wikipedia* W.2.18., *Zabrodzki J.* Z.1.1.

4.8.4. Rasteryzacja krzywych płaskich

4.8.4.10. **Wyjaśnienie.** Rasteryzacja – w grafice komputerowej to działanie polegające na jak najwierniejszym przedstawieniu płaskiej figury geometrycznej na urządzeniu rastrowym, dysponującym skończoną rozdzielczością. Rasteryzacji mogą podlegać krzywe: odcinki, okręgi, elipsy, łuki eliptyczne, krzywe sześcienne, krzywe sklejane (np. Béziera), przekroje stożkowe oraz powierzchnie wielokątów, kół, powierzchnie zdefiniowane krzywymi sklejanymi itp.

4.8.4.20. **Wyjaśnienie.** Dla niektórych figur istnieją bardzo proste i efektywne algorytmy. W 1965 roku *Bresenham* opracował tzw. algorytm *Bresenhama* z punktem środkowym, działający na liczbach całkowitych, służący do konwersji dowolnych odcinków, okręgów oraz elips; algorytm był i jest implementowany sprzętowo. Podobnie rzecz ma się z wypełnianiem powierzchni – wypełnianie dowolnych wielokątów jest skomplikowane i kosztowne obliczeniowo, jednak istnieją bardzo efektywne algorytmy wypełniające trójkąty oraz czworokąty wypukłe, które są implementowane sprzętowo w akceleratorach 3D.

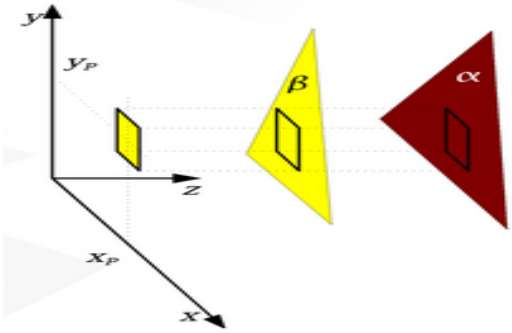


Rysunek 4.8.4.01. Przykład rasteryzacji odcinka, łuku oraz wielokąta.

4.8.4.30. **Wyjaśnienie.** W najprostszych przypadkach, gdy urządzenie rastrowe jest dwukolorowe, algorytm rasteryzujący zapala piksele leżące najbliżej krzywej/wewnątrz figury. Gdy docelowe urządzenie potrafi wyświetlać więcej kolorów (lub poziomów szarości), możliwe jest zastosowanie technik odkłócających (*antyaliasingu*), które powodują zniknięcie "schodków". W najogólniejszym przypadku intensywność koloru lub też stopień przezroczystości danego piksela jest proporcjonalny do pola powierzchni części wspólnej figury i piksela. Jeśli figura w całości pokrywa piksel, jego intensywność jest największa, gdy pokrywa tylko część – intensywność jest zmniejszana. Ta zależność może być liniowa (najmniejszy koszt obliczeniowy) albo nieliniowa (koszt większy, finalny efekt lepszy).

4.8.4.40. **Wyjaśnienie.** Sprawa trochę bardziej komplikuje się jeśli mamy do czynienia z grafiką trójwymiarową. Pierwszym etapem przygotowania obrazu 3D do wyświetlenia na ekranie jest

pozbycie się zbędnego - trzeciego – wymiaru danej bryły geometrycznej. Temu celowi służy wiele algorytmów, z których omówimy jeden – algorytm bufora głębokości.



Rysunek 4.8.4.02. Bufor głębokości zapamiętuje dla każdego punktu (x_p, y_p, z_p), współrzędną z_p.

Algorytm bufora głębokości (Z-bufora) został zaproponowany przez *Catmulla* w 1974 roku, o złożoności O(n), gdzie n-liczba przekrojów bryły równoległych do płaszczyzny XY – rzutu zobrazowania bryły. Jest jednym z najprostszych w implementacji algorytmów rozstrzygania widoczności. Algorytm zakłada istnienie bufora o rozmiarze całego wyświetlanego obszaru (ekranu) – występuje w tym przypadku odpowiedniość pozycji przyszłego piksela ekranu i odpowiadającej mu pozycji w buforze. Dla każdego punktu bryły geometrycznej w buforze zapamiętywana jest odpowiadająca mu głębokość czyli współrzędna *z*. Na początku pracy algorytmu bufor Z jest wypełniany maksymalną wartością współrzędnej $\boldsymbol{z_{max}}$, jaka może wystąpić w analizowanym obszarze. Jednocześnie wszystkie przyszłe piksele obrazu przyjmują barwę tła. Następnie rysowane są wielokąty (patrz rys. 4.8.4.02) – przekroje bryły geometrycznej, dla kolejnych wartości współrzędnej *z* oraz wypełniane są te przekroje odpowiednią barwą. Podczas wypełniania zwykła procedura wypełniająca jest poszerzona o sprawdzenie głębokości odpowiadającej danemu przyszłemu pikselowi. Przyszły piksel jest wypełniony tylko wtedy, kiedy jego współrzędna ***z*** ma wartość mniejsze - niż wartość zapisana w buforze. Mechanizm ten powoduje, że podczas wypełniania kolejnych wielokątów szukany jest punkt, którego współrzędna ***z*** jest najmniejsza – to znaczy szukany jest punkt leżący najbliżej obserwatora – czyli punkt rzeczywiście widoczny. Algorytm bufora głębokości rozstrzyga widoczność dla dowolnych scen wielościennych, jest niewrażliwy na zasłanianie cykliczne ani przecinanie obiektów. Nie wymaga żadnych dodatkowych struktur danych ani operacji wstępnych.

Dodatkowo, biorąc pod uwagę fakt, że wielokąty są obiektami płaskimi można wyznaczyć proste zależności przyrostowe wiążąc kolejne zmiany współrzędnych z wypełnianiem wielokąta liniami. Powoduje to, że dodatkowy czas potrzebny na rozszerzenie algorytmu wypełniania o rozstrzyganie widoczności jest niewielki. Przyjmuje się, że jest to algorytm o złożoności liniowej ze względu na liczbę wielokątów, ale dla ich dużej liczby czas rozstrzygania widoczności staje się w przybliżeniu praktycznie niezależny od liczby wielokątów w danej bryle widzenia. Ostatnim krokiem algorytmu, jest *rasteryzacja* płaskiej figury, do której sprowadzona została bryła po przekształceniu algorytmem bufora głębokości

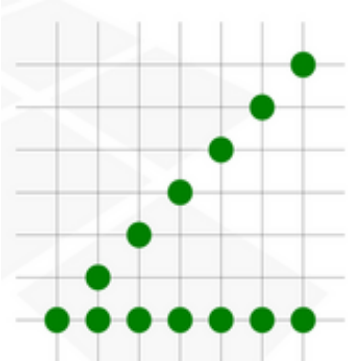
4.8.4.50. **Wyjaśnienie.** Algorytm ma tylko jedną wadą – potrzebuje pamięci o rozmiarze obrazu pozwalającej zapisać odległość dla każdego piksela. Jeszcze do niedawna (koniec dwudziestego wieku) był to warunek trudny do spełnienia. Realizacja algorytmu wymagała dodatkowo wykonania podziału obrazu na fragmenty, które mogły być razem z odpowiadającym mu buforem zmieszczone do pamięci. Dzisiaj nie stanowi to żadnego problemu. Prostota algorytmu

sprzyja również implementacji sprzętowej. Stacje graficzne i większość dobrych kart graficznych ma dzisiaj możliwość sprzętowej realizacji bufora głębokości.

Piśmiennictwo: *de Berg M.* B.4.1., *Cormen T.* C.5.1., *Shirley P.* S.5.1., *Zabrodzki J.* Z.1.1.

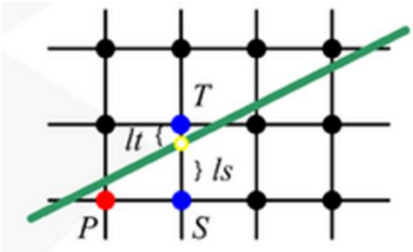
4.8.5. Przykłady algorytmów

Tworzenie obrazu na monitorze lub urządzeniu drukującym wymaga wypełnienia wybranego obszaru pikseli określonymi barwami w taki sposób, aby powstał zamierzony rysunek. Niestety korzystanie z rastra również i w przypadku rysowania odcinka może prowadzić do pewnych problemów. Pierwszym jest fakt, że odcinek może nie być symetryczny gdy zamienimy końce startowe. Problem ten wymaga korekty podejmowania decyzji dla d=0 w zależności od wzrostu lub spadku wartości współrzędnych. Poważniejszym problemem jest zmiana jasności. Oba odcinki na rysunku składają się z 7 punktów. Tylko że odcinek po przekątnej rastra jest $\sqrt{2}$ razy dłuższy od odcinka poziomego. Najprostszym rozwiązaniem wydaje się poprowadzenie prostej przez końce odcinka i opisanie jej równaniem. Następnie wyznaczenie wartości dla całkowitych wartości odpowiadających kolejnym kolumnom pikseli.



Rysunek 4.8.5.01.

4.8.5.10. **Wyjaśnienie.** Algorytm z punktem środkowym zaproponowany przez *Bresenhama* służy do rasteryzacji krzywych 2D, czyli jak najlepszego przybliżania matematycznych prostych oraz krzywych na siatce pikseli. Jego implementacja jest bardzo prosta i jednocześnie efektywna – np. ukośny odcinek przechodzi w pobliżu punktu P. Algorytm *Bresenhama* podejmuje decyzję bliżej, którego z punktów T czy S ma dalej przebiegać odcinek, w tym celu porównuje odległości punktów T oraz S od odcinka, czyli liczby l_t oraz l_s (patrz rys. 4.8.5.02). Jeśli teraz przybliżymy współrzędne do wartości całkowitej, to otrzymamy współrzędne dla kolejnych pikseli tworzących odcinek tzn., gdzie jest operacją zaokrąglania do najbliższej wartości całkowitej. Tak skonstruowany algorytm przyrostowy ma podstawową wadę. Wymaga operacji zmienno-pozycyjnych (mnożenia, dodawania, zaokrąglania).



Rysunek 4.8.5.02.

Na lewej części rys. 4.8.5.03 pokazany jest odcinek narysowany z pomocą algorytmu *Bresenhama*, na prawej części tego rysunku pokazane jest powiększenie tegoż odcinka, wyjaśniające co jest istotą przybliżania punktami środkowymi linii łamanej odcinka linii prostej.

4.8.5.20. **Wyjaśnienie.** Algorytm może działać zarówno na liczbach zmiennoprzecinkowych jak i całkowitych, ale ze względów praktycznych wykorzystuje się najczęściej realizacje całkowitoliczbowe. Siła algorytmu tkwi w jego prostocie, bowiem w głównej pętli algorytmu wykorzystywane są zaledwie dwie operacje: porównania i dodawania.

Rysunek 4.8.5.03. Oto przykładowy wynik działania algorytmu *Bresenhama* (rozmiar oryginalny po lewej, po prawej powiększenie).

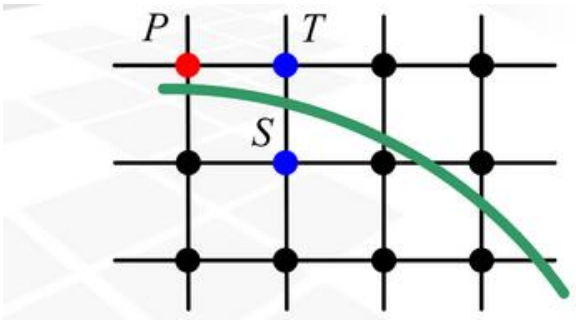
4.8.5.30. **Wyjaśnienie.** Algorytm rekursywny. Alternatywnym, dla algorytmu *Bresenhama*, jest algorytm rekursywny; jego kod bardzo prosty i krótki, oczywiście odcinek jest rasteryzowany prawidłowo.

```
def line_recursive(x0, y0, x1, y1):

  if x0 == x1 and y0 == y1:
    return

  x = (x0+x1)/2 # albo (x0+x1) >> 2
  y = (y0+y1)/2

  putpixel(x, y)

  line_recursive(x0,y0, x,y)
  line_recursive(x1,y1, x,y)
```

4.8.5.40. **Wyjaśnienie.** Rysowanie odcinka i łuku. Rozwiązanie oparte w całości na arytmetyce stałopozycyjnej zaproponował *Bresenham* w 1965 roku. Jest to algorytm przyrostowy, w którym jedna współrzędna np. x wzrasta w każdym kroku o 1, tzn. bez zmniejszania ogólności można przyjąć, że wystarczy opracować algorytm dla nachylenia odcinka od 0 do 1. Jest to 1/8 układu współrzędnych. Dla innych wartości można bowiem odpowiednio zamienić zmienne lub znaki przed nimi. Wzrost y w takim przypadku zależy od wyboru piksela (S lub T), który jest bliżej teoretycznej prostej. O wyborze decyduje zmienna kontrolna d, która jest uaktualniana w każdym kroku. Pełny algorytm *Bresenhama* dla x2 > x1, y2 > y1, 0 < m < 1.

```
procedure Bresline (x1,x2, y1,y2,)
begin
        dx := x2-x1; dy := y2-y1;
        p1 := 2*dy-dx; p2 := 2*(dy-dx);
        x := x1; y := y1;
        xend := x2;
        set_pixel(x,y);
        while (x<xend) do begin
                x := x+1;
                if (d<0) d := d+p1;
                else begin
                        d := d+p2;
                        y := y+1;
                end
                set_pixel(x,y);
        end
end
```

Jest to problem związany ze skończoną rozdzielczością rastra i podobnie jak usuwanie wrażenia schodków odcinka rozwiązuje się go zmieniając barwy pikseli sąsiednich. Na podobnej zasadzie jak rysowanie odcinka, *Bresenham* opracował algorytm rysowania łuku okręgu. Wychodząc z piksela *P* wybieramy piksel następny z pośród pikseli *S* lub *T*.



Rysunek 4.8.5.04. Wybór następnego punktu S oraz T przy kreśleniu łuku

Oczywiście zmieniają się w tym przypadku warunki wyboru pikseli, ale algorytm ten również wykorzystuje tylko operacje na liczbach całkowitych. Przy okazji rysowania okręgu warto zwrócić uwagę na parametr charakteryzujący proporcje boków rastra pikseli w pionie i w poziomie, czyli aspekt.

4.8.5.50. **Wyjaśnienie.** Jeśli proporcje rozdzielczości dla kart graficznych 1024x768 wynoszą 4:3, natomiast 1280x1024 wynoszą 5:4, to jeśli chcemy wyświetlić takie obrazy na tym samym monitorze, to proporcje trzeba przeliczyć także w odległości między pikselami. Inaczej narysowany okrąg albo w jednym albo w drugim przypadku będzie miał kształt elipsy.

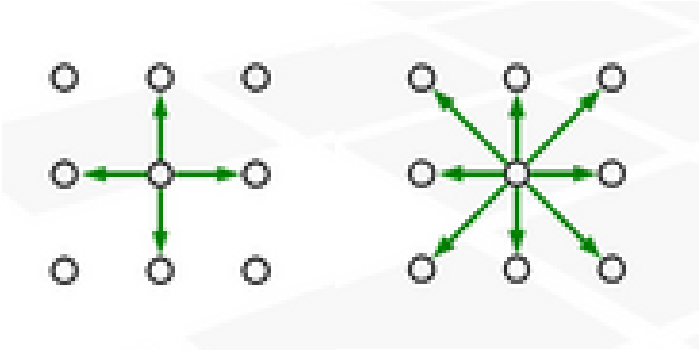
Wypełnianie obszaru jest kolejnym po rysowaniu odcinka lub łuku, najczęściej występującym problemem związanym z prymitywami. Zadanie dla szczególnych przypadków (np. dla prostokąta) jest zadaniem trywialnym. Natomiast w ogólnym przypadku algorytm powinien pracować poprawnie dla dowolnego wielokąta (także wklęsłego), również dla wielokątów z „dziurami". Można zauważyć, że jeśli wybierzemy jeden punkt, który jest wewnątrz wypełnianego obszaru, to punkt sąsiedni będzie również punktem wewnątrz albo będzie punktem brzegowym.

4.8.5.60. **Wyjaśnienie.** Wypełnianie przez spójność zakłada znajomość punktu startowego (tzw. „ziarna") wewnątrz obszaru. Punkt ten jest wypełniany, a następnie startując z niego wypełniamy punkty sąsiednie (jeśli oczywiście istnieją – jeśli nie są już wypełnione, ani nie są punktami granicznymi obszaru). Jednocześnie punkty sąsiednie stają się wyjściowymi dla wypełniania w następnym kroku. Procedura ta jest powtarzana dopóki można wskazać punkty wyjściowe (niewypełnione) wewnątrz obszaru. Algorytm wypełniania przez spójność dla siatki czterospójnej (patrz rys. 4.8.5.04). Przyjęto: c_b – barwa brzegu, c_f – barwa wypełnienia

```
procedure wypełnij1(x,y)
begin
        set_pixel(x,y,c_f);
        if (barwa(x-1,y) inna niż c_b i inna niż c_f) wypełnij1(x-1,y);
        if (barwa(x+1,y) inna niż c_b i inna niż c_f) wypełnij1(x+1,y);
        if (barwa(x,y-1) inna niż c_b i inna niż c_f) wypełnij1(x,y-1);
        if (barwa(x,y+1) inna niż c_b i inna niż c_f) wypełnij1(x,y+1);
end
```

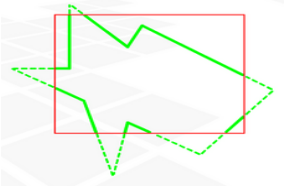
Zaproponowana postać algorytmu wypełniania przez spójność jest algorytmem rekurencyjnym. Daje to łatwość opisu ale także problemy realizacyjne. Z tego powodu znane są wersje niniejszego algorytmu w postaci nie-rekurencyjnej. Algorytm wypełniania przez spójność jest

czasem nazywany algorytmem przez sianie (punkt startowy jest „ziarnem”). Można również stosować wersję algorytmu dla siatek ośmiospójnych.



Rysunek 4.8.5.04. Siatki: cztero-spójna (lewa część) i ośmio-spójna (prawa część).

4.8.5.70. **Wyjaśnienie.** Wypełnianie przez kontrolę parzystości wykorzystuje pewną właściwość przecięcia brzegu linią prostą. Jeśli punkty przecięcia ponumerujemy kolejnymi liczbami naturalnymi zgodnie z orientacją prostej (na rysunku od lewej do prawej dla prostej poziomej) i jeśli będziemy poruszać się po prostej zgodnie z jej orientacją to każde nieparzyste przecięcie będzie „wejściem” do wnętrza obszaru, natomiast każde parzyste będzie „wyjściem” na zewnątrz. Zatem, aby wypełnić obszar należy go przecięć prostymi odpowiadającymi kolejnym rzędom pikseli, a następnie wypełnić odcinkami pomiędzy każdym nieparzystym przecięciem, a najbliższym parzystym.

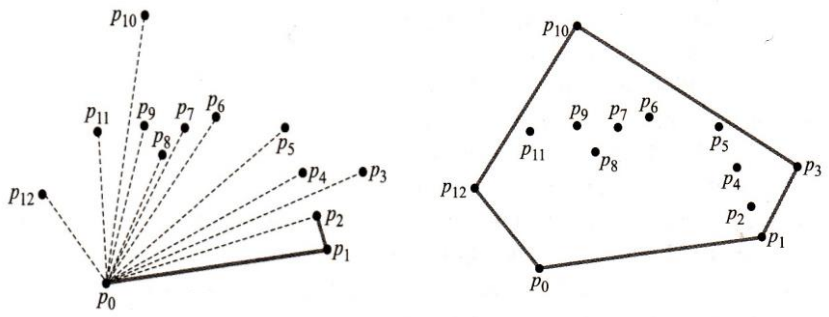


Rysunek 4.8.5.05. Obcinanie wieloboku do rozmiarów okna.

4.8.5.80. **Wyjaśnienie.** Algorytm *Cohena-Sutherlanda* (1974) służy do obcinania odcinków do prostokątnego okna. Oznacza to, że należy wybrać odcinki (lub ich fragmenty) do obcięcia na podstawie położenia ich końców.

Płaszczyzna została podzielona na 9 obszarów. Prostokąt centralny odpowiada obszarowi okna. Jednocześnie krawędzie okna wyznaczają cztery proste: prawą, lewą, górna i dolną. Każdemu obszarowi został przypisany czterobitowy kod. Kolejne bity kodu określają poziome i pionowe pasy. Operacja AND przeprowadzona na kodach końców odcinka pozwala odrzucić te odcinki, które na pewno są poza oknem. Spośród pozostałych odcinków należy wybrać te, które rzeczywiście mają wspólne punkty z oknem oraz przyciąć do jego rozmiaru. Operacja AND pozwala w tym przypadku wybrać prostą obcinającą.

4.8.5.90. **Wyjaśnienie.** W algorytmie *Grahama* problem otoczki wypukłej jest rozwiązywany z użyciem stosu S, który zawiera kandydatów na wierzchołki otoczki. Każdy punkt z wejściowego zbioru Q jest raz wkładany na stos, a punkty niebędące wierzchołkami CH(Q) są w końcu ze stosu zdejmowane. Po zakończeniu działania algorytmu stos S zawiera wyłącznie wierzchołki otoczki CH(Q) w przeciwnej do ruchu wskazówek zegara kolejności ich występowania na brzegu.



Rysunek 4.8.5.06. Algorytm Grahama

```
GRAHAM-SCAN(Q)
niech p0 będzie punktem w Q o minimalnej współrzędnej y lub,
    w przypadku remisu, pierwszym takim punktem z lewej
niech ⟨p1, p2, ..., pm⟩ będą pozostałymi punktami Q, posortowanymi
    ze względu na współrzędną kątową względem p0 w kierunku
    przeciwnym do ruchu wskazówek zegara (jeśli więcej niż
    jeden punkt ma taką samą współrzędną kątową, usuń wszystkie
    oprócz położonego najdalej od p0)
if m < 2
    return „otoczka wypukła jest pusta"
else niech S będzie pustym stosem
    PUSH(p0, S)
    PUSH(p1, S)
    PUSH(p2, S)
    for i = 3 to m
        while kąt utworzony przez punkty NEXT-TO-TOP(S),
                TOP(S) i pi nie stanowi skrętu w lewo
            POP(S)
        PUSH(pi, S)
    return S
```

Dane wejściowe dla procedury GRAHAM-SCAN stanowi zbiór punktów Q, gdzie |Q| > 3. Procedura korzysta z funkcji TOP(S), której wynikiem jest element na szczycie stosu S bez zmiany S, oraz NEXT-TO-TOP(S) dającej w wyniku element położony o jedno miejsce poniżej szczytu stosu, bez zmiany stosu S. Jak zaraz pokażemy, stos S obliczany przez procedurę GRACHAM-SCAN zawiera, patrząc od dołu do góry, wszystkie wierzchołki otoczki CH(Q) w kolejności przeciwnej do ruchu wskazówek zegara.

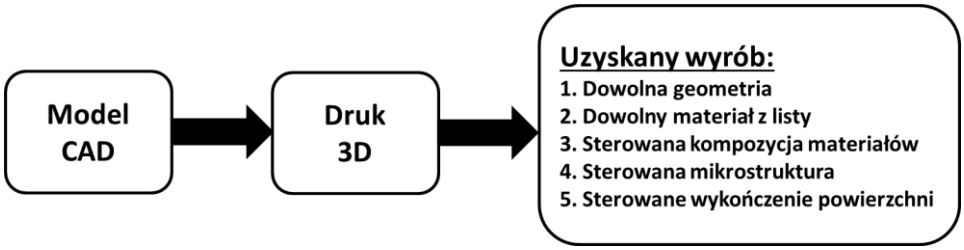
Piśmiennictwo: *de Berg M.* B.4.1., *Cormen T.* C.2.1., *Shirley P.* S.5.1., *Zabrodzki J.* Z.1.1.

4.8.6. DRUKARKI 3D I ROBOTYKA

4.8.6.10. **Wyjaśnienie.** Drukarka 3D oraz roboty w sposób istotny różnią się od dotychczas omawianej problematyki geometrii obliczeniowej i grafice komputerowej. Dotychczas zajmowaliśmy się albo figurami 2D, albo odwzorowaniem brył 3D na płaszczyźnie 2D, a następnie transformowaniem na raster ciągłego opisu 2D uzyskanej figury geometrycznej. Natomiast w przypadku druku 3D i robotyki, mamy do czynienia, z jakościowo inną transformacją. Z jednej strony mamy bryłę geometryczną, np. zaprojektowaną metodą CAD (*Computer Aided Design*), a z drugiej strony raster 3D według którego, odbywa się przemieszczanie: głowicy nakładającej warstwę (przypadek druku 3D), albo ramienia robota (przypadek robotyki).

4.8.6.20. **Wyjaśnienie.** Drukowanie przestrzenne (*3D printing*) – proces wytwarzania trójwymiarowych, fizycznych obiektów na podstawie komputerowego modelu. Początkowo była

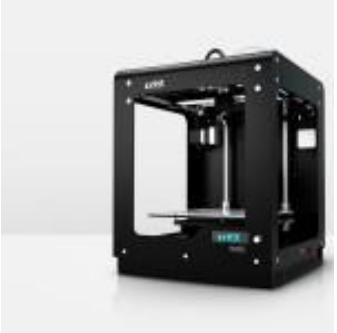
to jedynie jedna z metod szybkiego prototypowania używana zarówno do budowania form i samych prototypów. Wraz z postępami dokładności wykonania obiektów przez drukarki 3D, stała się to także metoda wykonywania gotowych obiektów, w tym zabawek, ubrań, czekoladek, a nawet protez.



Rysunek 4.7.6.01. Schemat wytwarzania wyrobu przy użyciu druku 3D.

4.8.6.30. **Wyjaśnienie.** Robot przemysłowy – jest automatycznie sterowaną, programowaną, wielozadaniową maszyną manipulacyjną stacjonarną lub mobilną, o wielu stopniach swobody, posiadającą właściwości manipulacyjne lub lokomocyjne, dla ważnych zastosowań np. przemysłowych. Roboty przemysłowe stosuje się w celu zastąpienia ludzi w pracy na stanowiskach uciążliwych i niebezpiecznych. Najczęściej wykonują one zadania ryzykowne (np. obsługa prasy lub praca w środowisku agresywnym chemicznie), monotonne (np. obsługa taśmy produkcyjnej), czy wymagające dużej siły fizycznej (np. rozładunek, załadunek), bądź wyjątkowej precyzji (np. zaawansowana obróbka materiałowa).

4.8.6.40. **Wyjaśnienie.** Robotyka – interdyscyplinarna dziedzina wiedzy działająca na styku mechaniki, automatyki, elektroniki, sensoryki, cybernetyki oraz informatyki. Domeną robotyki są również rozważania nad sztuczną inteligencją – w niektórych środowiskach robotyka jest wręcz z nią utożsamiana.



Rysunek 4.8.6.02. Drukarka 3D: Zortrax M200

Należy zauważyć, że w Polsce w 2013 roku, opracowano drukarkę 3D – sfinansowaną za pośrednictwem wiyryny *Kickstartera*. Pomysłodawcy drukarki 3D nazwanej „Zortrax M200" zaprojektowali urządzenie, które sprawiło, że druk 3D będzie pożądany nie tylko przez specjalistów, ale też szerokiego grona zwykłych entuzjastów. M200 do druku wykorzystuje jako materiał nakładany cienkimi warstwami: ABS, PC-ABS lub nylon. Posiada też łatwe w obsłudze oprogramowanie. Sama drukarka wykonana jest z aluminium, POM-u, DryLinu, FR4, stali nierdzewnej i mosiądzu. Dołączone oprogramowanie ułatwia projektowanie obiektów 3D tak, by te po wykonaniu się nie rozpadły. To ważne, biorąc pod uwagę to, że M200 posiada tylko jedną wytłaczarkę. Na początku 2014 roku, firma Dell zamówiła u polskiego producenta z Olsztyna Zortrax 5000 drukarek 3D model M200.

Piśmiennictwo: *Wikipedia* W.2.31.

4.9. URZĄDZENIA SZYFRUJĄCE I KRYPTOGRAFIA

4.9.1. UWAGI O KRYPTOGRAFII

Jak się powszechnie uważa, kryptografia to nauka o metodach szyfrowania (ukrywania) informacji. Związane z nią zagadnienia mieszczą się w obrębie zarówno informatyki, jak i matematyki, Istnieje wiele metod kryptograficznego zabezpieczania informacji, między innymi używanie haseł, biometryk lub innych urządzeń, szyfrowanie danych za pomocą algorytmów, używanie kluczy itp.

4.9.1.10. **Wyjaśnienie.** Szyfrowanie oznacza użycie funkcjonału przekształcania informacji na postać danych, które tracą swój kontekst. Deszyfrowanie to użycie funkcjonału odwrotnego, to znaczy przekształcanie danych z powrotem na postać informacji, które mogą być odczytane i zrozumiałe. Oba funkcjonały, czyli dwa algorytmy odpowiedzialne za szyfrowanie i deszyfrowanie, nazywamy kodowaniem. Niektóre systemy kodowania wymagają używania kluczy, który stanowi informację wykorzystywaną do modyfikacji operacji kodowania.

4.9.1.20. **Wyjaśnienie.** Klucz – w kryptografii informacja umożliwiająca wykonywanie pewnej czynności kryptograficznej – szyfrowania, deszyfrowania, podpisywania, weryfikacji podpisu itp.

4.9.1.30. **Wyjaśnienie.** Kryptografia symetryczna – w algorytmach symetrycznych klucz służy do szyfrowania i deszyfrowania wiadomości. Do obu tych czynności używa się tego samego klucza, dlatego powinien być znany tylko uczestnikom. Taki klucz jest przypisany do danej komunikacji, nie do posiadacza, dlatego zwykle do każdego połączenia jest generowany nowy klucz.

4.9.1.40. **Wyjaśnienie.** Kryptografia asymetryczna – w algorytmach wyróżniamy parę kluczy: *publiczny* oraz *prywatny*. Ten pierwszy może być zupełnie jawny, drugi powinien znać tylko właściciel. Konstrukcja kluczy powinna być taka, żeby wygenerowanie prywatnego klucza na podstawie publicznego klucza, było jak najtrudniejsze obliczeniowo. Dwie najważniejsze funkcje kryptografii asymetrycznej to:

(1) *szyfrowanie* – wtedy klucz *publiczny* służy do szyfrowania, a *prywatny* do deszyfrowania;
(2) *podpisy cyfrowe* – klucz *prywatny* służy do generacji podpisu, klucz *publiczny* zaś do weryfikacji podpisu.

Wydaje się, że dwa najważniejsze wydarzenia w rozwoju kryptografii związane z informatyką, to:

- zbudowanie maszyny szyfrującej w 1918 roku oraz
- wynalezienie algorytmów z parą kluczy publicznym i prywatnym.

4.9.1.50. **Wyjaśnienie.** Trzy najlepiej znane algorytmy kryptograficzne stosowane w informatyce to:

1. *Data Encryption Standard* (DES). Opracowany przez firmę IBM i wybrany w 1956 roku przez Narodowe Biuro Standardów jako oficjalny Federalny Standard Przetwarzania Informacji (Federal Information Processing Standard - FIPS) rządu USA. Algorytm DES używa algorytmu klucza symetrycznego oraz klucza 56-bitowego. Obecnie DES nie jest uznawany za bezpieczny, ale różne jego odmiany, na przykład 3DES i następcy *Advanced Encryption Standard* (AES), pozostają w powszechnym użyciu.
2. Algorytm *Diffie-Hellman Key Agreement* jest algorytmem uzgadniania kluczy, który pozwala na otrzymanie przez dwie strony tej samej liczby, niemożliwej do odgadnięcia przez

podsłuchanie. Algorytm D-H został opublikowany po raz pierwszy w roku 1976 przez *Whitfielda Diffiego* oraz *Martina Hellmana* i bazował na rozwiązaniu polegającym na użyciu rozprowadzania klucza publicznego. Rozwiązanie to zostało opracowane przez *Ralpha Merkle'a* z wielkiej Brytanii i było utrzymywane w tajemnicy do roku 1997. Z tego powodu czasami (naprawdę rzadko) można się spotkać z określeniem tej metody jako *Diffie-Hellman-Merkle*. *Diffie* jest teraz szefem bezpieczeństwa w firmie Sun Microsystems (przejętej w roku 2009 przez Oracle).

3. Algorytm klucza publicznego RSA bazuje na rozwiązaniu *Roniego Rivasta*, *Adiego Shamira* oraz *Leonarda Adlemana* z MIT i zostało opublikowane w roku 1983, a opatentowane również w roku 1983. Algorytmy RSA obejmują generowanie kluczy, szyfrowanie i deszyfrowanie za pomocą pary kluczy publicznego i prywatnego. Klucz publiczny jest używany do szyfrowania danych, które mogą zostać odszyfrowane jedynie za pomocą klucza prywatnego.

4.9.1.60. **Wyjaśnienie.** Podstawowe zastosowania kryptografii w informatyce, to:

1. Zapewnienie integralności komunikatów.
2. Uwierzytelnianie punktów końcowych.
3. Używanie kryptograficznych funkcji skrótu.
4. Stosowanie podpisu cyfrowego.
5. Certyfikowanie kluczy publicznych.
6. Zabezpieczanie poczty elektronicznej.
7. Zabezpieczanie połączeń TCP (protokół SSL).
8. Zabezpieczanie w warstwie sieci – IPsec i sieci VPN.
9. Zabezpieczanie bezprzewodowych sieci lokalnych.

Piśmiennictwo: *Kurose J*. K.8.1., *Sosińsky B. S.9.1.*

4.9.2. Zarys działania algorytmu RSA

Algorytm RSA wykonuje wiele operacji arytmetycznych z wykorzystaniem arytmetyki modulo-*n*. Opiszmy pokrótce, za *J. Kurose* i *K. Rossem* - podstawowe własności arytmetyki modularnej. Przypominamy, że x mod n oznacza po prostu resztę z dzielenia x przez n. Np. 19 mod 5 = 4. W arytmetyce modularnej wykonywane są normalne działania dodawania, mnożenia i potęgowania, przy czym wynik każdej operacji jest zastępowany przez całkowitoliczbową resztę z dzielenie go przez n. Dodawanie i mnożenie w arytmetyce modularnej ułatwiają poniższe wygodne zależności:

4.9.2.10 $$[(a \bmod n) + (b \bmod n)] \bmod 1 = (a + b) \bmod n$$

4.9.2.11 $$[(a \bmod n) - (b \bmod n)] \bmod n = (a - b) \bmod n$$

4.9.2.12 $$[(a \bmod n) * (b \bmod r)] \bmod n = (a * b) \bmod n$$

Z zależności 4.9.2.12 - wynika, że

4.9.2.13 $$(a \bmod n)^d \bmod n = a^d \bmod n.$$

Tożsamość 4.9.2.13 - okaże się dalej bardzo przydatna.

Przy analizie algorytmu RSA zawsze warto pamiętać o tym, że wiadomość to nic więcej jak wzorzec bitów, a każdy taki wzorzec ma niepowtarzalną reprezentację w postaci liczby całkowitej oraz długości wzorca. Załóżmy np., że komunikat to wzorzec bitów 1001. Można go przedstawić jako dziesiętną liczbę całkowitą 9. Dlatego szyfrowanie wiadomości za pomocą algorytmu RSA odpowiada szyfrowaniu niepowtarzalnych liczb całkowitych reprezentujących komunikat.

4.9.2.20. **Wyjaśnienie.** RSA posiada dwa powiązane ze sobą aspekty: (1) wybór klucza publicznego i klucza prywatnego; (2) algorytm szyfrowania i deszyfrowania.

4.9.2.30. **Wyjaśnienie.** Aby wybrać klucz publiczny i prywatny, należy wykonać następujące czynności:

1. Wybrać dwie duże liczby pierwsze p i q. Jak duże powinny być te liczby? Im są większe, tym trudniej złamać RSA, ale tym dłużej trwa kodowanie i dekodowanie. Firma RSA Laboratories zaleca, aby iloczyn p i q miał długość rzędu 1024 bitów. Sposoby znajdowania dużych liczb pierwszych można znaleźć w Wikipedii.
2. Obliczyć $n = p*q$ oraz $z = (p - 1)*(q - 1)$.
3. Wybrać liczbę e mniejszą niż n, która nie ma wspólnych dzielników (poza jedynką) z liczbą z (w takim przypadku mówi się, że liczby e i z są względnie pierwsze). Uwaga: użyto litery e, ponieważ wartość e będzie służyć do szyfrowania (*encryption*).
4. Znaleźć liczbę d taką, że $e*d$ -1 dzieli się bez reszty przez z. Użyto litery d, ponieważ wartość ta będzie służyć do deszyfrowania (*description*). Innymi słowy, dysponując wartością e, wybieramy liczbę d taką, aby: $e*d \bmod z = 1$
5. Klucz publiczny K^+, który udostępnia się całemu światu, jest parą liczb (n, e); klucz prywatny K^- zaś, jest parą liczb (n, d).

Przykład użycia RSA: osoba A szyfruje, a osoba B odszyfrowuje wiadomość w następujący sposób:

- Przypuśćmy, że A chce wysłać B wzorzec bitowy reprezentowany przez liczbę całkowitą m taką, że $m < n$. Aby ją zakodować, A używa klucza publicznego (n, e) oblicza m^e, a następnie oblicza resztę z dzielenia m^e przez n. Zatem zaszyfrowana wartość c tekstu jawnego m wysyłana przez A to $c = m^e \bmod n$. Wzorzec bitowy odpowiadający szyfrogramowi c jest wysyłany do B.
- Aby odszyfrować odebrany szyfrogram c, B oblicza: $m = c^d \bmod n$ co, wymaga użycia jego klucza prywatnego (n, d).

Rozwińmy dalej nasz prosty przykład użycia RSA. Przypuśćmy, że B wybiera dwie liczby pierwsze $p = 5$ $q = 7$ (oczywiście liczby te są zbyt małe, aby były bezpieczne). W takim przypadku $n = 35$, a $z = 24$. B wybiera $e = 5$, ponieważ 5 i 24 nie mają wspólnych dzielników. Wreszcie B wybiera $d = 29$, ponieważ 5*29 - 1 (tzn. $e*d$ - 1) dzieli się bez reszty przez 24. B upublicznia te dwie wartości ($n = 35$ i $e = 5$), ale nie ujawnia wartości $d = 29$. Znając dwie publiczne wartości, A chce wysłać do B litery „l", „o", „v" i „e". Interpretując każdą z nich jako wartość z zakresu 1 - 26 (według pozycji alfabetycznej: „a" to 1, „z" to 26), A i B szyfrują i deszyfrują wiadomość w sposób przedstawiony w tabelach 4.8.2.31 oraz 4.8.2.32.

Zauważmy, że w tym skrajnie prostym przykładzie, każda z czterech liter jest traktowana jak odrębny komunikat. Bardziej realistyczne rozwiązanie wymaga przekształcenia liter na ich 8-bitowe reprezentacje w formacie ASCI i zaszyfrowanie liczby całkowitej odpowiadającej

uzyskanemu 32-bitowemu wzorcowi. Jednak taki realistyczny przykład prowadzi do uzyskania liczb, które są zdecydowanie zbyt długie!

Tabela 4.9.2.31. Szyfrowanie RSA			
Litera tekstu jawnego	m: reprezentacja liczbowa	m^e	Szyfrogram $c=m^e \bmod n$
l	12	238832	17
o	15	759375	15
v	22	5153632	22
e	5	3125	10

Tabela 4.9.2.32. Deszyfrowania RSA			
Szyfrogram cc	c^d	$m = c^d \bmod n$	Litera tekstu jawnego
17	4B196857210675091509141182522307169 7	12	l
15	127834039488589391112327575683593 75	15	o
22	851643319086537701956194499721106030592	22	v
10	10000000000000000000000000000	5	e

Zważywszy, że już skrajnie prosty przykład z tabel 4.8.2.31 i 4.8.2.32 - doprowadził do powstania bardzo dużych liczb, a wcześniej dowiedzieliśmy się, że wartości p i q powinny liczyć kilkaset bitów, trzeba zastanowić się nad praktycznymi aspektami RSA. Jak wybiera się duże liczby pierwsze? Jak następnie wybiera się e i d? Jak wykonuje się potęgowanie, kiedy wykładnikami są duże liczby? Omówienie tych ważnych kwestii wykracza poza ramy niniejszej książki; szczegóły można znaleźć w piśmiennictwie oraz wymienionych tam źródłach.

4.9.2.40. **Wyjaśnienie.** Szyfrowanie i deszyfrowanie RSA wydaje się niemal magiczne. Dlaczego zastosowanie opisanych wyżej algorytmów pozwala odtworzyć oryginalną wiadomość? Aby zrozumieć, jak działa RSA, przyjmiemy $n = p*q$, gdzie p i q są dużymi liczbami pierwszymi używanymi w algorytmie RSA. Przypomnijmy, że w szyfrowaniu RSA wiadomość m (reprezentowana przez liczbę całkowitą) jest najpierw podnoszona do potęgi e z wykorzystaniem arytmetyki modulo n:

4.9.2.41 $$c = m^e \bmod n.$$

Deszyfrowanie polega na podniesieniu c do potęgi d, znów z wykorzystaniem arytmetyki modulo n. Wynik szyfrowania, po którym następuje operacja deszyfrowania, można zatem zapisać jako:

4.9.2.42 $$c^d \bmod n = (m^e \bmod n)^d \bmod n.$$

Zobaczmy, co możemy powiedzieć o tej wartości. Jak wcześniej wspomnieliśmy, jedną z ważnych cech arytmetyki modularnej jest, że

4.9.2.43 $(a \bmod n)^d \bmod n = a^d \bmod n$ - dla dowolnych wartości a, n i d.

Dlatego, wykorzystując zależność $a = m^e$ z tej właściwości, otrzymujemy:

4.9.2.44 $$(m^e \bmod n)^d \bmod n = m^{ed} \bmod n.$$

Pozostaje więc wykazać, że

4.9.2.45 $$m^{ed} \bmod n = m.$$

Choć próbujemy wyeliminować magię z algorytmu RSA, będziemy musieli teraz posłużyć się dość magicznym twierdzeniem z teorii liczb. Mówiąc ściślej, udowodniono, że jeśli p i q są liczbami pierwszymi, a

4.9.2.46 $n = p*q$ i $z = (p - 1)*(q - 1)$, to $x^y \bmod n$ jest równe $x^{(y \bmod n)} \bmod n$.

Stosując twierdzenie 4.8.2.46 dla $x = m$ i $y = e*d$ otrzymamy:

4.9.2.46 $m^{ed} \bmod n = m^{(ed \bmod z)} \bmod n$.

Jak jednak wiemy, wybraliśmy takie wartości e i d, że $e*d \bmod z = 1$. Otrzymujemy więc:

4.9.2.47 $m^{ed} \bmod n = m^1 \bmod n = m$

Właśnie takiego wyniku oczekiwaliśmy! Najpierw podnosząc wartość do potęgi e (tzn. szyfrując), a następnie podnosząc ją do potęgi d (tzn. deszyfrując), otrzymaliśmy pierwotną wartość m. Jeszcze bardziej zdumiewające jest to, że najpierw podnosząc wartość do potęgi d, a później do potęgi e - tzn. odwracając kolejność szyfrowania i deszyfrowania - również uzyskujemy pierwotną wartość m! Te wspaniałe zależności wynikają bezpośrednio z arytmetyki modularnej:

4.9.2.48 $(m^d \bmod n)^e \bmod \underline{n} = m^{de} \bmod n = m^{ed} \bmod n = (m^e \bmod n)^d \bmod n$.

4.9.2.50. **Wyjaśnienie.** Bezpieczeństwo RSA wynika z tego, że nie są znane żadne szybkie algorytmy dzielenia liczby (w tym przypadku publicznej wartości n) na czynniki pierwsze p i q. Gdybyśmy potrafili ustalić wartości p i q, to znając publiczną wartość e, moglibyśmy łatwo obliczyć składową tajnego klucza d. Z drugiej strony, nie wiadomo, czy nie istnieje jakiś szybki algorytm dzielenia liczb na czynniki pierwsze, więc w tym sensie bezpieczeństwo RSA nie jest gwarantowane.

Piśmiennictwo: *Kurose J*. K.8.1.

4.9.3. Historia odtworzenia Enigmy przez polski wywiad

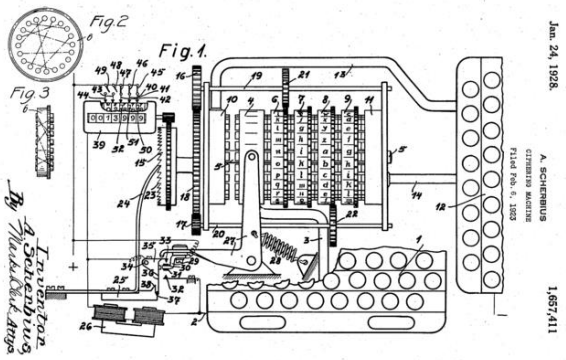
Jak podają różne źródła internetowe, w tym Wikipedia: ≪W 1918 roku niemiecki wynalazca *Arthur Scherbius* opatentował maszynę szyfrującą, nazwaną później *Enigmą* (z greckiego ainigma - tajemnica). Początkowo armia niemiecka nie była zainteresowana zamianą powszechnego w tych czasach kodu ręcznego na maszynę szyfrującą.

Jednakże fakt, iż służby Królestwa Brytyjskiego regularnie czytały depesze niemieckie w czasie I wojny światowej, spowodował, że dowództwo niemieckie zdecydowało się na wprowadzenie kodu maszynowego, stanowiącego pełną gwarancję zachowania bezpieczeństwa przekazywanych informacji. Po raz pierwszy na wyposażeniu armii niemieckiej, *Enigma* pojawiła się w 1926 roku, najpierw w marynarce wojennej, a dwa lata później w siłach lądowych. W kolejnych latach, podczas których Niemcy coraz intensywniej szykowali się do działań wojennych, *Enigma* przechodziła kilkakrotne modyfikacje, często bardzo znacznie zmniejszające szanse ewentualnego jej dekryptarzu.

W końcowym efekcie suma wszystkich jej kombinacji wynosiła czterysta sekstylionów, czyli czwórka i dwadzieścia sześć zer. *Marian Rejewski*, jeden z polskich matematyków, którzy złamali szyfr *Enigmy*, następująco zobrazował te liczbę: "jest ona większa od liczby sekund, jakie upłynęły od początku świata, jeśli przyjąć, że świat istnieje od pięciu miliardów lat". Dzięki temu uważana była za najdoskonalszą maszynę szyfrującą na świecie.≫ …

≪Enigma posiadała klawiaturę, taką jak maszyna do pisania, ale zamiast jednego zbioru liter (czcionek), posiadała trzy zbiory (a potem więcej) liter, rozmieszczonych na bębenkach. Naciśnięcie klawisza z literą, np. "a", uruchamiało na pierwszym bębenku literę odmienną do "a", tej z kolei odpowiadała inna litera na drugim bębenku i jeszcze inna na trzecim bębenku. Po każdym naciśnięciu klawisza bębenki obracały się i po ponownym naciśnięciu klawisza "a" odpowiadała mu już inna kombinacja liter na bębenkach. Polscy kryptolodzy słusznie domyślali się, że Niemcy zastosowali maszyny szyfrujące, odkąd nieczytelne okazały się dla nich depesze Kriegsmarine oraz korespondencja Wehrmachtu. Na zwróceniu uwagi na *Enigmę* dopomógł im

przypadek. W 1928 roku wywiad wojskowy został powiadomiony przez Urząd Celny o przesyłce z Niemiec, której zwrotu, jako nadanej omyłkowo, natarczywie żądał, koniecznie przed dokonaniem odprawy celnej, nadawca. W przesyłce miał być sprzęt radiowy, lecz po jej otwarciu ukazała się maszyna szyfrująca.



Rysunek 4.9.3.01. Opis patentu amerykańskiego z 1928 roku na maszynę *Enigma*, uzyskanego przez niemieckiego inżyniera *Artura Scherbiusa*.

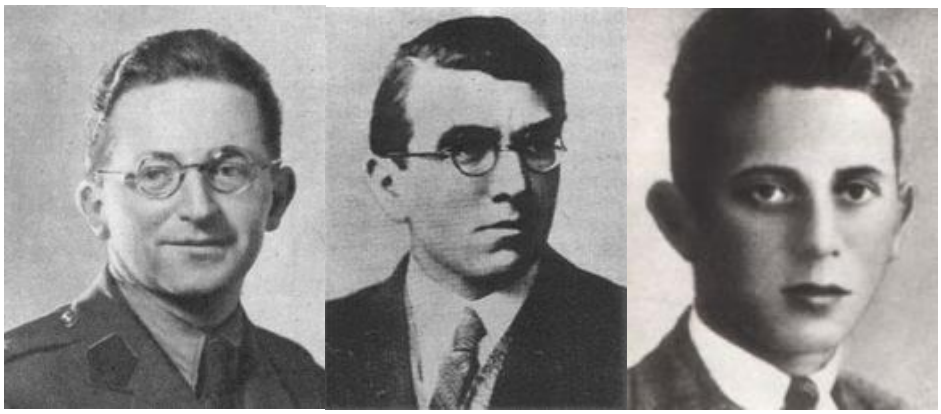
Biuro Szyfrów zakupiło jeden jej egzemplarz, która w wersji cywilnej była dostępna na rynku i natychmiast rozpoczęto badania. Pracowali nad tym polscy analitycy[82]: szef Biura Szyfrów ppłk *Karol Gwido Langer*, jego zastępca kpt. *Maksymilian Ciężki* i inż. *Antoni Palluth*. Nie przyniosły one jednak sukcesu, więc w 1929 roku zorganizowali dla wybranych studentów Uniwersytetu Poznańskiego kurs kryptologii. Ośmiu najbardziej obiecującym ofiarowali pracę w ekspozyturze Biura Szyfrów w Poznaniu. Po okresie próbnym trzech z nich: *Marian Rejewski*, *Jerzy Różycki* oraz *Henryk Zygalski* rozpoczęło prace nad *Enigmą*. Ich dalsze losy znamy głównie z relacji *Mariana Rejewskiego*, który początkowo został odizolowany od kolegów. Jego zadaniem było rozpoznanie wewnętrznych połączeń w maszynie, głównie pomiędzy klawiaturą i bębenkami. Pomogła mu w tym matematyczna teoria permutacji a także zdobyta przez francuski wywiad niemiecka instrukcja obsługi, a także "klucz" dzienny. Już po czterech miesiącach *Rejewski* wiedział jak działa maszyna. Udało mu się także odtworzyć wojskową *Enigmę*.

Za drugie zadanie przydzielono Rejewskiemu pracę nad tzw. *kluczem depeszy*, trzyliterowym oznaczeniem wzajemnego położenia bębenków w momencie rozpoczęcia szyfrowania, żeby odczytujący depeszę mógł ustawić bębenki w identycznym położeniu. Tu założył on, że szyfranci chętnie będą wystukiwać na klawiaturze trzykrotnie tę samą literę. O dziwo, hipoteza ta potwierdziła się i była to, według *Rejewskiego*, szczęśliwa okoliczność, gdyż niedługo po tym zauważono, że przełożeni zabronili szyfrantom używać takich połączeń. "Śledziliśmy bardzo uważnie ewolucję upodobań szyfrantów i gdy wkrótce zabronili im używania jako kluczy trzech identycznych liter, udawało się zawsze odkryć jakieś inne ich nawyki. Chociażby ten, że skoro nie wolno im było używać trzech jednakowych, unikali dwukrotnego powtarzania jakiejkolwiek litery, a to wystarczało, żeby dojść, jakich kluczy używano do depesz przed ich zaszyfrowaniem".

Trzecim a zarazem najważniejszym zadaniem było znalezienie metody odtwarzania "*kluczy*" dziennych, czyli obowiązujących danego dnia w całej sieci. Z tym zadaniem *Rejewski*, *Różycki* i *Zygalski* uporali się w styczniu 1933 roku. Następnie znów podzielili się zadaniami. *Jerzy Różycki* opracował "*metodę zegara*", czyli określania, który z trzech bębenków w danym dniu funkcjonuje jako pierwszy. Gdy liczba bębenków w wojskowych Enigmach została podniesiona

[82] Nadzór nad pracami związanymi z odtwarzaniem *Enigmy* sprawował w imieniu kierownictwa wywiadu II RP – dr *Tadeusz Modelski* (wówczas formalnie zajmujący stanowisko jednego ze starostów miasta stołecznego Warszawy, a następnie do września 1939 roku wiceprezydent Warszawa), późniejszy szef wywiadu przy generale *Sikorskim* w Londynie.

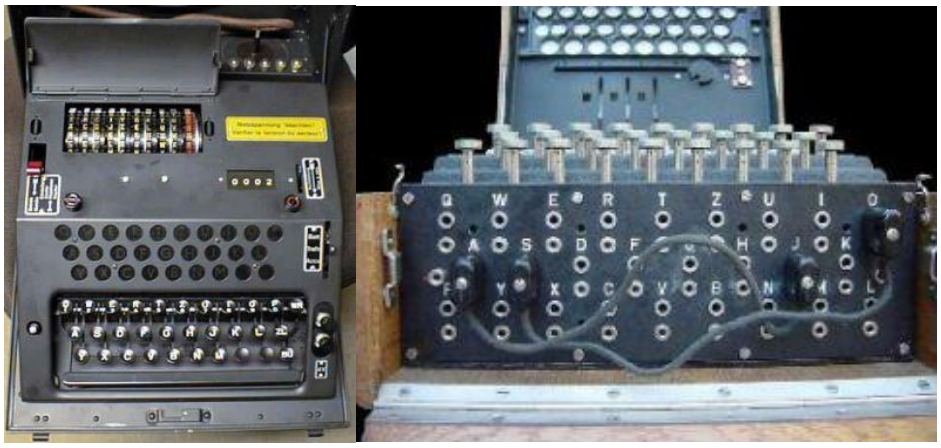
do pięciu, *Rejewski* skonstruował tzw. "*bomby*". Były to urządzenia do określania, które trzy z opięciu bębenków są aktualnie w użyciu. Owe "*bomby Rejewskiego*" w lipcu 1939 roku zrobiły ogromne wrażenie na angielskich specjalistach. *Henryk Zygalski* miał za zadanie opracować arkuszy perforowanych (nazwanych później "płachtami Zygalskiego"), które zastępowały książki kodowe i umożliwiały odczytywanie szyfru.
"Niemcy wciąż doskonalili *Enigmę* i wprowadzali coraz to nowsze udoskonalenia. Z czasem wynalazki polskich matematyków przestały wystarczać do odkodowywania tajnych niemieckich depesz. Zaistniała także konieczność stworzenia sześćdziesięciu "*bomb Rejewskiego*" zamiast istniejących trzech, a także skonstruowania do każdej nowe "*płachty Zygalskiego*". Siły polskiego Biura Szyfrów były na to za słabe. Większy potencjał wywiadu francuskiego i angielskiego stwarzał szansę uporania się z tym problemem. Polacy darowali więc sojusznikom wszystko, co mieli: dwie zrekonstruowane w Polsce wojskowe *Enigmy* z pięcioma bębenkami, "*zegary Różyckiego*", "*bomby Rejewskiego*" "*płachty Zygalskiego*", wszystko w dwóch kompletach. ≫ …



Marian Rejewski (1905-80) Henryk Zygalski (1908-78) Jerzy Różycki (1909-42)

≪Na spotkaniu 25 lipca 1939 roku powiedzieliśmy wszystko, co wiedzieliśmy i pokazaliśmy, co mieliśmy do pokazania" - napisał *Marian Rejewski*. W ten sposób tajemnica *Enigmy* została przekazana sprzymierzonym. Zastępca szefa brytyjskiego wywiadu, pułkownik *Steward Mazines,* osobiści odebrał cenna przesyłkę na dworcu kolejowym Victoria w Londynie w dniu 16 sierpnia, na dwa tygodnie przed wybuchem wojny. ≫
≪W 1939 r. w Pyrach koło Warszawy, gdzie mieściło się Biuro Szyfrów II Oddziału Sztabu Głównego WP, przyjechali przedstawiciele angielskiego i francuskiego wywiadu, którzy również pracowali nad rozszyfrowaniem Enigmy. 25 lipca Polacy ujawnili swoje wyniki prac aliantom, co w każdym razie bardzo ich to zdziwiło. Pokazano im kopię *Enigmy*, a także urządzenia, które pomagały rozszyfrowywać wiadomości, gdy Niemcy zmieniali klucz szyfrowania wiadomości. Polacy tuż przed wybuchem Wojny przekazali informacje dotyczące *Enigmy*, aliantom. Na początku Anglicy współpracowali z Polakami, ponieważ byli im potrzebni. We Francji otworzono w tym czasie, Warszawskie Biuro Szyfrów na Obczyźnie.
Polacy podczas współpracy kontaktowali się z naukowcami z Batchley. Jednak kontakty zerwały się po zajęciu Francji przez wojska Niemieckie. Polscy naukowcy pracowali w konspiracji, współpracując z Francuskim podziemiem odszyfrowywali wiadomości. Później uciekając przed tajną policją, przedostali się przez Pireneje i zostali zatrzymani na granicy i wsadzeni do więzienia w Hiszpanii. W 1943 r. A. Turing przyleciał do Stanów Zjednoczonych by zademonstrować urządzenie, które skonstruował po przeróbkach urządzeń stworzonych przez polskich naukowców, w tymże czasie Polacy przedostali się na Gibraltar, później przybyli do Anglii, gdzie dobrzy Anglicy odsunęli ich od ich własnego dzieła, nad którym pracowali.

Skierowano ich do podrzędnych prac. Polscy naukowcy już do końca Wojny rozwiązywali szyfry SS dla Polskich Sił Zbrojnych.



Rysunek 4.9.3.02. Enigma maszyna szyfrująca.

Niemcy do końca wojny nie wierzyli ani przez chwilę, by nieprzyjaciel mógł kiedykolwiek odczytywać depesze szyfrowane mechanicznie przez Enigmę.

… *Enigma* wywarła znaczący wpływ na konstrukcje wielu maszyn szyfrujących, głównie opartych na wirnikach. Brytyjska maszyna szyfrująca *Typex* została zainspirowana patentami *Enigmy*, także tymi, które nie zostały wykorzystane w najsłynniejszej wojskowej wersji maszyny. Rząd brytyjski w trosce o zachowanie tajemnicy nie ujawnił informacji o wykorzystanych rozwiązaniach i w związku z tym nie płacił żadnych tantiem autorom projektu. Japończycy wykorzystywali własną maszynę szyfrującą, która przez kryptologów amerykańskich została określona kryptonimem *GREEN*. Posiadała ona cztery wirniki, ale zainstalowane współpłaszczyznowo obok siebie (tzn. nie na jednej osi). Japońska maszyna nie znalazła jednak aż tak szerokiego zastosowania, jak niemiecka. Amerykański kryptolog *William Friedman* zbudował własną maszynę, oznaczoną jako *M-325*, odmienną konstrukcyjnie, ale kodującą w podobny do *Enigmy* sposób…≫

Piśmiennictwo: *Wikipedia* W.2.10.

Część 5.
Walidacja oprogramowania

5.0. WSTĘP DO WALIDACJI

5.0.0. KILKA NOWYCH POJĘĆ

Rozpocznijmy od wprowadzenia rozróżnienia pomiędzy dwoma podstawowymi klasami programów komputerowych:

5.0.0.01. **Definicja.** Programy sekwencyjne, mające swój jednoznacznie wyznaczony początek - rozpoczęcia działania (wykonywania) oraz jednoznacznie wyznaczone zakończenie działania. Większość tradycyjnych aplikacji, jest realizowane z pomocą programów sekwencyjnych.

5.0.0.02. **Definicja.** Programy współbieżne, a w szczególności programy współbieżne - reaktywne działające w środowisku rozproszonym. Programami współbieżnymi (reaktywnymi), które wprawdzie mają również jednoznacznie określony swój początek rozpoczęcia działania, ale z góry nie wiadomo, kiedy zakończą swoje działanie – są np. programy – moduły systemu operacyjnego. Podobną własność mają również programy sieciowych systemów operacyjnych lub sieciowych (rozproszonych) baz danych.

Kolejna sprawa, którą się zainteresujemy, jest sprawa, jakości programów, zarówno programów sekwencyjnych, jak również programów współbieżnych. W tym celu przypomnijmy najpierw, co rozumiemy pod pojęciem, jakości.

5.0.0.10. **Definicja.** Norma NF ISO 8402 [ISO-8402 94] definiuje pojęcie jakości w następujący sposób: *„zespół własności obiektu, wiążących się z jego zdolnością zaspokajania potrzeb stwierdzonych i oczekiwanych”.*

Powyższa norma rozważa, jakość w dwóch aspektach. Po pierwsze precyzuje, „czego” jakość jest własnością, czyli, o jaki obiekt chodzi (w tym przypadku o program komputerowy), a z drugiej strony "czyje" potrzeby są brane pod uwagę (w naszym przypadku użytkownika – wyrażone w postaci specyfikacji wymagań na dany program). Wymagania użytkownika odnoszące się, do jakości, a dotyczące samego produktu, jakim jest program komputerowy, zostały określone i podzielone w normie ISO 9126 [ISO-9126 92] na 6 *własności:*

- *Niezawodność programu*. Obejmuje *„zespół własności odnoszących się do zdolności programu do utrzymania poziomu usług zgodnego z ustalonymi warunkami i w czasie ustalonego okresu czasu*”. Z własnością tą łączą się pojęcia takie jak tolerancja błędów, możliwość odtworzenia stanu obliczeń po awarii, częstość awarii.
- *Łatwość użytkowania*. Obejmuje *„zespół własności odnoszących się do wysiłku niezbędnego do użycia i do indywidualnej oceny tego użytkowania przez określony jawnie lub domniemany zbiór użytkowników*". Z własnością tą łączą się pojęcia takie jak łatwość zrozumienia, łatwość szkolenia i łatwość eksploatacji.
- *Sprawność*. Ta własność obejmuje *„zespół własności odnoszących się do wzajemnego stosunku pomiędzy poziomem usług świadczonych przez dany program a wielkością użytkowanych zasobów w określonych warunkach*”. Z własnością tą łączą się pojęcia takie jak czas odpowiedzi, wielkość użytkowanych zasobów (obszar pamięci, przestrzeń na dysku, sprzęt, użytkowane usługi lub wsparcie logistyczne, itp.).

- *Zdolność funkcjonalna*. Charakteryzuje ona zdolność programu do zaspokajania potrzeb użytkowników w danej kategorii usług. Z własnością tą łączą się pojęcia takie jak zdolność oferowania oczekiwanej funkcjonalności, dostarczanie stosownych i dokładnych wyników, interoperacyjność (zdolność wzajemnej współpracy), zgodność z normami czy bezpieczeństwo.
- *Przenośność*. Obejmuje *„zespół właściwości odnoszących się do zdolności programu do przenoszenia z jednego środowiska komputerowego do innego*". Z własnością tą łączą się pojęcia takie jak łatwość adaptacji, łatwość instalacji, zgodność z normami i przyjętymi konwencjami, mające związek z możliwością przenoszenia oprogramowania.
- *Utrzymywalność*, (czyli *łatwość pielęgnowania)*. Własność ta odnosi się do wysiłku niezbędnego przy wprowadzaniu modyfikacji do programu. Z własnością tą, łączą się pojęcia takie jak: czytelność dokumentacji programu, łatwość analizy programu, łatwość modyfikacji, stabilność programu przy wprowadzaniu modyfikacji.

Wprowadzimy kolejne dwa bardzo istotne pojęcia, a mianowicie: *weryfikację* oraz *walidację* programu:

5.0.0.20. **Definicja.** *Weryfikacja* programu ma na celu skontrolowanie, czy dany (np. napisany) program jest zgodne ze specyfikacją wymagań na dany program.

5.0.0.30. **Definicja.** *Walidacja* to sprawdzenie, czy dany program działa zgodne ze swoją specyfikacją lub oczekiwaniami użytkownika. Tak, więc *weryfikacja* programu dotyczy funkcjonalności danego programu, ze względu na zgodność z funkcjonalnością opisaną w specyfikacji wymagań, natomiast *walidacja* dotyczy stwierdzenia poprawności (*correctness*) działania albo jej braku, (*falsification*). Gdzie przez poprawność działania programu zarówno rozumiemy poprawne wykonywanie funkcjonalności obliczeniowej, jak również (można powiedzieć, że przede wszystkim) poprawnego funkcjonowania sterowania wykonywaniem poszczególnych sekwencji czynności (np. brak martwych pętli w programie, brak zakleszczeń programu, itp.) oraz prawidłowego zakończenia działania programu.

Od samego początku istnienia komputerów, pisanie poprawnie działających programów jak również poprawna wzajemna współpraca programów, tworzących dany system programów (czyli oprogramowanie), była kluczowy zadaniem do rozwiązania. Pierwszym podejściem naturalnym dla środowisk naukowych i przemysłowych zajmujących się nową dziedziną – pisaniem programów komputerowych, było prowadzenie sprawdzenia mającego za zadanie odpowiedzenie na pytanie, czy napisany program działa zgodnie z wcześniejszą specyfikacją wymagań. Tych prób, dokonywano w oparciu o przygotowane dane próbne – testowe, dla który znane były oczekiwane wyniki poprawnego działania danego programu. Testowanie programu, stało się, więc – procesem związanym z pisaniem programów komputerowych. Tak, więc testowanie, stało się procesem służącym zapewnieniu wymaganej jakości poszczególnych programów jak i systemów programów (czyli tzw. oprogramowaniu). Testowanie może mieć na celu, zarówno *weryfikację*, jak i *walidację* programów komputerowych.

5.0.0.40. **Wyjaśnienie.** Testowanie jednak nigdy nie jest w stanie wykryć wszystkich błędów danego programu lub systemu programów, jednak może dostarczyć informacji o jego/ich poprawności działania, zgodnie z wybraną (będącą podstawą opracowanych i wykonanych testów), częścią specyfikacji wymagań, czy też oczekiwań klienta. Trzeba pamiętać, że proces testowania - nie sprawdza pod kątem wszelkich możliwych wariantów działania programu, lecz

jedynie w zakresie wybranych do badania wariantów działania. Tym samym, poprawne wykonanie testów przez dany program, oznacza jedynie, że program działa poprawnie, tylko w zakresie tych sprawdzanych przypadków działania, przez wykonany test.

Zasady dokumentowania procesu testowania poszczególnych programów komputerowych, jak również systemów programów (oprogramowania) reguluje szereg norm i standardów międzynarodowych.

5.0.0.50. **Definicja.** Podstawowym standardem dla testowania oprogramowania jest *IEEE 829-2008 (829 Standard for Software Test Documentation)*. Jest to standard określający formę zbioru ośmiu dokumentów potrzebnych w każdej z faz testowania oprogramowania. W efekcie każdej z tych faz tworzony jest 1 dokument wynikowy. Standard ten określa dokładnie format dokumentów, jednak nie wymaga, aby wszystkie były wykonane. Nie zawiera także informacji o tym, co dokładnie mają zawierać.

- *Test Plan* – dokument planowania zarządzania projektem, który składa się z informacji o tym, w jaki sposób będą prowadzone testy, kto będzie je przeprowadzał, co będzie testowane, jak długo potrwa cały proces oraz jaki będzie zakres testów.
- *Test Design Specification* – szczegóły na temat warunków testowania, oczekiwanych wyników a także kryteriach przejścia testu.
- *Test Case Specification* – specyfikuje dane testowe do użycia podczas wdrażania warunków testowania określonych w *Test Design Specification*.
- *Test Procedure Specification* – zawiera szczegóły na temat przeprowadzenia każdego testu włączając w to założenia oraz poszczególne kroki testów.
- *Test Item Transmittal Report* – zawiera raporty na temat czasu przejścia testowanych fragmentów oprogramowania między etapami.
- *Test Log* – zawiera informacje o tym, które przypadki testowania zostały użyte, kto je użył i w jakim porządku oraz informacje o ich powodzeniu.
- *Test Incident Report* – zawiera informacje o testach zakończonych niepowodzeniem. Informacje o wynikach oraz dlaczego dany test nie powiódł się.
- *Test Summary Report* – raport ten zawiera wszystkie istotne informacje ujawnione podczas zakończonych testów oraz wyceny, jakości procesów testowania, jakości oprogramowania poddanego testowi, a także statystyki uzyskane z *Incident Report*. Raport referuje również do typów i czasu trwania wykonanych testów w celu usprawnienia wszelkich planów związanych z testami w przyszłości. Ostateczna forma dokumentu jest wykorzystywana w celach weryfikacji poprawności testowanego systemu względem wymagań zdefiniowanych przez zleceniodawców.

W naszych dalszych rozważaniach, standardy prowadzenia i dokumentowania testów programów nie są zbyt istotne. Natomiast warto podkreślić, że mimo wielu zgłaszanych przez lata zastrzeżeń, użycie procesu testowania odnośnie programów sekwencyjnych, zapewnia na ogół, dość wysoką, jakość programów. Natomiast, proces testowania programów współbieżnych, a w szczególności programów działających w środowisku rozproszonym (np. sieciowym lub wieloprocesorowym), nastręcza wiele wątpliwości, a tym samym, celowości jego stosowania. Mówiąc, wprost, ponieważ wzajemne współdziałanie programów współbieżnych, jest, co do zasady - trudno przewidywalnym, bardzo wręcz trudnym zadaniem, a często - zadaniem niewykonalnym, jest przygotowanie zestawu testowego - zdolnego do sprawdzenia oddziaływani – współdziałania wielu programów współbieżnych w złożonym środowisku. Tak, więc, skuteczne zapewnienie, jakości programów współbieżnych na drodze ich testowania, jest

praktycznie niemożliwe. Przywoływana wcześniej (patrz przedmowa) katastrofa samolotu Lufthansa Airbus A320-200[83] na lotnisku im. Fryderyka Chopina w dniu 14 września1993 roku w Warszawie, jest tego wymownym przypadkiem.

Pierwsze próby analitycznego dowodzenia poprawności działania programów komputerowych (podkreślić należy, że dotyczyło to jedynie programów sekwencyjnych), podjął w 1969 roku - *Charles A. R. Hoare* - brytyjski teoretyk informatyki, tworząc tzw. *logikę Hoare'a*[84], służącą do weryfikowania poprawności programów metodą przekształceń analitycznych. Zdanie logiczne:

5.0.0.61 $\{P\}S\{Q\}$

oznacza, że fragment kodu S o ile na wejściu będzie miał stan spełniający warunek P, oraz zakończy swoje działanie, to, na wyjściu da stan spełniający warunek Q. Formułę P nazywamy warunkiem wstępnym, a formułę Q nazywamy warunkiem końcowym. W przypadku logiki *Hoare'a* dozwolone jest m.in. następujące rozumowanie:

5.0.0.62 jeśli $\{P_1\}S\{P_2\}$ oraz $\{P_2\}D\{P_3\}$, to $\{P_1\}S;D\{P_3\}$.

Pozwala nam to rozbijać złożone fragmentu kodu na instrukcje elementarne, dla których weryfikacja poprawności zapisu $\{P\}S\{Q\}$ jest łatwa. Rozumowanie *Hoare'a* dopropwadziło do sformułowanie podstaw teoretycznych (logicznych) poprawności programów komputerowych:

5.0.0.71. **Hipoteza**. Hipotezą takiej teorii poprawności, jest pewien warunek, spełniany przez zmienne programu bezpośrednio przed wykonaniem danego programu. Ten warunek jest zwykle zwany warunkiem początkowym (czyli „*pre condition*").

5.0.0.72. **Teza**. Tezą tej teorii poprawności, jest pewien warunek, spełniany przez zmienne programu bezpośrednio po wykonaniu danego programu. Ten warunek jest zwykle zwany warunkiem końcowym (czyli „*post condition*").

5.0.0.73. **Teoria poprawności programu komputerowego** (wywodząca się z badań Floyd'a, a następnie Hoare'a). Jeżeli warunek P – (*pre condition*), jest prawdziwym przed wykonaniem programu S, wówczas warunek Q – (*post condition*), będzie prawdziwy po wykonaniu programu S. Co można zapisać krótko: $\{P\}S\{Q\}$.

Kazimierz Trzęsicki (patrz T.3.1.), opisuje historię opracowania metod *walidacji* oprogramowania (zwane również metodami formalnymi), przyjmującą za punkt wyjścia logikę modalna:[85] „W 1974 roku *Rod M. Burstall* - brytyjski naukowiec informatyk, zauważył możliwość zastosowania logiki modalnej do rozwiązywania problemów informatyki. Logika dynamiczna programów została wynaleziona przez *Vaughan R. Pratta* (1980)." Dalej Trzęsicki pisze: „Zadanie użycia TL (*temporal logic*) do inżynierii oprogramowania, zostało podjęte przez *Krögera* (1977, 1987, 1991, 2008). Dalszy rozwój zastosowania TL do nauk komputerowych zawdzięczamy *Amirowi Pnueli*. Został on zainspirowany książką zatytułowaną ≪Temporal Logic≫, napisaną przez *Reschera* i *Urquharta* (1971). ≪The Temporal Logic of Programs≫ (1977), artykuł autorstwa *Pnueli*, jest klasycznym źródłem zastosowania TL do specyfikacji wymagań i walidacji oprogramowania. Ta praca jest uważana za przełomową w użyciu TL w

[83] Opis katastrofy jest dostępny na stronie www.crashdatabase.com

[84] logika Hoare'a – formalizm matematyczny służący do opisu poprawności algorytmów.

[85] Tłumaczenie z języka angielskiego – nie autoryzowane. MJG

naukach komputerowych. *Amir Pnueli* uzasadnił przydatność stosowania formalizmu logiki temporalnej, zarówno do badania zachowań programów sekwencyjnych, jak również do działających powtarzalnie (*nonterminating*) programów współbieżnych." Kilka stron dalej, w wyżej wzmiankowanej publikacji, Trzęsicki dokonuje formalnego podziału na dwie podstawowe kategorie metod zapewniania poprawności programów komputerowych:

(1) pisania matematycznych dowodów poprawności;
(2) opracowanie modelu sterowania programu, a następnie sprawdzanie poprawności takiego modelu, krok po kroku – wzdłuż skończonej przestrzeni stanów (tzw. *model checking).*

To drugie podejście, dała praktycznie istotny postęp w zakresie zapewnienia poprawności programów. Użycie metod formalnych – podobnych w swojej idei do metody zero – jedynkowego, czyli tabliczkowego sprawdzania poprawności formuł rachunku zdań (porównaj 2.2.3).

Piśmiennictwo: *Ben-Ari M.* B.2.1., B.2.2., *Dijkstra E.* D.2.1. *Holzmann G.* H.2.1., *Trzęsicki K.* T.3.1., T.3.2.

5.0.1. PROGRAM KOMPUTEROWY, JAKO ZŁOŻONE WYRAŻENIE LOGICZNE

Jednym z głównych zastosowań logiki formalnej w informatyce (dokładniej mówiąc, szczególnie logik temporalnych), jest weryfikacja poprawności programów komputerowych. Każdy program komputerowy jest przecież, złożonym wyrażeniem logicznym. Tym samym, można dokonać jego walidacji, czyli odpowiedzieć na pytanie, kiedy (dla jakich wartości parametrów, w tym parametrów środowiskowych oraz dla jakich przedziałów wartości danych wejściowych programu), jest on prawdziwy (czyli działający poprawnie), a kiedy fałszywy (czyli działający niepoprawnie). Oczywiście, w wyniku walidacji mamy w wyniku dwie kategorie programów komputerowych:

1. Programy komputerowe (zarówno sekwencyjne, jak i współbieżne), które dla wszelkich możliwych zestawów wartości parametrów środowiskowych oraz dla wszelkich danych wejściowych - działają prawidłowo (można by określić, jako programami pozytywnie *walidowanymi*), czyli są odpowiednikami wyrażeń logicznych zawsze prawdziwych, czyli tautologiom.
2. Programy komputerowe, które dla pewnych możliwych zestawów wartości parametrów środowiskowych oraz dla pewnych danych wejściowych (mieszczących się w zadanych przedziałach wartości) - działają nie zawsze prawidłowo (do tej kategorii zliczyć należy większość nowo napisanych programów komputerowych).

Pierwsza kategoria, nie jest dla nas interesująca, bo jeśli w każdej sytuacji program działa prawidłowo, to możemy go eksploatować, bez obawy na pojawienie się nieprzewidywalnych wyników lub zachowań.

Druga z wymienionych kategorii programów, jest dla nas najbardziej interesująca, ponieważ jak już powiedzieliśmy, większość nowo pisanych programów, zawiera pewną liczbę błędów, których obecność może spowodować nieprawidłowe działanie danego programu, w najmniej oczekiwanym momencie. Jak wiemy, upewnienie się, co do poprawności działania danego programu komputerowego w określonym środowisku systemu operacyjnego i sieci komputerowej, dla zadanych przedziałów wartości danych wejściowych, jest warunkiem koniecznym dla zapewnienia bezpiecznego działania informatycznego systemu aplikacyjnego, w skład, którego wchodzi dany program komputerowy.

Ponieważ bardzo często program komputerowy zawiera z jednej strony procedury obliczeniowe, których poprawność działania jest stosunkowo łatwa to sprawdzenia, zaś z

drugiej strony, strukturę sterowania wykonywaniem programu, zależną od wcześniej uzyskanych wyników częściowych wykonywania danego programu, to nasunął się dość naturalny pomysł, aby zastąpić walidację programu, jako całości oddzielnymi walidacjami: struktury sterowania oraz procedur obliczeniowych. Takie podejście, jest wzorowane na podejściu inżynierskim do nowych konstrukcji technicznych. Zanim opracujemy konstrukcję nowego produktu, opracowujemy model użytkowy danego produktu i sprawdzamy jego funkcjonalność – badają zbudowany model.

W dalszym rozwoju metod i technik *walidacji* programów komputerowych, takie podejście okazało się podejściem dominującym. Ale zanim zajmiemy się budowaniem modeli programów, popatrzmy na rozwój podejścia dowodzenia poprawności oraz walidacji programów, począwszy od prób dotyczących posłużenia się klasycznym rachunkiem zadań, poprzez pierwsze usiłowania zastosowania logiki temporalnej, aż do współczesnych metod formalnych walidacji modeli sterowania programów.

Piśmiennictwo: *Ben-Ari M.* B.2.2., *Dijkstra E.* D.2.1. *Holzmann G.* H.2.1., *Trzęsicki K.* T.3.1.

5.0.2. Podejście do dowodów poprawności programów wg. C.A.R. Hoare'a.

C.A.R. Hoare - jest twórcą tzw. *Logiki Hoare'a*[86], pierwszego opracowanego narzędzia dowodzenia poprawności (walidacji) programów sekwencyjnych, logiki typu systemu dedukcyjnego – pochodnej próby zastosowania klasycznego rachunku zdań w jednym z działów informatyki. Logika Hoare'a opiera się na zestawie sześciu postulatów. Wprowadźmy następujące oznaczenia, potrzebne do sformułowania postulatów:

1. Niech p – oznacza predykat warunku początkowego spełniany przez wyrażenie (*precondition*).
2. Niech q – oznacza predykat warunku końcowego spełniany przez wyrażenie (*postcondition*).
3. Niech S – oznacza segment programu sekwencyjnego (blok programu) spełniający warunek początkowy oraz warunek końcowy, od zmiennych x.
4. Niech B – oznacza warunek logiczny, przyjmujący jedną z dwu wartości 1 (TRUE) lub 0 (FALSE).
5. Niech x – oznacza zmienne (*variables)* programu S.
6. Niech l – oznacza stan licznika rozkazów (*program counter*), odpowiadający danej wartości zmiennych x.
7. Niech $< l, x >$ – oznacza krotkę wiążącą stan licznika rozkazów z wartościami przyjmowanymi przez zmienne programu.

Instrukcję (*statement*) każdego języka programowania, jak również S - segment programu sekwencyjnego, może być traktowany, jako funkcję, której wartość jest zmieniona w wyniku wykonania tej instrukcji lub segmentu programu wraz z odpowiednim przyrostem wartości licznika rozkazów, co możemy opisać z pomocą krotki $< l, x >$.

Poniżej przedstawiamy zestaw sześciu postulatów wg *C.A.R. Hoare'a*:

5.0.2.11. **Postulat domeny**. Każda prawdziwa formuła określa domenę (domeny) zmiennych programu.

5.0.2.12**. Postulat podstawienia**.

$$\vdash \{p(x)\{x \leftarrow t\}\}x = t\{p(x)\}.$$

[86] Porównaj Mordechai Ben-Ari „Mathematical Logic for Computer Science" [B.2.1].

Dla wyjaśnienia postulatu podstawienia, rozważmy wyrażenie:

$$\vdash \ \{\ ?\ \}\, x \ = \ t\{p(x)\}.$$

Po wykonaniu operacji podstawienia, chcielibyśmy żeby wyrażenie p(x) było prawdziwe, wtedy, kiedy wartość przypisana x – jest wartością wyrażenia t. Jeśli wyrażenie $p(x)\{x \leftarrow t\}$, w wyniku podstawienia, jest prawdziwe, to wówczas x aktualnie podstawione przez t, powoduje, że predykat $p(x)$ będzie miał wartość *true*.

5.02.13. **Postulat składania ról**.

$$\frac{\vdash \{p\}S1\ \{q\} \qquad \vdash \{q\}\ S2\ \{r\}}{\vdash \{p\}\ S1\ S2\ \{r\}}.$$

Postulat składania ról nie wymaga chyba wyjaśnień.

5.02.14. **Postulat alternatywnej roli**.

$$\frac{\vdash \{p\wedge B\}\ S1\ \{q\} \qquad \vdash \{p\wedge\sim B\}\ S2\ \{q\}}{\vdash \{p\}\ \text{if (B) S1 else S2}\ \{q\}}.$$

Podobnie, jaka postulat składania ról, postulat alternatywnej roli nie wymaga komentarza.

5.02.15. **Postulat roli w pętli**.

$$\frac{\vdash \{p\wedge B\}\ S\{p\}}{\vdash \{p\}\text{while (B) S}\ \{p\wedge\sim B\}}.$$

Predykat p w pętli *while* jest nazywany niezmienniczym (*invariant*), ponieważ opisuje zachowanie pojedynczego wykonania segmentu programu S wewnętrznej – powtarzalnej zawartości pętli. Dowodząc: $\vdash \{\ p\ \}\ \textit{while}$ (B) S $q0$, stwierdzamy, że predykat p jest niezmienniczy (*invariant*):

$$\vdash \ \{p\ \wedge\ \text{B) S}\ \{\ p\ \}$$

z rolą pętli

$$\vdash \{p\}\text{while (B) S}\ \{p \wedge \neg B\}.$$

Jeśli potrafimy udowodnić, że $p0 \rightarrow p$ oraz $(p\ \wedge \neg B) \rightarrow q0$, to postulat konsekwencji roli (5.02.16), może być użyty do wnioskowania o poprawności formuły. Nie wiemy ilokrotnie formuła będzie wykonywana, ale wiemy natomiast, że wyrażenie p $\wedge \neg$B – będzie prawdziwe.

5.02.16. **Postulat konsekwencji roli.**

$$\frac{\vdash p1 \rightarrow p \quad \vdash \{p\}\ \text{S}\ \{q\} \quad \vdash q \rightarrow q1}{\vdash \{p1\}\ \text{S}\ \{q1\}}.$$

Dla dowiedzenia poprawności segmentu programu (nie mówimy jeszcze o całości programu), należy znaleźć odpowiednie niezmienniki. Np. najsłabsza z możliwych formuła prawdziwości (*true formula*) jest niezmiennikiem każdej pętli, ponieważ: $\vdash \{\ \textit{true}\ \wedge$ B) S $\{\ \textit{true}\ \}$, jest prawdziwe dla każdego wyrażenia B oraz S. Oczywiście, ta formuła jest zbyt słaba, ponieważ jest niepodobnym, że udowodnimy $\{\ \textit{true}\ \wedge$ B) S $\{\ \textit{true}\ \} \rightarrow q0$. Z drugiej strony, jeśli formuła jest zbyt silna, nie będzie niezmiennikiem.

Dotychczas mówiliśmy o dedukcyjnym dowodzeniu prawdziwości poszczególnych instrukcji lub segmentów programu sekwencyjnego, obecnie zajmiemy się programami sekwencyjnymi, jako całością. Załóżmy, że nasz program sekwencyjny składa się z n – segmentów: S_1, S_2, … , S_n. Uwzględniając formuły predykatów warunków początkowych i końcowych poszczególnych segmentów programu sekwencyjnego, otrzymamy:

5.02.20 $\{\ p_1\ \}\ S_1\ \{\ p_2\ \}\ S_2\ \{\ p_3\ \} \ldots \{\ p_n\ \}\ S_n\ \{\ p_{n+1}\ \}.$

Jeśli więc udowodnimy, że { p_i } S_i { p_{i+1} }, dla wszelkich i, jest z walidowane – to tym samym udowodniliśmy zgodnie z Logiką Hoare'a, poprawność danego programu sekwencyjnego.

Piśmiennictwo: *Ben-Ari M.* B.2.1., *Hoare C.A.R.* H.4.1.

5.0.3. Zastosowanie przez Prueli'a - LTL do badania poprawności programów

O ile wyniki prac *C.A.R. Hoare'a* - dotyczyły programów sekwencyjnych, o tyle praca *Amira Pnueli'a* dotyczyła zarówno programów sekwencyjnych jak i programów współbieżnych (reaktywnych). Uniwersalne podejście zarówno do programów sekwencyjnych, jak również programów współbieżnych, oparł *Pnueli* na definicji dynamicznego systemu dyskretnego z wykorzystaniem logiki temporalnej TL:

5.0.3.10 $< S, R, s_0>$

gdzie:

S – jest zbiorem stanów, jaki system może przyjąć (być może nieskończony);

R – jest relacją przejścia pomiędzy parą (poprzednikiem i następcą) stanów systemu $R \subseteq S \times S$;

s_0 – jest stanem początkowym.

Dodatkowym założeniem przyjętym niejawnie (prawdopodobnie domyślnie) przez *Pnueli'a*, było założenie o dyskretnym, do tego asynchronicznym czasie, (czyli czasie dyskretnym, którego poszczególne kwanty, są wynikiem zachodzenia określonego zdarzenia, np. przejścia od jednego stanu do następnego). W dalszym ciągu, takie kwanty czasu dyskretnego, będziemy nazywali *chronami*.

Wykonywanie systemu prowadzi do powstawania w kolejnych chronach t = 0, 1, 2, …, i, … sekwencji kolejnych stanów należących do przestrzeni stanów, a dokładniej mówiąc do podprzestrzeni stanów dostępnych w wyniku wykonywania danego programu:

5.0.3.11 $\varsigma = s_0, s_1, s_2, \ldots, s_i, \ldots$ dla każdego $i \geq 0$, gdzie spełniona jest relacja $R(s_i, s_{i+1})$.

Ogólnie mówiąc, jeśli relacja R jest relacją niedeterministyczną – to możliwe są różne sekwencje stanów systemu dynamicznego.

W przypadku programów sekwencyjnych, ograniczymy się do deterministycznej formy relacji R. Wówczas wartości s_i, dla i > 0 są wyznaczone przez parę uporządkowaną $<\pi_i, u_i>$, gdzie π_i jest komponentem sterującym; zaś u_i jest komponentem danych należącym do nieskończonych domen. Poszczególnym komponentom sterującym, mogą być przypisane etykiety lokalizacji należące do zbioru:

5.0.3.12 $L = \{ l_0, l_1, \ldots, l_n \}$.

Relacja R, w przypadku programów sekwencyjnych może być rozłożona na parę funkcji. Mianowicie funkcje następnej lokacji $N(\pi, u)$, oraz funkcję transformacji danych $T(\pi, u)$. Natomiast w przypadku, gdy komponent π jest instrukcją warunkową, funkcja N zależy wyłącznie od komponentu u. Możemy, więc, w przypadku programów sekwencyjnych, wyrazić relację R za pomocą funkcji N i T. Czyli:

5.0.3.13 $R(<\pi_i, u_i>, <\pi_{i+1}, u_{i+1}>)$ można wyrazić przez: $\pi_{i+1} = N(\pi_i, u_i)$ oraz $u_{i+1} = T(\pi_i, u_i)$.

Zajmiemy się z kolei przypadkiem programów współbieżnych. Dopuszczając postać wektorową komponentu sterującego w miejsce komponentu skalarnego, przechodzimy od przypadku programu sekwencyjnego do opisu działania zestawu programów współbieżnych. Stan bieżący jest wyznaczony przez (n+1) składowe tzw. *krotki*:

5.0.3.21 $s = <\pi^1, \pi^2, \ldots, \pi^n; u>$,

gdzie każdy π^j może być traktowany, jako skończony program sekwencyjny j-tego procesora (dla $0 \leq j \leq n$), natomiast u jest współdzielonym komponentem danych n programów sekwencyjnych wykonywanych współbieżnie. Zakładamy dalej, że zarówno funkcja stanu następnego j-tego programu $N(\pi^j, u)$, jak i funkcja transformacji danych przez j-ty program $T(\pi^j, u)$, są dalej funkcjami deterministycznymi i zależą jedynie od jednego komponentu sterującego. Natomiast wybór następnego procesora, w każdym chronie (kwancie czasowym) odbywa się niedeterministycznie.

Intuicyjnie, tak określony model pozwala na wykonywanie równoczesne n programów przez n procesorów. W każdym kroku całego systemu, jeden procesor o numerze j zostaje wybrany a instrukcja π^j_i - zlokalizowana pod adresem wskazanym przez IC (*Instruction Counter* – licznik rozkazów) tegoż procesora, zostaje wykonana (nie dotyczy procedur). Na pierwszy rzut oka, może się wydawać ograniczeniem niepozwalającym na modelowanie różnorodnych oddziaływań pomiędzy różnymi fazami wykonywalności instrukcji. Jak jednak wiadomo, to od programisty zależy wybór jednostek, które dla potrzeb modelowania zostaną określone, jako niepodzielne (*atomic*).

Podchodząc do sprawy formalnie, możemy przedstawić ogólne zasady przejścia:

5.0.3.22 $R(<\pi^1_{k+1}, \pi^2_{k+1}, \ldots, \pi^n_{k+1}; u_{k+1}>, <\pi^1_k, \pi^2_k, \ldots, \pi^n_k; u_k>)$ dla $0 \leq j \leq n$ oraz $0 \leq k \leq m$,

z pomocą indywidualnych funkcji przejścia dla poszczególnych procesorów, jako:

5.0.3.23 $<\pi^1_{k+1}, \pi^2_{k+1}, \ldots, \pi^n_{k+1}> = <\pi^1_{k+1}, \pi^2_{k+1}, \ldots, \pi^{j-1}_{k+1}, N_j(\pi^j_k; u_k), \ldots, \pi^{j+1}_{k+1}, \ldots, \pi^n_{k+1}>$; oraz

5.0.3.24 $u_{k+1} = T_j(\pi^j_k, u_k)$.

Z kolei dla wyrażenia właściwości systemu oraz hierarchię tych zmian w czasie, użyjemy relacji stanów w czasie q(s) sformułowaną w odpowiednim języku. Stosując tą relację do naszego zestawu programów współbieżnych otrzymamy relację pomiędzy $<\pi^1, \pi^2, \ldots, \pi^n; u>$ wartościami danych oraz lokacjami wskazanymi przez IC (*Instruction Counter* - liczniki rozkazów) poszczególnych procesorów. Podstawowym zadaniem, jest sformułowanie zasad opisywanych zmian q(s) w kolejnych chronach (kwantach czasu). Wprowadzając bezpośrednią sekwencje zmiennych chronów t_1, t_2, … , które w naszym modelu numerujemy liczbami naturalnymi, i pomiędzy nimi zachodzą relację: „<”, „=” – możemy wprowadzić funkcję czasu:

5.0.3.30 $H(t, q) \equiv q(s_t)$,

umożliwiającą opisanie szerokiej gammy zależności. Możliwym jest dokonanie selekcji przypadków w zależności od liczby rozróżnianych zmiennych zależnych od czasu i wyrażenie ich bezpośrednich zależności.

W dalszym ciągu rozważań ograniczymy się do jednej zmiennej czasu, przebiegającą sekwencje chronów, to w dalszym ciągu będziemy rozróżniać pięć różnych zależności:

5.0.3.31. **Definicja**. Niezmienniczość (*Invariance*) – czyli własność utrzymującą się we wszystkich możliwych stanach, we wszystkich możliwych sekwencjach przejść pomiędzy stanami.

Jeżeli rozszerzymy relację R o jej domknięcie R*, zdefiniujemy zbiór dostępnych stanów:

$$X = \{ s \mid R^*(s_0, s) \}$$

Natomiast predykat p(s), jest niezmiennikiem zachowywanym przy wszelkich możliwych zmianach stanu systemu $s \in X$, własność p(s) jest spełniana. Co można zapisać:

$$(\forall s \in X) \; p(s) \rightarrow \forall t \; H(t, p).$$

5.0.3.32. **Definicja**. Częściowa poprawność (*Partial Correctness*). Jeśli mówimy o fragmencie kodu programu sekwencyjnego o etykiecie wejściowej l_0 - posiadającym również etykietę końcową l_m, określamy częściową poprawność fragmentu kodu od etykiety pierwszej instrukcji l_0 – do ostatniej linii kodu etykietowanej przez l_m, jeśli własność s jest inwariantem tegoż fragmentu kodu programu sekwencyjnego.

5.0.3.33. **Definicja**. Poprawne wykonanie (*Clean Execution*) kodu programu. Jeżeli fragment kodu programu sekwencyjnego o etykiecie wejściowej l_0 - posiadającym również etykietę końcową l_m, jest wykonywany zgodnie ze specyfikacją wymagań na dany fragment kodu oraz nie pojawi się ani razu w ciągu takiego wykonania sytuacja „nielegalna" ze względu na wzmiankowaną specyfikację, to ma miejsce poprawne wykonanie kodu programu. Przykładem może być fragment kodu zawierający instrukcje dzielenia zmiennopozycyjnego. Niech $l_1, l_2, \ldots, l_k$ – będą miejscami w kodzie gdzie znajduje się instrukcja dzielenia, zaś $y_1, y_2, \ldots, y_k$ są wartościami dzielników, to stwierdzeniem zero błędów dzielenia, jest niezmiennik postaci:

$$(\pi = l_1 \rightarrow y_1 \neq 0) \wedge \ldots \wedge (\pi = l_k \rightarrow y_k \neq 0).$$

5.0.3.34. **Definicja**. Wzajemne wykluczenie (*Mutual Exclusion*). Jeśli dwa programy sekwencyjne mają wzajemnie wykluczający się dostęp w jednym chronie czasu - do danego obszaru krytycznego (np. sekcji krytycznej w systemie operacyjnym), oznacza to, że oba programy mogą korzystać z danego obszaru krytycznego, ale nigdy równocześnie, to oznacza mają one prawo do wzajemnie wykluczającego dostępu do obszaru krytycznego. Oznacza to, że spełniony jest niezmienniczo własność, że jeśli pierwszy z programów korzysta z obszaru krytycznego, to drugi program musi czekać z dostępem do momentu, w którym pierwszy program opuści obszar krytyczny, dopiero wówczas uzyska dostęp do tegoż obszaru – i odwrotnie.

5.0.3.35. **Definicja**. Brak zakleszczeń (*Deadlock Freedom*). Rozważmy z kolei, przykładowo zestaw dwóch współbieżnie wykonywanych programów sekwencyjnych P_1 oraz P_2. Zakleszczeniem nazywamy sytuację, w której oba programy oczekują na możliwość równoczesnego dostępu do pary zasobów, które nazywamy A oraz B odpowiednio (każdy z tych zasobów, może pozwolić na dostęp jednoczesny tylko jednego programu – czyli jest obszarem krytycznym, patrz 5.0.3.34), w różnej kolejności. Czyli pierwszy z programów P_1, musi uzyskać - najpierw dostęp do zasobu A, a następnie do zasobu B (utrzymując też dalej dostęp do zasobu A, aż do wykonania operacji na zasobie B); natomiast program drugi P_2 odwrotnie, najpierw musi uzyskać dostęp do zasobu B, a następnie do zasobu A (utrzymując też dalej dostęp do zasobu B, aż do wykonania operacji na zasobie A). Jeżeli w danym chronię (kwancie czasu) pierwszy z programów P_1 uzyska dostęp do zasobu A, a równocześnie drugi program P_2 uzyska dostęp do zasobu B, to nastąpi zakleszczenie wzajemne obu programów, ponieważ P_1 blokuje dostęp do zasobu A i usiłuje uzyskać dostęp do zasobu B, zaś P_2 odwrotnie. Żaden z obu programów, nie może uzyskać dostępu do drugiego z zasobów, tym samym blokują wzajemnie wykonywanie dalszych instrukcji każdego z dwóch programów. *Pnueli* pokazał, że wystąpieniu wzajemnego zakleszczenia dwóch programów, towarzyszy naruszenie niezmienniczości pewnego wyrażenia logicznego.

Kolejnym krokiem rozważań *Pnueli* było zajęcie się poprawnością całkowitą (*Total Correctness*) programu sekwencyjnego. W tym celu *Pnueli* rozważał dwa przypadki: (1) przypadek programu liniowego (programu, w którym kolejne sekcje kodu programu, są wykonywane kolejno, od pierwszej do ostatniej; oraz (2) przypadek programu z pętlą (czyli sekcją kodu wykonywaną wielokrotnie), a w szczególności programu powtarzalnego, który praktycznie nigdy nie kończy swojego działania, jak to ma miejsce z systemem operacyjnym. Rozważania poprawności całkowitej dotyczą przynajmniej pary uporządkowanej chronów t_1 oraz t_2, z których pierwszy chron t_1, poprzedza wystąpienie drugiego chrona t_2 – tworząc tzw. następstwo w czasie (*Temporal Implication* oznaczoną symbolem ⇒), co można zapisać:

5.0.3.40 $\varphi \Rightarrow \psi \,\square\, \forall\, t_1\, \exists\, t_2\, (t_1 \leq t_2)\, H(t_1, \varphi) \rightarrow H(t_2, \psi)$

5.0.3.41. **Definicja**. Całkowita poprawność programu liniowego (*Total Correctness*). Przez całkowitą poprawność programu liniowego rozumiemy osiągnięcie końca kodu w wyniku wykonania danego programu oraz dostarczenie poprawnego wyniku działania programu. Co jest spełnianiem implikacji temporalnej.

5.0.3.42. **Definicja**. Całkowita poprawność programu powtarzalnego (*Total Correctness of Cycling Program*). Przez całkowitą poprawność programu zapętlonego – powtarzalnie działającego „bez końca", rozumiemy poprawne wykonywanie działań, czynności będących odpowiedzią na zgłoszone zapotrzebowanie (*Responsiveness*).

W dalszym ciągu rozważań *Pnueli* określił tzw. trzy zasady przeprowadzenia dowodu całkowitej poprawności programu sekwencyjnego, obejmujące: (1) niezmienniczość, (2) poprawne określenie zbioru stanów oraz (3) rozumowanie dotyczące ewentualności.

5.0.3.51. **Definicja**. Niezmienniczość można zdefiniować indukcyjnie, dane $\varphi(s_0)$, jeśli

$$\forall s, s_1\ \varphi(s) \wedge R(s, s_1) \rightarrow \varphi(s_1) \text{ to } (\forall s \in X)\ \varphi(s).$$

Co jest oczywistą zasadą indukcji obliczeniowej. Inaczej mówiąc, własność, która jest spełniona oraz przekazuje to spełnianie wzdłuż dopuszczalnej sekwencji przekształceń, jest niezmiennikiem.

5.0.3.52. **Definicja**. Poprawne określenie zbioru stanów - opiera się o następującą zależność. Niech A(s, n) będzie predykatem zależnym od stanu s i jakiejś naturalnej liczby $n \geq 0$. Wówczas

$$\varphi(s) \rightarrow \exists\, n\, A(s, n)$$

$$A(s, n) \wedge R(s, s_1) \rightarrow A(s, n\text{-}1) \vee \psi(s)$$

$$\varphi \,\square\, \psi.$$

Powyższa zasada, zawiera w sobie zarówno zasadę niezmienniczości realizowanej przez rodzinę niezmienników A(s, n) oraz zasadę poprawnie określonego zbioru.

5.0.3.53. **Definicja**. Rozumowanie dotyczące ewentualności. W niniejszym podejściu, prosta relacja ewentualności (możliwości), wyprowadzana jest bezpośrednio z zasady przejścia R od danego stanu do kolejnego oraz zastosowania wielokrotnego tejże zasady R. Podstawą są dwa przyjęte aksjomaty:

(A1) jeśli $\forall\, (s, s_1)\ p(s) \wedge R(s, s_1) \rightarrow q(s1)$ to $p \,\square\, q$

(A2) jeśli $p \rightarrow q$ to $p \,\square\, q$

Wprowadźmy następujące reguły wnioskowania:

(R1) $p \,\square\, q, \forall\, (s, s_1)\ r(s) \wedge R(s, s_1) \rightarrow r(s_1)$ wówczas $(p \wedge r) \,\square\, (q \wedge r)$

(R2) $(p \square q)$ oraz $(q \square r)$ to $(p \square r)$

(R3) $(p_1 \square q)$ oraz $(p_2 \square q)$ to $(p_1 \vee p_2) \square q$

(R4) $(p \square q)$ to $((\exists p) \square q)$

Dodatkowo, przyjmijmy całą teorię logiki pierwszego rzędu, jako dodatkowe aksjomaty. Tak przyjęte aksjomaty pozwalają na wyprowadzenie elementarnych ewentualności (możliwości). Aksjomat (A1) mówi, że jeśli dla wszelkich jednokrotnych przejść, p przed przejściem implikuje q, to po wykonaniu przejścia zachodzi związek $p \square q$. Natomiast aksjomat (A2) mówi, że implikacja w rozumieniu rachunku zdań, jest szczególnym przypadkiem implikacji temporalnej. Reguły wnioskowania, dotyczą zależności pozwalających ze złożonych implikacji uzyskać proste. Przeto (R1) może być rozpatrywana zarówno, jako aksjomat ramowy, albo rolę niezmiennika dodawanego, jako niezależną regułę dodawaną, jako ewentualność. Przykładowo zauważmy, schemat ogólnej całkowitej indukcji, dostarcza szczególny schemat indukcji:

5.0.3.60 Jeśli $p(0) \square q$ oraz $\forall n\ p(n) \square q$ to $p(n+1) \square q$.

Użycie w dowodzie niniejszego podejścia wydaje się bardziej intuicyjnym. Natomiast oparcie się o zasadę 5.0.3.52, prowadzi do dowodu stosującego podejście negowania, pokazując, że nieskończone obliczenia lub błędne są niemożliwe do realizacji. Natomiast niniejsze podejście zapewnia raczej pozytywne rozumowanie, w postaci łańcucha niewidocznych przejść, jedno po drugim, prowadząc do poprawnego zakończenia. Dlatego też, podobnie do innych metod stwierdzania, nie mamy jedynie z formalnym dowodem poprawności programu, ale dostarcza również udowodniony, czytelny wywód poprawności programu.

Dalsze rozważania *Pnueli*, obejmują sformułowanie twierdzenia o zupełności oraz rozstrzygalności stworzonej przez niego tzw. logiki programów, opartej o logikę temporalną oraz prezentację szkicu dowodu części twierdzenia dotyczącą zupełności. Kolejnymi krokami, zaprezentowanymi przez *Pnueli'a*, jest przedstawienie zastosowania opracowanej teorii do programów sekwencyjnych, a następnie rozszerzenie opracowanego podejścia na programy współbieżne (reaktywne) działające w środowisku wieloprocesorowym.

Przyjęte rozszerzenia obejmują pojęcia: (1) niezmienników, (2) pełnego i częściowego przydziału zasobu. Rozważania zilustrowane są przykładem pary współbieżnych programów nazwanej *Producer-Consumer*. Programy te, komunikują się za pośrednictwem semafora i charakteryzują się następującymi własnościami: *Producer* i *Consumer* korzystając z tej samej sekcji krytycznej, przestrzegają wzajemnego wykluczenia równoczesności dostępu do tejże sekcji krytycznej, nigdy nie powodują przepełnienia bufora za pośrednictwem, którego przekazują sobie zasób, nie maj miejsca zakleszczenia pomiędzy nimi. A to dzięki korzystaniu ze wspólnych niezmienników.

W kolejnej części swojej publikacji *Pnueli* wprowadza funkcje i prawa logiki temporalnej oraz dokonuje interpretacji wyłożonej przez siebie teorii poprawności programów sekwencyjnych oraz współbieżnych (reaktywnych), w zachowaniu funkcji logiki temporalnej. Ostatnie dwie części publikacji *Pnueli*, dotyczą programów o skończonym zbiorze stanów systemu oraz porównaniu uzyskanych wyników z pracami innych teoretyków informatyki.

Analityczne dowodzenie poprawności programów komputerowych (sekwencyjnych i współbieżnych) wymaga zarówno posiadania odpowiedniej wiedzy matematycznej, jak i uporu, ale przecież nie każdy obdarzony jest talentem i dysponuje czasem dla prowadzenia formalnego rachunku, niezbędnego dla przeprowadzenia potrzebnego dowodu. Podobnie jednak, jak w

przypadku dowodzenia formuł np. rachunku zdań, możemy zamiast żmudnych rozważań dedukcyjnych, dokonać *walidacji* - sprawdzenia prawdziwości programu komputerowego we wszystkich osiąganych przez ten program stanach, pamiętając, iż każdy program komputerowy jest złożonym wyrażeniem logicznym.

Piśmiennictwo: *Ben-Ari* M. B.2.1., *Pnueli A.* P.3.1., *Trzęsicki K.* T.4.2.

5.1. WALIDACJA CYFROWA SYSTEMU PROGRAMÓW

5.1.0. METODY FORMALNE

Metody formalne, to logiczne podejście do informatyki (dokładniej mówiąc: głównie do oprogramowania i rozwoju systemów), oparte o przestrzeganie formalnych zasad opracowania specyfikacji, projektowania i walidacji programów i systemów programów. Używanie notacji oraz specjalizowanych języków specyfikacji i notacji (jak np. omawiana Notacja Z) o formalnie (logicznie) zdefiniowanym znaczeniu poszczególnych zdań, pozwala na precyzyjne wyrażenie specyfikacji, bez dwuznaczności oraz z ewentualną możliwością dokonania przekształceń tejże specyfikacji (tzw. *refinement*, czyli uszczegółowiania specyfikacji wymagań, czyli rozszerzania specyfikacji wymagań) do postaci zawierającej znacznie więcej szczegółów, wygodniejszej (np. ze względu na wyższą jednoznaczność specyfikacji wymagań) np. dla przyszłej implementacji. Dodatkową zaletą formalnie opisanej specyfikacji, jest możliwość łatwiejszej jej weryfikacji przez klienta, na rzecz, którego tę specyfikację opracowano.

Projektowanie programów i systemów programów w oparciu o sformalizowaną specyfikację, daje możliwość badania zgodności projektu z założeniami. W tym przypadku, możliwe jest również przekształcanie specyfikacji wymagań (translacji) w kod programu lub systemu programów. Takie przekształcanie jednego sformalizowanego zapisu w inny sformalizowany zapis (kod) w innym języku, często również jest określane, jako rodzaj refinement*'u*. Coraz więcej języków programowania, może być wynikiem automatycznego przekształcania np. projektu systemu programów zaprojektowanych w UML, z ewentualnym interakcyjnym wspomaganiem przez projektanta/programistę - do kodu w języku programowania, takim jak C, C#, Java, itd.

Metody formalne odwołują się do rygorystycznego przestrzegania zasad logiki - pisania specyfikacji wymagań, projektowaniu i walidacji, zarówno pojedynczych programów, jak systemów programowania, jak również sprzętu komputerowego. Gdzie „rygorystyczne przestrzeganie zasad logiki" oznacza, w odniesieniu do specyfikacji wymagań, że jest ona zapisana w poprawnych zdaniach języka do pisania wymagań (np. w Notacji Z), umożliwiając weryfikację jednoznaczności i niesprzeczności tychże wymagań. Odnośnie opracowanego projektu, że jest napisany w języku projektowania (np. w UML) i można zweryfikować opracowany projekt, przez formalne stwierdzenie jego zgodności ze specyfikacją wymagań. Np. stwierdzić, że projekt jest uszczegółowieniem specyfikacji wymagań. Odnośnie wreszcie kodu programu czy kod systemu programów utworzonego na podstawie projektu, powinny być spełnione, aż dwa wymagania:

1. Kod jest uszczegółowieniem opracowanego projektu;
2. Kod jest poprawny, w rozumieniu walidacji programu czy też systemu programów.

Zajmijmy się z kolei uszczegółowieniem części pojęć, którymi powyżej posługiwaliśmy się.

5.1.0.10. **Definicja**. *Dowód* to argumentacja logiczna przekonująca o prawdziwości wyrażenia logicznego. W logice formalnej argumentacja taka, nie może pozostawiać żadnych wątpliwości.

5.1.0.20. **Definicja**. *Twierdzenie* to wyrażenie logiczne, którego prawdziwość została udowodniona.

W dowodach często pojawia się kilka podstawowych typów uzasadnień. W informatyce najczęściej występuje trzy podstawowe typy dowodów, które dalej omówimy:

5.1.0.30. **Definicja**. *Dowód konstrukcyjny*. Jeśli mowa o istnieniu obiektu o pewnych szczególnych własnościach, to dowód konstrukcyjny polega na skonstruowanie takiego obiektu posługując się pojęciami podstawowymi oraz np. aksjomatami danego rachunku logicznego.

5.1.0.40. **Definicja**. *Dowód przez sprowadzenie do sprzeczności*. Często metodą dowodzenia twierdzenia jest przyjęcie założenia, że twierdzenie to jest fałszywe i wykazanie, że takie założenie prowadzi do ewidentnie błędnego wniosku nazywanego sprzecznością.

5.1.0.50. **Definicja**. *Dowód przez indukcję*. Dowód indukcyjny to zaawansowana metoda wykorzystywana do pokazania, że wszystkie elementy nieskończonego zbioru mają określoną własność. Każdy dowód indukcyjny zawsze składa się z dwu części: (1) *kroku indukcyjnego* i (2) *podstawy* (bazy) indukcji. Krok indukcyjny polega na wykazaniu, że dla każdego $i \geq 1$ zachodzi implikacja:, jeśli P(i) - jest prawdą, to P(i+1) - jest również prawdą. Podstawa indukcji polega na wykazaniu, że P(1) – jest prawdą. W kroku indukcyjnym założenie, że zachodzi P(i) – nazywamy hipotezą (złożeniem) indukcyjną.

Tak rozumiane metody formalne obejmują zarówno podejście oparte o dowody polegające na dokonywaniu przekształceń formalnych z wykorzystaniem podanych wyżej definicji typów dowodów, jak również uogólnienie metody sprawdzania (patrz 2.2.3), czy dana zależność logiczna jest prawdziwa dla wszystkich możliwych kombinacji wartości argumentów. W dalszym ciągu, wprowadzimy pojęcie metody formalnej w węższym rozumieniu tegoż pojęcia, które ograniczymy do metod dowodu opartych na sprawdzaniu wszystkich przyjmowanych wartości zmiennych (argumentów).

Piśmiennictwo: *Ben-Ari M.* B.2.1., *Guttag J.* G.5.1., *Pnueli A.* P.3.1., *Spivey J.* S.10.1

5.1.1. TROCHĘ HISTORII

W roku 1981 – *Edmund Clarke* oraz *E. Allen Emerson*, Amerykanie pracujący w USA i *Joseph Sifakis*, Grek z pochodzenia pracujący we Francji – autoryzowali bardzo interesującą publikację, dotyczącą obiecującego kierunku prac nad opracowaniem algorytmu walidacji systemów komputerowych. Prace nad algorytmem, który by umożliwiał: zarówno walidację sprzętu komputerowego, jak również oprogramowania oraz protokołów komunikacyjnych. Technologia oparta o zapowiadany algorytm zakładała, że w oparciu o specyfikację systemu (sprzętowego, programowego lub protokołu komunikacyjnego) wyrażoną w postaci zależności logiki temporalnej - powstanie model formalny, który można będzie traktować jak automat skończony i badać poprawność jego działania. Jeśli działanie nie było poprawne, (czyli zgodne ze specyfikacją) założono, że algorytm pozwoli na zbudowanie kontrprzykładu niespełniania specyfikacji. Podstawowym problemem wymagającym rozwiązania w tym podejściu, było poradzenie sobie z tzw. eksplozją liczby stanów automatu skończonego, które należy zbadać.

W ciągu następnych 30 lat, pomysł algorytmu został rozwinięty w szereg dojrzałych technologii walidacji systemów, między innymi do walidacji obwodów scalonych wielkiej skali integracji, programów współbieżnych - reaktywnych, różnorodnych systemów rozproszonych. Prace zainicjowane przez Clarke'a, Emerson'a oraz Sifakis'a były z sukcesem kontynuowane, tworząc liczne środowisko badawcze, rozwijające: nowe logiki specyfikacji wymagań, nowe algorytmy walidacji i uzyskujące zupełnie nowe, zaskakujące wyniki badań. Tematyka dotycząca algorytmów formalnej walidacji systemu, spowodowała powstanie zarówno zespołów badawczych, jak i przemysłowych. Przykładowo wywarła istotny wpływ, na jakość projektowania układów elektronicznych wielkiej skali integracji. Jednym z podstawowych wyników było jednak inne opracowanie - wykonanej przez zespół Geralda Holzmann'a. A mianowicie opracowanie: (1) języka modelowania systemów PROMELA i (2) analizatora modeli napisanych w tym języku PROMELA - procesora SPIN.

W roku 2007, panowie: *Edmund Clarke, E. Allen Emerson* i *Joseph Sifakis* – otrzymali ACM A.M. Turing Award, za stworzenie podstaw teoretycznych procesu walidacji systemów współbieżnych – reaktywnych.

Piśmiennictwo: Clarke E. C.3.1.

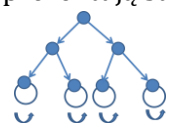
5.1.2. ZARYS ALGORYTMU WALIDACJI

5.1.2.10. **Wyjaśnienie**. Formalna walidacja poprawności programu wymaga użycia logiki matematycznej. Każdy program komputerowy jest dobrze zdefiniowanym wyrażeniem logiki temporalnej. W okresie początków rozwoju informatyki uważano, że dla wykazania poprawności programu komputerowego koniecznym jest przeprowadzenie odpowiedniego dowodu analitycznego poprawności danego programu. Przeciwną sytuację mamy w przypadku formalnej walidacji, która jest podejściem nie wymagającym budowy analitycznych dowodów poprawności. Nie trzeba chyba nikogo przekonywać, że możliwość zastąpienia istotnego wysiłku intelektualnego, niezbędnego do przeprowadzenia dowodu analitycznego, rachunkiem formalnym, jest rozwiązaniem pożądanym.

5.1.2.11. **Wyjaśnienie**. Jeśli program komputerowy daje się opisać z pomocą logiki temporalnej, to staje się automatem skończonym, którego działanie (zachowanie) można badać formalnie. A to z kolei jest sugestią, jak ma działać algorytm nazywany w języku angielskim *model checking.*

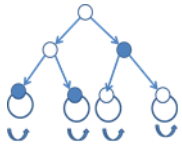
5.1.2.12. **Wyjaśnienie**. Jak wiemy, sugestia użycia do opisu specyfikacji programu w kategoriach wyrażeń logiki temporalnej pochodzi od *Amina Pnueli*. Używał on logiki LTL, operując funkcjonałami: $\Box$ - zawsze, <> - czasami, $\circ$ – następny chron czasu, oraz $\boldsymbol{U}$ – dopóki. Kwantyfikatory, są nieco inaczej interpretowane: $\forall$ – przez wszystkie kolejne chrony działania, $\exists$ – przez wybrany przynajmniej jeden chron z okresu działania.

5.1.2.13. **Wyjaśnienie**. Wyrażenie LTL: $\forall\Box p$ - we wszystkich przyszłych stanach, zawsze $p \equiv$ true (porównaj rys. 5.2.1.14, kółka reprezentują stany, zaciemnienie odpowiada $p \equiv$ true).



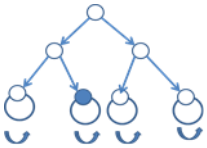
Rysunek 5.1.2.14. Interpretacja wyrażenia LTL - $\forall\Box p$

5.1.2.15. **Wyjaśnienie**. Wyrażenie LTL: $\forall <> p$ - wśród przyszłych stanów będą takie, że $p \equiv$ true (porównaj rys. 5.2.1.16, kółka reprezentują stany, zaciemnienie odpowiada $p \equiv$ true, brak zaciemnienia $p \equiv$ false).



Rysunek 5.1.2.16. Interpretacja wyrażenia LTL - $\forall <> p$

5.1.2.17. **Wyjaśnienie**. Wyrażenie LTL: $\exists <> p$ - wśród przyszłych stanów jest taki, że $p \equiv$ true (porównaj rys. 5.2.1.18, kółka reprezentują stany, zaciemnienie odpowiada $p \equiv$ true, brak zaciemnienia $p \equiv$ false).

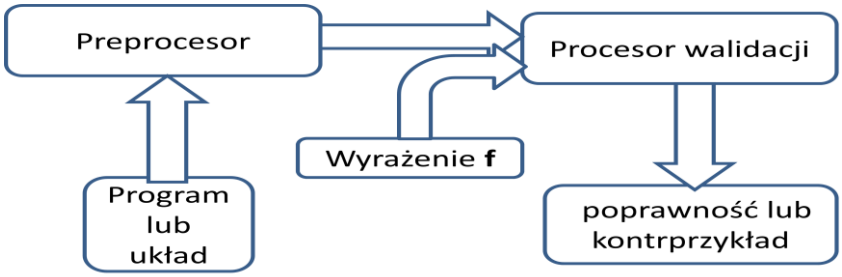


Rysunek 5.1.2.18. Interpretacja wyrażenia LTL - $\exists <> p$

5.1.2.19. **Wyjaśnienie**. Wyrażenie LTL można interpretować jako pewien graf stanów, zwanych również strukturami Kripke'go M rozpostarty nad zbiorem stanów S, ze zdefiniowaną relacją przejścia $R \subseteq S \times S$, etykietowanymi zbiorem etykiet L, z ewentualnie określonym stanem początkowym s_0.

5.1.2.20. **Wyjaśnienie**. Wyrażenie LTL: $\Box\neg(C_1 \wedge C_2)$ zapewnia wzajemne wykluczenie w dostępie do sekcji krytycznej, w tym przypadku procesów C_1 oraz C_2 – odpowiednio.

5.1.2.21. **Wyjaśnienie**. Techniczne sformułowanie algorytmu „*model checking*" jest następujące: Dana jest skończona struktura M, stan s oraz wyrażenie LTL f, wyznaczamy zbiór $\{s : M, s \models f\}$ i sprawdzamy czy we wszystkich punktach s – spełniane jest wyrażenie $f \equiv true$. Wizja schematu realizacji algorytmu „*model checking*" pokazana jest na rys. 5.1.2.23.



Schemat 5.1.2.23. Wizja realizacji algorytmu „model checking" wg Clarke, Emerson i Sifakis

5.1.2.22. **Wyjaśnienie**. SAT jest problemem sprawdzania, czy formuła (wyrażenie) rachunku zdań w postaci CFN (*conjunctive normal form*), czyli koniunkcyjnej postaci normalnej, po podstawieniu wartości *true* (prawda) na wszystkie zmienne formuły, uzyskuje wartość *true* (prawda). Należy zauważyć, że problem SAT jest klasy NP-zupełna. W ciągu ostatnich 20 lat, zostały opracowane efektywne narzędzia typu *SAT-SOLVER* , które są z powodzeniem wykorzystywane w tzw. podejściu BMC (*bounded model checking*) do sprawdzania poprawności modeli logicznych sprzętu komputerowego (np. obwodów scalonych) i oprogramowania. W wyniku zastosowania podejścia BMC do badanego modelu logicznego w LTL, otrzymujemy:

stwierdzenie poprawności działania modelu albo kontrprzykład na brak poprawności działania modelu.

5.1.2.24. **Wyjaśnienie**. Podstawowa idea BMC, jest stosunkowo prosta, prowadząc do poszukiwania ścieżki, o długości k, kolejnych stanów s_i prowadząca do cyklu (pętli stanów), gdzie w każdy z tych stanów s_i nie jest spełniony predykat p = f , gdzie f wyrażeniem, o którym mowa na rys. 5.1.2.23 (czyli spełnione jest formuła odwrotna $\neg f$), jak pokazano na tys. 5.1.2.25.

Rysunek 5.1.2.25. Schemat kontrprzykładu o długości ścieżki co najmniej k stanów

Zakładając że nasz system M - przejść pomiędzy stanami, posiada łącznie n stanów. Każdy stan może być opisany jako wektor $\vec{v}$ zmiennych Boole'owskich. Zbiór stanów początkowych określimy jako wyrażenie logiczne $I(\vec{v})$, gdzie wektorowi $\vec{v}$ przypisano wartości stanów początkowych. Podobnie, relacje przejścia określa wyrażenie logiczne $R(\vec{v}, \vec{v}')$. Ścieżkę przejścia pomiędzy kolejnymi stanami o długości k rozpoczynającą się w stanie początkowym, można zapisać w sposób następujący:

5.1.2.26 $\quad path\ (k) = I\ (\vec{v}_0) \wedge R\ (\vec{v}_0, \vec{v}_1) \wedge \ldots \wedge R\ (\vec{v}_{k-1}, \vec{v}_k).$

Ścieżka kończy się pętlą (cyklem)

5.1.2.27 $\quad cycle\ (k) = R\ (\vec{v}_k, \vec{v}_0) \vee \ldots \vee R\ (\vec{v}_k, \vec{v}_{k-1}) \vee R\ (\vec{v}_k, \vec{v}_k).$

wtedy i tylko wtedy, jeżeli wyrażenie f, jest fałszywe w każdym z k kroków:

5.1.2.28 $\quad property\ (k) = \neg f\ (\vec{v}_0) \wedge \neg f\ (\vec{v}_1) \wedge \ldots \wedge \neg f\ (\vec{v}_k).$

Kontrprzykład o długości k wystąpi tylko wtedy, jeśli koniunkcja wyrażeń, jest spełniona (przyjmuje wartość *true*):

5.1.2.29 $\quad \Omega(k) = path\ (k) \wedge cycle\ (k) \wedge property\ (k).$

5.1.2.30. **Algorytm.** Rozpatrzmy następujący algorytm: Rozpoczynamy dla k = 1. Jeśli formuła $\Omega(k)$ jest spełniona, to wiemy, że istnieje kontrprzykład o długości k. Ścieżka wykonanego przejścia od stanu początkowego do stanu k, może być uzyskana z $\Omega(k)$. Jeśli formuła $\Omega(k)$ nie jest spełniona, to mamy do czynienia albo z przypadkiem gdy system działa poprawnie dla wszystkich ścieżek przejścia rozpoczynających się w stanie początkowym, albo iż istnieje kontrprzykład o ścieżce dłuższej niż k. Kiedy formuła $\Omega(k)$ nie jest spełniona, to mamy dwie możliwości: (1) zwiększyć wartość k i szukać dłuższego kontrprzykładu, albo (2) zatrzymać postępowanie, jeśli przekraczamy dostępną pojemność pamięci lub czas badania.

5.1.2.40. **Wyjaśnienie**. Dowód poprawnośći działania algorytmu podanego w 5.1.2.30, jest dość skomplikowany. Dowód wykorzystuje istnienie punktu stałego (patrz twierdzenie *Knastra-Tarskiego*, podrozdział 2.2.8). Przykładowo, niech $f(Z)$ oznacza $p \vee \forall(\circ\ (Z))$. Widzimy, że $\forall(<>p\ ) = f\ (\forall(<>p))$ jest punktem stałym funkcji $f(Z)$, ponieważ $\forall(<>p)$ ma zadaną wartość *true* (prawda), wtedy i tylko wtedy p ma zadaną wartość *true* (prawda) lub $\forall(\circ\ \forall(<>p\ ))$ ma tą zadaną wartość *true* (prawda). Ogólnie mówiąc, może występować kilka punktów stałych. Można pokazać, że $\forall(<>p)$, jest najmniejszym punktem stałym, który oznaczymy $mZ = f(Z)$, gdzie $f(Z)$ jak wyżej. Punkt stały oznaczony jako $mZ = f(Z)$, tworzy iteracyjny zbiór stanów dla których

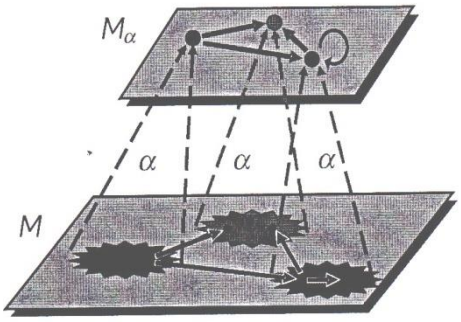
$\forall(<>p)$ ma zadaną wartość *true*. Wykorzystujemy tu fakt, że każda formuła odpowiada zbiorowi stanów, w których przyjmuje ona wartość *true*(prawda). Wyznaczamy, maksymum rosnącego zbioru stanów: $false \subseteq f(false) \subseteq f^2(false) \subseteq \ldots \subseteq f^k(false) = f^{k+1}(false)$, gdzie k jest rozmiarem (skończonym) tej przestrzeni stanów. Twierdzenie *Knastra-Tarskiego*, zezwala na prowadzenie interaktywnego rachunku $\bigcup f^i(false)$ dowolnej zależnej od czasu własności *r*, charakteryzującej się posiadaniem co najmniej jednego punktu stałego $\mu Z = f(Z)$, zakładając monotoniczność $f(Z)$, co jest zapewnione, jeśli Z nie ma wartości zanegowanych. W przypadku liniowej logiki temporalnej (LTL) walidacja modelu M, może być wykonana w czasie $O(|M| \cdot \exp(|f|))$, gdzie $|M|$ jest liczbą osiągalnych stanów przez model M (programu lub układu), zaś $|f|$ jest liczbą stanów w których kryterium formuły *f* nie jest spełnione (z reguły jest to liczba mała) – składnik wykładniczy można zakceptować (patrz: Złożoność czasowa i analiza algorytmów, w podrozdziale 3.8.5.).

Piśmiennictwo: Clarke E. C.3.1.

5.1.3. TWORZENIE ABSTRAKCJI I PROBLEM EKSPLOZJI LICZBY OSIĄGANYCH STANÓW

Jednym z podstawowych problemów wymagających rozwiązania było opracowanie algorytmu umożliwiającego zastąpienie badanego obiektu (oprogramowania albo obwodu scalonego), uproszczonym modelem logicznym opisującym badany obiekt ze względu na istotne cechy działania, takie np. jak sterowanie wyboru drogi przejścia pomiędzy istotnymi stanami.

5.1.3.10. **Wyjaśnienie**. Abstrakcją badanego obiektu nazwano - model logiczny obiektu. Przyjęto konwencję oznaczania abstrakcji systemu $M = \langle S, s_0, R, L\rangle$ jako $M_a = \langle S_a, s_0^a, R_a, L_a\rangle$, ze względu na odwzorowanie *a* (przykładowo odwzorowaniem stanu początkowego s_0 jest stan abstrakcji s_0^a). Zakładamy przy tym, że odwzorowujemy wybrane elementy niepodzielne obiektu (*atomic*), tym samym powodujemy, że pewne zachowania obiektu nie mają swoich odpowiedników w zachowaniu abstrakcji obiektu (patrz rys. 5.1.3.11). Dlatego też, kontrprzykłady znalezione dla abstrakcji obiektu, nie muszą automatycznie być również kontrprzykładami dla obiektu.



Rysunekt 5.1.3.11. Pierwotna zasada tworzenia abstrakcji dla walidacji na podstawie obiektu

5.1.3.20. **Wyjaśnienie**. Może się okazać, że abstrakcja nie odwzorowuje istotnych cech obiektu, w wyniku walidacji utworzony kontrprzykład nie dotyczy obiektu wyjściowego. Dlatego też, opracowano metodę ze sprzężeniem zwrotnym (patrz rys. 5.1.3.21) nazwanej CEGAR (*Counterexample Guided Abstraction Refinement*).

5.1.3.30. **Wyjaśnienie**. Eksplozja liczby stanów przyjmowanych przez walidowany program lub układ, okazał się centralnym problemem opracowania algorytmu „*model checking*". Asynchroniczna kompozycja n - współbieżnych procesów, z których każdy może przyjmować m różnych stanów, prowadzi łącznie do wyróżnienia n^m – przejść pomiędzy stanami. Co to

oznacza? Dla systemów współbieżnych z mała liczbą procesów, z których z kolei każdy proces ma niewielką liczbę stanów, nie nastręcza to trudności. Natomiast dla systemów o dużej liczbie współbieżnych procesów, globalna liczba występujących stanów i relacji przejścia między nimi jest zbyt duża, żeby można sobie z nimi poradzić. Jesienią 1987 roku, *McMillan* kończący studia na Uniwersytecie Carnegie Mellon, stwierdził korzystając z symbolicznej reprezentacji systemu relacji przejść pomiędzy stanami, że możliwym jest weryfikacja znacznie większych systemów.

Rysunek 5.1.3.21. Schemat działania metody CEGAR

5.1.3.31. **Wyjaśnienie**. Ta nowa symboliczna reprezentacja oparta była wymyślone przez *Bryant'a* "*ordered binary decision diagrams* (w skrócie OBDDs)". Zapis OBDD dostarcza kanoniczną postać wyrażeń Bool'owskich, które przeważnie dostarczają znacznie bardziej zwięzły zapis, niż normalna kanoniczna postać wyrażeń oparta o alternatywę, koniunkcję i negację, zaś algorytmy w zapisie OBDD są krótsze i wygodniejsze do operowania nimi. Reprezentacja symboliczna wychwytuje pewne regularności występujące w przestrzeni stanów, wynikające z powtarzalności oraz protokołów powiązań pomiędzy procesami, co z kolei umożliwia walidację systemów o bardzo dużej liczbie osiąganych stanów. Umożliwiło to walidowanie systemów osiągających 10^{20} stanów, a po udoskonaleniu techniki OBDD, możliwości wzrosły do walidacji systemów osiągających ponad 10^{120} stanów.

5.1.3.40. **Wyjaśnienie**. Walidacja systemów programów (oprogramowania) dostarcza istotnych problemów dla tworzonych algorytmów. Oprogramowanie ma tendencję do mniejszej strukturyzacji niż układy sprzętowe. Dodatkowo, współbieżny – reaktywne oprogramowanie jest z zasady asynchroniczne, czyli większość aktywności realizowanych przez poszczególne procesy, jest wykonywana niemal niezależnie, bez wspólnego zegara synchronizacji. Z tego powodu, problem eksplozji stanów jest szczególnie poważny dla programów. Dlatego też, w pierwszej kolejności rozwinęły się metody walidacji układów elektronicznych (urządzeń). Postęp odnośnie oprogramowania, a właściwie dotyczący systemów asynchronicznych, nastąpił dopiero po opracowaniu techniki nazwanej: *partial order reduction.* Ta technika z kolei wykorzystuje niezależność równolegle pojawiających się zdarzeń. Intuicyjnie mówiąc, dwa zdarzenia są pomiędzy sobą niezależne, jeśli osiągnięty wynik prowadzi do takiego samego stanu globalnego przestrzeni stanów. Algorytmy wykorzystujące technikę *partial order reduction*, są opisane w wielu niezależnych publikacjach. Przykładowo, procesor SPIN opracowany przez zespól Holzmann'a wykorzystuje z powodzeniem tą technikę.

Piśmiennictwo: Clarke E. C.3.1.

5.1.4. PRAKTYKA INŻYNIERSKA A ELEMENTY KONSTRUKCJI MODELU

Projektowanie systemu informatycznego jest procesem odkrywczym. Specyfikacja wymagań na system informatyczny (program lub system programów) stwarza ramy, które należy wypełnić szczegółowymi rozwiązaniami, które następnie dadzą się przetłumaczyć na kod. Duże programy

sekwencyjne, typowo są podzielone na mniejsze moduły, z których każdy realizuje dobrze określoną funkcję. Interfejsy pomiędzy tymi modułami, są z zasady możliwie jak najmniejsze, redukując tym samym liczbę założeń dotyczących zadań, jakie dany moduł musi wykonywać na rzecz pozostałych modułów. W tym celu, interfejsy są przeznaczone do przekazywania danych (*data-oriented*), a nie do przekazywania sterowania (*control-oriented*). Moduły typowych sekwencyjnych aplikacji nie zawierają wewnętrznej informacji o swoich stanach niezależnie od pozostałych modułów programu.

Zupełnie inaczej wygląda to w systemach rozproszonych (np. sieciowych). W systemach rozproszonych struktura modułowa jest typowo wyznaczana przez fizyczny podział poszczególnych komponentów części sprzętowej systemu. Każdy moduł systemu rozproszonego posiada własny system sterowania, który w szczególności przechowuje informacje o stanie. W tym przypadku sterowanie, a nie same dane – są podstawowym czynnikiem przekazywanym przez interfejsy.

Podobnie jak w innych działach techniki, w przypadku projektowania rozproszonego systemu informatycznego, zasadniczą sprawą, jest poprzedzenie projektowanie ostatecznego już rozwiązania, zbudowanie wstępnego modelu współpracy poszczególnych modułów systemu rozproszonego. Modelu, który nacelowany jest - na sprawdzenie poprawności współdziałania pomiędzy poszczególnymi modułami systemu, przy jednocześnie minimalizacji funkcjonalności przetwarzania danych, realizowanej przez poszczególne moduły. Model taki, nie musi stać się ostatecznie częścią konstrukcji finalnego rozwiązania, czyli systemu wynikowego. Ale podobnie jak w innych dziedzinach techniki, służy sprawdzeniu poprawności przyjętych rozwiązań w zakresie sterowania złożonym wielomodułowym rozwiązaniem.

Budując model sterowania systemu współbieżnego, należy pamiętać o dwu podstawowych zasadach:
1. Model opisuje zawsze zachowanie systemu o skończonej liczbie osiągalnych stanów;
2. Model musi być zbudowany według jednoznacznie określonej specyfikacji wymagań.

Żeby *walidować* system musimy opisać dwie rzeczy: *zbiór faktów*, które chcemy zweryfikować; oraz związane z nimi *aspekty systemu*, które są potrzebne do weryfikacji tych faktów. Badając typy faktów możemy chcieć uzyskać dowody dotyczące systemów rozproszonych. Zaczniemy od opisu zachowań systemów rozproszonych na względnie wysokim poziomi abstrakcji, tak, aby automatyzacja weryfikacji istotnych faktów zachowania systemu było możliwym.

Piśmiennictwo: *Holzmann G.* H.3.1., H.3.2.

5.1.5. MODEL: ASYNCHRONICZNE PROCESY, DANE ORAZ KANAŁY KOMUNIKAJI

Bazowymi blokami konstrukcyjnymi modeli, są procesy asynchroniczne, buforowane oraz niebuforowane kanały przekazywania komunikatów pomiędzy procesami, wyrażenia synchronizujące oraz dane strukturyzowane. Rozmyślnie, nie mówimy o notacji dotyczącej czasu, czy też zegara, nie posługujemy się liczbami zmiennoprzecinkowymi. Te ograniczenia powodują trudności przy modelowaniu obliczeń, ale równocześnie umożliwiają modelowanie systemu sterowania programu i weryfikację zachowania klientów i serwerów w sieci procesorów.

Poszczególne procesy składają się z sekwencji czynności, w szczególności z czynności niepodzielnych, czyli atomowych (*atomic*). Czynności mogą zawierać warianty wykonania, (czyli

rozgałęzienia) oraz pętle (rekursje). Każdy utworzony proces, a dokładniej mówiąc każda czynność utworzonego procesu, może mierz maksymalnie dwa przełączane statusy: (1) wykonywany; (2) blokowany. Status *wykonywany* czynności oznacza, że czynność (a tym samym proces w skład, którego wchodzi) jest wykonywana. Status *blokowany* czynności oznacza, że czynność nie jest wykonywana (a tym samym proces - w skład, którego wchodzi) jest niewykonywany tylko zablokowany. Stan każdego statusu zależy od spełnienia (*true*) albo niespełnienia (*false*) danego warunku w wyrażeniu logicznym i może się wielokrotnie zmieniać w toku działania modelu. Należy podkreślić, że część typowych czynności ma zawsze status wykonywany.

Nowo utworzony proces – może, ale nie musi, rozpocząć działanie (aktywność) natychmiast po swoim utworzeniu. Podobnie, nowy proces może, ale nie potrzebuje – ogólnie nie zakończyć swoje działanie, zanim proces, który go utworzył przejdzie do następnej instrukcji. Czyli: proces nie zachowuje się jak funkcje. Każdy proces, niezależnie jak został utworzony, określa pewną asynchroniczną nić działania, która może przeplatać się z niezależnym wykonywaniem instrukcji innych procesów.

Przyjmiemy założenie, że modele mają rozpiętość tylko dwu poziomów: poziom *globalny* oraz poziom *lokalny* procesu. Oczywiście, na każdym poziomie – wszystkie obiekty danych (zmienne i stałe), muszą zostać zadeklarowane, zanim pierwszy raz ma miejsce odwołanie się do nich. Ze względu na fakt, że niema poziomu pośredniego rozpiętości, rozpiętość poziomu globalnego danej zmiennej, nie może być ograniczona do jakiegoś podzbioru procesów, jak również rozpiętość lokalnej zmiennej procesu nie może być ograniczony do wskazanych czynności procesu. Można odwoływać się do lokalnej zmiennej, od punktu jej deklaracji do końca ciała opisu procesu, w którym się pojawia. Zmienne i stałe zawsze muszą mieć ściśle określone granice przedziałów wartości.

Kanały komunikacji są używane do modelowania wymiany danych (komunikatów) pomiędzy procesami. Kanał może być deklarowany zarówno lokalnie jak i globalnie. Posługiwać się będziemy dwoma typami kanałów: (1) kanałem bezpośredniego przekazywania komunikatu, oraz (2) kanałem buforowanym, gdzie bufor ma określoną pojemność przeznaczoną do przechowywania komunikatów. Z kanałami związane są dwa typy operacji: (1) przekazywania komunikatu do kanału, oraz (2) odczytywanie komunikatu z kanału.

Tak, więc, model zbudowany jest z trzech podstawowych typów obiektów:
- Procesów asynchronicznych
- Obiektów danych
- Kanałów komunikacji pomiędzy procesami.

Celem użycia modelu nie jest weryfikowanie aspektów obliczeniowych aplikacji, ale odpowiedzialna identyfikacja problemów, jakie mogą pojawić się przy współdziałaniu procesów. Oznacza to, że interesują nas zawiłe problemy koordynacji procesów, a nie własności lokalnych sekwencyjnych lub deterministycznych obliczeń. Model walidacji programu lub systemu programów – różni się, co najmniej w dwu podejściach, od programu napisanego w typowym języku programowania, np. w C:
- Model reprezentuje *abstrakcję* projektu i zawiera tylko te aspekty projektowanego systemu, które odpowiadają własnościom – podlegającym walidacji.
- Model zawiera z reguły elementy, które nie są częścią projektowanej realizacji. To może być np. założenia dotyczące skrajnie pesymistycznych zbiegów okoliczności zachowania

środowiska, w którym systemem działa, mogącego również oddziaływać na system, a co ważniejsze, może zawierać bezpośrednio lub pośrednio specyfikację pożądanej poprawności.

Chociaż mogłoby być to atrakcyjnym, posiadanie jednej wspólnej specyfikacji - opracowaną zarówno dla potrzeb modelu oraz implementacji projektowanego systemu – walidacja oraz implementacja mają zasadniczo inne cele. Model (w naszym rozumieniu), może być porównany ze względu na swoje przeznaczenie do prototypu lub modelu projektowego w konstrukcjach inżynierii lądowej, a służy do udowodnienia poprawności założeń konstrukcyjnych. Modele projektowe inżynierii lądowej z reguły nie staje się częścią docelowo wdrażanego systemu.

System informatyczny przeznaczony do wdrożenia, typowo zawiera znacznie więcej funkcjonalności oraz szczegółów, niż jego model. Oznacza to, że trudno jest znaleźć transformację dla przekształcania modelu w implementowany projekt systemu. Oczywiście transformacja w odwrotna – nie jest już tak trudną.

Celem badań modelu systemu jest określenie, co jest możliwe – a co nie. Często, takie stwierdzenie, co jest logicznie możliwe - będzie przedmiotem pewnego zbioru rozważań na temat kontekstu wykonywania modelu, takich jak możliwe zachowania zewnętrznych czynników, z którymi model wzajemnie oddziałują. Wykonując logiczną weryfikację, jesteśmy szczególnie zainteresowani w określeniu, kiedy specyfikacja wymagań projektu mogłyby zostać naruszona, kiedy i w jakim stopniu jest to możliwe, a w jakim naruszenie specyfikacji wymagań nie jest możliwe.

Dramatyczne awarie systemu są niemal zawsze wynikiem pozornie niemożliwych sekwencji zdarzeń: to dokładnie tak, kiedy pewne sekwencje zdarzeń, są pomijane w rozważaniach fazy projektowania. Kiedy jednak zrozumiemy, jak oryginalna specyfikacja założeń projektowych może zostać naruszona, to należy wprowadzić (w specyfikacji wymagań) odpowiednie zmiany, aby zapobiegać błędom działania. Poprawność logiczna, jest w pierwszej kolejności związana z możliwością, a nie z prawdopodobieństwem. Koncentrowanie się na możliwości, a nie prawdopodobieństwie, ma dwie konsekwencje. Po pierwsze, należy podkreślić, że dowód poprawności otrzymujemy, jako wynik weryfikacji modelu. Jeśli weryfikator powie nam, że niema możliwości naruszenia zadanych wymagań, to brzmi istotnie mocniej – niż werdykt, że naruszenie poprawności ma niskie prawdopodobieństwo wystąpienia. Po drugie, ograniczenie mówiące o możliwości wykonania weryfikacji bardziej efektywnie, jeśli również uwzględniamy w rozważaniach prawdopodobieństwo wystąpień poszczególnych scenariuszy. Wyniki oceny prawdopodobieństw, są przecież podkopane przez trudności przypisania jakiejś metryki prawdopodobieństwom wystąpienia poszczególnych zdarzeń w modelu systemu. Błąd ocen może ograniczyć wartość uzyskanego wyniku. Powinniśmy być w stanie, sprawdzić istotę logiki właściwości - poprawności rozproszonego systemu, niezależnie od każdego założenia dotyczącego relatywnej szybkości wykonywania procesów, gdzie czas jest rozumiany, jako szybkość wykonywania poszczególnych czynności, lub prawdopodobieństwo wystąpienia poszczególnych typów zdarzeń, takich jak strata komunikatu w kanale transmisji lub błąd urządzenia zewnętrznego.

Dowód poprawności algorytmu, jest również niezależny od implementacji. W szczególności, poprawność algorytmu – również nie zależy od tego, na jak szybkim komputerze jest implementowany. Dlatego też w procesie walidacji nie możemy przyjmować takich założeń.

Przedstawione dalej zasady, są specyficzne dla obszaru naszych zainteresowań:, czyli dla weryfikacji oprogramowania systemów rozproszonych. Inne zasady dotyczą np. weryfikacji poprawności działania urządzeń sprzętowych. Poprawność działania układu scalonego może krytycznie zależeć od opóźnienia propagacji sygnału oraz szybkości działania poszczególnych elementów obwodu. Czas propagacji sygnału oraz rozmieszczenie elementów obwodu, są częścią funkcjonalności i projektu obwodu scalonego: nie może być niezależnie zmieniana. Poprawność protokołu komunikacyjnego lub działanie rozproszonego systemu operacyjnego, z drugiej strony, nigdy nie może zależeć od takich czynników. Szybkość wykonywania systemu oprogramowania, na pewno będzie się zmieniać w sposób zasadniczy – w ciągu cyklu życia danego projektu.

Przy projektowaniu systemów rozproszonych, jest standardem rozróżnianie pomiędzy dwoma typami wymagań poprawności: bezpieczeństwo systemu oraz żywotność systemu. Bezpieczeństwo jest zazwyczaj określane, jako zbiór własności, których system nie może naruszyć, natomiast żywotność jest określana, jako zbiór własności, które system musi spełniać. Przeto bezpieczeństwo określa *złe* sytuacje, których system powinien unikać, zaś żywotność określa *dobre* sytuacje, które realizuje funkcjonalność systemu. Funkcja walidacji modelu systemu, nie może stwierdzić, co jest dobre a co złe; może jedynie pomóc projektantowi, wskazując, co jest *możliwe*, a co nie jest możliwe. Z punktu widzenia walidatora, mają miejsce dwa typy żądań poprawności: żądanie dotyczące osiągania albo nie osiągania pewnych stanów oraz żądanie dotyczące możliwości albo niemożliwości wykonania (np. określonej sekwencji stanów). Tę pierwszą czasami nazywamy *własności stanu* (*state properties*), tę drugą *własnościami ścieżki* (*path properties*). Ścieżka jak i wykonanie, może być skończona albo nieskończona (np. pętla).

Prostym typem własności stanu jest *niezmienniczość* (*invariant*) modelu systemu, która zapewnia trzymanie się osiągalnych stanów składających się na - wykonywaną ścieżkę przejść. Trochę słabszą wersją, jest - *proces zapewniony* (*process assertion*), który trzyma się tylko stanów wyspecyfikowanych. Własności stanów wzięte razem, po połączeniu – tworzą własność ścieżki. Na przykład, własnością ścieżki jest: każde osiągnięcie stanu o własności P, musi ostatecznie prowadzić do przejścia do stanu o własności Q, a to po bezpośrednim osiągnięciu stanu R. Pewne typy właściwości są tak podstawowe, że nie wymagają dodatkowych wyjaśnień. Jedną z takich własności jest, przykładowo, nieobecność osiągania stanu *zakleszczenia systemu*. Oczywiście, użytkownik może modyfikować semantykę wbudowując sprawdzanie na drodze prostego etykietowania czynności. Przykładowo, zakleszczenie można domyślnie rozważane, jako niezamierzone osiągnięcie stanu końcowego systemu. Możemy powiedzieć walidatorowi, że pewne specyfikowane stany końcowe są celowo etykietowane, jako stany końcowe. Własności poprawności są formułowane poprzez użycie np. następujących konstrukcji:

- Stwierdzenia podstawowe (*Basic assertions*)
- Etykiety stanu końcowego (*End-state labels*)
- Etykiety stanu postępu (*Progress-state labels*)
- Etykieta stanu akceptacji (*Accept-state labels*)
- Żądania niemożliwe (*Never claims*)
- Stwierdzenia trasy (*Trace assertions*)

Żądania niemożliwe można napisać ręcznie, można je generować automatycznie z formuły logicznej lub z opisu własności zależności od czasu.

Piśmiennictwo: *Holzmann G.* H.3.1., H.3.2.

5.1.6. AUTOMAT BÜCHI – A WALIDACJA PROGRAMÓW WSPÓŁBIEŻNYCH

Kolejnym pojęciem, które wprowadzamy, jest język ω-regularny. Klasa języków ω-regularnych, jest rozszerzeniem klasy języków regularnych na języki o nieskończenie długich słowach.

5.1.6.10. **Definicja**. L jest językiem ω-regularnym, gdy: A^{ω} – jest zbiorem słów nieskończenie długich, zbudowanych w wyniku nieskończenie krotnej konkatenacji słów ze zbioru A, nie zawierającego pustych słów.

Języki ω-regularne, stanowią klasę języków ω, będącą uogólnieniem języków regularnych przez dopuszczenie języków zbudowanych ze słów nieskończenie długich. Język L jest ω-regularnym, jeśli jego wyrazy tworzące słownik A^{ω}, powstają z wyrazów należących do słownika A niepustego języka regularnego (języka nie zawierającego słów pustych), po przez nieskończenie krotną konkatenację jego wyrazów.

W informatyce, jak również w teorii automatów, automat Büchi (należący do klasy ω-automatów) jest rozszerzeniem pojęcia automatu skończonego poprzez przyjęcie słów o nieskończonej długości. Automat Büchi akceptuje słowa wejściowe o nieskończonej długości, które pozwalają uzyskiwać przebiegi o skończonej liczbie stanów, który osiąga (przynajmniej jeden) stan końcowy nieskończenie często. Automat Büchi rozpoznaje języki ω-regularne, zawierającą wersję o nieskończenie długich słowach językach regularnych. Nazwa automatu pochodzi od nazwiska Juliusa Richarda Büchi, szwajcarskiego matematyka, który opublikował w 1962 roku, wynik swoich badań nad wynalezionym przez siebie nowym rodzajem uogólnionego automatu[87]. ω-automaty są w szczególności przydatne do badania zachowań systemów, które z założenia działają bez przewidywanego zakończenia swojej pracy, tak jak niektóre typy urządzeń automatycznych, systemy operacyjne oraz systemy automatycznego sterowania. Dla takich systemów, można chcieć określić własność, jako „każde żądanie – powoduje określoną reakcję", lub poprzez negację „niema żądania, którego następstwem jest nieokreślona czynność". To drugie sformułowanie odpowiada nieskończonemu słowu: nie można podać skończonej sekwencji, która spełnia taką własność.

Do klasy ω-automatów należą obok automatu Büchi, automat Rabina, automat Streetta, automat „parity" i automat Mullera, każdy z nich zarówno w formie deterministycznej, jak również niedeterministycznej. Poszczególne klasy ω-automatów różnią się między sobą sformułowaniami warunku akceptacji. Każdy z tych ω-automatów rozpoznaje regularne ω-języki, za wyjątkiem deterministycznego automatu Büchi, który jest znacznie słabszy niż pozostałe ω-automaty. Chociaż wszystkie te rodzaje ω-automatów rozpoznają ten sam typ ω-języków, to jednak różnią się w zwięzłości reprezentacji danego ω-języka. Automaty Büchi, są w szczególności wykorzystywane w procesie „*model checking*" w powiązaniu z liniową logiką temporalną LTL.

Przyjmiemy następującą notację dla nieskończonych słów: Niech Σ będzie skończonym alfabetem. Nieskończone słowo $\alpha \in \Sigma^{\omega}$ jest nieskończoną sekwencją symboli należących do Σ. Możemy przedstawić α, jako funkcję - $\alpha : N_0 \rightarrow \Sigma$, gdzie N_0 jest zbiorem liczb naturalnych

---

[1] Büchi, J.R. ,"On a decision method in restricted second order arithmetic". *Proc. International Congress on Logic, Method, and Philosophy of Science. 1960* (Stanford: Stanford University Press): 1–12.

{0, 1, 2, …}. Wprowadzimy notację α(i) oznacza, że ta litera pojawia się na i-tej pozycji słowa wejściowego.

Ogólnie, jeśli S jest zbiorem, zaś σ jest nieskończoną sekwencją symboli należących do zbioru S – to, $\sigma : N_0 \to \Sigma$ – to inf(σ) oznacza zbiór symboli z S, pojawiających się nieskończenie często w słowie σ. Formalnie $\inf(\sigma) = \{s \in S \mid \exists^{\omega} n \in N_0 : \sigma(n) = S\}$, gdzie $\exists^{\omega}$ oznacza kwantyfikator „istnieje nieskończenie wiele".

5.1.6.11 **Definicja**. Automat, jest trójką uporządkowaną $A = (S, R, S_{in})$, gdzie S jest zbiorem stanów, przy czym zbiór stanów początkowych, jest podzbiorem $S_{in} \subseteq S$, zaś $R \subseteq S \times \Sigma \times S$ jest relacją przejścia.

5.1.6.12 **Definicja**. Przebieg. Niech $A = (S, R, S_{in})$ będzie automatem oraz $\alpha : N_0 \to \Sigma$ niech będzie słowem wejściowym. Przebiegiem - nazywamy nieskończoną sekwencję $\sigma : N_0 \to S$ taki, że $\sigma(0) \in S_{in}$ oraz dla wszystkich $i \in N_0$,

$$\sigma(i)\ R(\alpha(i))\ \sigma(i+1)$$

Tak, więc przebieg, jest „legalną" sekwencją stanów automatu, które przechodzi automat czytając słowo wejściowe. Ogólnie, słowo wejściowe może odpowiadać wielu przebiegom, ze względu na niedeterminizm. Dlatego też, niedeterministyczny automat może mieć stany, które nie odpowiadają przejściom dla pewnych wejściowych liter, jest również możliwym, że nie spowoduje przebiegu w tym szczególnym przypadku. Każdy potencjalny przebieg prowadzi do stanu, z którego niema możliwości przejścia odpowiadającego kolejnej literze słowa wejściowego. Natomiast, jeśli automat jest deterministyczny, każdemu słowu wejściowemu odpowiada dokładnie jeden przebieg.

5.1.6.21 **Definicja**. Deterministyczny automat Büchi jest piątką uporządkowaną:
$A = (Q, \Sigma, \delta, q_0, \mathbf{F})$, złożoną z następujących składowych:

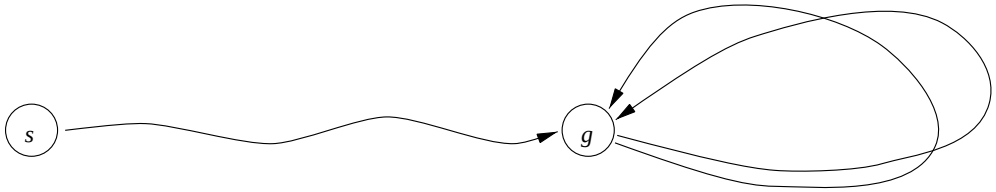
- Q – jest skończonym zbiorem. Elementy zbioru Q, są nazywamy *stanami* automatu A.
- Σ – jest skończonym zbiorem nazywanym *alfabetem* automatu A.
- $\delta: Q \times \Sigma \to Q$ jest funkcją, nazywaną *funkcją przejścia* automatu A.
- q_0 - jest elementem zbioru Q, nazwanym stanem *początkowym automatu A*.
- $\mathbf{F} \subseteq Q$ - jest nazywane zbiorem *stanów warunkowej akceptacji*. A akceptuje te przebiegi, w których przynajmniej jeden nieskończenie często pojawia się stan należący do **F**.

5.1.6.22. **Definicja**. W niedeterministycznym automacie Büchi, funkcja przejścia δ jest zastąpiona relacją przejścia Δ, która przypisuje każdemu przejściu podzbiór stanów, przy czym pojedynczy stan początkowy q_0, jest zastąpiony zbiorem I_{in} stanów początkowym.

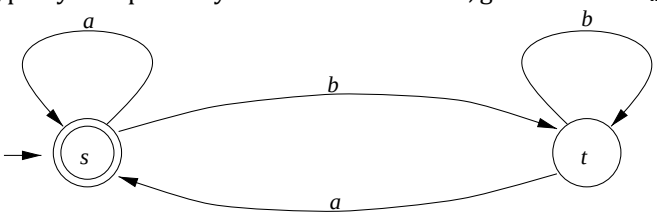
Zgodnie z definicją, automat Büchi akceptuje słowo wejściowe, jeśli istnieje przebieg, w którym pewien podzbiór zbioru Q pojawia się nieskończenie często. Ponieważ zbiór Q jest zbiorem skończonym, łatwo zauważyć, że musi istnieć stan $g \in Q$, który pojawia się nieskończenie często w przebiegu σ. Innymi słowy, jeśli potraktujemy przestrzeń stanów automatu Büchi, jako graf, to trasa akceptowanego przebiegu jest nieskończoną ścieżką – rozpoczynającą się stanem s, będącym stanem początkowym I_{in}, dochodząc do osiągalnego stanu $g \in I$, tworząc pętlę kolejnych stanów, prowadzących do stanu g nieskończenie często (parz rys. 5.1.3.23).

5.1.6.26. **Definicja**. Dopełnieniem języka L (rozpoznający język L - automat, jest pokazany na rys. 5.1.6.24), które zapisujemy $\overline{L}$, jest zbiór wszystkich nieskończonych słów α, takich, które mają skończoną liczbę wystąpień a. Drugą literę, która może wystąpić w nieskończonym słowie

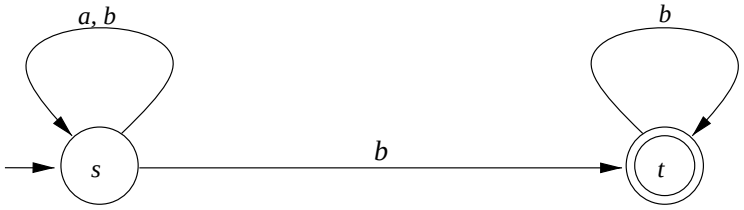
wejściowym, jest litera b. Automat rozpoznający dopełnienie języka L, jest pokazane na rys. 5.1.6.25. Automat przypuszcza, że istnieje punkt w słowie wejściowym, poza którym nie pojawi się więcej wystąpień litery a – wiadomo, że taki punkt musi istnieć w każdym nieskończonym słowie wejściowym, posiadającym tylko skończoną liczbę wystąpień litery a. Po pojawieniu się ostatniego wystąpienia litery a, słowo wejściowe może zawierać już tylko kolejne wystąpienia litery b.



5.1.6.23. Rys. Typowy akceptowany stan automatu Büchi, gdzie stan s ∈ I_{in}, zaś stan g ∈ Q.



5.1.6.24. Rys. Automat Büchi dla języka L.



5.1.6.25. Rys. Automat Büchi dla języka $\overline{L}$.

Innowacyjny pomysł Edmund Clarke, Allen Emerson oraz Joseph Sifakis, polegał na zaproponowaniu interakcji dwóch automatów skończonych, jako narzędzia sprawdzającego poprawność programu współbieżnego (reaktywnego). Jednym z tych automatów jest automat Büchi – reprezentujący przebieg badanego programu (kod badanego programu staje się segmentem nieskończonego słowa wejściowego, jeśli badany program zawiera tzw „martwą pętlę", powtarzającą się w nieskończoność), zaś drugim automat skończony realizujący kryterium poprawności. Przy czym, kryterium własności poprawności programu, jest sformułowane w liniowej logice temporalnej LTL.

5.1.6.31. **Wyjaśnienie**. Prze formalizację rozumiemy: Niech E będzie pełnym zbiorem wszystkich możliwych ω-przebiegów systemu. Dana poprawność własności f formalnie własność LTL. Możemy powiedzieć, że system spełnia f wtedy i tylko wtedy, gdy może wykonać wszystkie przebiegi w E. Możemy to wyrazić, jako:

5.1.6.32 $$(\mathrm{E} \models f) \leftrightarrow \forall p, (\rho \in \mathrm{E} \rightarrow \rho \models f).$$

SPIN, jak wiadomo, nie próbuje dowodu wprost. Możemy jedynie użyć SPIN dla usiłowania obalenia spełniania żądania poszukując kontrprzykładu pokazującego, że $\neg f$ jest spełnione przez

przynajmniej jeden przebieg. Co oznacza, że zamiast bezpośrednio dowodzić wyrażenie 5.1.6.32, SPIN próbuje dowieść coś przeciwnego:

5.1.6.33 $\neg (E \models f) \leftrightarrow \exists \rho, (\rho \in E \wedge \neg(\rho \models f))$

gdzie oczywiście $\neg(\rho \models f)$ oznacza $(\rho \models \neg f)$

5.1.6.40. **Wyjaśnienie**. Automaty wprowadzone, jako podstawa metody *model checking*, przez *Clarke'a*, *Emersona* oraz *Sifakisa*, charakteryzują się skończoną liczbą stanów i możliwością nieskończonych przebiegów.

5.1.6.41. **Definicja**. Akceptujący przebieg automatu skończonego $A = (S, s_0, L, T, F)$; gdzie S – to skończony zbiór stanów automatu A, s_0 – jest wyróżnionym stanem początkowym $\in S$, L – skończonym zbiorem etykiet, T – zbiór przejść $T \subseteq (S \times L \times S)$, zaś F - zbiór stanów końcowych $F \subseteq S$; to kończącym się przebieg z przejściem $(s_{n-1}, T[l_{n-1}]. s_n)$, który ma własność $s_n \in F$.

5.1.6.42. **Definicja**. Przebieg ω-akceptujący automatu Büchi $A = (S, s_0, L, T, F)$; jest to każdy nieskończony przebieg ρ, który ma własność $\exists s_f, s_f \in F \wedge s_f \in \sigma^{\omega}$.

Piśmiennictwo: *Ben-Ari M.* B.2.1., *Büchi, J.* B.3.1., *Holzmann G.* H.3.1, H.3.2.

5.2. MODEL WALIDACJI STANÓW PROGRAMU

5.2.0. UWAGI WSTĘPNE

Jak pisze *Mordechai Ben-Ari* - czerwcowy numer czasopisma *ACM Inroads,* z roku 2009, zawierał specjalny dział zatytułowany *Formal Methods in Education and Training,* w którym kilkanaście publikacji było poświęconych nowemu podejściu do walidacji (dowodzenia poprawności) oprogramowania (w szczególności współbieżnego i rozproszonego w sieci), podejściu nazwanemu *model checking*. Metoda *model checking* waliduje oprogramowanie – używając specjalizowanych narzędzi programowych pozwalających na sprawdzanie stanu w locie (*on-the-fly*), a następnie badanie przestrzeni stanów; w przeciwieństwie do wcześniejszych metod dedukcyjnych - zainicjowanych przez pionierskie prace *C.A.R. Hoare'a* (programy sekwencyjne) i kontynuowane między innymi przez *Amira Pnueli* i *Zohara Manna* (programy współbieżne - reaktywne i sieciowe). *Edmund Clarke*, *Allen Emerson* oraz *Joseph Sifakis* otrzymali w roku 2008 nagrodę *ACM Turing Award* za ich wkład innowacyjny w opracowanie metody *model checking*. Metoda *model checking* - stało się kamieniem węgielnym zarówno dla potrzeb projektowania obwodów scalonych wielkiej skali integracji, jak i dla projektowania współbieżnego i sieciowego oprogramowania.

Wcześniej wydawało się, że sprawdzenie stanów przyjmowanych przez program komputerowy działający w środowisku współbieżnym, jest ze względu na astronomiczną skalę koniecznych do przeprowadzenia obliczeń, praktycznie nie wykonalne. W roku 1981, *Clarke*, *Emerson* and *Sifakis* – wykazali, że fizycznie możliwym jest sprawdzenie wszystkich stanów programu komputerowego działającego w środowisku programów współbieżnych. Jak już zostało powiedziane, kluczem do rozwiązania, był pomysł zastosowania równoczesnego działania dwu *nondeterministic finite automata (NDFA)*, czyli niedeterministycznych automatów skończonych, z których jeden odpowiada za walidację programu dla danego osiągniętego stanu, zaś drugi za wyznaczanie w biegu tegoż stanu. Jeśli dla jakiegoś stanu stwierdzono by falsyfikację warunku walidacji badanego programu, to należy uznać, że program nie jest poprawny, zaś

przeprowadzony rachunek pokazuje przykład błędnego działania badanego (walidowanego) programu.

Piśmiennictwo: *Ben-Ari M.* B.2.2., *Holzmann G.* H.3.1., H.3.2.

5.2.1. TRYWIALNY PRZYKŁAD PARY PROGRAMÓW WSPÓŁBIEŻNYCH

Ben-Ari pokazał następnie trywialny przykład współbieżnego programu (patrz 5.2.1.00), zmieniającego zawartość komórki pamięci, w czasie wykonywania działań arytmetycznych na zawartości rejestru. W tym celu wyobraźmy sobie program aplikacyjny zawierający dwa procesy współbieżne oraz rutynowy program obsługi przerwania. W momencie pojawienia się przerwania, zawartość rejestrów (programu aplikacyjnego) zostaje zapamiętana na stosie, a następnie odtworzona po zakończeniu działania rutynowego programu obsługi przerwania. Efekt polega na tym, że program aplikacyjny i rutynowy program obsługi przerwań – operują oddzielnymi zestawami rejestrów. Przykład napisany jest w języku symbolicznym - zbliżonym do języka C.

```
integer n = 0;
process P
integer regP = 0;
p1: load n into regP
p2: increment regP
p3: store regP into n
p4: end

process Q
integer regQ = 0;
q1: load n into regQ
q2: increment regQ
q3: store regQ into n
q4: end
```

5.2.1.00. Kod - trywialnego przykładu programu zawierającego procesy współbieżne

Na stan powyższego trywialnego przykładu programu aplikacyjnego, składa się pięć komponentów:

1. Zawartość rejestru licznik rozkazów (*instruction pointer*) dla każdego z procesów: P i Q;
2. Trzy zmienne: jeden rejestr globalny `n`, oraz dwa rejestry lokalne `regP` i `regQ`.

Jednym z możliwych stanów jest stan: (`IP(P)=p3, IP(Q)=q1, regP=1, regQ=0, n=0`); co w skrócie możemy zapisać: `(p3,q1,1,0,0)`. Stan początkowy zaś, to: `(p1,q1,0,0,0)`, z którego istnieją dwa przejścia, jedno po wykonaniu instrukcji `p1`, zaś drugie po wykonaniu instrukcji `q1`. Liczba możliwych stanów jest skończona: mamy do czynienia z czterema możliwymi stanami dla każdego z dwóch liczników rozkazów oraz z trzema możliwymi stanami dla każdej z trzech zmiennych, co daje razem 4 x 4 x 3 x 3 x 3 = 432 różnych stanów. Co więcej, stwierdzamy, że istnieją co najwyżej dwa możliwe przejścia z każdego ze stanów. Stosunkowo łatwym jest utworzenie *NDFA* – odpowiednika sekwencji wyznaczania stanów. Jedną z możliwych sekwencji jest np.:

`(p1,q1,0,0,0) → (p2,q1,0,0,0) →(p3,q1,1,0,0) → (p4,q1,1,0,1) →(p4,q2,1,1,1) → (p4,q3,1,2,1) → (p4,q4,1,2,2).`

Ostatni ze stanów powyższej sekwencji, jest stanem końcowym *NFDA,* ponieważ nie ma dalszego przejścia po osiągnięciu wartości `p4` lub `q4`.

Kolejnym krokiem omawianym przez *Ben-Ari'ego* jest sformułowanie żądania poprawności dla omawianego trywialnie prostego programu: „Poprawne zakończenie programu wymaga aby $n = 2^n$, lub bardziej formalnie: `terminates->(n=2)`, czyli negacja `terminates &&!(n=2)`, co można wyrazić `terminates &&(n!=2)`.

Negacja żądania poprawności (*correctness claim negation*), można przedstawić w postaci trywialnego automatu *NDFA* posiadającego jeden stan odpowiadający tej formule oraz pętle powrotu do tegoż stanu. Sprawdzając odpowiedni fragment przestrzeni stanów, łatwo zauważyć, że istnieje sekwencja stanów, która zarówno spełnia *NDFA* walidowanego programu (prowadząca do zakończenia wykonywania programu), jak również przez *NDFA* odpowiadający negacji żądania poprawności:

```
(p1,q1,0,0,0) → (p2,q1,0,0,0) → (p2,q2,1,0,0) → (p3,q2,1,0,0) →
(p3,q3,1,1,0) → (p4,q3,1,1,1) → (p4,q4,1,1,1).
```

W ten sposób, jak pisze *Ben-Ari*, można udowodnić falsyfikację negacji żądania poprawności, ale również skonstruować kontrprzykład - pozwalający na znalezienie błędu w programie (*bug*) oraz negacji żądania poprawności tegoż programu.

Można więc stwierdzić, że jeśli żądanie poprawności jest prawdziwe (*true*), badanie przestrzeni stanów nie napotka żadnej sekwencji stanów, która zapewnia zarówno prawidłowe (w sensie formalnym) wykonanie programu, jak również spełnienia negacji żądania poprawności dla danego programu. Zauważmy, że podwójna negacja się znosi: program jest poprawny jeśli nie istnieje sekwencja stanów programu, która zawiera negację poprawności.

Piśmiennictwo: *Ben-Ari M.* B.2.2., *Holzmann G.* H.3.1., H.3.2.

5.2.2. Algorytm sprawdzania poprawności programu

Teoretyczne rozważania *Clarke'a, Emersona* i *Sifakisa* były dalej rozwijane w kierunku doskonalenia algorytmu (5.1.2.30) wraz z techniką jego implementacji, tak, aby poradzić sobie z astronomiczną wręcz liczbą stanów, które spowodują istotne utrudnienia obliczeniowe. Wykonywanie rzeczywistych obliczeń na współczesnych komputerach, ze względu na skończoną liczbę bitów w ich pamięciach, nie odbywa się na liczbach dowolnie dużych. Zmienna komputerowa typu całkowitego (*integer*), przyjmuje np. maksymalną wartość 2^{32}, w przeciwieństwie do matematycznych liczb całkowitych, które mogą przyjmować nieskończoną liczbę wartości.

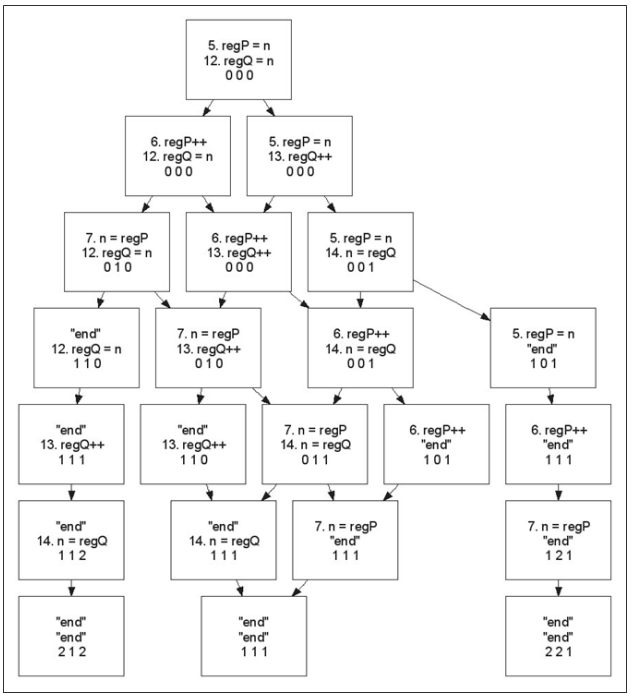
Przykładowo, jeśli dany program korzysta z jednomega-bajtowej pamięci roboczej, gdzie każdy bajt zawiera 2^8 możliwych różnych wartości, to taka pamięć umożliwia zapisanie $2^8 \times 2^{20}$ różnych stanów przetwarzania; natomiast, jeśli przemnożymy liczbę możliwych przejść w NDFA (*nondeterministic finite automaton*), dla każdego z modelowanych procesów i przez liczbę możliwych przejść w NDFA, to otrzymamy liczbę stanów, jakie określa dany program, który być może, że wymagają sprawdzenia – dla stwierdzenia poprawności walidowanego programu; która to liczba różnych stanów może przekroczyć wartość 2^{28}.

Wracając do rozważań dotyczących przestrzeni stanów omawianego przykładu programu, należy stwierdzić, że nie wszystkie stany przestrzeni stanów programu są osiągalne. Inaczej mówiąc, istnieją stany, do których nie prowadzi żadna sekwencja przejścia pomiędzy stanami. Ta własność, okazuje się być bardzo istotną. W omawianym przykładzie programu (w

podrozdziale 5.1.1), jak to zostało pokazane wcześniej, na przestrzeń stanów składa się z 432 stanów, z których znaczna większość jest nieosiągalna.

Trzy zmienne, muszą mieć wartość zero, dla przypadku, gdy proces `P` jest w stanie `p1` oraz proces `Q` jest w stanie `q1`. Innymi słowy, jedynym osiągalnym stanem dla `(p1,q1,…,…,…)` jest `(p1,q1,0,0,0)`; pozostałych 26 stanów postaci `(p1,q1,…,…,…)` jest nieosiągalnych i możemy ich nie uwzględniać badając przestrzeń stanów. Na rys. 5.2.2.00. pokazany jest fragment przestrzeni stanów, z omawianego przykładu, obejmujący jedynie stany osiągalne.

Możemy, więc ograniczyć liczbę badanych stanów do tych, które są osiągalne przy tworzeniu i badaniu stanów w tzw. locie (*on-the-fly*). Tak, więc, osiągalne stany to te, które mogą być tworzone w locie, startując ze stanu początkowego i budując sekwencje kolejno osiągalnych stanów. Synchronicznie do tworzenia nowo - osiągniętego stanu, weryfikowana jest negacja żądania poprawności (*correctness claim negation*) dla tegoż stanu. Czyli jak tylko pojawia się nowy stan, żądane kryterium poprawności jest sprawdzane; jeśli żądanie poprawności daje wartość - fałsz (*false*), czeker sygnalizuje błąd, natomiast, jeżeli żądanie poprawności daje wartość prawda (*true*) – czeker kontynuuje tworzenie nowego osiągalnego stanu.



Rys. 5.2.2.00. Przestrzeń osiągalnych stanów

Dla uniknięcia powtórnego sprawdzania stanów, które były już badane w poprzednich sekwencjach przejść, wykorzystywany jest stos badanych stanów i zrealizowanych przejść pomiędzy stanami w kolejnych sekwencjach przebiegów. Użycie stosu powoduje, że prezentowanie kontrprzykładów jest trywialne; sprowadza się do pokazania sekwencji przejścia, zarejestrowanej na stosie, prowadzącej do ujawnionego błędu.

Ponownie zajmiemy się naszym przykładem z podrozdziału 5.2.1. Załóżmy, że chcemy dokonać walidacji naszego przykładowego programu z żądaniem poprawności:

```
terminated -> (n <= 2)
```

oraz załóżmy, że nasza pierwsza sekwencja osiągniętych stanów, ma postać:

```
(p1,q1,0,0,0) → (p2,q1,0,0,0) → (p3,q1,1,0,0).
```

Ponieważ, żaden błąd nie został stwierdzony w badanej sekwencji stanów, generujemy kolejny stan `(p2,q2,0,0,0)` i kontynuujemy walidację; ponownie nie napotykając na błąd. Ostatecznie stwierdzamy, że rozpoczynając od procesu `Q`, trafimy na kolejną generowaną sekwencję postaci:

`(p1,q1,0,0,0) → (p1,q2,0,0,0) → (p2,q2,0,0,0).`

Ale ostatni z osiągniętych stanów, był już wcześniej walidowany i nie stwierdzono wystąpienia błędu. Koniecznym jest, więc pamiętanie przez czeker stanów wcześniej walidowanych, aby nie duplikować wysiłku związanego z ponowną walidacją tych stanów, które mogą wystąpić w więcej niż jednej z generowanych sekwencji stanów. Czeker musi, więc przechowywać strukturę walidowanej już części przestrzeni stanów programu. W tym celu przeważnie używa się tzw. *hash tables*, umożliwiających sprawdzenie, czy dany stan jest już zapamiętany w strukturze przestrzeni danych, czy też nie.

Piśmiennictwo: *Ben-Ari M*. B.2.2., *Holzmann G*. H.3.1., H.3.2.

5.2.3. MODELOWANIE PROGRAMU I SYSTEMU PROGRAMÓW

W praktyce zamiast walidacji programów pisanych w różnych językach programowania, tworzymy, jak to już zostało powiedziane w podrozdziale 5.1.3, abstrakcji - modelu działania sterowania danego programu czy też systemu programowania, a dopiero ten model (napisany w specjalnym języku opisu działania procesów współbieżnych), jest przedmiotem badania z pomocą czekera. Walidacja modelu, jest podstawową metodą sprawdzania poprawności programów współbieżnych (reaktywnych) oraz działających w środowisku sieciowym (rozproszonym). Strona obliczeniowa, jak również badanie zawartości (*content*) przekazywanych komunikatów, ma na ogół małe znaczenie z punktu widzenia działania sterowania modelu programu. Na ogół jesteśmy zainteresowani faktem, czy wysłane komunikaty docierają do odbiorcy, natomiast ich zawartość, jest z punktu widzenia badania poprawności działania sterowania programu, sprawą drugorzędną. Podobnie, chcemy wiedzieć, czy działają poprawnie mechanizmy synchronizacji, czy procesy korzystając z zasobów (np. rozproszonych) mogą potencjalnie podlegać zakleszczeniom, czy wreszcie poprawnie działa protokół komunikacyjny. Dlatego też, pierwszym krokiem badania poprawności funkcjonowania programu współbieżnego pracującego w środowisku rozproszonym (np. sieciowym), jest stworzenie abstrakcji, czyli napisanie modelu tegoż programu. Kolejnym krokiem, jest dokonanie walidacji opracowanego modelu z pomocą tzw. *czekera, czyli* systemu programów walidujących modele oraz sprawdzenie, czy uzyskany wynik dotyczy modelowanego programu.

Model programu lub systemu programów, jest opisem na wyższym poziomie abstrakcji, niż sam program (lub system programów) przeznaczony do implementacji. Abstrakcyjność modeli jest istotną, ponieważ umożliwia inżynierom oprogramowania – pisanie zrozumiałych specyfikacji, które równocześnie nie narzucają zbędnych ograniczeń na implementację. Modele pisane w języku sformalizowanych specyfikacji, mają dodatkową zaletę, że poprawność własności w nim zapisanych, może być weryfikowana. Jeśli zostanie stwierdzone, że implementacja modelu abstrakcyjnego prowadzi do błędnych sytuacji, błędy przy implementacji takiego rozwiązania, są na ogół, łatwiejsze do lokalizacji i skorygowania, niż błędy w tradycyjnej niesformalizowanej specyfikacji.

Przypuśćmy, że chcemy zbudować model kontrolera temperatury elektrowni. Zbiór dopuszczalnych wartości temperatury może zmieniać się w przedziale kilkuset stopni Celsjusza, ale - jak wiadomo z wielu złych doświadczeń, błędy dotyczą ograniczonych wartość zmiennych. Możemy np. w modelu zastąpić typowy zakres dopuszczalnego przedziału temperatur, kilkoma

dyskretnymi wartościami temperatury: (1) najniższą dopuszczalną; (2) najwyższą dopuszczalną; (3) oraz dwoma wartościami z poza przedziału dopuszczalnego. Wyzwaniem dla inżynierów oprogramowania jest zbudowanie modelu, który wystarczająco dokładny, żeby wiernie opisywać rzeczywistość, ale jest równocześnie opracowany na takim poziomie abstrakcji, że jest łatwy do zrozumienia i zachowuje się dostatecznie wiernie w zakresie naśladowania rzeczywistości.

Piśmiennictwo: *Ben-Ari M*. B.2.2.

5.2.4. PROCESOR SPIN A INNE CZEKERY

Odnośnie procesora SPIN, często używamy rozszerzonej nazwy *„model czeker"*, zamiast powiedzieć krótko „czeker". Jednym z najbardziej znanych czekerów (obsługujący język modelowania procesów *PROMELA*), jest procesor SPIN (*Simple Promela Interpreter*). Procesor SPIN[88] jest tzw. czekerem modeli sterowania programami współbieżnymi – działającymi w środowisku rozproszonym, opracowanym i rozwijanym przez szereg lat przez *Gerarda J. Holzmanna,* obecnie pracującego w NASA/JPL. Procesor SPIN waliduje modele programów - napisane w prostym języku modelowania PROMELA. W roku 2001, *Holzmann* za pracę nad procesorem SPIN otrzymał ACM Software Systems Award. Początkowo celem prac prowadzonych przez *Holzmanna* było narzędzie do weryfikacji protokołów komunikacyjnych, które to oprogramowanie zostało następnie rozbudowane w procesor do walidacji modeli sterowania programami współbieżnymi w środowisku sieciowym (rozproszonym), programami zbudowanymi z niepodzielnych (*atomic*) sekwencji-bloków instrukcji poszczególnych procesów programu.

Zanim przystąpimy do przedstawienia bardziej szczegółowego opisu funkcjonalnego języka modelowania PROMELA oraz czekera SPIN, zasygnalizujemy fakt - istnienia szeregu innych czekerów, równie ważnych, ale o nieco innym przeznaczeniu:

- SMV/NuSMV[89], jest czekerem korzystającym z innej logiki temporalnej, niż wykorzystywanej w SPIN, dla specyfikowania własności poprawności oraz innym sposobem przechowywania i przeszukiwania przestrzeni stanów. To podejście znajduje szerokie zastosowanie w walidacji systemów synchronicznych, np. realizowanych sprzętowo (*hardware system*), gdzie wiele komponentów zmienia swój stan równocześnie, w wyniku otrzymywania sygnałów zegarowych.
- JAVA PATHFINDER[90], jest czekerem przeznaczonym do walidacji programów pisanych w języku programowania, w przeciwieństwie do SPIN, który posługuje się prostym językiem modelowania. Weryfikacja programu napisanego w języku programowania, jest niewątpliwie podejściem realistycznym, w Java PathFinder – niestety, odbywa się to kosztem istotnych ograniczeń wielkości kodu, ponieważ użycie języka Java prowadzi do znacznie bogatszej przestrzeni stanów, niż w przypadku języka modelowania PROMELA.
- UPPAAL[91] model checker, jest czekerem korzystającym z innej logiki temporalnej, niż użytej w SPIN (jest to wersja logiki temporalnej czasu rozgałęzionego, czyli CTL), a służącym do walidacji programów czasu rzeczywistego (*real-time*). Tak, więc, jest to rozwiązanie abstrahujące od pojęcia czasu absolutnego i opiera się „wklejaniu" sekwencji niepodzielnych (*atomic)* instrukcji współbieżnych procesów.

[88] Patrz http://spinroot.com
[89] Patrz http://nusmv.fbk.eu
[90] Patrz http://javapathfinder.sourceforge.net
[91] Patrz http:// www.uppaal.org

Oczywiście, przytoczona lista czekerów jest jedynie niewielkim wycinkiem listy opracowanych programów – narzędzi walidacji. Obszerniejszą listę można znaleźć np. w publikacjach Kazimierza Trzęsickiego.

Piśmiennictwo: Ben-Ari M. B.2.2., *Holzmann G.* H.3.1, *Trzęsicki K.* T.3.2.

5.2.5. SYMULACJA NIEDETERMINIZMU W SPIN

Mogłoby się wydawać, że czekery modeli – takie jak SPIN, wydają się niewłaściwe do prezentacji zjawiska niedeterminizmu, ponieważ semantyka współbieżności jest uniwersalna (wszystkie czynności przetwarzania muszą być poprawne), kiedy semantyka NDEA (nie-deterministycznych automatu skończonego) jest egzystencjonalna (łańcuch symboli jest akceptowalny, jeśli istnieje, zaś przetwarzanie kończy się na stanie akceptowalnym po przeczytaniu danego łańcucha). Jednakże, prosty zabieg techniczny umożliwia procesorowi Spin implementować NDEA i również inne niedeterministyczne algorytmy.

Proces NDFA, pozwala łatwo oprogramować niedeterministyczne przejście (*transition*), przy użyciu instrukcji typu – `if` ze strażnikiem. Przykładowo, rozważmy takie NDFA, którego zbiór możliwych przejść ze stanu `q3`, jest:

`{(q5,a,q7),(q5,a,q3),(q5,b,q5)}.`

Taką funkcję przejścia można z łatwością modelować z pomocą instrukcji ze strażnikiem (definicja strażnika, symbol ::, podana jest dalej, w podrozdziale 5.3.6):

```
q5:
if
:: input == 'a' ->
      input = next-symbol; goto q7
:: input == 'a' ->
      input = next-symbol; goto q3
:: input == 'b' ->
      input = next-symbol; goto q5
fi
```

Następnie dodajemy przekształcenie:

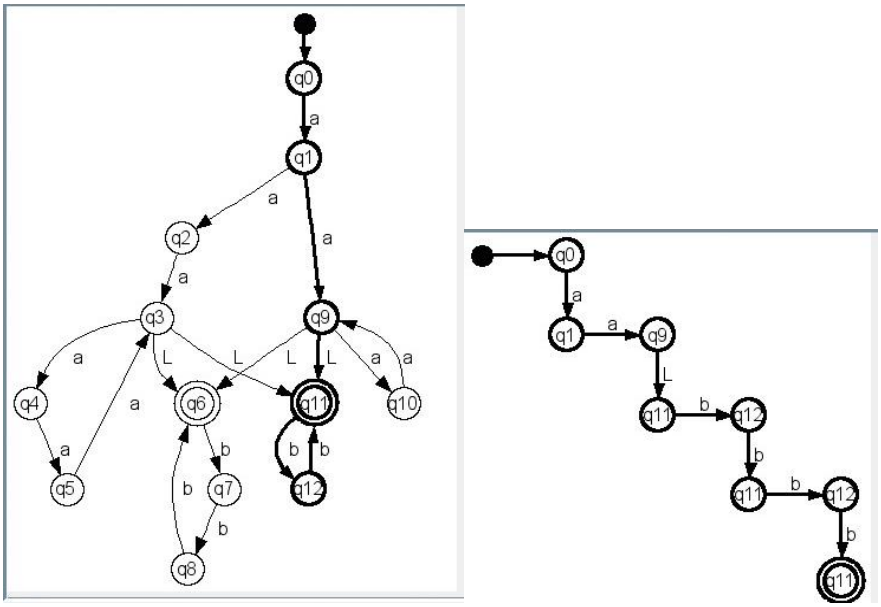
```
:: end-of-input -> assert(false)
```

będące instrukcją ze strażnikiem dla każdego stanu akceptacji. Akceptacja przetwarzania naszego NDFA, odpowiada, więc przetwarzaniu wykonującemu `assert(false)`. SPIN w takiej sytuacji, może jedynie zaraportować wystąpienie błędu, co oznacza, że nasze przetwarzanie NDFA jest „poprawne" (czyli, że zostało zaakceptowane przez NDFA), ponieważ błąd został stwierdzony.

Każdy z trybów działania Spin koresponduje z poszczególnymi drogami działania NDFA:

- Losowa symulacja oznacza wyboru decyzji przejścia niedeterministycznego (jednego z trzech możliwych).
- Interaktywna symulacja jest wykonaniem NDFA według wyroczni (ciebie) upewniając się, że akceptujące przetwarzanie zostało wybrane.
- Weryfikacja odpowiada meta-poziomowi, czyli istnienia akceptujące przetwarzanie, czy też nie istnienia.

Graficzne zobrazowanie powyższego algorytmu dla przykładowej nie-deterministycznego wektor-funkcji przejścia (patrz: rys. 5.2.5.00 – strona lewa) i wybranej w wyniku przetwarzania ścieżki przejścia (patrz: rys. 5.2.5.00 – strona prawa).



Rys. 5.2.5.00. Wizualizacja przejść NDFA (rys. lewy) i prezentacja wybranej ścieżki przejść NDFA (rys. prawy).

Piśmiennictwo: *Ben-Ari M.* B.2.2.

5.2.6. PODSUMOWANIE I WNIOSKI

Jak widać z dotychczasowych rozważań, metody formalne (*formal methods*), przestały wreszcie być ciekawostką - typową dla rozważań akademickich, a stały się praktycznie przydatnymi narzędziami. Celem użycia metod formalnych jest zagwarantowanie użytkownikom programów komputerowych i systemów programów działających w środowisku rozproszonym (np. sieciowym) poprawność, czyli bezbłędność, a to w przeciwieństwie do częstych dotychczas praktyk – braku odpowiedzialności za błędy w pakietach aplikacyjnych, zgodnie z niesławnym stwierdzeniem „*end-use license agreements*". Przykładowo, oferowane użytkownikom - jądro (*kernel*) system operacyjny Linux, jest walidowane.

Walidacja programów, w oparciu o metody formalne - wykorzystuje takie pojęcia jak: środowisko rozproszone, przestrzeń stanów, współbieżność, poprawność lokalna i całkowita, non-determinizm, itp. Dalszy rozwój metod walidacji - wymaga kontynuacji badań w dziedzinie matematyki dyskretnej, teoretycznych podstaw informatyki i użycie uzyskiwanych wyników np. w: teorii automatów skończonych; działach logiki - takich jak temporalny rachunek zdań i formułowanie warunków i żądań poprawności; teorii grafów - odnośnie badania przestrzeni stanów; funkcji haszujących oraz kompresji danych. SPIN, jako profesjonalne narzędzie oprogramowania, znalazł już szerokie zastosowanie w przemyśle oraz badaniach kosmicznych, między innymi ze względu na prostą składnię języka PROMELA i efektywność ciągle doskonalonych programów – komponentów czekera. Szerszy zakres użycia czekerów, wymaga między innymi - opracowania narzędzi umożliwiających generowanie modeli sterowania w prostym języku modelowania (jak np. PROMELA), na podstawie kodu napisanego w popularnych dziś językach programowania (np. Java, C#, itp.).

Piśmiennictwo: *Ben-Ari M.* B.2.2., *Trzęsicki K.* T.4.1.

5.3. ZARYS JĘZYKA PROMELA

5.3.0. UWAGI WSTĘPNE

Procesor SPIN jest narzędziem do analizowania: logicznej spójności modeli procesów współbieżnych działających w rozproszonym środowisku (a w szczególności protokołów komunikacyjnych), modelu systemu zbudowanego z wielu procesów współbieżnych, opisanych w języku PROMELA (*Process Meta Language*). Wzmiankowany język pozwala na utworzenie opisów współbieżnych procesów asynchronicznych. Związki pomiędzy poszczególnymi procesami odbywają się za pośrednictwem kanałów komunikacji przekazywania wiadomości (komunikatów). Kanały komunikacji mogą być definiowane zarówno, jako synchroniczne (*rendezvous*), jak również, jako asynchroniczne (*buffered*).

Model systemu napisany w PROMELA, procesor SPIN może zarówno wykonać symulując działanie modelu, jak również wygenerować program w języku C, który z kolei wykona efektywną online weryfikację poprawności systemu. W toku wykonywania symulacji lub weryfikacji - procesor SPIN sprawdza występowanie pętli – zagłodzeni lub zakleszczeń, pojawianie się działań zaskakujących (niewyspecyfikowanych) dla zachowań systemu, jak również pojawienie się niewykonalnych kodów. SPIN może również być użyty, jako weryfikator poprawności niezmienników (*invariants*), wystąpienia martwych pętli (*nonprogress execution cycles*), jak również może weryfikować własności wyrażone, jako następstwa czasu w formułach logiki temporalnej.

Procesor Spin został zorganizowany tak, żeby korzystał minimalnej ilości pamięci. Wyczerpujące weryfikacja wykonana przez procesor SPIN, odpowiada z matematyczną dokładnością na pytanie: „czy dany model systemu jest, czy też nie jest wolny od błędów?" Bardzo duże problemy weryfikacji, które bezpośrednio nie mogłyby być wykonywane ze względu na ograniczenia współczesnych systemów komputerowych, mogą zostać podjęte przez procesor SPIN w sposób oszczędny, nazwany „*bit state storage*" techniką, zwaną również *supertrace*. Metoda ta pozwala na upakowanie danych w pamięci na minimalnej niezbędnej liczbie bitów na zapisanie każdego z osiąganych stanów weryfikowanego modelu.

Pierwsza część niniejszego rozdziału poświęcona jest wprowadzeniu do języka PROMELA, natomiast druga część dotyczy podstaw procesora SPIN. Szczegółowy opis języka Promela można znaleźć w [H.3.1.]. Niniejszy rozdział poświęcony jest jedynie opisowi podstawowych informacji dotyczących PROMELA. Rozdział 5.4 - poświęcony będzie podstawowym własnością procesora SPIN.

Piśmiennictwo: *Dijkstra E.* D.2.1., *Holzmann G*. H.3.1, H.3.2.

5.3.1. PROMELA – JĘZYK MODELOWANIA

Jak już zostało powiedziane, język PROMELA (*Process Meta Language*) jest przeznaczonym dla celów weryfikacji - językiem modelowania procesów współbieżnych. Innymi słowy, PROMELA jest wehikułem tworzenia abstrakcji rozproszonych systemów – zbudowanych z reaktywnych procesów współbieżnych, pozwalającym pominąć nieistotna szczegóły ze względu na wzajemne oddziaływanie procesów. Celem użycia procesora SPIN, jest weryfikacja poszczególnych procesów, których zachowanie z jakiegoś powodu może być podejrzane. Te procesy o podejrzanym zachowaniu są modelowane i weryfikowane. Pełna weryfikacja wykonywana jest, jako seria kroków, skonstruowanych z odpowiednią szczegółowością w modelu (abstrakcji programów) napisanym w PROMELA. Każdy taki model może być weryfikowany z różnymi typami założeń dotyczących środowiska (np. w przypadku protokołu komunikacyjnego:

zgubienie komunikatu, duplikacja komunikatu, itd.). Kiedy poprawność modelu zostanie stwierdzona przez procesor SPIN, fakt ten może być wykorzystany w konstrukcji i weryfikacji modeli pochodnych.

Program modelowany w PROMELA – składa się z *procesów*, *kanałów* komunikacji i *zmiennych*. Procesy są obiektami globalnymi. Kanały komunikacyjne i zmienne mogą być deklarowane zarówno, jako globalne ze względu na model, lub lokalne ze względu na proces modelu. Procesy określają zachowanie modelu, kanały i zmienne globalne określają środowisko, w którym działa proces.

Piśmiennictwo: *Dijkstra E.* D.2.1., *Holzmann G.* H.3.1, H.3.2.

5.3.2. WYKONYWALNOŚĆ PROCESU

W języku PROMELA nie ma rozróżnienia pomiędzy warunkami i instrukcjami, nawet pojedynczy warunek boolowski może działać jak instrukcja. Wykonanie dowolnej instrukcji jest warunkowane bieżącą wykonalnością, ponieważ każda instrukcja w danej chwili - w zależności od stanu środowiska, może być wykonywalna albo blokowana. Wykonywalność jest podstawowym środkiem synchronizacji. Każdy proces może oczekiwać na wystąpienie zdarzenia, które powoduje jego wykonywalność. Na przykład zamiast napisać pętlę oczekiwania, jak niżej.

```
while (a != b)
        skip    /* wait for a==b */
```

można uzyskać taki sam efekt w PROMELA, używając instrukcji

```
(a == b)
```

Warunek może być wykonywany (przejść), jeśli jest spełniony. Jeśli warunek nie jest spełniony, wykonanie instrukcji (warunku) jest blokowane.

Zmienne są używane do przechowywania zarówno globalnych dotyczących całego modelu systemu, jak również lokalnych wskazanego procesu, w zależności, w jaki miejscu deklaracja zmiennej została umieszczona. Poniższe deklaracje

```
bool flag;
int state;
byte msg;
```

definiują zmienne – służące do przechowywania wartości całkowito liczbowe z trzech różnych przedziałów wartości. Zakres wartości jest globalny, jeśli deklaracja zmienne znajduje się na zewnątrz deklaracji procesu, albo lokalny, jeśli deklaracja znajduje się wewnątrz deklaracji procesu.

Piśmiennictwo: *Dijkstra E.* D.2.1., *Holzmann G.* H.3.1, H.3.2.

5.3.3. TYPY DANYCH ORAZ TABLICE ZMIENNYCH

Poniższa tabela 5.3.3.00 zawiera podstawowe informacje dotyczące typów danych stosowanych w PROMELA, wielkości, przedziałów wartości dla komputera o 32 – bitowym słowie.

Nazwy `bit` oraz `bool` są synonimami jednobitowych informacji. Natomiast `byte` jest liczbą bez znaku, której wartość należy do przedziału od `0` do `255`. Symbole `short` oraz `int`, są liczbami ze znakiem o wartościach z odpowiednich przedziałów.

Tabela 5.3.3.00 - PODSTAWOWE TYPY DANYCH			
Typ	C-ekwiwalent	Zakres/liczba elementów	Macro
bit	bit-field	0, 1	-
bool	unchar	false, true	CHAR_BIT()
byte	unchar	0..255	
chan		1..255	
mtype		1..255	
pid	unchar	0..255	
short	short	-2^{15} .. $2^{15}-1$	SHRT_MIN .. SHRT_MAX
int	int	-2^{31} .. $2^{31}-1$	INT_MIN .. INT_MAX
unsigned		0 .. $2^{n}-1$	

Zmiennym typu `mtype` mogą mieć przypisane symboliczne wartości, deklarowane w ramach definiowania wartości:
`mtype = { ... }`, o której dalej. Zmienne mogą tworzyć tablice. Przykładowo:

```
byte state[N]
```

gdzie, deklarowane element tabeli o `N` – bajtowej, mogą być dostępne z pomocą poniższej instrukcji

```
state[0] = state[3] + 5 * state[3*2/N]
```

w której `N` jest stałą lub zmienną gdzieś zadeklarowaną w modelu. Indeksem tablicy zmiennych może być dowolne wyrażenie pozwalające wyrazić jednoznacznie wartość całkowitoliczbową. Użycie indeksu z poza przedziału `0 .. (N-1)` jest nieokreślona; prawdopodobnie zostanie wskazany jako błąd przebiegu procesora SPIN (Uwaga: wielowymiarowe tablice mogą być deklarowane pośrednio z pomocą konstrukcji `typedef`).

Typowe deklaracje zmiennych tych typów podstawowych obejmują:

```
bit x, y;               /* two single bits, initially 0 */
bool turn = true; /* Boolean value, initially true */
byte a[12];       /* all elements initialized to 0 */
chan m;                 /* uninitialized message chanel */
mtype n;                /* uninitialized mtype value */
short b[4] = 89;        /* all elements initialized to 89 */
int cnt = 67;           /* integer scalar, initially 67 */
unsigned v : 5;         /* unsigned stored in 5 bits */
unsigned w : 3 = 5;     /* value range 0..7, initially 5 */
```

Tylko jedno wymiarowe tablice zmiennych są obsługiwane, aczkolwiek jest pośrednia droga definiowania wielowymiarowych tablic, korzystając z definicji struktury, co pokrótce pokażemy. Wszystkie zmienne, wliczając w to tablice, są domyślnie inicjalizowane zerem, nie zależnie od tego, czy są globalnymi czy też lokalnymi.

Zmienne zawsze muszą mieć ściśle określone granice przedziałów wartości. Zmienne z ostatniego przykładu, mogą zawierać wartości, które dają się zapisać z pomocą trzech bitów: od zera do siedmiu. Zmienne typu short, mogą zawierać wartości, które można zapisać w szesnastu bitach pamięci. Ogólnie, jeśli wartość przypisana zmiennej – leży poza przedziałem wartości deklarowanej domeny, to przypisywana wartość jest automatycznie odcinana. Przykładowo:

```
byte a = 300;
```

wynikiem podstawienia będzie 44 (300%256). Jeśli takie przypisanie zostaje wykonane w toku kierowanej lub losowej symulacji, to SPIN drukuje komunikat o błędzie, żeby zaalarmować użytkownika o wystąpieniu obcięcia. Ostrzeżenia takie nie są generowane w toku przebiegu weryfikacji, dla unikania generacji dużego wolumenu powtarzających się wydruków.

Normalnie, wielokrotne zmienne danego typu są grupowane za pojedynczą nazwą typu zmiennych:

```
byte a, b[3] = 1, c = 4;
```

W tym przypadku, zmienna `a` – domyślnie inicjowana jest zerem; wszystkie elementy tablicy `b` są inicjowane wartością jeden, zaś zmienna `c` jest inicjalizowana wartością cztery.

Zmienne typu `mtype` służą do składowania wartości symbolicznych, które należy określić w jednej lub więcej deklaracji zmiennych typu `mtype`. Każda deklaracja `mtype` jest typowo umieszczana na początku specyfikacji, co jedynie powoduje ponumerowanie nazw, na przykład jak następuje:

```
mtype = { apple, pear, orange, banana };
mtype = { fruit, vegetables, cardboard };
init {
            mtype = pear n = pear; /* initialize n to pear */
            printf("the value of n is ");
            printm(n);
            printf("\n");
}
```

5.3.1.10. Kod deklaracji `mtype`

Oczywiście, żadna z nazw wymienionych w deklaracji `mtype` nie może odpowiadać słowom kluczowym PROMELA, np. takich jak `init`, czy też `short`.

Jak dotąd, omówiliśmy dwa typy deklaracji instrukcji PROMELA: warunek boolowski oraz podstawienie. Deklaracje oraz podstawienia są zawsze wykonywalne. Natomiast warunki, są wykonywalne tylko wtedy, kiedy są one spełnione (mają wartość `true`).

Piśmiennictwo: *Holzmann G.* H.3.1, H.3.2.

5.3.4. TYPY PROCESÓW I ICH INSTANCJONOWANIE

Stany zmiennych oraz kanałów komunikacji mogą być jedynie zmieniane lub sprawdzane przez procesy. Zachowanie procesu jest definiowane za pomocą deklaracji `proctype`. Następująca przykładowo deklaracja tworzy proces z jedną lokalną zmienną `state`.

```
active proctype A()
{       byte state;

        state = 3
}
```

5.3.4. 00. Kod deklaracji procesu.

Nazwa typu deklarowanego procesu to `A`. Ciało deklarowanego typu procesu jest zawarte w nawiasach klamrowych. Ciało deklaracji może zawierać listę obejmującą zero lub więcej deklaracji zmiennych lokalnych i/lub instrukcji. Powyższa deklaracja zawiera tylko jedną deklarację lokalnej zmiennej oraz pojedynczą instrukcję: podstawienie wartości `3` na zmienną `state`.

Średnik – jest separatorem instrukcji (*separator* a nie terminator instrukcji, skąd niema średnika po ostatniej instrukcji deklaracji). PROMELA dopuszcza dwa separatory instrukcji: strzałkę „->” oraz średnik „;”. Te dwa symbole separacji są równoważne. Czasami strzałka jest również wykorzystywana, jako nieformalny wskaźnik relacji przyczynowo - skutkowej dwóch instrukcji. Rozważmy następujący przykład (patrz kod 5.3.4.10).

```
byte state = 2;
```

```
proctype A()
{       (state == 1) -> state = 3
}

proctype B()
{       state = state - 1
}
```

5.3.4.10. Kod przykładu deklaracji pary procesów

W powyższym przykładzie zadeklarowano dwa typy procesów, `A` oraz `B`. Zmienna `state` jest teraz zmienną globalną, z wartością początkową równą dwa. Proces typu `A` zawiera dwie instrukcje, odseparowane strzałką. W przykładzie, deklaracja procesu typu `B` zawiera pojedynczą instrukcję, która powoduje zmniejszenie wartości zmiennej o jeden. Ponieważ instrukcja podstawienia jest zawsze wykonywalna, proces typu `B` jest zawsze tworzony bez żadnego opóźnienia. Natomiast proces typu `A`, jak już wiemy, jest tworzony dopiero wtedy, kiedy zmienna `state` przyjmie wartość jeden.

Instrukcja `proctype` deklaruje jedynie zachowanie procesu, ale nie powoduje jego wykonywania. Jedyny proces modeli napisanych w PROMELA, który zawsze będzie wykonywalny – jest proces typu `init`, który musi zostać zadeklarowany bezpośrednio w każdym definiowanym modelu. Dlatego też, najmniejszy model napisany w PROMELA, jest następujący:

```
init { skip }
```

gdzie instrukcja `skip`, jest instrukcją pustą – nic nierób. Bardziej interesujący jest inny model od powyższego, w którym proces inicjujący `init` – może zainicjować zmienne globalne oraz instancjonować dwa typy procesów. Poniższa instrukcja `init`, może mieć postać, jak dalej:

```
init
{       run A(); run B()
}
```

gdzie słowo `run` zostało użyte jako jednostkowy operator odziaływujący na proces danego typu. Wykonywalność operatora `run` ma miejsce tylko wówczas, gdy proces danego typu jest zainicjowany. Natomiast jest niewykonywalny, np. gdy zbyt wiele procesów już zostało uruchomionych (działa).

Instrukcja `run` przenosi wartości bazowe poszczególnych typów danych wszystkich parametrów do nowo tworzonych procesów. Odpowiednia deklaracja, napisana jak niżej dokonuje tegoż przepisania (kod 5.3.4.10):

```
proctype A(byte state; short foo)
{
        (state == 1) -> state = foo
}

init
{
        run A(1, 3)
}
```

5.3.4.10. Kod przykładu z użyciem instrukcji `run`

Tablice danych (`data arrays`) oraz typy procesów nie mogą być przekazane jak parametry. Jak pokazano poniżej, jest tylko jeden inny typ danych, który może być użyty, jako parametr. Tym typem danych jest kanał komunikacji.

Instrukcja `run` może być użyta z każdym rozmnażanym procesem, nie tylko z procesem `init`. Użycie instrukcji `run` tworzy nowe procesy. Wykonywany proces znika po swoim zakończeniu

(*terminates*), czyli po osiągnięciu końca zadeklarowanego typu procesu, ale nie wcześniej niż wszystkie te procesy wcześniej uruchomione - musiały się zakończyć. Za pomocą instrukcji `run` możemy tworzyć dowolną liczbę kopii procesów typu `A` oraz `B`. Jeśli jednak, więcej niż jeden współbieżny proces jest upoważniony zarówno do czytania lub pisania wartości zmiennej globalnej, to jak wiadomo może to doprowadzić do pojawienia się problemów (szczegóły patrz Holzmann H.3.1). Rozpatrzmy przykładowo, następujący przypadek modelu (kod 5.3.4.20) złożonego z dwóch procesów, dzielących dostęp do globalnej zmiennej `state`.

```
byte state = 1;

proctype A()
{       byte tmp;
        (state==1) -> tmp = state; tmp = tmp+1; state = tmp
}

proctype B()
{       byte tmp;
        (state==1) -> tmp = state; tmp = tmp-1; state = tmp
}

init
{       run A(); run B()
}
```

5.3.4.20. Kod przykładu dwóch procesów, dzielących dostęp do globalnej zmiennej `state`

Jeśli, jeden z tych dwu procesów zakończy działanie (*terminate*), zanim jego konkurent wystartuje, inne procesy będą na zawsze blokowane w stanie początkowym. Jeśli natomiast obydwa zostaną uruchomione jednocześnie, to obydwa wykonają swoje czynności, ale wynikowy stan zmiennej `state` – jest nieprzewidywalny. Może mieć wartość `0`, `1` albo `2`.

Wiele rozwiązań tego problem było rozpatrywane, począwszy np. od zakazu użycia zmiennych globalnych, do wyposażenia komputera w specjalny zestaw instrukcji – gwarantujących indywidualne sprawdzanie i ustawianie kolejności dostępu do współdzielonych pomiędzy procesy zmiennych. Podany dalej przykład rozwiązania problemu, był pierwszym publikowanym rozwiązaniem, pochodzących od holenderskiego matematyka *Dekkera*. Rozwiązanie opiera się na parze procesów wzajemnie wykluczające dostęp do zadanej *sekcji krytycznej* kodu, operując trzema dodatkowymi zmiennymi globalnymi. Pierwsze cztery linie kodu, podanego poniżej, napisanego w PROMELA – są typowymi w stylu języka C makrodefinicjami. Pierwszym dwu makrodefinicje przypisują stanowi `true` – wartość `1`, zaś wartości `false` wartość `0`. Podobnie `Aturn` oraz `Bturn` - są definiowane, jako stałe (kod 5.3.4.30).

```
#define true   1
#define false  0
#define Aturn  false
#define Bturn  true

bool x, y, t;

proctype A()
{       x = true;
        t = Bturn;
        (y == false || t == Aturn);
        /* critical section */
        x = false
}
```

```
proctype B()
{       y = true;
        t = Aturn;
        (x == false || t == Bturn);
        /* critical section */
        y = false
}

init
{       run A(); run B()
}
```

5.3.4.30. Kod przykładu poprzedzony makrodefinicjami

Powyższy algorytm może być wykonywany wielokrotnie i nie zależy - od relatywnej szybkości pary swoich procesów.

Inną drogą uzyskania rozwiązania tegoż problem, jest wprowadzone w języku PROMELA sekwencji atomowych (`atomic sequences`) instrukcji. Używając prefiksu – słowo kluczowe `atomic`, można wskazać sekcję instrukcji, która ma być wykonywana jako niepodzielna jednostka, nieprzesłanialną przez żaden inny proces. W przypadku powstania błędu w toku przebiegu (*run-time error*) – wykonywania jakiejkolwiek instrukcji, różnej od pierwszej instrukcji modelu, ma miejsce blokowanie wszystkich sekwencji atomowych. Tak, więc, można użyć sekwencji atomowych do ochrony współbieżnego dostępu do zmiennej globalnej `state`, omawianej w poprzedzającym przykładzie (kod 5.3.4.40).

```
byte state = 1;

proctype A()
{       atomic {
          (state==1) -> state = state+1
        }
}

proctype B()
{       atomic {
          (state==1) -> state = state-1
        }
}

init
{       run A(); run B()
}
```

5.3.4.40. Kod przykładu użycia sekwencji atomowych

W powyższym przypadku – końcowa wartość zmiennej globalnej `state` – jest zerem albo dwa, w zależności, który z procesów wykonywano. Wszelkie pozostałe procesy będą trwale blokowane. Sekwencje atomowe mogą być ważnym środkiem redukcji złożoności modeli weryfikacji. Zauważmy, że użycie sekwencji atomowych pozwala ograniczyć współbieżność procesów dopuszczalną w systemach rozproszonych. Innymi słowy trudne do kierowania modele, mogą stać się znacznie łatwiejszymi, przykładowo na drodze etykietowania operowania lokalnymi zmiennymi za pośrednictwem sekwencji atomowych. Na tej drodze, można uzyskać bardzo istotną redukcję złożoności modelowanego systemu.

Piśmiennictwo: *Holzmann G.* H.3.1, H.3.2.

5.3.5. MIĘDZYPROCESOWE PRZEKAZYWANIE KOMUNIKATÓW

Kanały przekazywania komunikatów są używane do modelowania przekazywania danych pomiędzy procesami. Kanały mogą być deklarowane zarówno lokalnie, jak globalnie. Np. jak niżej:

```
chan qname = [16] of { short }
```

Jest to deklaracja kanału komunikacji mogącego zawierać 16 komunikatów typu `short`. Nazwy kanałów mogą być przekazywane z jednego procesu do innego za pośrednictwem kanałów lub jako parametry w toku inicjacji procesu. Jeśli komunikaty, które mają być przekazywane kanałem muszą mieć więcej niż jedno pole, deklaracja może wyglądać następująco:

```
chan qname = [16] of { byte, int, chan, byte }
```

Tym razem kanał przechowuje również szesnaście komunikatów, każda składa się z dwóch jedno-bajtową wartości, jednej 32-bitowej wartości i nazwy kanału.

Instrukcja: `qname!expr` przesyła komunikat - wartość wyrażenia `expr` za pośrednictwem wyżej utworzonego kanału, dokładniej mówiąc dołączając wartość komunikatu do końca kolejki kanału. Natomiast `qname?msg`instrukcja, pobiera komunikat z czoła kolejki kanału i umieszcza pobrany komunikat w zmiennej `msg`. Kanał przekazuje komunikaty na zasadzie *first-in-first-out*. W omawianym przykładzie tylko pojedynczy komunikat przechodzi przez kanał. Jeśli więcej niż jedna wartość składa się na komunikat, poszczególne wartości dzielone są przecinkami – tworząc listę jak pokazano niżej:

```
qname!expr1,expr2,expr3
qname?var1,var2,var3
```

Błędem jest wysyłanie albo otrzymywanie większej liczby parametrów składających się na komunikat, niż zostało to wcześniej zadeklarowane. Stosujemy konwencję, pierwsze pole komunikatu często jest używane do określenia typu komunikatu (np. jakieś wartości stałej). Alternatywna notacja, będąca równoważną omawianej, jest na początku określenie typu komunikatu, a następnie w nawiasach okrągłych listę pól składających się na komunikat. Ogólnie można zapisać w formie, jak niżej:

```
qname!expr1(expr2,expr3)
qname?var1(var2,var3)
```

Operacja wysyłania jest wykonywalną tylko wtedy, kiedy adresowany kanał niej jest pełen. Podobnie operacja otrzymywania, jest wykonywalną, – jeśli kanał nie jest pusty. Opcjonalnie, pewne argumenty operacji otrzymywania mogą być stałymi:

```
qname?cons1,var2,cons2
```

W powyższym przykładzie, warunkiem wykonywalności operacji otrzymania, jest zgodność wartość wszystkich pól wymienionych w komunikacie, czyli pola wyspecyfikowane, jako stałe, muszą być zgodna z odpowiadającymi im deklaracjami w nagłówku kanału. Jeśli instrukcja będzie niewykonywalna, proces gotowy do wykonywania będzie ocZekiwać, aż do momentu, w którym instrukcja stanie się wykonywalną.

Poniżej (kod 5.3.4.50) pokazany jest przykład użycia omawianych wyżej mechanizmów:

```
proctype A(chan q1)
{       chan q2;
        q1?q2;
        q2!123
}

proctype B(chan qforb)
{       int x;
        qforb?x;
        printf("x = %d\n", x)
}
```

```
init {
        chan qname = [1] of { chan };
        chan qforb = [1] of { int };
        run A(qname);
        run B(qforb);
        qname!qforb
}
```

5.3.4.50. Kod przykładu korzystania z przekazu asynchronicznego kanałami

W wyniku działania powyższego modelu – wydrukowany zostanie napis: `1 2 3`.

Predefiniowana funkcja `len(qname)` zwraca liczbę komunikatów aktualnie składowanych w kanale `qname`. Zauważmy, że a len jest raczej użyta jako instrukcja, niż jako prawostronne podstawienie, będzie niewykonywalne, jeśli kanał jest pusty: zwraca wówczas wartość zero, co z definicji oznacza, że dana instrukcja jest czasowo niewykonywalną.

```
(qname?var == 0)        /* syntax error */
```

jak również

```
(a > b && qname!123)    /* syntax error */
```

są niepoprawnymi instrukcjami PROMELA (zauważmy, że te warunki nie mogą być wykonywane bez efektu ubocznego). Dla instrukcji pobierania, można zastosować alternatywny zapis użycie za znakiem zapytania - nawiasów kwadratowych:

```
qname?[ack,var]
```

co jest traktowane, jako warunek. Warunek zwracający 1, jeśli odpowiadające mu instrukcja otrzymania

```
qname?ack,var
```

jest wykonywalną, czyli tak jakby komunikat `ack` znajdował się na czole kanału. W przeciwnym przypadku zwraca wartość zero. W żadnym przypadku ocena instrukcji takiej jak

```
qname?[ack,var]
```

nie tworzy żadnego efektu ubocznego: otrzymywany komunikat jest oceniany, a niewykonywany.

Zauważ dalej, że nie-atomowa sekwencja dwóch instrukcji, takich jak

```
(len(qname) < MAX() -> qname!msgtype
```

albo

```
qname?[msgtype] -> qname?msgtype
```

druga instrukcja, *niekoniecznie* jest wykonywalna po wykonaniu pierwszej z instrukcji. Może wystąpić wyścig warunków, w przypadku, gdy dostęp do kanałów jest dzielony pomiędzy szereg procesów. W pierwszym z przypadków – inny proces może wysłać komunikat do kanału `qname`, natychmiast po tym, kiedy pierwszy z procesów stwierdzi, że kanał nie jest pełny. W drugim przypadku, inny proces może przejąć komunikat, bezpośrednio po tym jak pierwszy z procesów stwierdzi obecność tegoż komunikatu.

Dotychczas mówiliśmy jedynie o asynchronicznej komunikacji pomiędzy procesami – z użyciem kanałów, deklarowanych jak niżej:

```
chan qname = [N] of { byte }
```

gdzie `N` jest dodatnią stała całkowitoliczbową – definiującą rozmiar bufora kanału. Logicznie myśląc, rozszerzeniem naturalnym powyższej deklaracji jest instrukcja:

```
chan port = [0] of { byte }
```

definiująca tzw. `randezvous port`, pozwalający na przesłanie jednobajtowego komunikatu.

Rozmiar kanału (a raczej buforu kanału) jest zerowy, czyli że kanał o nazwie `port` - może przekazywać komunikaty bez ich składowania. Współdziałanie procesów za pośrednictwem portów `randezvous`, jest z definicji synchroniczne. Rozważmy następujący przykład (kod 5.3.4.60):

```
#define msgtype 33

chan name = [0] of { byte, byte };

proctype A()
{       name!msgtype(124);
        name!msgtype(121)
}

proctype B()
{       byte state;
        name?msgtype(state)
}

init
{       atomic { run A(); run B() }
}
```

5.3.4.60. Kod przykładu korzystania z przekazu synchronicznego kanałem

Kanał o nazwie `name`, jest globalnym portem *rendezvous*. Dwa procesy będą synchronizowały wykonania ich pierwszych instrukcji: „*handshake*" z pomocą komunikatu `msgtype` oraz przesłanie wartości `124` do zmiennej lokalnej `receive` operacją procesu `B`. Druga instrukcja procesu A będzie niewykonywalną, ponieważ niema dopasowania operacji w procesie B.

Jeśli kanał o nazwie `name` jest zdefiniowany z niezerową pojemnością bufora, zachowanie będzie inne. Jeśli rozmiar bufora jest, co najmniej `2`, wówczas proces typu `A` może zakończyć wykonywanie, zanim zdarzenie równoważne nastąpi. Jeśli rozmiar bufora jest `1`, sekwencja zdarzeń będzie następująca. Proces typu A może wykonać pierwszą czynność wysyłania, a następnie zostanie zablokowany na drugiej czynności, ponieważ bufor kanału jest już pełny. Natomiast proces typu B może uzyskać pierwszy komunikat i wykonać swoje czynności. W tym momencie proces A stanie się znowu wykonywalnym i zostanie wykonany, pozostawiając ostatni komunikat w buforze kanału

Komunikacja rendezvous ma charakter binarny: tylko dwa procesy w niej uczestniczą, nadawca i odbiorca, a operacja jest synchronizowana z pomocą *rendezvous handshake*. Dalej poznamy przykład pokazujący jak wykorzystać powyższą konstrukcję do budowy semaforów. Wcześniej jednak, omówimy struktury sterowania, które będą nam potrzebne we wzmiankowanym przykładzie semafora.

Piśmiennictwo: *Holzmann G.* H.3.1, H.3.2.

5.3.6. STRUKTURY STEROWANIA

W ramach dotychczasowych rozważań, wprowadziliśmy trzy rodzaje struktur sterowania wykonywania modelu, a mianowicie: konkatenacją instrukcji języka PROMELA, równoległym wykonywaniem procesów oraz sekwencjami atomowymi. Obok wymienionych, istnieją jeszcze trzy inne podstawowe struktury sterowania języka PROMELA, które dalej omówimy. Są to: wybór przypadku, powtórzenia oraz skok bezwarunkowy. Rozważania uzupełnimy posługiwaniem się: podstawieniami, wykorzystywaniem tzw. *timeoutów* oraz specjalnych etykiet.

5.3.6.01. **Wybór przypadku**. Jest to najprostsza struktura sterowania. Używając względnych wartości dwóch zmiennych `a` oraz `b`, możemy wybrać dwie opcje. Przykładowo, piszemy:

```
if
:: (a != b) -> option1
:: (a == b) -> option2
fi
```

Struktura wyboru przypadku zawiera dwie wykonywalne sekwencje instrukcji, każdą poprzedzoną podwójnym dwukropkiem. Tylko jedna z podanych sekwencji może być wykonana. Dana sekwencja jest wybrana tylko wtedy, kiedy jej pierwsza instrukcja jest wykonywalna. Pierwsza instrukcja sekwencji nazywana jest strażnikiem.

W powyższym przykładzie obaj strażnicy są wzajemnie zależni, ale tak nie musi być zawsze. Jeśli więcej niż jeden strażnik jest wykonywalny, jedna z odpowiadających mu sekwencji jest wybierana niedeterministycznie. Jeśli natomiast wszyscy strażnicy są niewykonywalni, ten proces będzie blokowany do moment, w którym przynajmniej jeden strażnik może być wybranym. Niema żadnych ograniczeń dotyczących typu instrukcji, która zostanie użyta, jako strażnik. Następujący przykład (kod 5.3.6.02), używa w tym celu instrukcji wejścia:

```
#define a 1
#define b 2

chan ch = [1] of { byte };

proctype A()
{       ch!a
}

proctype B()
{       ch!b
}

proctype C()
{       if
        :: ch?a
        :: ch?b
        fi
}

init
{       atomic { run A(); run B(); run C() }
}
```

5.3.6.02. Kod przykładu z wzajemnie zależnymi strażnikami

Ten przykład definiuje trzy procesy i jeden kanał. Pierwsza opcja w strukturze selekcji, jest proces typu `C`, który jest wykonywalny, jeśli kanał zawiera komunikat `a`, gdzie `a` jest stałą o wartości `1`, zdefiniowaną w makrodefinicji na początku modelu. Druga opcja jest wykonywalna, jeśli zawarty komunikat jest `b`, gdzie podobnie `b` jest stałą. Jaki komunikat będzie dostępny – zależy od nieznanej względnej szybkości procesów. Proces poniższego typu (kod 5.3.6.03) będzie albo zwiększał, albo zmniejszał wartość zmiennej `count`.

```
byte count;

proctype counter()
{
        if
        :: count = count + 1
```

```
        :: count = count - 1
        fi
}
```

5.3.6.03. Kod przykładu korzystania z wcześniej nieznanej względnej szybkości procesów

5.3.6.04. **Powtórzenia** (repetycje). Logicznym rozszerzeniem struktury selekcji jest struktura powtarzania (repetycji). Możemy zmodyfikować wcześniej omawiany model, tak, aby otrzymać pętlę, w której losowo zmienia się wartość zmiennej - poprzez jej zmniejszanie albo zwiększanie.

```
byte count;
proctype counter()
{
        do
        :: count = count + 1
        :: count = count - 1
        :: (count == 0) -> break
        od
}
```

5.3.6.05. Kod przykładu zawierającego powtórzenia

Tylko jedna opcja może być wybrana w czasie wykonywania modelu. Po zakończeniu wykonywania wybranej opcji, wykonanie struktury jest powtarzane. Naturalnym sposobem zakończenia powtarzań struktury jest wykonanie instrukcji `break`. W tym przykładzie, pętla może być przerwana, kiedy zmienna `count` przyjmie wartość zero. Zauważmy, że oczywiście pętla nie może się zakończyć, jak długo dwie pozostałe opcje są wykonywalne. Żeby wymusić zakończenie, kiedy licznik `count` przyjmie wartość zero, możemy zmodyfikować nasz model jak dalej (kod 5.3.6.06).

```
proctype counter()
{
        do
        :: (count != 0) ->
                if
                :: count = count + 1
                :: count = count - 1
                fi
        :: (count == 0) -> break
        od
}
```

5.3.6.06. Kod przykładu przerwania pętli przy liczniku równym zero

Innym przykładem jest współdziałanie pary procesów producent i konsument (*producer and consumer*). Pierwsza linia deklaracji dwu symbolicznych zmiennych modelu: `P` oraz `C`. Efektem tej deklaracji, bardzo podobnej do deklaracji `enum`[92] w programie C. Parser SPIN przypisuje unikalną dodatnią wartość całkowitą każdej symbolicznej zmiennej, będącą wewnętrzną interpretacją. Jak sugeruje to nazwa typu, ten typ deklaracji jest często stosowany dla zdefiniowania listy nazw komunikatów wykorzystywanych używanych w komunikacji pomiędzy procesami.

Następną deklaracją po `mtype`, jest globalna deklaracja zmiennej nazwanej `turn` typu `mtype`. Zmienna ta może przyjmować wartości dla typu `mtype (1,.., 255)`, co w tym przypadku

[92] Język C umożliwia zastosowanie typu wyliczeniowego począwszy od standardu C89, np. `enum typ_wyliczeniowy {PUNKT, PROSTA, TROJKAT, KWADRAT, KOLO = 360, WIEKSZE_KOLO};`

oznacza, że można podstawić wartości przypisane symbolicznie zmiennym `P` oraz `C`. Dla uniknięcia niejednoznaczności zmiennej `turn` typu `mtype` przypisano wartość początkową `P`. Jako następne mamy dwie deklaracje `proctype`, z których każda poprzedzona jest słowem `active`, powodującą tworzenie instancji automatyczne każdego z tych procesów (kod 5.3.6.07).

```
mtype = {P,C};
mtype turn = P;
active proctype producer()
{
        do
        :: (turn == P) ->
                printf("Produce\n");
                turn = C
        od
}
active proctype consumer ()
{
        do
        :: (turn == C) ->
                printf("Consume\n");
                turn = P
        od
}
```

5.3.6.07. Kod przykładu tworzenia automatycznego instancji procesów

Struktura sterowania w obu `proctype` jest taka sama: obydwa zawierają pętlę. Pętla w PROMELA rozpoczyna słowo kluczowe `do`, kończy słowo kluczowe `od`. Ciało pętli może zawierać jedną lub więcej opcjonalnych sekwencji instrukcji poprzedzonych podwójnym dwukropkiem `::`. W naszym przykładzie, pętle posiadają po jednej sekwencji. Każda z tych sekwencji zawiera trzy instrukcje. Pierwsza z instrukcji po podwójnym dwukropku ma specjalne znaczenie: nazywana jest instrukcją straży (*guard*), która decyduje czy następująca sekwencja instrukcji będzie wykonywana, czy też nie. W naszym przykładzie sekwencja instrukcji `producer`, jedynym przypadkiem prowadzącym do wykonywania sekwencji jest spełnianie warunku straży `(turn == P)`. Oznacza to, że instrukcje składające się na sekwencję po instrukcji straży, mogą być wykonane tylko wtedy gdy zmienna `turn` ma wartość `P`. PROMELA używa podwójnego znaku równości (==) jako symbol operacji równości, zaś pojedynczego symbolu równości (=) jako symbolu operacji podstawienia. Pętla w PROMELA – jest podobna do innych konstrukcji sterowania używanych w tym języku. Jeśli napiszemy pętle procesu `producer`, korzystając z instrukcji wyboru z użyciem instrukcji skoku bezwarunkowego do etykiety - dla uzyskania powtarzalności wykonywania sekwencji instrukcji, to wówczas:

```
active proctype producer()
{
again:  if
        :: (turn == P) ->
                printf("Produce\n");
                turn = C
        fi;
        goto again
}
```

5.3.6.08. Kod przykładu

Zasada wykonywania opcjonalnych sekwencji instrukcji pozostaje niezmieniona. Jedyna różnica pomiędzy strukturą opartą o wybór, a strukturą powtarzania, jest w tym, że pętla jest automatycznie powtarzana od startu poprzez okres spełniania warunku sterującego, podczas gdy struktura oparta o wybór powoduje przejście do następnej instrukcji, którą w tym przypadku jest skok, który powoduje przejście do początku sekwencji, umożliwiający ponowne wykonanie sekwencji. Wykonanie przerwania powtarzanej sekwencji umożliwia zastosowanie instrukcji *break*. Podobnie jak w C zastosowanie instrukcji *break* powoduje natychmiastowe przerwanie wykonywania powtarzalnej sekwencji instrukcji.

Gdy wszystkie warunki straży w pętli dają wartość *false* (w naszym przykładzie mamy tylko jeden warunek straży), nie ma już warunków do dalszego wykonywania powtarzalnej sekwencji instrukcji pętli i dalsze wykonywanie procesu jest *blokowane* (*blocks*). Ta semantyka pozwala na wyrażenie synchronizacji procesów w sposób zwarty i jasny. Zauważmy, na przykład, że zamiast kodowania oczekiwania w postaci cyklu:

```
wait:  if
        ::(turn == P) ->
        :: else -> goto wait
        fi;
         …
```

wystarczającym jest napisanie w PROMELA:

```
(turn == P)-> …
```

Strzałki oraz średniki są równoważne w PROMELA, tak więc powyższa instrukcja, może być zapisana w postaci:

```
(turn == P); …
```

oczywiści zakładając, że następują dalsze instrukcje. W ogólnie przyjętym stylu pisania specyfikacji, tylko potencjalnie blokujące instrukcje są zakończone strzałką, jako separatorem pomiędzy kolejnymi instrukcjami, wszelkie inne typy instrukcji są zakańczane symbolem średnika, jest to jednak przyjęty styl pisania programów, a nie gramatyka.

Wprowadziliśmy dodatkowe słowo kluczowe PROMELA: *else*. Użycie tego kluczowego słowa jako instrukcji stróżującej w strukturze wyboru albo powtórzenia – określone przez warunek *true*, jest możliwe jedynie wtedy i tylko wtedy wszystkie inne warunki straży w danej strukturze przyjmują wartość *false*. W poprzednio podanym przykładzie, użycie słowa *else* jest równoważne instrukcji:

```
!(turn == P)
```

Zauważmy, że w danej strukturze wielokrotne warunki stróżujące przyjmują równocześnie wartość *true*. W tym przypadku, SPIN wybiera do wykonania jeden z tych warunków straży jako warunek niedeterministyczny. Jak wiadomo, może być nie więcej niż jedna opcja *else* przypadająca na jedną konstrukcję *if* albo *do*, dlatego też kiedy opcja *else* zostaje wybrana, jest to jedyna wykonalna opcja.

Innym punktem wartym omówienia, jest sytuacja gdy opcja *else* zostaje usunięta (nie może być wybrana), wykonywanie zostaje zablokowane (*blocks*) do puki - co najmniej jeden warunek wartujący przejdzie do stanu *true*. To czasowe (lub permanentne) blokowanie dostarcza odpowiedniego środka do modelowania międzyprocesowej synchronizacji procesów współbieżnych.

Użycie *nie-determinizmu* przy definiowaniu procesu pozwala na rozważną swobodę – konstruowania modelu weryfikacji. Mówiąc, że dwa możliwe zdarzenia - są możliwe w danym

punkcie wykonywania, możemy usunąć nieistotne szczegóły z modelu i efektywniej weryfikować poprawność projektu niezależnie od tego szczegółu. Jest również drugi typ *niedeterminizmu* w modelu PROMELA. Jeśli w dowolnym punkcie wykonywania więcej niż jeden proces ma wykonywalną instrukcję która mógłby odbywa się, to semantyka PROMELA powoduje, że któryś z tych procesów zostanie wybrany do wykonania; zaś wybór z założenia jest *niedeterministyczny*. W istocie, ten poziom systemowego *niedeterminizmu* oznacza, że nie przyjmujemy *a priori* założeń dotyczących zachowania się scheduler'a. Zauważmy przykładowo, że procesy asynchroniczne, są często sterowane niezależnym scheduler'am działającym na wydzielonym sprzęcie w sieci procesorów. Podsumowując - temat harmonogramowania procesów, nawet na jednoprocesorowym systemie, ma miejsce właściwie podejrzenie istnienia ograniczonej niepewności przy weryfikacji.

5.3.6.09. **Skoki bezwarunkowe**. Inną drogą przerwania pętli, jest użycie bezwarunkowego skoku: niesławnej instrukcja `goto`, oraz etykiety `done` – do której wykonywany jest skok. Pokazujemy użycie tego rozwiązania na przykładzie implementacji algorytmu Euklidesa – znajdowania największego wspólnego dzielnika dwóch niezerowych liczb dodatnich (kod 5.3.6.10).

```
proctype Euclid(int x, y)
{
        do
        :: (x >  y) -> x = x - y
        :: (x <  y) -> y = y - x
        :: (x == y) -> goto done
        od;
done:
        skip
}
```

5.3.6.10. Kod przykładu algorytmu Euklidesa w języku PROMELA

W powyższym przykładzie instrukcja `goto` przekazuje sterowanie do instrukcji oznaczonej etykietą nazwaną `done`. Etykieta może pojawić się tylko bezpośrednio przed instrukcją i jest zakończona symbolem dwukropka. W przykładzie skaczemy do zakończenia modelu. Ze względu na konieczność umieszczenia instrukcji po użyciu etykiety, użyjemy instrukcji `skip` – przejdź dalej, wykonywalnej zawsze, podobnie jak instrukcja `goto`.

5.3.6.11. **Filtrowanie komunikatów**. Następujący przykład zawiera specyfikację filtru otrzymywanych komunikatów z kanału `in`, a następnie dokonuje podziału otrzymanych komunikatów pomiędzy dwa kanały `large` oraz `small` wartości, w zależności od otrzymanej wartości komunikatów. Stałej `N` zostaje nadana wartość `128`, zaś rozmiar jest określony na równy `16` w dwóch makrodefinicjach (kod 5.3.6.12).

```
#define N    128
#define size  16

chan in    = [size] of { short };
chan large = [size] of { short };
chan small = [size] of { short };

proctype split()
{       short cargo;

        do
        :: in?cargo ->
                if
```

```
                :: (cargo >= N) ->
                        large!cargo
                :: (cargo <  N) ->
                        small!cargo
                fi
        od
}

init
{       run split()
}
```

5.3.6.12. Kod przykładu filtrowania kmunikatów

Proces typu łączenia `merge` dwóch strumieni z kanałów `large` oraz `small`, w kierunku odwrotnym I umieszczający wyniki łączenia w jednym kanale `in`, prawdopodobnie w odwrotnej kolejności, można opisać jak następuje (kod 5.3.6.13):

```
proctype merge()
{       short cargo;

        do
        ::      if
                :: large?cargo
                :: small?cargo
                fi;
                in!cargo
        od
}
```

5.3.6.13. Kod przykładu łączenia dwóch strumieni komunikatów

Jeśli z kolei zmodyfikujemy proces `init` jak dalej (kod 5.3.6.14), procesy rozdzielania, czyli `split` oraz składania, czyli `merge` - mogłyby gorliwie wykonywać swoje czynności w nieskończoność.

```
init
{       in!345; in!12; in!6777;
        in!32;  in!0;
        run split();
        run merge()
}
```

5.3.6.14. Kod przykładu rozdzielania strumienia komunikatów na dwa strumienie

Jako przykład końcowy tych rozważań, rozważymy implementację semafora Dijkstry (kod 5.3.6.15), używając do tego binarną komunikację `rendezvous`.

```
#define p       0
#define v       1
        chan sema = [0] of { bit };
        proctype dijkstra()
{       byte count = 1;

        do
        :: (count == 1) ->
                sema!p; count = 0
        :: (count == 0) ->
                sema?v; count = 1
        od
}
        proctype user()
{       do
        :: sema?p;
```

```
            /*     critical section */
            sema!v;
            /* non-critical section */
        od
}
        init
{       run dijkstra();
        run user();
        run user();
        run user()
}
```

5.3.6.15. Kod przykładu implementacji semafora Dijkstry

Semafor Dijkstry gwarantuje, że tylko jeden procesów będzie miał – w danym przedziale czasu dostęp do sekcji krytycznej. Co równocześnie nie powoduje monopolizacji dostępu do sekcji krytycznej przez tylko jeden z procesów.

5.3.6.16. **Modelowanie procedur i rekursji**. Procedury mogą być modelowane, jako procesy, w szczególności procesy rekursywne. Zwracana wartość może być ponownie przekazywana, do wywołania procesu poprzez zmienną globalną lub poprzez komunikat. Poniższy model ilustruje takie postępowanie (kod 5.3.6.17).

```
proctype fact(int n; chan p)
{       chan child = [1] of { int };
        int result;

        if
        :: (n <= 1) -> p!1
        :: (n >= 2) ->
                run fact(n-1, child);
                child?result;
                p!n*result
        fi
}
init
{       chan child = [1] of { int };
        int result;

        run fact(7, child);
        child?result;
        printf("result: %d\n", result)
}
```

5.3.6.17. Kod przykładu procedury rekursywnej

Proces `fact(n,p)` metodą rekursji oblicza wartości funkcji `n!` (n - silnia), przekazując kolejne wyniki za pomocą komunikatów do swego rodzicielskiego procesu `p`.

5.3.6.18. **Instrukcja `timeout`**. Omawialiśmy już dwa typy instrukcji o wstępnie definiowanych działaniach w języku PROMELA: `skip` oraz `break`. Kolejną tego typu instrukcją jest `timeout`. Ta instrukcja z kolei modeluje specjalny warunek, który pozwala procesowi na przerwanie oczekiwania na spełnienie warunku, który może nigdy nie być prawdziwy (`true`), np. w sytuacji oczekiwania na komunikat z pustego kanału. Słowo kluczowe `timeout` pozwala w języku Promela zapisać ucieczkę procesu ze stanu zawieszenia (`hang state`). Towarzyszący słowu kluczowemu `timeout` warunek, przyjmuje wartość prawda (`true`), tylko wtedy, gdy w systemie rozproszonym (współbieżnym) – żadna inna instrukcja, nie jest wykonywana. Zauważmy, że jest

to rozwiązanie całkowicie abstrahujące od rozważań czasowych, ale niezbędne w toku weryfikacji. Tym samym nie określiliśmy, jak ze względu na czas instrukcja `timeout`, jest zaimplementowana. Prosty przykład zastosowania instrukcji `timeout` w procesie `watchdog`, dla wysłania komunikatu `reset` (w przypadku wstrzymania działania modelu), za pośrednictwem kanału `guard` – pokazany jest poniżej (kod 5.3.6.19).

```
proctype watchdog()
{
        do
        :: timeout -> guard!reset
        od
}
```

5.3.6.19. Kod przykładu użycia instrukcji `timeout`

5.3.6.20. **Stwierdzenie** (`assert`). Kolejną ważną konstrukcją w języku PROMELA, wymagająca dodatkowych wyjaśnień jest instrukcja `assert` (stwierdzenia). Instrukcja ta ma postać:

```
assert(any_boolean_condition)
```

Instrukcja `assert` jest zawsze wykonywalna. Jeśli podany warunek boolowski jest spełniony (`true`), instrukcja nie daje żadnego efektu. Jeśli natomiast, podany warunek nie jest spełniony (`false`), instrukcja wytworzy raport błędu – w toku wykonywania przebiegu weryfikacji procesora SPIN.

5.3.6.21. **Wychwytywanie zachowań**. Język modelowania PROMELA posiada kilka dodatkowych własności przeznaczonych dla potrzeb aspektów weryfikacji, a dokładniej mówiąc własności przeznaczonych do wychwytywania szczególnych zachowań modelu. Są to, odpowiednio: (1) sposoby stosowania kluczowych etykiet (`label`); (2) posługiwanie się instrukcją `timeout` oraz (3) omawiana wyżej instrukcja stwierdzenia (`assert`). Kolejne sekcje poświęcimy stosowaniu kluczowych etykiet.

5.4.3.22. **Etykiety** „`end state`". Kiedy język PROMELA jest używany do celów weryfikacji poprawności działania modelu, użytkownik musi posiadać możliwości rozróżniania subtelnych różnic w zachowaniu modelu. W szczególności, jeżeli weryfikowany model jest sprawdzany na obecność istnienia wzajemnych zakleszczeń procesów (`deadlock`), weryfikator musi mieć możliwość rozróżnienia pomiędzy normalnym stanem zakończenia działania modelu od stanu nienormalnego (np. powstałego w wyniku zakleszczenia).

Jednoznaczne osiągnięcie stanu końcowego modelu ma miejsce wtedy, gdy: (1) każdy z procesów modelu, który został uruchomiony (*instancjonowany*) osiągnął poprawnie swoje zakończenie (*terminated*) zdefiniowane w ciele procesu - oraz (2) wszystkie kanały komunikacji są puste. Często mamy jednak sytuację niejednoznaczną, nie wszystkie procesy modelu mogły osiągnąć koniec swojego ciała. Niektóre procesy mogły zostać w stanie bezczynnym (*idle*) lub cierpliwie oczekiwać w gotowości podjęcia działania np. po pojawieniu się nowych danych wejściowych.

Dla uzyskania jednoznaczności stanu, weryfikator musi mieć możliwość rozróżnienia pomiędzy poprawnym osiągnięciem stanu końcowego, a sytuacją wzajemnego zakleszczenia procesów. W tym celu, język PROMELA daje możliwość użycia tzw. etykiet stanów (`state labels`).

Przykładowo, w tym celu w omawianym wcześniej modelu `dijkstra()`, (kod 5.3.6.23) użyliśmy etykiety `end`.

```
proctype dijkstra()
{       byte count = 1;

end:    do
        :: (count == 1) ->
                sema!p; count = 0
        :: (count == 0) ->
                sema?v; count = 1
        od
}
```

5.3.6.23. Kod przykładu z użyciem etykiety `end`

Tym samym pokazujemy, że nie jest to sytuacja błędna, proces typu `dijkstra()` nie osiągnął jeszcze końcowego nawiasu klamrowego swojego ciała, ale oczekuje w pętli. Oczywiście, taki stan oczekiwania może również pojawić się w stanie zakleszczenia procesów (`dedlock`), ale w tym przypadku nasz proces jest w stanie zaznaczonym etykietą `end`.

Należy zauważyć, że weryfikowany model może zawierać więcej niż jeden poprawny stan końcowy (`end state`). Ponieważ w ramach jednego procesu etykiety muszą być unikalne, wszystkie etykiety oznaczające poprawnie osiągnięte stany końcowe zaczynają się od trzech liter „`end`" i różnią się dalszymi znakami (np. literami lub cyframi), np.: `enddne`, `end0`, `end_appel`, itp.

5.3.6.24. **Etykiety „`progress state`"**. W podobnym duchu jak etykiety `end`, użytkownik może definiować w ramach modelu inny rodzaj etykiet, zwanych etykietami `progres state`. W tym przypadku, etykiety są używane do zaznaczenia stanów, które muszą być osiągnięte przez model, aby stwierdzić postępy w wykonywaniu modelu. Każda wykonywana w nieskończoność pętla modelu, który nie przechodzi przynajmniej raz przez etykietowany stan postępu wykonywania (`progres state`), jest potencjalnym pętlą zagłodzenia (`starvation loop`). W przykładzie `dijkstra`, możemy etykietować dokonanie przejścia przez semafor, jako „`progress`" i zapytać weryfikator, czy niema takich powtarzalnych przejść przy wykonywaniu modelu, w których przynajmniej jeden proces nie wszedłby do krytycznej sekcji modelu, chronioną przez strażnika semafora (kod 5.3.6.25).

```
proctype dijkstra()
{       byte count = 1;

end:    do
        :: (count == 1) ->
progress:       sema!p; count = 0
        :: (count == 0) ->
                sema?v; count = 1
        od
}
```

5.3.6.25. Kod przykładu użycia etykiet `end` oraz `progress`

Zauważmy, że taka etykieta - nie może być umieszczona bezpośrednio przed lub bezpośrednio po symbolu „::", tak więc wymaga to pewnej kreatywności, żeby znaleźć właściwe miejsce do wstawienia etykiety. Jeśli więcej niż jeden stan powinien zostać zaetykietowany etykietą typu

progress, dla zapewnienia jednoznaczności użyjemy wspólnego prefiksu etykiety, tworząc np. etykiety w rodzaju: progress0, progress_foo, itp.

Wszystkie analizatory walidacji generowane przez procesor SPIN z flagą -a, muszą (po dokonaniu kompilacji) posiadać opcję „*runtime*" nazwaną -l. Wywołanie wygenerowanego analizatora z taką flagą powoduje szybkie szukanie pętli typu „*non-progress*". zamiast żmudnego poszukiwania zakleszczeń. Szukanie takich pętli wymaga na ogół krótszego czasu wykonania (oraz użycia dwukrotnie mniej pamięci), niż domyślne szukanie zakleszczeń.

5.3.6.26. **Dostarczanie klauzuli**. Wykonywanie procesu normalnie jest ukierunkowane przez zasady synchronizacji zawarte instrukcjach semantyki specyfikacji proctype. Możliwym jest, definiowanie dodatkowych globalnych więzów na wykonywanie procesu. To może być wykonane z pomocą słowa kluczowego provided, które umieszczamy za listą parametrów deklaracji proctype, jak pokazano na następnym przykładzie (kod 5.3.6.27):

```
bool  toggle = true                        /* global variables */
short cnt;                                 /* global variables */
active proctype A() provided (toggle == true)
{
L:    cnt++;                               /* means: cnt = cnt+1 */
            printf("A: cnt=%d\n", cnt);
            toggle = false;                     /* yield control to B
       */         goto L                            /* do it again
*/          }
                  active proctype B() provided (toggle == false)
                  {
                        L:    cnt--;                                /*
means: cnt = cnt-1 */         printf("B: cnt=%d\n", cnt);
      toggle = true;                       /* yield control to A    */
      goto L
}
```

5.3.6.27. Kod przykładu definiowania dodatkowych więzów

Dostarczone klauzule użyte w tym przykładzie wymuszają wykonywanie procesu wymiennie, tworząc nieskończony strumień wyjścia (kod 5.3.6.28):

```
$ spin toggle.pml | more
A:    cnt=1
            B:    cnt=0
A:    cnt=1
            B:    cnt=0
A:    cnt=1
…
```

5.3.6.28. Kod przykładu tworzenia nieskończonego strumienia wyjścia

Proces nie może wykonać żadnego kroku dopóki dostarczona (provided) klauzula przyjmie wartość true. Natomiast brak klauzuli, powoduje domyślne wyrażenie true, wobec braku klauzuli.

Dostarczenie klauzuli może być użyte do zastosowania niestandardowego algorytmu harmonogramowania danego procesu. Omawiana własność umożliwia wprowadzanie dodatku

ceny do weryfikacji systemu. Jednak użycie klauzuli `provided` – może uniemożliwić użycie bardzo silnego algorytmu optymalizacji wyszukiwania przez SPIN.

5.3.6.29. **Zasady wykonalności.** Centralną definicją PROMELA jest składnia *wykonalności* (*executability*), dostarczającą podstawowe znaczenia w tym języku dla synchronizacji procesów. W zależności od stanu systemu, dowolna instrukcja w modelu SPIN jest albo *wykonywalną* (*executable*) albo *blokowaną* (*blocked*). Widzieliśmy już cztery rodzaje instrukcji PROMELA: instrukcje drukowania, instrukcje podstawiania, instrukcję wejścia/wyjścia oraz instrukcje realizacji wyrażeń. Ciekawostką PROMELA jest to, że wyrażenia mogą być użyte jako instrukcje w każdym kontekście. Są one wykonywalne (*passable*) – wtedy i tylko wtedy jeśli ocenia ona wartość boole'owską jako *true*, lub równoważną niezerową wartość całkowitą. Zasady składni PROMELA określają, że instrukcje drukowania i podstawiania, są zawsze bezwarunkowo wykonywalne. Jeśli proces osiągnie punkt swojego kodu, który jest niewykonywalną instrukcją – po prostu zostaje blokowany.

Na przykład, zamiast pisać aktywną pętle oczekiwania:

```
while (a != b)    /* while is not a keyword in Promela */
            skip; /* do nothing, while waiting for a==b */
```

Ten sam efekt w PROMELA można uzyskać z pomocą prostej instrukcji:

```
(a == b);         /* block until a equals b */
```

Taki sam efekt można uzyskać w PROMELA z pomocą poniższych konstrukcji:

```
L:                /* dubious */
                  if
                  :: (a == b) -> skip
                  :: else -> goto L
                  fi
```

lub

```
do                /* also dubious */
:: (a == b) -> break
od
```

ale jest to zawsze mniej efektywne, i nie zalecane przez fachowców PROMELA. Pokazaliśmy wcześniej, że wyrażenia PROMELA powinny być wolne od efektów ubocznych. Przyczyna będzie jasna: blokująca instrukcja wyrażenia, może być oceniana wiele razy, ilekroć pojawi się efekt uboczny, w wyniku może pojawić się chaos. Jest jeden wyjątek od tej zasady. Jak mówiliśmy wcześniej, wyrażenie, które zawiera operator `run` i tworzy efekt uboczny, jest przyczyną pewnego ograniczenia składni. Główne ograniczenie polega na tym, że może pojawiać się tylko jeden operator `run` w wyrażeniu, a jeśli się już pojawi nie może być połączony z żadnym innym operatorem. Oczywiście, pozwala nam to na użycie instrukcji z `run`, jako potencjalnie blokującą operację. Możemy pokazać jednoznacznie ten efekt, jeśli zamiast pisać:

```
run you_run(0);               /* potentially blocking */
```

bez zmiany znaczenia – piszemy:

```
(run you_run(0)) ->     /* potentially blocking */
```

Rozważmy przykładowo, jaki uzyskamy efekt jeśli użyjemy takie wyrażenie `run` w poniższym modelu, jako wariant modelu wcześniej omawianego

```
active proctype new_splurge(int n)
{
      printf("%d\n", n);
      run new_splurge(n+1)
}
```

Jak poprzednio, ze względu na ograniczenie liczby procesów istniejących równolegle, 255-te usiłowanie inicjacji nowego procesu padnie. Przyczyna błędu wyrażenie `run` oceniane na zero, powoduje permanentne blokowanie danego procesu. Zablokowany proces nie może osiągnąć końca swojego kodu i dlatego nie może się zakończyć lub umrzeć. W wyniku, żaden z jego poprzedników także nie umrze. System złożony z 255 procesów dochodzi do rozkruszającego stopu, 254 procesy kończą się, ale są zablokowane przed swoim usiłowaniem śmierci, a pojedynczy proces blokuje usiłowanie rozpoczęcie nowego procesu.

Jeśli ocena wyrażenia z `run` zwróci zero, wykonanie zostaje blokowane, ale żaden efekt uboczny nie pojawi się, przez co niema zagrożenia pojawienia się efektu ubocznego powtarzając kolejny test wykonalności. Jeśli zwracana ocena jest różna od zera, mamy efekt uboczny po zakończeniu wykonywania instrukcji, ale instrukcja, jako całość nie blokuje. To może z całą pewnością stwarzać wątpliwość, czy złożone wyrażenie może być budowane z operatorem `run`. Np.:

```
run you_run(0) && run you_run(1)          /* not valid */
```

wyrażenie mogłoby blokować w przypadku, gdy oba nie mogłyby być instancjami, ale zjawisko mogłoby się nie ujawnić w zależności od tego czy jeden z procesów był utworzony, czy też żaden z nich. Podobnie:

```
run you_run(0) || run you_run(1)          /* not valid */
```

wyrażenie mogłoby blokować w przypadku, gdy oba procesy nie byłyby zainicjowane, ale nie dawałoby odpowiedzi, który z dwu procesów został utworzony.

5.3.6.30. **Podstawienia i wyrażenia.** Podobnie jak w języku C, podstawienia:

```
c = c + 1; c = c - 1                      /* valid */
```

mogą być skrócone do postaci:

```
c++; c--                                  /* valid */
```

natomiast inaczej niż w C:

```
b = c++
```

nie jest poprawnym podstawieniem w PROMELA, ponieważ operand prawostronny, nie jest wolnym od skutków ubocznych wyrażeniem. Niema ekwiwalentu skrótowych zapisów:

```
--c; ++c                                  /* not valid */
```

w PROMELA, ponieważ instrukcje podstawiania takie jak

```
c = c-1; c = c1                           /* valid */
```

brane jako jednostka nie jest równoważna wyrażeniom w PROMELA. Z tymi ograniczeniami, instrukcja taka jak `--c` nie różni się od poprawnego wyrażenia `c--`.
W podstawieniach takich jak

```
variable = expression
```

wartość wszystkich operandów po prawej stronie wyrażenia, są w pierwszej kolejności liczby całkowite ze znakiem, zanim jakikolwiek operand zostanie zastosowany. Zasady pierwszeństwa operatorów dla języka C, determinujące kolejność rozwijania, są podane w tabeli 5.3.6.31 precedensu operatorów (od najwyższego do najniższego). Po zakończenia rozwijania prawej strony wyrażenia, ale przed dokonaniem podstawienia, wyznaczana wartość jest przekształcana w typ zmiennej docelowej. Jeśli prawa strona wychodzi poza zakres domeny docelowego typu, stosowane jest obcięcie wyniku. W toku działania SPIN w trybie symulacji, powstają ostrzeżenia, iż dany typ obcięcia nie jest dopuszczalny.

Jest również możliwym, używanie w stylu języka C – wyrażeń warunkowych w każdym kontekście, w którym wyrażenie jest dopuszczalne. Składnia, oczywiście, nieco różni się, od tej używanej w C.

5.3.6.31 Tabeli precedensu operatorów (od najwyższego do najniższego)		
operator	kierunek	komentarz
() []	→	nawiasy, granice bloków
! ‘ ++ --	←	negacja, uzupełnienie, zwiększenie, zmniejszenie
* / %	→	mnożenie, dzielenie modulo
+ -	→	dodawanie, odejmowanie
<< >>	→	przesuwanie w lewo oraz prawo
< <= > >=	→	operatory relacji
== !=	→	równość, nierówność
&	→	operator logiczny *end* dla bitów
^	→	operator logiczny *xor* dla bitów
|	→	operator logiczny *or* dla bitów
&&	→	operator logiczny *end*
||	→	operator logiczny *or*
-> :	←	operator warunkowy
=	←	operator podstawienia

W języku C można napisać:

```
expr1 ? expr2 : expr3                /* not valid */
```

natomiast w PROMELA piszemy:

```
(expr -> expr2 : expr3)              /* valid */
```

Symbol strzałki jest użyty tutaj dla uniknięcia możliwego nieporozumienia z użyciem znaku „?” w operacji pobierania zawartości kanału. Wartość wyrażenia warunkowego jest równa wartości `expr2` wtedy i tylko wtedy, gdy `expr1` przyjmie wartość *true*, w pozostałych przypadkach jest równy `expr3`. Wyrażenie warunkowe w PROMELA musi być zawarte w nawiasach, dla uniknięcia błędnej interpretacji strzałki, jako separatora instrukcji.

5.3.6.32. **Sterowanie przepływem: instrukcje złożone.** Jak dotychczas, koncentrowaliśmy się na podstawowych instrukcjach PROMELA oraz na sposobach łączenia instrukcji w model zachowań procesowych. Główny typ instrukcji, wzmiankowanych to: drukowanie, podstawienia, wyrażenia oraz instrukcje wysyłania oraz otrzymywania. Widzimy, że `run` jest operatorem, który umożliwia tworzenie takich instrukcji wyrażenia, jak `run sender()`. Podobnie, `skip` nie jest instrukcją, ale wyrażeniem: równoważnym wartości `(1)` lub *true*.

PROMELA posiada pięć typów złożonych instrukcji:

- Sekwencje atomowe
- Kroki deterministyczne
- Selekcje
- Powtórzenia
- Sekwencje opuszczane

Innym mechanizmem sterowania przepływem jest możliwość definiowania w PROMELA makrofunkcji „w wierszu”.

5.3.6.33. **Sekwencje atomowe.** Najprostszą instrukcją złożoną – jest sekwencja atomowa. Prostym przykładem sekwencji atomowej jest na przykład (kod 5.3.6.34):

```
atomic {                /* swap the values of a and b */
            tmp = b;
            b = a;
            a = tmp
}
```

5.3.6.34. Kod przykładu najprostszej sekwencji atomowej

W tym przykładzie, wartość dwu zmiennych `a` oraz `b` są między sobą zamieniane – instrukcjami, których sekwencji wykonania nie można przerwać. Jest to przykład procesu, którego wykonywanie nie może być przerwane (wykonywane są kolejno wszystkie instrukcje sekwencji atomowej) przez żaden inny proces modelu. Częstym wykorzystaniem metody *sekwencji atomowych*, jest przypadek inicjowania serii procesów, w taki sposób, żeby nie uruchomić działania żadnego z tych procesów, zanim inicjalizacja wszystkich z nich zostanie wykonana (kod 5.3.6.35):

```
init {
      atomic {
                  run A(1,2);
                  run B(2,3)
}     }
```

5.3.6.35. Kod przykładu inicjowanie serii procesów sekwencją atomową

Sekwencja atomowa może być niedeterministyczna. Jeśli, jednak, któraś z instrukcji sekwencji atomowej okaże się niewykonalną (to znaczy z blokadą wykonania), łańcuch instrukcji sekwencji atomowej zostaje przerwany i inny proces może przejąć sterowanie modelem. Kiedy jednak, później zablokowana instrukcja stanie się wykonalną, nastąpi niedeterministyczny powrót do procesu, a dalsze wykonywanie sekwencji atomowej, przebiega jakby nie było przerwane.

Zauważmy, że bez pojęcia sekwencji atomowej, dwie poniższe instrukcje, takie jak:

```
nfull(qname) -> qname!msg0
```

lub

```
qname?[msg0] -> qname?msg0
```

druga z tych instrukcji nie musi być wykonalne po wykonaniu pierwszej instrukcji. Może mieć miejsce wyścig kilku procesów o dostęp do wskazanego kanału. Pierwszy z przykładów, to po sprawdzeniu czy kanał nie jest pełny, wysyłany jest komunikat `msg0`. W drugim przykładzie, inny proces usiłuje pobrać komunikat, bezpośrednio po stwierdzeniu obecności komunikatu. Z drugiej strony, sekwencja z redundancją:

```
atomic { qname?[msg0] -> qname?msg0 }
```

jest równoważna pojedynczej instrukcji:

```
qname?msg0
```

5.3.6.36. **Kroki deterministyczne.** Innym sposobem definiowania w całość wykonywanej sekwencji instrukcji, jest użycie słowa kluczowego `d_step`. Nawiązując do poprzednio omawianego przykładu (kod 5.3.6.37):

```
d_step {          /* swap the values of a and b */
            tmp = b;
            b = a;
            a = tmp
```

5.3.6.37. Kod przykładu użycia instrukcji `d_step`

Inaczej jednak, niż w przypadku sekwencji atomowej, sekwencja zaczynająca się od słowa kluczowego `d_step` – jest traktowana, jako pojedyncza instrukcja PROMELA i jest w całości wykonywana. Można powiedzieć, że `d_step` służy do definiowania prostych instrukcji. Prowadzi to, do narzucenia pewnych restrykcji na używanie sekwencji instrukcji `d_step`:

- Wykonywanie sekwencji `d_step` jest zawsze deterministyczne. Jeśli w sekwencji zostanie umieszczona instrukcja nie-deterministyczna, to wykonywane jest stałe rozwiązanie, np. w

przypadku instrukcji z strażnikiem, wykonywana jest pierwsza spełniona nie-deterministyczna instrukcja lub struktura powtarzalna.

- Występowanie instrukcji `goto` wyprowadzającej poza sekwencję `d_step` nie jest dozwolone, będzie sygnalizowane, jako błąd przez parser SPIN.
- Wykonywanie sekwencji rozpoczynającej się od `d_step`, nie może być przerwane prze instrukcję blokującą. Jest błędem, jeśli jakakolwiek instrukcja inna niż pierwsza (instrukcja straży) w sekwencji `d_step`, okaże się niewykonalną.

Żadna z wyżej wymienionych trzech restrykcji, nie dotyczy sekwencji atomowych. Oznacza to, że słowo kluczowe `d_step` może być zawsze zamienione słowem `atomic`, ale nie odwrotne. Bezpiecznym jest umieszczanie sekwencji `d_step` wewnątrz sekwencji atomowej, ale odwrotnie nie jest dopuszczalnym.

5.3.6.38. **Selekcja.** Używając względnych wartości dwu zmiennych `a` oraz `b` – możemy określić wybór pomiędzy wykonywaniem dwóch różnych opcji, z pomocą struktury selekcji, jak niżej:

```
if
:: (a != b) -> option1
:: (a == b) -> option2
fi
```

Powyższa struktura selekcji zawiera dwie sekwencje do wyboru. Jedna z sekwencji instrukcji z listy zawsze będzie wykonana. Sekwencja instrukcji będzie wybrana tylko wtedy, gdy pierwsza z instrukcji sekwencji poprzedzająca podwójne dwukropki zostanie wykonana. Pierwsza instrukcja sekwencji jest nazywana strażnikiem (*guard*) opcji sekwencji.

W ostatnim przykładzie strażnik jest wzajemnie ekskluzywny, co nie oznacza, że tak musi być zawsze. Jeśli więcej niż jeden strażnik jest wykonywalny, jedna z możliwych sekwencji jest nie-deterministycznie wybierana. Jeśli wszyscy wartownicy są niewykonalni, proces zostaje zablokowany aż któryś z nich stanie się wykonalny. Niema żadnych ograniczeń dotyczących typu instrukcji, która może być użyta jako strażnik: może być wysyłającą lub odbierającą, podstawieniem, `printf`, `skip`, itd. Rolę wykonalności określa w każdym przypadku – całość struktury selekcji. Poniższy przykład, ilustruje użycie instrukcji wysłania jako strażnika selekcji (kod 5.3.6.39):

```
mtype = { a, b };
chan ch = [1] of { mtype };
active proctype A() { ch?a }
active proctype B() { ch?b }
active proctype C()
{     if
      :: ch!a
      :: ch!b
      fi
}
```

5.3.6.39. Kod przykładu wyboru nie-determonistycznego

Powyższy przykład definiuje trzy procesy oraz jeden kanał. Pierwszą opcją struktury procesu `C` jest wykonanie na kanale `ch`, który nie jest zapełniony – warunku początkowego. Jeśli obie straży są wykonalne, proces `C` pobiera jeden z komunikatów i umieszcza go w kanale `ch`. Proces typu `A`, może wykonać swoją instrukcję, jeśli komunikatem wysłanym jest `a`, gdzie `a` jest

symboliczną stała definiowaną w deklaracji typu `mtype` na początku modelu. Jeśli równorzędny proces typu `B` może wykonać swoją samodzielną instrukcją umieszczając komunikat `b` w kanale. Jeśli przełączymy wszystkie instrukcje wysyłania na instrukcje odbierania i odwrotnie, otrzymamy również poprawny model PROMELA. W tym przypadku, wybór w `C` jest wymuszony przez komunikat, który został umieszczony w kanale, którego stan zależy od tego, czy wykonano proces `A` albo `B`. W obu wersjach modelu, jeden z trzech uruchomionych procesów jest zawieszony na koniec wykonywania, upadnie kończąc działanie.

Proces następującego typu albo zwiększy, albo zmniejszy wartość zmiennej `count`. Ponieważ podstawienia są zawsze wykonalne, wybór dokonany jest naprawdę nie-deterministyczny oraz niezależny (kod 5.3.6.40) od początkowej wartości zmiennej `count`.

```
byte count        /* initial value defoults to zero */
active proctype counter()
{
      if
      :: count++
      :: count--
      fi
}
```

5.3.6.40. Kod przykładu wyboru nie-deterministycznego

5.3.6.41. **Sekwencje opuszczane.** Ostatnim typem złożonej struktury, którą przedyskutujemy jest instrukcja `unless`. Ten raczej rzadko używany, wymaga jednak dodatkowych wyjaśnień.

Składnia sekwencji jest następująca:

```
{ P } unless { E }
```

gdzie litery `P` i `E` reprezentują pewne fragmenty kodu PROMELA. Wykonanie instrukcji `unless` rozpoczyna się z wykonywaniem instrukcji `P`. Zanim każda z instrukcji z `P` jest wykonywana, pierwsza instrukcja `E` jest sprawdzana, korzystając z normalnej składni PROMELA dotyczącej wykonalności. Wykonanie instrukcji z `P` ma miejsce, tylko wtedy, gdy pierwsza instrukcja z `E` pozostaje niewykonalną. Za pierwszym razem, gdy „straż opuszczanej sekwencji" stwierdzi wykonalność, wykonanie jest kontynuowane zgodnie z `E`. Wykonanie poszczególnych instrukcji pozostaje niewidoczne, natomiast sterowanie może być przekazane z wnętrza `P` na początek `E`, pomiędzy wykonywaniem poszczególnych instrukcji. Jeśli straż opuszczanej sekwencji nie stanie się wykonalną w toku wykonywania `P`, to ma miejsce przeskoczenie całości po zakończeniu `P`.

Przykładem użycia sekwencji opuszczanej jest (kod 5.3.6.42):

```
do
:: b1 -> B1
:: b2 -> B2
…
od unless { c -> C };
D
```

5.3.6.42. Kod przykładu sekwencji opuszczanej

Jak pokazano na przykładzie, nawiasy klamrowe zawierające główną sekwencję (lub sekwencję opuszczaną), mogą być pomijane, jeśli niema wątpliwości, co jest zawartością tych sekwencji. W powyższym przykładzie, warunek c działa jak pies łańcuchowy konstrukcji powtarzania w

głównej sekwencji. Zauważmy, że nie jest to koniecznie równoważne konstrukcji poniższej (kod 5.3.6.43):

```
do
:: b1 -> B1
:: b2 -> B2
…
:: c -> break
od; C; D
```

5.3.6.43. Kod przykładu sekwencji opuszczonej z pominięciem nawiasów

jeśli `B1` lub `B2` nie są puste. W pierwszej wersji przykładu, wykonywanie iteracji może zostać przerwane w każdym punkcie wewnątrz sekwencji opcji. W drugiej wersji, wykonanie może zostać przerwane tylko na początku sekwencji opcji.

Przykład zastosowania sekwencji opuszczania, pokazany jest prostym modelu systemu telefonii (*pots – plain old telephone service*). W tym systemie mamy do czynienia z dwoma procesami: *subskrybentem* i *serwerem pots*. Proces subskrybent działa według regulaminu. Po podniesieniu słuchawki, zawsze czeka na sygnał tonowy wybierania, a następnie wysyła numer, z którym chce zostać połączony po otrzymaniu sygnału tonowego. Następnie oczekuje na otrzymanie albo sygnału zajętości albo sygnału tonowego dzwonka. Po otrzymaniu sygnału zajętości, nasz wyidealizowany proces subskrybenta odwiesza słuchawkę i ewentualnie próbuje uzyskać połączenie ponownie. Po otrzymaniu sygnału dzwonka, albo czeka na sygnał odbioru połączenia, albo niecierpliwie odwiesza słuchawkę. Gdy następuje połączenie, proces serwer *pots* oczekuje na zakończenie połączenia, ale jeśli zbyt długo sygnał zachodzenia połączenia nie nadchodzi, *timeout* powoduje zakończenie połączenia.

Ten model subskrybenta zachowuje się w sposób standardowy. Musimy być jednak bardziej kreatywni modelując serwer *pots*. Proces serwera rozpoczyna działanie w swoim stanie bezczynności, czyli oczekując na subskrybenta, który podniesie słuchawkę i tym samym wyśle sygnał kanałem, przez który komunikuje się z serwerem. Serwer z kolei odpowie wysyłając sygnał ton wybierania i oczekuje na wybranie numeru. Po otrzymaniu numeru, proces serwera wybiera połączenie z odbiorcą, w wyniku wysyła subskrybentowi albo sygnał dzwonka, albo sygnał zajętości. Po tonie dzwonienia wysyłany jest sygnał ustanowienia połączenia, po czym proces serwer przechodzi w stan `zombi`, oczekując, na odwieszenie słuchawki – ewentualnie poprzedzając tę czynność wysłaniem komunikatu `hangup`. Zauważmy, że `skip` oraz instrukcja `goto zombi` – prowadzi do przejścia do tego samego stanu, w obu przypadkach (oznacza to, że instrukcja `goto`, jest tutaj nadmiarem).

Zauważmy, nie musieliśmy włączać żadnej dodatkowej reakcji w przypadku komunikatu `hangup` z procesu subskryptora w głównym strumieniu zachowań procesu *pots*.

```
mtype = { offhook, dialtone, numer, ringing,
            busy, connected, hangup, hungup };
chan line = [0] of { mtype, chan };
active proctype pots()
{     chan who;
idle: line?offhook,who;
      {     who!dialtone;
            who?number;
```

```
            if
            :: who!busy; goto zombie
            :: who!ringing ->
                  who!connected;
                  if
                  :: who!hungup; goto zombie
                  :: skip
                  fi
            if
      } unless
      {     if
            :: who?hangup -> goto idle
            :: timeout -> goto zombie
            fi
      }
zombie:     who?hangup; goto idle
}
active proctype subscriber()
{     chan me = [0] of { mtype };
idle: line!offhook,me;
      me?dialtone;
      me!number;
      if
      :: me?busy
      :: me?ring ->
            if
            :: me?connected;
                  if
                  :: me?hungup
                  :: timeout
                  fi
            :: skip
            fi
      fi;
      me!hangup; goto idle
}
```

5.3.6.44. Kod przykładu działania centrali telefonicznej

Powodem, jest fakt, że proces serwer pots – może być przerwany w każdej chwili i komunikat `hangup` może pojawić się w każdej chwili. Podobnie, jeśli proces serwera *pots* w dowolnym momencie zostanie zawieszony, musi pojawić się opcja `timeout`. Klauzula opuszczenia `unless` zawiera dwa warunki, które powodują aborcje głównego strumienia modelu, w każdej sytuacji. Po komunikacie `hangup`, serwer po prostu powraca do stanu `idle`, ponieważ wiadomo, że subskrybent odwiesił słuchawkę (jest ponownie w stanie `onhook`). Po komunikacie `timeout`, przechodzi do stanu `zombie` (kod 5.3.6.44).

Piśmiennictwo: *Dijkstra E.* D.2.1., *Holzmann G.* H.3.1, H.3.2.

5.3.7. DEFINIOWANIE TYPÓW KOMUNIKATÓW

Wiemy już jak definiować zmienne i stałe – używając makrodefinicji w stylu języka C. Język PROMELA pozwala również na definiowanie komunikatów, według zasady pokazanej niżej:

```
mtype = { ack, nak, err, next, accept }
```

Jest to preferowany sposób specyfikowania `message types`, ponieważ pozwala abstrahować od używania wartości, jednocześnie wprowadzając nazwy stałych wygodne do stosowania, ułatwiające równocześnie raportowanie błędów.

Użycie słów kluczowych typu `mtype` do deklaracji komunikatów kanałowych, zapewniamy również, że odpowiadające im komunikaty będą interpretowane symbolicznie, a nie numerycznie. Przykładowo:

```
chan q = [4] of { mtype, mtype, bit, short };
```

Piśmiennictwo: *Holzmann G.* H.3.1, H.3.2.

5.3.8. PSEUDOINSTRUKCJE

Dotychczas omówiliśmy podstawowe typy instrukcji języka PROMELA, a mianowicie: instrukcję podstawienia, instrukcje warunkowe, instrukcję wysyłanie komunikatów kanałem i pobieranie komunikatów z kanału, instrukcję `assert`, instrukcję `timeout`, instrukcję `goto`, instrukcję `break` oraz instrukcję `skip`. Zauważmy, że słowa kluczowe `chan`, `len` oraz `run` – nie są instrukcjami, ale unarnymi operatorami, które możemy użyć zarówno w warunkach jak podstawieniach.

Wzmiankowaliśmy wcześniej, że instrukcja `skip` jest bardzo przydatną dla zapewnienia zgodności syntaktycznych, chociaż nie wykonuje żadnego działania. W rzeczywistości, nie jest częścią języka, ale zaledwie *pseudoinstrukcją*, podobnie jak synonimem innej instrukcji dającej taki sam efekt: prosty warunek o stałej wartości `(1)`. W tym samym duchu, mogą być definiowane inne pseudoinstrukcje (ale nie są), takie jak `block` albo `hang`, są ekwiwalentem `(0)` albo `halt`, tak jak ekwiwalent `assert(0)`, ... Inną pseudoinstrukcją jest `else`, które to słowo kluczowe może być użyte, jako inicjująca instrukcja ostatniej wybieranej opcji sekwencji selekcji lub iteracji.

```
if
:: a > b -> ...
:: else -> ...
fi
```

Warto podkreślić, że pseudoinstrukcja `else` jest wykonywalna (`true`) tylko wtedy, kiedy wszystkie pozostałe opcje tej selekcji są niewykonywalne (`false`).

5.3.8.10. **Predefiniowane zmienne i funkcje**. Niektóre predefiniowane zmienne i funkcje mogą być szczególnie użyteczne w stwierdzeniu `trace` oraz żądaniu `never`. Są tylko cztery predefiniowane zmienne. A mianowicie:

```
_
np_
_pid
_last
```

Dwie zmienne `_pid` oraz `_` były omawiane wcześniej. Są one bez ograniczeń używane w deklaracjach `proctype`. Natomiast zmienne `np_` oraz `_last`, są używane jedynie wewnątrz stwierdzeń `trace` oraz żądania `never`.

5.3.8.20. **Predefiniowana zmienna `np_`** przechowuje wartość boolowską *false*, dla wszystkich stanów procesu, kiedy co najmniej jeden proces jest bieżąco w stanie *control-flow* oznaczonych etykietą `progress`. Przeto zmienna `np_` mówi, czy system jest bieżąco stanie `progress`, czy też `non-progress`. Możemy łatwo użyć tej zmiennej, żeby zbudować żądanie `never`, która może wykryć istnienie pętli `non-progress`, np. w następujący sposób (kod 5.3.8.21):

```
never { /* non-progress cycle detector */
            do
            :: true
            :: np_ -> break
            od;
accept:
            do
            :: np_
            od
}
```

5.3.8.21. Kod przykładu użycia zmiennej predefiniowanej

Po skończonym prefiksie o dowolnej długości, opcjonalnie przechodzimy przez pewną liczbę stanów `non-progress`, żądanie automatycznie przechodzi ze stanu inicjacji do stanu `final accepting`, w którym możemy pozostać jeśli istnieje co najmniej jeden nieskończenie długa sekwencja, która nigdy nie przechodzi w stan `progress`.
Prawdziwym celem użycia zmiennej `np_` nie polega na definicji żądania, kiedy odpowiednie żądanie jest użyte automatycznie przy wywołaniu przez SPIN domyślny wyszukiwanie pętli `non-progress`.

5.3.8.30. **Predefiniowana zmienna** `_last` przechowuje numer instancyjny procesu, który wykonał ostatnio krok sekwencji wykonywania systemu. Jego wartość nie jest częścią systemu dopóki eksplicite użyty w specyfikacji. Wartość początkowa tej zmiennej jest równa zero.

5.3.8.40. **Trzy predefiniowane funkcje**. Użycie następujących trzech predefiniowanych funkcji, jest ograniczone do żądania `never`:

```
pc_value(pid)
enabled(pid)
procname[pid]@label
```

Pierwsza z tych funkcji `pc_value(pid)` zwraca bieżący stan kontrolowanego procesu wraz z numerem `pid` lub zero jeśli taki proces nie istnieje. Zwracany numer jest zawsze nie-zerową liczbą całkowitą i odpowiada numerowi wewnętrznemu który SPIN nadaje jako licznik programu wykonywanego procesu.

W następującym przykładzie, jeden proces będzie drukował swój wewnętrzy numer stanu dla trzech kolejnych kroków, oraz drugi blok dopóki pierwszy proces osiągnie stan z numerem wyższym niż dwa (kod 5.3.8.41).

```
active proctype A()
{
      printf("%d\n", pc_value(_pid));
      printf("%d\n", pc_value(_pid));
      printf("%d\n", pc_value(_pid));
}
active proctype B()
{
```

```
        (pc_value(0) > 2);
        printf("ok\n")
}
```

5.3.8.41. Kod przykładu użycia zmiennej predefiniowanej z wewnętrznym numerem stanu

Ostatnia z predefiniowanych funkcji `procname[pid]@label` zwraca nie-zerową wartość tylko wtedy, gdy następna instrukcja która może być wykonywana jest proces z numerem instancyjnym `pid` a instrukcja jest oznaczona etykietą `label` w `proctype procname`. Jest błędem, jeśli proces odwołuję się do numeru `pid`, który nie jest instancją wymienionego procesu.

Następujący przykład pokazuje jak można użyć zdalnego odwołania wewnątrz żądania `never` (kod 5.3.8.42):

```
/*
 * Processes 1 and 2 cannot enter their
 * critical sections at the same time.
 */
never {
            do
            :: user[1]@critical && user[2]@critical ->
                break /* implicitly accepting */
            :: else /* repeat */
            od
}
```

5.3.8.42. Kod przykładu użyć zdalnego odwołania wewnątrz żądania `never`

Jeśli jest jedna instancja danego `proctype` w systemie, identyfikator tego procesu jest w rzeczywistości zbyteczny, oraz (w SPIN version 4.0 lub wyższej) referencja z ostatniego przykładu, może w tym przypadku jako `user@critical`. Jeśli natomiast, przejdziemy do więcej niż jednej instancji typu procesu, ten typ referencji wybierze arbitralnie jeden z nich. Symulator SPIN dostarczy ostrzeżenia w przypadku pojawienia się przypadku jak wyżej.

Piśmiennictwo: *Holzmann G.* H.3.1, H.3.2.

5.3.9. GRAMATYKA JĘZYKA PROMELA (SPIN V.6)

Poniżej przedstawiamy gramatykę języka PROMELA w wersji, z której korzysta procesor SPIN v.6. Konwencje notacji przedstawiamy, jak następuje:

- Wybory są rozdzielane pionowymi kreskami: `|`
- Części będące opcjami, są zawarte w nawiasach kwadratowych: `[ … ]`
- Gwiazdka wg notacji Kleene'go wskazuje zero lub więcej krotne powtórzenie bezpośrednio poprzedzającą dany fragment.
- Literały są zamknięte w pojedynczych cudzysłowach: `' … '`
- Nazwy pisane dużymi literami (*Uppercase*) dotyczą tokenów odnoszących się do słów kluczowych języka PROMELA. W modelach przetwarzanych przez procesor SPIN, te same słowa kluczowe są pisane małymi literami – zamiast dużymi.
- Pisane małymi literami nazwy odnoszą się do zasad gramatyki.

Nazwa *`any_ascii_char`* odnosi się dowolnego drukowanego znaku ze zbioru ASCII, za wyjątkiem podwójnego znaku cudzysłowu: `"`.

Słowo `separator` używane w niniejszym opisie reguł gramatyki, znaczy to samo, co symbol `';'`. W wielu miejscach znak `';'` (separator), może być zastąpiony parą symboli: `'->'`, bez zmiany znaczenia.

```
spec    : module [ module ] *

module  : proctype      /* proctype declaration */
        | init          /* init process       */
        | never         /* never claim        */
        | trace         /* event trace        */
        | utype         /* user defined types */
        | mtype         /* mtype declaration  */
        | decl_lst      /* global vars, chans */

proctype: [ active ] PROCTYPE name '(' [ decl_lst ]')'
          [ priority ] [ enabler ] '{' sequence '}'

init    : INIT [ priority ] '{' sequence '}'

never   : NEVER '{' sequence '}'

trace   : TRACE '{' sequence '}'

utype   : TYPEDEF name '{' decl_lst '}'

mtype   : MTYPE [ '=' ] '{' name [ ',' name ] * '}'

decl_lst: one_decl [ ';' one_decl ] *

one_decl: [ visible ] typename  ivar [',' ivar ] *

typename: BIT | BOOL | BYTE | SHORT | INT | MTYPE | CHAN
        | uname /* user defined type names (see utype) */

active  : ACTIVE [ '[' const ']' ]    /* instantiation */

priority: PRIORITY const       /* simulation priority */

enabler : PROVIDED '(' expr ')' /* execution constraint */

visible : HIDDEN | SHOW

sequence: step [ ';' step ] *

step    : stmnt [ UNLESS stmnt ]
        | decl_lst
        | XR varref [',' varref ] *
        | XS varref [',' varref ] *

ivar    : name [ '[' const ']' ] [ '=' any_expr | '=' ch_init ]

ch_init : '[' const ']' OF '{' typename [ ',' typename ] * '}'

varref  : name [ '[' any_expr ']' ] [ '.' varref ]

send    : varref '!' send_args        /* normal fifo send */
        | varref '!' '!' send_args    /* sorted send */

receive : varref '?' recv_args        /* normal receive */
        | varref '?' '?' recv_args    /* random receive */
```

```
        | varref '?' '<' recv_args '>' /* poll with side-effect */
        | varref '?' '?' '<' recv_args '>'    /* ditto */

poll    : varref '?' '[' recv_args ']'/* poll without side-effect */
        | varref '?' '?' '[' recv_args ']'    /* ditto */

send_args: arg_lst | any_expr '(' arg_lst ')'

arg_lst  : any_expr [ ',' any_expr ] *

recv_args: recv_arg [ ',' recv_arg ] *  |  recv_arg '(' recv_args ')'

recv_arg : varref | EVAL '(' varref ')' | [ '-' ] const

assign  : varref '=' any_expr  /* standard assignment */
        | varref '+' '+'       /* increment */
        | varref '-' '-'       /* decrement */

stmnt[93] : IF options FI        /* selection */
        | DO options OD        /* iteration */
        | FOR '(' range ')' '{' sequence '}'          /* iteration */
        | ATOMIC '{' sequence '}'      /* atomic sequence */
        | D_STEP '{' sequence '}'      /* deterministic atomic */
        | SELECT '(' range ')' /* non-deterministic value selection */
        | '{' sequence '}'     /* normal sequence */
        | send
        | receive
        | assign
        | ELSE                 /* used inside options */
        | BREAK                /* used inside iterations */
        | GOTO name
        | name ':' stmnt       /* labeled statement */
        | PRINT '(' string [ ',' arg_lst ] ')'
        | ASSERT expr
        | expr                 /* condition */
        | c_code '{' ... '}'   /* embedded C code */
        | c_expr '{' ... '}'
        | c_decl '{' ... '}'
        | c_track '{' ... '}'
        | c_state '{' ... '}'

range   : varref ':' expr '..' expr
        | varref IN varref

options : ':' ':' sequence [ ':' ':' sequence ] *

andor   : '&' '&' | '|' '|'

binarop : '+' | '-' | '*' | '/' | '%' | '&' | '^' | '|'
        | '>' | '<' | '>' '=' | '<' '=' | '=' '=' | '!' '='
        | '<' '<' | '>' '>' | andor

unarop  : '~' | '-' | '!'

any_expr: '(' any_expr ')'
        | any_expr binarop any_expr
        | unarop any_expr
        | '(' any_expr '-' '>' any_expr ':' any_expr ')'
        | LEN '(' varref ')'   /* nr of messages in chan */
```

[93] Słowo kluczowe `stmnt` – jest skrótem słowa *statement*, które tłumaczymy, jako instrukcja języka PROMELA.

```
        | poll
        | varref
        | const
        | TIMEOUT
        | NP_                  /* non-progress system state */
        | ENABLED '(' any_expr ')'            /* refers to a pid */
        | PC_VALUE '(' any_expr ')'           /* refers to a pid */
        | name '[' any_expr ']' '@' name      /* refers to a pid */
        | RUN name '(' [ arg_lst ] ')' [ priority ]
        | get_priority( expr )         /* expr refers to a pid */
        | set_priority( expr , expr )  /* first expr refers to a pid */

expr    : any_expr
        | '(' expr ')'
        | expr andor expr
        | chanpoll '(' varref ')'      /* may not be negated */

chanpoll: FULL | EMPTY | NFULL | NEMPTY
string  : '"' [ any_ascii_char ] * '"'
uname   : name
name    : alpha [ alpha | number ] *
const   : TRUE | FALSE | SKIP | number [ number ] *

alpha   : 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j'
        | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't'
        | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'
        | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J'
        | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T'
        | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'
        | '_'

number  : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Piśmiennictwo: *Holzmann G.* H.3.1, H.3.2.

5.4. PROCESSOR SPIN (SPIN CHECKER)

5.4.0. UWAGI WSTĘPNE

Jeśli mamy dany model system opisany w języku PROMELA, to procesor SPIN może albo wykonać losową symulację działania modelu lub może generować program w języku C, który z kolei może wykonać szczegółową weryfikację przestrzeni stanów systemu. Taka weryfikacja może przykładowo sprawdzić czy wartości niezmiennicze (*invariants*) nie zostały naruszone w toku wykonywania modelu.

Jeśli procesor Spin zostanie wywołany bez żadnych opcji to wykona on losową symulację działania modelowanego systemu. Natomiast z opcją `-nN`, gdzie `N` krotność kroków symulacji, wykona dokładnie `N` – krokową symulację.

Grupa opcji „`p g l r s`” może być użyta, do ustawienia pożądanego poziomu informacji, które użytkownik zamierza uzyskać w wyniku wykonanych kroków przebiegu symulacji. Każda z linii wynikowych zawiera odniesienie do numeru linii specyfikacji modelu.
„`-p`” służy do pokazania zmian stanu poszczególnych procesów PROMELA w każdym kroku;
„`-g`” pokazuje zmiany stanu zmiennych globalnych po wykonaniu każdego kroku;
„`-l`” pokazuje zmiany stanu zmiennych lokalnych po wykonaniu każdego kroku, najlepiej korzystać łącznie z opcją `-p`;

„-r” pokazuje wszystkie zdarzenia otrzymania komunikatu. Pokazuje jak wykonywane procesy przyjmują komunikaty, ich nazwy i numery, numery linii źródłowych, numery parametrów komunikatów (po jednej linii na każdy z parametrów), typ komunikatu oraz numer kanału i nazwę;
„-s” pokazuje zdarzenia związane z wysyłaniem komunikatów;

SPIN rozumie trzy dalsze opcje „-a, -m, -t”:
„-a” powoduje generowanie specyfikacji dla analizy. Wynik tworzy zbiór plików C, nazwanych pan.[cbhmt], które po kompilacji tworzą analizator (wykonanie którego dostarcza wyniki analizy). Gwarantując wyczerpującą eksploracje pamięci stanów, program można skompilować po prostu jako:

```
$ gcc -o pan pan.c
```

w przypadku większego systemu, należy użyć odpowiednio dużej pojemności pamięci. Duże i bardzo duże systemy mogą być analizowane, używając metody efektywnego zapisu bitowego stanów:

```
$ gcc -DBITSTATE -o pan pan.c
```

Wskaźnik posługiwania się taką metodą wyszukiwania oparty jest o tzw. *hash factor*. Uogólniony analizator, nazwano run, ma swój własny zbiór opcji, który można oglądać po wprowadzaniu sekwencji „./pan -?” (zobacz dalej 5.5.2. *Używanie analizatora*).

„-m” opcja ta może być użyta do zmiany domyślnej semantyki czynności wysyłania komunikatu. Normalnie, operacja wysyłania komunikatu jest tylko wtedy wykonywana, jeśli docelowy kanał nie jest pełen. Nakazuje to bezpośrednią synchronizację, która nie zawsze jest uzasadniona. Opcja -m powoduje, że operacja wysłania komunikatu jest zawsze wykonywalna, a komunikat wysłany do pełnego kanału - zostaje pominięty. Jeśli ta opcja jest połączona z opcją -a, semantyka wysyłania generowana dla analizatora jest podobna, zaś efektem weryfikacji będzie wzięcie pod uwagę utracony komunikat.

„-t” opcja - jest nazywana ciągnięciem śladu. Jeśli analizator znajdzie naruszenie poprawności, zakleszczenie albo nieprzewidywane podstawienie (zwane dalej *naruszeniem twierdzenia*), zapisuje ślad błędu do tworzonego pliku o nazwie pan.trail. Ten ślad błędu, będzie następnie szczegółowo sprawdzony prze wywołanie procesora SPIN z opcją -t. W połączeniu z opcjami p g l r s różne prezentacje sekwencji błędu z łatwością są do uzyskania.

Ze względu na skrótowy charakter niniejszego opisu, pozostałe opcje procesora SPIN nie są omawiane.

Piśmiennictwo: *Holzmann G.* H.3.1, H.3.2.

5.4.1. SYMULATOR

Weźmy z kolei pod uwagę następujący przykład protokołu przebiegu, który jest przechowywany w pliku komputerowy o nazwie lynch (kod 5.4.1.00). Poniższy protokół korzysta z trzech typów komunikatów: ack, nak, oraz specjalnego komunikatu typu err – używanego do modelowania zafałszowań w kanale łączącym dwa różne procesy. Zachowania kanału jest modelowane przez proces kanału. Istnieje również instrukcja stwierdzenia wystąpienia usterek w zachowaniu niezmienniczych relacji - pomiędzy dwiema lokalnymi zmiennymi w procesie transferu.

```
 1  #define MIN      9
 2  #define MAX      12
 3  #define FILL     99
 4
 5  mtype = { ack, nak, err }
 6
 7  proctype transfer(chan chin, chout)
 8  { byte o, i, last_i=MIN;
 9
10    o = MIN+1;
11    do
12    :: chin?nak(i) ->
13            assert(i == last_i+1);
14            chout!ack(o)
15    :: chin?ack(i) ->
16            if
17            :: (o <  MAX) -> o = o+1
18            :: (o >= MAX) -> o = FILL
19            fi;
20            chout!ack(o)
21    :: chin?err(i) ->
22            chout!nak(o)
23    od
24  }
25
26  proctype channel(chan in, out)
27  { byte md, mt;
28    do
29    :: in?mt,md ->
30            if
31            :: out!mt,md
32            :: out!err,0
33            fi
34    od
35  }
36
37  init
38  { chan AtoB = [1] of { mtype, byte };
39    chan BtoC = [1] of { mtype, byte };
40    chan CtoA = [1] of { mtype, byte };
41    atomic {
42            run transfer(AtoB, BtoC);
43            run channel(BtoC, CtoA);
44            run transfer(CtoA, AtoB)
45    };
46    AtoB!err,0;     /* start */
47    0               /* hang */
48  }
```

5.4.1.00. Kod przykładu protokołu przebiegu SPIN

Działanie procesora SPIN bez użycia opcji, prowadzi do losowej symulacji dostarczającej raport wynikowy tylko wtedy, kiedy wykonanie zostanie zakończone lub jeśli instrukcja `printf` – zostanie napotkana.

W takim przypadku (raport 5.4.1.01):

```
$ spin lynch   # no options, not recommended
spin: "lynch" line 13: assertion violated
#processes: 4
proc  3 (transfer)     line 11 (state 15)
```

```
proc  2 (channel)      line 28 (state 6)
proc  1 (transfer)     line 13 (state 3)
proc  0 (:init:)       line 48 (state 6)
4 processes created
$
```

5.4.1.01. Raport przykładu losowej symulacji

W specyfikacji powyższego modelu niema instrukcji `printf`, otrzymaliśmy jednak raport wyniku, ponieważ nastąpiło *naruszenie twierdzenia*. Jeśli chcemy znać więcej szczegółów, musimy powtórzyć przebieg z bardziej rozbudowanym raportem wyniku, np. obejmującym wszystkie zdarzenia, jakie wystąpiły w czasie przebiegu. Wynik takiegoż przebiegu o rozszerzonej dokumentacji jest pokazany na kod 5.5.1.00. Większość pozycji wyjścia jest samo-dokumentowana.

Pokazany niżej przebieg symulacji, również zakończył się *naruszenie twierdzenia*. Odkąd przebiegi symulacji dokonują niedeterministycznych wyborów, w trybie losowym – nie zawsze może mieć miejsce taki przypadek zakończenia. Dla ograniczenia powtarzalności kroków przebiegu, należy użyć opcji `-nN`. Przykładowo:

```
$ spin -r -n100 lynch
```

ustawi generator liczb losowych na całkowitą krotność generacji liczb równą 100, co z kolei zagwarantuje, że wykonanych zostanie dokładnie 100 kroków przebiegu.

Inne dostępne opcje procesora SPIN mogą istotnie powiększyć uzyskane wyniki z przebiegu symulacji, ale mogą również spowodować gwałtowne rozrośnięcie się tekstu raportu wynikowego. Jednym z prostych rozwiązań jest filtrowanie tekstu wynikowego z pomocą programu *grep.* Przykładowo, jeśli w omawianym wyżej przykładzie - jesteśmy zainteresowani jedynie zachowywaniem się procesu w kanale, to możemy użyć programu *grep*, w następujący sposób:

```
$ spin -n100 -r lynch | grep "proc  2"
```

Raport wyniku działania przebiegu symulacji, pokazano na 5.5.1.02.

```
$ spin -r lynch
proc  1 (transfer) line  21, Recv err,0  <- queue 1 (chin)
proc  2 (channel)  line  29, Recv nak,10 <- queue 2 (in)
proc  3 (transfer) line  12, Recv nak,10 <- queue 3 (chin)
proc  1 (transfer) line  15, Recv ack,10 <- queue 1 (chin)
...
proc  1 (transfer) line  15, Recv ack,12 <- queue 1 (chin)
proc  2 (channel)  line  29, Recv ack,99 <- queue 2 (in)
proc  3 (transfer) line  15, Recv ack,99 <- queue 3 (chin)
proc  1 (transfer) line  15, Recv ack,99 <- queue 1 (chin)
proc  2 (channel)  line  29, Recv ack,99 <- queue 2 (in)
proc  3 (transfer) line  21, Recv err,0  <- queue 3 (chin)
proc  1 (transfer) line  12, Recv nak,99 <- queue 1 (chin)
spin: "lynch" line 13: assertion violated
#processes: 4
proc  3 (transfer) line 11 (state 15)
proc  2 (channel)  line 28 (state 6)
proc  1 (transfer) line 13 (state 3)
proc  0 (:init:)   line 48 (state 6)
4 processes created
$ spin -n100 -r lynch | grep "proc  2"
proc  2 (channel) line 29, Recv nak,10 <- queue 2 (in)
proc  2 (channel) line 29, Recv ack,11 <- queue 2 (in)
```

```
proc  2 (channel) line 29, Recv ack,12 <- queue 2 (in)
proc  2 (channel) line 28 (state 6)
```

5.4.1.02. Raport wyniku przebiegu symulacji z opcji `-r` (wszystkie zdarzenia)

Interesujący start przebiegu symulacji, uzyskamy korzystając z opcji –c (nowa opcja od wersji 3.0), która dostarcza następujący wynik:

```
$ spin -c lynch
proc 0 = :init:
proc 1 = transfer
proc 2 = channel
proc 3 = transfer
q\p   0   1   2   3
  1   AtoB!err,0
  1   .   chin?err,0
  2   .   chout!nak,10
  2   .   .   in?nak,10
  3   .   .   out!err,0
  3   .   .   .   chin?err,0
  1   .   .   .   chout!nak,10
  1   .   chin?nak,10
  2   .   chout!ack,10
  2   .   .   in?ack,10
  3   .   .   out!ack,10
  3   .   .   .   chin?ack,10
  1   .   .   .   chout!ack,11
  1   .   chin?ack,11
  2   .   chout!ack,11
  2   .   .   in?ack,11
  3   .   .   out!ack,11
  3   .   .   .   chin?ack,11
  1   .   .   .   chout!ack,12
  1   .   chin?ack,12
  2   .   chout!ack,12
  2   .   .   in?ack,12
  3   .   .   out!ack,12
  3   .   .   .   chin?ack,12
  1   .   .   .   chout!ack,99
  1   .   chin?ack,99
  2   .   chout!ack,99
  2   .   .   in?ack,99
  3   .   .   out!err,0
  3   .   .   .   chin?err,0
  1   .   .   .   chout!nak,99
  1   .   chin?nak,99
spin: line  13 "lynch", Error: assertion violated
-------------
final state:
-------------
#processes: 4
 79:    proc  3 (transfer) line  11 "lynch" (state 15)
 79:    proc  2 (channel) line  28 "lynch" (state 6)
 79:    proc  1 (transfer) line  13 "lynch" (state 3)
 79:    proc  0 (:init:) line  47 "lynch" (state 6)
4 processes created
```

5.4.1.03 . Raport wynikowy symulacji z opcją –c

Pierwsza kolumna raportu 5.4.1.03 wynikowego podaje numer identyfikacyjny kanału, pierwszy wiersz podaje natomiast id - numery identyfikacyjne procesu uczestniczącego w przekazywaniu komunikatu wymieniony w pozostałej części tablicy.

Piśmiennictwo: *Holzmann G.* H.3.1, H.3.2.

5.4.2. ANALIZATOR

Przebiegi symulacyjne są bardzo przydatne w szybkim wyszukiwanie błędów (*debugging*). Jednak jedynie z pomocą symulacji, nie możemy dowieść, że badany *model systemu* nie posiada błędów. Walidacja nawet największych modeli - powinna być wykonywana z włączonymi opcjami procesora SPIN: `-a` oraz `-t`.

Bardzo szczegółowe badanie przestrzeni stanów walidowanego modelu systemu, odbywa się na podstawie zawartości pięciu plików, nazywanych odpowiednio `pan.[b c h m t]`(raport 5.4.2.10).

```
$ spin -a lynch
$ wc pan.[bchmt]
     99     285    1893 pan.b
   3158   10208   70337 pan.c
    356    1238    7786 pan.h
    216     903    6045 pan.m
    575    2099   14017 pan.t
   4404   14733  100078 total
```

5.4.2.10. Raport przebiegu badania przestrzeni stanów modelu systemu

Szczegóły dotyczące zawartości powyższych plików, nie są zbyt interesujące: plik `pan.c` zawiera głównie kod w języku C, służący do analizy protokołu przebiegu symulacji; plik `pan.t` zawiera macierz przejścia kodów końcowych protokołu przepływu sterowania; pliki `pan.b` oraz `pan.m` zawierają kod w języku C przejść do przodu i do tyłu przez przestrzeń stanów; zaś plik `pan.h` jest plikiem nagłówków. Tak opisany przez wzmiankowane pliki – program o nazwie `pan`, można kompilować na szereg sposobów, np. z pełną przestrzenią stanów albo z bitową przestrzenią stanów.

Piśmiennictwo: *Holzmann G.* H.3.1, H.3.2.

5.4.3. PRZESZUKANIE PRZESTRZENI STANÓW Z WYCZERPANIEM

Najkorzystniejszą metodą, umożliwiającą pracę z przestrzenią stanów modelu systemu, z grubsza szacowanej na 100.000 stanów, jest zastosowanie kompilacji programu `pan` z domyślnymi parametrami (patrz 5.5.2):

```
$ gcc -o pan pan.c
```

Uzyskany w wyniku kompilacji program `pan`, może wykonać walidację badanego modelu systemu. Walidacja jest całkowicie wyczerpująca, ponieważ testuje wszystkie możliwe sekwencje zdarzeń we wszystkich możliwych kolejnościach wystąpienia. Można więc otrzymać, wszelkie wcześniej już zauważone naruszenia twierdzenia (patrz raport 5.4.3.00).

```
$ ./pan
assertion violated (i == last_i + 1))
pan: aborted
pan: wrote pan.trail
search interrupted
vector 64 byte, depth reached 56
      61 states, stored
       5 states, linked
       1 states, matched
hash conflicts: 0 (resolved)
(size 2^18 states, stack frames: 0/5)
```

5.4.3.00. Raport przebiegu stwierdzający naruszenie poprawności (parametry domyślne)

(Należy podkreślić, że wynik uzyskany z pomocą ostatnich wersji procesora Spin, zawiera więcej szczegółów, które jednak łącznie przekazuje te same informacje.)

Pierwsza linia wyniku informuję, że miało miejsce naruszenie twierdzenia i próbuje dostarczyć wskazówkę o niezmienniku, który został zmieniony. Szczegóły naruszenia zostały znalezione po zbadaniu 61 stanów. Słowa „`hash conflicts`" określają liczbę wykrytych kolizji, w toku dostępu do przestrzeni stanów. Jak pokazuje raport wyniku, nie wystąpił żaden konflikt w toku kolejnych dostępów do pamięci stanów modelu systemu. Najważniejszą częścią raportu wyniku w tym przypadku, jest trzecia linia, która mówi, że plik `pan.trail` został utworzony i może zostać użyty w połączeniu z symulacją – do odtworzenia sekwencji powstawania błędu. Przykładowo, możemy użyć poniższą konfigurację procesor SPIN, aby wyznaczyć przyczynę wystąpienia błędu.

```
$ spin -t -r lynch | grep "proc  2"
```

Należy jednak podkreślić, że wprawdzie weryfikator gwarantuje znalezienie naruszenia stwierdzenia, ale tylko wtedy, kiedy jakieś naruszenie wystąpi. Jeśli wyczerpujące przeszukanie przestrzeni stanów nie stwierdzi takowego naruszenia, to pewnym jest, że żadne wykonywanie przebiegu wyznaczania przyczyny, nie stwierdzi wystąpienia przyczyny.

Piśmiennictwo: *Holzmann G.* H.3.1, H.3.2.

5.4.4. OPCJE ANALIZATORA

Wygenerowany w wyniku przebiegu symulacji wykonywalny analizator, posiada skromną liczbę opcji, które mogą być użyte, jak niżej (raport 5.4.4.00):

```
$ ./pan --
-cN stop at Nth error (default=1)
-l  find non-progress cycles   # when compiled with -DNP
-a  find acceptance cycles     # when not compiled with -DNP
-mN max depth N (default=10k)
-wN hash table of 2^N entries (default=18)
 ...etc.
```

5.4.4.00. Raport przebiegu (fragment) z opcjami analizatora

Używając zera jako argumentu pierwszej z opcji, wymuszamy przeszukiwanie pamięci stanów, w sytuacja znalezienia błędów w przebiegu analizy. Przegląd niewykonalnych (nieosiągalnych) kodów, jest podawany w raporcie każdego wykonanego przebiegu analizy: zarówno w przypadku przebiegu z domyślnym ustawieniem parametrów, jeśli nie zostanie znaleziony żaden błąd, albo w przebiegu wykonanym z opcją `-c0`. W tym przypadku raport wyniku przebiegu ma postać (raport 5.4.4.01):

```
$ ./pan -c0
assertion violated (i == (last_i + 1))

vector 64 byte, depth reached 60, errors: 5
     165 states, stored
       5 states, linked
      26 states, matched
hash conflicts: 1 (resolved)
(size 2^18 states, stack frames: 0/6)

unreached code :init: (proc 0):
        reached all 9 states
unreached code channel (proc 1):
         line 35 (state 9),
        reached: 8 of 9 states
unreached code transfer (proc 2):
```

```
 line 24 (state 18),
reached: 17 of 18 states
```

5.4.4.01. Raport przebiegu przeszukania przestrzeni stanów

Miało miejsce pięć *naruszeń twierdzenia*, oraz 165 unikalnych stanów systemu zostało wygenerowanych. Każdy opis stanu (`vector`) zajmuje 64 bajty pamięci; najdłuższa bez powtórzeń wykonywana sekwencja wyniosła 60. Wystąpiły nieosiągalne stany zarówno działaniu kanału, jak i procesach przesyłania. W obu przypadkach nieosiągalne stany w punktach przepływu sterowania, mają miejsce w wykonywaniu pętli w każdym z procesów. Zauważmy, że oznacza to, że obie pętle są nieskończonej krotności wykonywania.

Opcja `-l` spowoduje, że analizator wyszukuje raczej pętle bez-postępu (`non-progress`), niż zakleszczenia (`deadlocks`) lub *naruszeń twierdzenia* (`assertion violations`). Objaśnienia działania opcji jest szczegółowo omówione w dokumentacji eksploatacyjnej procesor SPIN.

Wykonywalny analizator ma jeszcze dwie inne opcje. Domyślnie głębokość wyszukiwania jest ograniczona do przyjętych raczej arbitralnie 10.000 kroków. Jeśli głębokość limitu zostanie osiągnięta, przeszukanie jest kończone, powodując, że weryfikacja nie jest wyczerpująca. Dla upewnienia się czy też nie, że wyszukiwanie „osiągnęło dno" przy maksymalnie głębokim wyszukiwaniu, można powtarzać przebieg analizy, z opcja `-m`. Opcja ta, oczywiści może być użyta do jawnego obcięcia głębokości przeszukania, gdy chcemy znaleźć możliwie najkrótszą z możliwych sekwencji stwierdzającej wystąpienie naruszenia wskazanego twierdzenia. Takie obcinanie przeszukań, jak wiadomo, nie gwarantuje znalezienia wszelkich możliwych *naruszeń twierdzeń*, nawet dla danej głębokości przeszukania.

Ostatnia z opcji `-wN`, ma wpływ jedynie na czas trwania przebiegu, a nie na jego zakres, analizy całej przestrzeni stanów. Parametr `hash table width` – powinien być ustawiony na liczbę równą lub jeszcze lepiej na większą niż, wartość logarytmu spodziewanej liczby stanów generowanej przez analizator. Jeśli parametr ten będzie miał wartość zbyt małą, liczba kolizji (`hash collisions`) wzrośnie i spowolni przeszukiwania. Wartość domyślna `N = 18`, pozwala operować liczbą stanów nie większą niż `262.144`, co jest liczbą wystarczającą dla większości aplikacji dokonujących analizę całej pamięci stanów.

Piśmiennictwo: *Holzmann G.* H.3.1, H.3.2.

5.4.5. ANALIZA BITOWEJ PRZESTRZENI STANÓW

Stosunkowo łatwo można oszacować zapotrzebowanie na pojemość pamięci – niezbędną do analizowania zawartości całej pamięci stanów[94]. Przykładowo, jeśli badany model 64 bitów pamięci na zakodowanie jednego stanu z przestrzeni stanów modelu, a mamy dostępnej pamięci 2MB dla przeszukania, możemy zapamiętać aż do 32.768 różnych stanów modelu. Jeśli okaże się, że model posiada większą liczbę osiągalnych stanów niż to wynika z dostępnej pamięci, to analiza załamie się. Dlatego też, autorzy procesora SPIN - wprowadzili unikalne rozwiązanie, pozwalające uniknąć takiej pułapki. Warto zauważyć, że wszystkie inne, dotychczas opracowane, programy automatycznej weryfikacji – padają, gdy przekroczą dostępną pojemność pamięci, nie dostarczając informacji użytkownikowi.

[94] Holzmann, G.J., "Algorithms for automated protocol verification." *AT&T Technical Journal* 69, 1 (Jan/Feb 1990). Special Issue on Protocol Testing, Specification, Verification.

SPIN dostarcza znacznie doskonalszego rozwiązania, wprowadzając tzw. „*bitową przestrzeń stanów*". Użycie bitowej przestrzeni stanów można włączyć do kompilacji i analizy, w sposób następujący:

```
$ gcc -DBITSTATE -o pan pan.c
```

Analizator w wyniku kompilacji ponownie (również) znajdzie naruszenie poprawności, co widać z poniższego raportu 5.4.5.00:

```
$ ./pan
assertion violated (i == ((last_i + 1))
pan: aborted
pan: wrote pan.trail
search interrupted
vector 64 byte, depth reached 56
      61 states, stored
       5 states, linked
       1 states, matched
hash factor: 67650.064516
(size 2^22 states, stack frames: 0/5)
$
```

5.4.5.00. Raport stwierdzający naruszenie poprawności

W rzeczywistości, dla małych lub średniej wielkości problemów, ma miejsce niewielka różnica czasu wykonywania analizy - pomiędzy metodami posługiwania się pełnym opisem stanu, a bitowym opisem stanu (z wyjątkiem tego, że pierwsza z metod jest trochę szybsza, ale wymaga więcej pamięci niż druga z metod). Istotna różnica ma miejsce dopiero przy bardzo dużych i wielkich problemach. Dwie ostatnie linie raportu zawierają bardzo przydatne informacje dotyczące oszacowania bardzo dużych przebiegów. Maksymalna liczba stanów, które bitowa przestrzeń stanów może obsłużyć jest zapisana w ostatniej linii raportu (w tym przypadku 2^{22} bitów, czyli około 32 milionów bitów, gdzie jeden bit opisuje 1 stan przestrzeni stanów. Linia powyżej – określa tzw. `hash factor`, który z grubsza przybliża maksymalną liczbę stanów dzieloną przez aktualną liczbę stanów. Jeśli ten współczynnik jest większy niż 100, oznacza z praktyczną pewnością, pokrycie 99% do 100% pojemnością pamięci zapotrzebowania przestrzeni stanów. Jeśli ten współczynnik jest bliski 1, to pokrywa 0% zapotrzebowania.

Celem użycia weryfikacji opartej o bitową przestrzeń stanów, jest uzyskanie współczynnika „*hash factor*" powyżej 100, na drodze alokacji maksymalnej części pamięci operacyjnej z przeznaczeniem na bitową przestrzeń stanów. Dla uzyskania najlepszego wyniku: należy użyć opcji `-wN` o wartości odpowiadającej faktycznie ilości fizycznej (nie wirtualnej) pamięci dostępnej w danym komputerze. Domyślna wartość `N` – to 22, co odpowiada alokacji na bitową przestrzeń stanów 4MB. Przykładowo, jeśli komputer ma 128MB fizycznej pamięci operacyjnej, ustawiając opcję –w27, uzyskujemy możliwość analizowania modelu, którego osiągalna przestrzeń stanów posiada do miliarda różnych stanów.

Piśmiennictwo: *Holzmann G.* H.3.1, H.3.2.

5.4.5. *DEFINIOWANIE ŻĄDAŃ POPRAWNOŚCI*

Celem systemu weryfikacji jest określenie, co jest możliwe – a co nie. Często, takie stwierdzenie, co jest logicznie możliwe będzie przedmiotem pewnego zbioru rozważań - na temat kontekstu wykonywania systemu, takich jak możliwe zachowania zewnętrznych składników, z którymi system wzajemnie oddziałuje. Wykonując logiczną weryfikację, jesteśmy szczególnie zainteresowani w określeniu, kiedy (w jakich warunkach) pojawiają się zdarzenia naruszające wymagania projektu, które mogłyby taką sytuację spowodować. Natomiast niekoniecznie

badamy, z jakim prawdopodobieństwem jest to możliwe, a z jakim prawdopodobieństwem naruszenie wymagań nie jest możliwe. Dramatyczne awarie systemu są niemal zawsze wynikiem pozornie niemożliwych sekwencji zdarzeń: to dokładnie tak, jakby pewne sekwencje zdarzeń zostały pomijane w rozważaniach - fazy projektowania. Kiedy jednak zrozumiemy jak oryginalne założenia projektowe mogą zostać naruszone, należy wprowadzić niezbędne w nich zmiany, aby zapobiegać błędom. Poprawność logiczna, jest w pierwszej kolejności związana z możliwością, a nie z prawdopodobieństwem.

5.4.5.11. **Mocniejszy dowód**. To ograniczenie do możliwości, a nie prawdopodobieństwa, ma dwie konsekwencje. Po pierwsze, należy podkreślić, że dowód poprawności otrzymujemy jako wynik walidacji systemu. Jeśli weryfikator powie nam, że niema możliwości naruszenia zadanych wymagań, to brzmi istotnie mocniej – niż werdykt, że naruszenie poprawności ma niskie prawdopodobieństwo wystąpienia. Po drugie, mówienie o możliwości wykonania weryfikacji bardziej efektywnie, jeśli uwzględniamy w rozważaniach prawdopodobieństwo wystąpień poszczególnych scenariuszy, jest mało wirogodne. Wyniki oceny prawdopodobieństw są przecież niepewne, między innymi przez trudności przypisania jakiejś rozsądnej metryki prawdopodobieństwom wystąpienia poszczególnych zdarzeń. Błąd ocen może istotnie ograniczyć wartość uzyskanego wyniku.

Powinniśmy być w stanie wyjaśnić istotę logiki właściwości poprawności - rozproszonego systemu, niezależnie od każdego założenia dotyczącego relatywnej szybkości wykonywania procesów, gdzie czas jest rozumiany, jako szybkość wykonywania poszczególnych instrukcji, lub prawdopodobieństwo wystąpienia poszczególnych typów zdarzeń, takich jak strata komunikatu w kanale transmisji lub błąd urządzenia zewnętrznego. Dowód poprawności algorytmu jest również niezależny od implementacji. W szczególności, poprawność algorytmu – również nie zależy od tego, na jak szybkim komputerze jest implementowany. Dlatego też w procesie weryfikacji nie możemy przyjmować takich założeń. Nie można się więc dziwić, że PROMELA efektywnie uniemożliwia badania poprawności ze względu na założenia dotyczące implementacji.

Przedstawione dalej zasady, są specyficzne dla obszaru naszych zainteresowań, czyli dla walidacji oprogramowania systemów rozproszonych. Inne zasady dotyczą np. weryfikacji poprawności działania urządzeń sprzętowych. Poprawność działania układu scalonego może krytycznie zależeć od opóźnienia propagacji sygnału oraz szybkości działania poszczególnych elementów obwodu. Czas propagacji sygnału oraz rozmieszczenie elementów obwodu, są częścią funkcjonalności i projektu obwodu scalonego, nie może być niezależnie zmieniana. Poprawność protokołu komunikacyjnego lub rozproszonego systemu operacyjnego, z drugiej strony, nigdy nie może zależeć od takich czynników. Szybkość wykonywania systemu oprogramowania, na pewno będzie się zmieniać w sposób zasadniczy – w ciągu cyklu życia danego projektu.

5.4.5.12. **Podstawowe typy żądań - poprawności**. Przy projektowaniu systemów rozproszonych, jest standardem rozróżnianie pomiędzy dwoma typami wymagań poprawności: bezpieczeństwo oraz żywotność. Bezpieczeństwo jest zazwyczaj określane, jako zbiór własności, których system nie może naruszyć, natomiast żywotność jest określana, jako zbiór własności, które system musi spełniać. Przeto bezpieczeństwo określa złe sytuacje, których system powinien unikać, zaś żywotność określa dobre sytuacje, które realizuje funkcjonalność systemu. Funkcja walidacji systemu, jak wiadomo nie jest zbyt wymagająca. Nie potrzebuje i rzeczywiście

nie może stwierdzić, co jest dobre a co złe; może jedynie pomóc projektantowi stwierdzić, co jest *możliwe*, a co nie jest możliwe.

Z punktu widzenia weryfikatora, mają miejsce dwa typy żądań poprawności: żądanie dotyczące osiągania albo nie osiągania pewnych stanów oraz żądanie dotyczące możliwości albo niemożliwości wykonania (np. określonej sekwencji stanów). Tę pierwszą czasami nazywamy *własności stanu* (*state properties*), tę drugą *własnościami ścieżki* (*path properties*). Ścieżka jak i wykonanie, może być skończona albo nieskończona (np. pętla).

Prostym typem własności stanu jest *niezmienniczość* (*invariant*) systemu, która zapewnia trzymanie się osiągalnych stanów składających się na ścieżkę. Trochę słabszą wersją *proces zapewnienia* (*process assertion*), który trzyma się wyspecyfikowanych stanów. Własności stanów wzięte razem, po połączeniu – tworzą własność ścieżki. Na przykład, własnością ścieżki jest: *każde osiągnięcie stanu o własności P, musi ostatecznie prowadzić do przejścia do stanu o własności Q, po bezpośrednim osiągnięciu stanu R*. Weryfikator SPIN może sprawdzić oba stany oraz własności ścieżki, oba rodzaje sprawdzania można wyrazić w specyfikacji napisanej w PROMELA.

Pewne typy właściwości są tak podstawowe, że nie wymagają dodatkowych wyjaśnień. SPIN sprawdza je domyślnie. Jedną z takich własności jest, przykładowo, nieobecność osiągania stanu *zakleszczenia systemu*. Oczywiście, użytkownik może modyfikować semantykę wbudowując sprawdzanie na drodze prostego etykietowania instrukcji. Przykładowo, zakleszczenie może być domyślnie rozważane, jako niezamierzone osiągnięcie stanu końcowego systemu. Możemy powiedzieć weryfikatorowi, że pewne specyfikowane stany końcowe są celowo etykietowane, jako stany końcowe. Własności poprawności są formułowane w PROMELA poprzez użycie następujących konstrukcji:

- Stwierdzenia podstawowe (*Basic assertions*);
- Etykiety stanu końcowego (*End-state labels*);
- Etykiety stanu postępu (*Progress-state labels*);
- Etykieta stanu akceptacji (*Accept-state labels*);
- Żądania niemożliwe (*Never claims*);
- Stwierdzenia trasy (*Trace assertions*).

Żądania niemożliwe (*Never claims*) - można napisać ręcznie, można je generować automatycznie z formuły logicznej lub z opisu własności zależności od czasu. Konstrukcje odpowiadające każdej z tych możliwości omówimy szczegółowo poniżej.

5.4.5.12. **Stwierdzenia podstawowe** (*Basic assertions*). Instrukcja mająca postać:

```
assert(expression)
```

jest nazywana w *stwierdzeniem podstawowym* (*basic assertions*) PROMELA, dla odróżnienia od terminu *stwierdzenie tropiące* (*trace assertion*), które to pojęcie omówimy później. Przydatność stwierdzeń tego typu było zauważone dawno temu. Wzmiankę na ten temat można znaleźć w pracy *Johna von Neumanna* (1903 - 1957). Jest ona następująca (oczywiście nie dotyczy ona języka programowania C, który opracowany 30 lat później):

≪It may be true, that whenever C actually reaches a certain point in the flow diagram, one or more bound variables will necessarily possess certain properties, or satisfy certain relations with each other. Furthermore, we may, at such a point, indicate the validity of these limitations. For this reason we will denote each area in which the validity of such limitations is being asserted, by special box, which we call an 'assertion box'.

Goldstein and von Neumann, 1947≫

Stwierdzenia podstawowe w PROMELA, są zawsze wykonywalne, podobnie jak podstawienia, drukowanie i instrukcja `skip`. Wykonanie instrukcji wzmiankowanego typu nie tworzy w wyniku wyrażenia, które przyjmuje boolowską wartość *true* lub alternatywnie nie-zerową wartość całkowito-liczbową. Wynik własności podstawowej poprawności, zapewnia niemożliwość przyjęcia wartości *false* (lub zero). Nie spełnianie stwierdzenia spowoduje wysłanie komunikatu błędu.

Jak pokazano już, model trywialny:

```
init { assert(false) }
```

daje po wykonaniu następujący wydruk (raport 5.4.5.13):

```
$ spin false.pml
spin: line 1 "false.pml", Error: assertion violated
#processes: 1
     1:   proc 0 (:ini:) line  1 "false.pml" (state 1)
1 process created
```

5.4.5.13. Raport przebiegu dla trywialnego modelu

Powyżej SPIN został użyty w trybie symulacji. Wykonanie zostało zatrzymane w punkcie programu, w którym został wykryty błąd stwierdzenia poprawności. Po zatrzymaniu przebiegu symulacji, ze zmienną `pid` zero, jest to tak zwany pierwszy stan wewnętrzny numerowany. Symulator zawsze tworząc listę stanów, przy których ma miejsce zatrzymanie. Wszystkich zainicjowane procesy, zostają zawieszone, ale nie zostają zakończone, można powiedzieć, że dalej żyjących. Jeśli dokonamy zmiany w modelu z wartości `false` na `true`, nie pojawi się żaden komunikat błędu, ponieważ niema żadnego uruchomionego procesu, który by pozostał po zakończeniu przebiegu. W wyniku stopu, zabity został jedyny zainicjowany proces.

```
$ spin true.pml
1 process created
```

Instrukcja `assert()`, jest jedynym rodzajem badania poprawności przez PROMELA, która umożliwia badanie poprawności w toku przebiegu symulacyjnego SPIN. Wszystkie pozostałe postacie będą dyskutowane dalej i dotyczą badania poprawności w toku przebiegu walidacji. Jeśli przebiegi Spin załamią się na skutek natrafienie na instrukcję powodującą naruszenie poprawności, to wcale nie oznacza, że instrukcja `assert()` wbudowana w badany model, nie może naruszyć przebiegu symulacji.

5.4.5.14. **Meta etykiety.** Etykiety w specyfikacji PROMELA służą głównie, jako cel dla instrukcji bezwarunkowego skoku `goto`. Omówimy poniżej trzy typy etykiet, mających specjalne znaczenie, gdy SPIN działa w trybie weryfikacji. Etykiet tych używa się do identyfikacji:

- Stanu końcowego (*End states*)
- Stanów kolejnych etapów (*Progress states*)
- Stanów uznających fakt (*Accept states*)

Stan końcowy: kiedy model PROMELA jest badany w trybie weryfikacji, pod kątem osiągnięcia stanu zakleszczenia (*deadlock*), weryfikator musi mieć możliwość rozróżnienia prawidłowe zakończenie wykonywania modelu, czyli osiągnięcie stanu `end` od stanu nieprawidłowego (raport 5.4.5.15).

```
mtype { p, v }
chan  sema = [0] of { mtype };
```

```
active proctype Dijkstra()
{     byte count = 1;
end:  do
      :: (count == 1) ->
            sema!p; count = 0
      :: (count == 0) ->
            sema?v; count = 1
      od
}
active [3]proctype user()
{     do
      :: sema?p;              /* enter */
critical:   skip;             /* leave */
      od
}
```

5.4.5.15. Raport badania modelu dla stwierdzenia zakleszczenia procesów

Domyślnie, jedynymi poprawnymi stanami końcowymi lub punktami terminalnymi, są te w których każdy zainicjowany proces PROMELA, osiągnął zakończenie swojego kodu (to jest nawias klamrowy w odpowiadającym mu ciele instrukcji `proctype`). Zauważmy, że wszystkie procesy PROMELA zmierzają do zakończenia swojego kodu. Niektóre mogą znajdować się w stanie oczekiwania lub mogą przebywać cierpliwie w pętli, żeby podjąć działanie, gdy pojaw się nowe wejście.

Dla jasności działania weryfikatora, te alternatywne stany końcowe, oznaczamy specjalnymi etykietami, nazywanymi etykietami końcowymi (`end-state`). Dla umożliwienia weryfikatorowi, rozpoznania które są ważne z poszczególnych alternatywnych stanów końcowych (*end-states*), wprowadzamy specjalne etykiety: *end-state*. Przykładowo użyto taką etykietę w modelu semafora typu Dijkstra, modelującego semafor o nazwie `sema` z kanałem `rendezvous`. Ten semafor gwarantuje, że tylko jeden z trzech procesów użytkowych może znajdować się w danym przedziale czasu w sekcji krytycznej. Etykieta `end` definiuje, że nie ma miejsca sytuacjas błędu oraz, że jest to koniec wykonania określonej sekwencji, a dany proces nie osiągnął zamykającego nawiasu klamrowego, oczekując przy etykiecie. Oczywiście, taki stan mógłby być częścią stanu zakleszczenia, ale to nie dotyczy w szczególności danego procesu. Model napisany w PROMELA, może zawierać wiele etykiet typu *end-state*, zakładając jednak, że każda z etykiet w obrębie ciała `proctype`, jest unikalna. Dla umożliwienia użycia więcej niż jednej etykiety typu *end-state* w tym samym ciele `protype`, PROMELA korzysta z zasady mówiącej, że każda etykieta posiadająca trzy literowy prefiks `end` określa etykietę typu *end-state*. Następujące nazwy etykiet, są poprawnymi etykietami typu *end-state*: `endme`, `end0`, `end_of_this_part`.

Działając w trybie weryfikacji SPIN sprawdza domyślnie niepoprawne etykiety typu *end-state*, z czego wynika, że żadnej szczególnej ostrożności nie trzeba zachować, żeby zapobiegać tego typu błędom. Z drugiej strony, jeśli użytkownik *nie* interesuje się tego rodzaju błędami, może użyć opcji `-E` dla pominięcia w raporcie sygnałów dotyczących takich błędów. Podobnie, użycie opcji `-A`, można wyłączyć pojawianie się raportu naruszenia poprawności (oznacza to, że szukamy błędów innych typów, zaś błędy dotyczące naruszenia poprawności będą badane w przebiegu weryfikacji). Żeby zablokować pojawianie się błędów obu typów dla przykładowego modelu „z etykietowanym stanem końcowym”, można użyć poniższego postępowania (raport 5.4.5.16):

```
$ spin -a dijkstra.pml
$ cc -o pan pan.c
$ ./pan -E -A     # add two restrictions
(Spin Version 4.0.7 -- 1 August 2003)
      + Partial Order Reduction
Full statespace search for:
      never claim                  - (none specified)
      assertion violations         - (disabled by -A flag)
      acceptance cycles            - (not selected)
      invalid end states           - (disabled by -E flag)
State-vector 36 byte, depth reached 8, errors: 0
      15 states, stored
       4 states, matched
      19 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)
1.573       memory usage (Mbytes)
unreached in proctype Dijkstra
            line 14, state 10, "-end-"
            (1 of 10 states)
unreached in proctype user
            line 22, state 7, "-end-"
            (1 of 7 states)
```

5.4.5.16. Raport z badania modelu - z etykietowanym stanem końcowym

Otrzymywany wydruk nie odbiega od poprzedniego, czyli nieużywającego zarówno opcji `-A`, jak opcji `-E`, ponieważ niema stanów `end` ani naruszenia poprawności w modelu. Jako wynik użycia restrykcji, weryfikator słusznie zauważa w swoim wydruku, że niemożliwe są takie dwa rodzaje sygnałów.

Generowanemu przez SPIN weryfikatorowi – można również nakazać przemilczanie dokładnych wymagań stanu `end`, jak naszkicowano to powyżej. Jeśli użyjemy dodatkowej opcji przebiegu `-q`, kompilując weryfikację, wszystkie procesy muszą osiągać poprawnie stany końcowe (`end states`) oraz wszystkie kanały komunikacji muszą być puste, dla dopuszczalnego stanu modelu. W normalnych przypadkach, kiedy nie korzystamy z opcji `-q`, wymaganie dotyczące pustych kanałów komunikacji, jest pomijane.

5.4.5.17. **Stany postępu** (*Progress States*). Podobne konwencje kontekstowe dotyczą użycia etykiet stanów postępu (*progress states*) PROMELA. Używamy etykiet stanu postępu dla zaznaczenia instrukcji w modelu napisanym w PROMELA, którym towarzyszy zjawisko uzyskania czegoś istotnego; czegoś, co jest wynikiem rzeczywistego postępu, a nie pustej czynności czy oczekiwania, że inny proces osiągnie rzeczywisty postęp. Używamy trybu weryfikacji SPIN, dla sprawdzenia czy każdy potencjalnie nieskończony cykl, który jest dozwolony w modelu, przeszedł przynajmniej jednokrotnie przez etykietę stanu postępu w modelu. Jeśli zostanie stwierdzone, że pętla nie posiada takiej własności, weryfikator może deklarować istnienie pętli bez postępu (*non-progress loop*), odpowiadającą możliwości zagłodzenia.

Przykładowo, możemy umieścić etykietę stanu postępu w omawianym algorytmie Dijkstra, w następujący sposób (kod 5.4.5.18):

```
active proctype Dijkstra()

      {           byte count = 1;
      end:        do
                  :: (count == 1) ->
      progress          sema!p; count = 0
                  :: (count == 0) ->
                        sema?v; count = 1
                  od
      }
```

5.4.5.18. Kod programu Dijkstra z etykietą postępu

Poprawne przejście w tym miejscu przez test semafora jest postępem i powoduje dla upewnienia się zapytanie weryfikatora, czy w wyniku nieskończenie wielu wykonań, proces semafora osiągnie etykietę postępu nieskończenie często. Żeby wykonać przebieg z kontrolą pętli bez postępu, musimy kompilować weryfikator ze specjalną opcją, która doda właściwy typ czasowego żądania do modelu (szczegóły tego żądania przedstawimy dalej). Dla sprawdzenia tego przejścia, postępujemy jak niżej (raport 5.4.5.19):

```
$ spin -a dijkstra_progress.pml
$ cc -DNP -o pan pan.c  # enable non-process checking
$ ./pan -l                    # search for non-progress cycles
(Spin Version 4.0.7 - 1 August 2003)
      + Partial Order Reduction
Full statespace search for:
      never claim       +
      assertion violations    + (if within scope of claim)
      non-progress cycles     + (fairness disabled)
      invalid end states      + (disabled by never claim)
State-vector 40 byte, depth reached 18, errors: 0
      27 states, stored (39 visited)
      27 states, matched
      66 transitions (= visited+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
1.573    memory usage (Mbyte)
unreached in proctype Dijkstra
      line 14, state 10, "-end-"
      (1 of 10 states)
Unreached in proctype user
      line 22, state 7, "-end-"
      (1 of 7 states)
```

5.4.5.19. Raport przebiegu z kontrolą pętli bez postępu

Na początku wydruku widzimy, że wyszukiwanie pętli bez-postępu (*non-progress cycles*) jest dostępne, oraz że został użyte żądanie `never`. Takie żądanie poprawności, jest zwykle definiowane przez użytkownika lub wynika z formuły logicznej. Żądanie `never`, może być również generowane wewnętrznie przez SPIN. W tym przypadku, żądanie jest generowane przez SPIN i automatycznie dołączane do modelu, żeby zdefiniować sprawdzenie własność stanu postępu (*non-progress*). Dołączenie tego żądania, jest spowodowane dyrektywą kompilacji –

DNP. W dalszej części, omówimy ze szczegółami - używanie definiowanego przez użytkownika żądanie never.

Wydruk z przebiegu weryfikacji mówi nam, że zarówno naruszenia poprawności jak również istnienia cyklu bez-postępu – nie miały miejsce. Licznik błędów ma wartość zero, co oznacza, że nie miało miejsca ani naruszenie poprawności, ani nie wystąpił przynajmniej jeden cykl bez-postępu. Możemy na tej podstawie wnioskować, że model „Etykietowany stan końcowy" – pozwala na wielokrotne wykonywanie operacji P semafora.

Aby umożliwić wyszukiwanie własności „pętla bez-postępu", – czyli „*livens property*", automatycznie uniemożliwiamy wyszukiwania niepoprawnych stanów końcowych, czyli zapewniamy „*safety property*". Warto też zauważyć, że porównując ostatnie sprawdzanie, liczba osiąganych stanów prawie podwoiła się. Kiedy omawiamy algorytm wyszukiwania używany przez SPIN dla sprawdzenia różnych typów własności, dowiemy się, co jest przyczyną takiego wyniku. Jeśli więcej niż jeden stan jest oznakowany etykietą „*progress label*", wariacje dotyczące wspólnego prefiksu dalej obowiązują i mamy etykiety takie jak: progress0, progression. Jak może wyglądać bardzo prosty przykład cyklu *bez-postępu*, zobaczymy na modelu zawierającego dwa procesy (kod 5.4.5.20):

```
byte x = 2;
active proctype A()
{     do
      :: x = 3 - x
      od
}
active proctype B()
{     do
      :: x = 3 - x
      od
}
```

5.4.5.20. Kod programu bez etykiet postępu z pętlami bez-postępu

Wyjaśniając, te dwa procesy będą powodować, że wartość zmiennej globalnej x będzie *ad infinitum* alternatywnie 2 lub 1. Żadnej etykiety postępu nie używamy, natomiast każdy przebieg pętli modelu jest *pętlą bez-postępu*. Możemy wykonać sprawdzanie *pętli bez-postępu*, jak poniżej (raport 5.4.5.21):

```
$ spin -a fair.pml
$ cc -DNP -o pan pan.c  # enable non-progress checking
$ ./pan -l                    # search for non-progress cycles
pan: non-progress cycle (at depth 2)
pan: wrote fair.pml.trail
(Spin Version 4.0.7 - August 2003)
Warning: Search not completed
            + Partial Order Reduction
Full statespace search for:
      never claim       +
      assertion violations    + (if within scope of claim)
      non-progress cycles     + (fairness disabled)
      invalid end states      - (disable by never claim)
State-vector 24 byte, depth reached 7, errors: 1
      3 states, stored (5 visited)
```

```
      4 states, matched
      9 transitions (= visited+matched)
      0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)
1.573 memory usage (Mbyte)
```

5.4.5.21. Raport przebiegu sprawdzania istnienia pętli bez-postępu

Jak można się było spodziewać, *pętla bez postępu* została znaleziona. Przywołanie wykrycia błędu przez weryfikator, zapisane zostało wraz ze ścieżką dojścia, w pliku `pan.trail`. Użycie tego pliku, pozwala na odtworzenie ścieżki błędu, z pomocą opcji symulacji przebiegu SPIN'u, w sposób poniższy (raport 5.4.5.22):

```
$ spin -t -p fair.pml
spin: couldn't find claim (ignored)
      2: proc 1 (B) line 12 "fair.pml" (state 1) [x = (3-x)]
      4: proc 1 (B) line 12 "fair.pml" (state 1) [x = (3-x)]
      <<<<<START OF CYCLE>>>>>
      6: proc 1 (B) line 12 "fair.pml" (state 1) [x = (3-x)]
      8: proc 1 (B) line 12 "fair.pml" (state 1) [x = (3-x)]
spin: trail ends after 8 steps
#processes 2
            x = 2
      8: proc 1 (B) line 11 "fair.pml" (states 2)
      8: proc 0 (A) line  5 "fair.pml" (states 2)
2 processes created
```

5.4.5.22. Raport przebiegu odtworzenie ścieżki błędu

Ostrzeżenie, że SPIN nie może zlokalizować żądania identyfikacji błędu jest naiwne: żądanie użyte do stwierdzenia pętli bez postępu – nie może się pojawić, ponieważ jest predefiniowane (wyjście z żądaniem byłoby błędem modelu). Kroki, których numery nie pojawiły się na wydruku (kroki 1, 3, 5 i 7), są krokami wykonanymi automatycznie przez ukryte żądanie.

5.4.5.23. **Poprawne pętle** (*Fair Cycles*). Kontrprzykład pokazuje tylko nieskończenie krotne wykonanie procesu typu `B`, bez udziału żadnego innego procesu systemu. Uderzający jest fakt, że SPIN nie pozwala nam na przyjęcie założeń dotyczących względnej szybkości wykonywania procesów, dopuszczalny jest specjalny przypadek procesu typu A pauzującego w nieskończoność, przy czym kontprzykład obowiązuje. Ciągle jednak może mieć miejsce przypadek istnienia naruszenia własności, przy bardziej realistycznych założeniach. Jednym z takich założeń, jest założenie typu - skończony postęp (*finite progres*). Mówi ono, że każdy proces mogący wykonać instrukcję, będzie wykonywalny.

Mamy dwa warianty tegoż założenia. Dokładna wersja stwierdza, że jeśli dany proces osiąga punkt, w którym znajduje się wykonalna instrukcja, to nigdy się nie zmieni wykonywalność tejże instrukcji. Bardziej ogólny przypadek dotyczy sytuacji, kiedy dany proces osiąga punkt, w którym znajduje się instrukcja wykonywana nieskończenie często, wówczas przyjmujemy, iż jest to instrukcja wykonywalna. Pierwsza wersja często nazywana jest *słabą poprawnością* (*weak fairness*), zaś druga *silną poprawnością* (*strong fairness*). W naszym przykładzie, wymuszamy słabą poprawność poszukując pętli bez postępu, odgrywającą istotną rolę w kontrprzykładzie domyślnego poszukiwania. Możemy wymusić słabą poprawność, w toku weryfikacji następująco (raport 5.4.5.24):

```
$ ./pan -l -f
      pan: non-progress cycle(at depth 8)
      pan: wrote fair.pml.trail
      (Spin Version 4.0.7 -- 1 August 2003)
      Warning: Search not completed
                  + Partial Order Reduction
      Full statespace search for:
            never claim       +
            assertion violations    + (if within scope of claim)
            non-progress cycles     + (fairness enabled)
            invalid end states      - (disabled by never claim)
      State-vector 24 byte, depth reached 15, errors: 1
             4 states, stored (12 visited)
             9 states, matched
            21 transitions (= visited+matched)
             0 atomic steps
      hash conflicts: 0 (resolved)
      (max size 2^18 states)
      1.573    memory usage (Mbytes)
```

5.4.5.24. Raport przebiegu walidacji ze „słabą poprawnością”

Nowa pętla, którą opisano powyżej, powinna być spójną ze słabą poprawnością z założenia o skończonym postępie. Szybkie spojrzenie na nowy kontrprzykład potwierdza to (raport 5.4.5.25).

```
      $ spin -t -p fair.pml
      spin: couldn't find claim (ignored)
        2:   proc 1 (B) line 12 "fair.pml" (state 1) [x = (3-x)]
        4:   proc 1 (B) line 12 "fair.pml" (state 1) [x = (3-x)]
        6:   proc 1 (B) line 12 "fair.pml" (state 1) [x = (3-x)]
        8:   proc 0 (A) line  6 "fair.pml" (state 1) [x = (3-x)]

<<<<<START OF CYCLE>>>>>

       10:   proc 1 (B) line 12 "fair.pml" (state 1) [x = (3-x)]
       12:   proc 1 (B) line 12 "fair.pml" (state 1) [x = (3-x)]
       14:   proc 1 (B) line 12 "fair.pml" (state 1) [x = (3-x)]
       16:   proc 0 (A) line  6 "fair.pml" (state 1) [x = (3-x)]
      spin: trail ends after 16 steps
      #processes: 2
                        x = 2
       16:   proc 1 (B) line 11 "fair.pml" (state 2)
       16:   proc 0 (A) line  5 "fair.pml" (state 2)
      2 processes created
```

5.4.5.25. Raport przebiegu tworzenia kontr-przykładu

Jako następną próbę, możemy dodać etykietę postępu (*progress label*) do jednego z procesów (ale nie do obu) z dwóch `proctypes`, przykładowo jak niżej (kod 5.4.5.26):

```
      active proctype   B()
      {
            do
```

```
        :: x = 3 - x; progress: skip
        od
}
```

5.4.5.26. Kod programu z etykietą `progress`

Proces typu `B` będzie teraz zmieniał się z typu *progress state* na typ *non-progress state*, co oznacza, że gdy rozpocznie się pętla, proces może w zasadzie oczekiwać zawsze w stanie *non-progress state*. Pomyślmy przez chwilę, czego mógłby się spodziewać weryfikator wyszukując z innymi opcjami pętle typu *non-progress state*, zarówno w przypadku bez i z opcją słabej poprawności. Wykonaj taką próbę i sprawdź czy wynik jest zgodny z rozumieniem podejścia[95]. Oczywiście, to jest dodanie dodatkowego ograniczenia do obliczeniowej złożoności problemu weryfikacji. Nie jest to zbyt widoczne w tym małym przykładzie, ale ogólnie, ilość pracy wykonywanej przez weryfikator musi – wzrośnąć o współczynnik N, gdzie N jest liczbą aktywnych procesów. Koszty mogą być jednak wyższe, jeśli będziemy dążyć do użycia silnej poprawności (*strong fairness*), co spowoduje wzrost współczynnika do wartości N^2. W praktyce, ten typ wartości narzutu, powoduje ograniczenie tejże opcji silnej poprawności jedynie do najprostszych testowych przypadków. Kiedy naprawdę potrzebne jest użycie opcji *finite progress* z silną poprawnością (*strong fairness*), można to wyrazić z pomocą formuły logiki temporalnej lub bezpośrednio jako własności ω-regularne z pomocą wyrażenia PROMELA nazwanego `never claim`. Zauważmy również, że wyobrażenie „sprawiedliwości" użyte w SPIN zastosowane do decyzji harmonogramowania (nie dotyczy rozkładów niedeterministycznych) wyborów wewnątrz procesów. Gdy zachodzi potrzeba, innego typu „sprawiedliwość" może zostać sformułowana dla formuł logicznych.

5.4.5.27. **Stany akceptacji** (*accept states*): ostatnim typem etykiet – stany akceptacji, są zarezerwowane do użycia z *brakiem żądania* (`never claim`), często mechanicznie generowanym z formuły logicznej. Przypadki użycia `never claim` omówimy w następnym podrozdziale. Chociaż ma to rzadko miejsce, etykiety stanów akceptacji, mogą być również użyte w dowolnym miejscu modelu PROMELA – bez koniecznej obecności żądania `never claim`. Oznaczając stan dowolną etykietą zaczynającą się prefiksem `accept` – prosimy weryfikator o wyszukanie wszystkich pętli, które przechodzą – co najmniej jeden raz przez tą etykietę. Podobnie do przypadku etykiety `progress-state`, etykieta `accept-state` – nie ma wpływu na przebieg symulacji: jest ona jedynie interpretowana przez SPIN w trybie weryfikacji. Bezpośrednie żądanie poprawności, wyrażone przez obecność jakiejś etykiety `accept-state` informuje, że nie istnieje takie wykonanie, które powoduje przejście przez etykietę `accept-state` nieskończenie często.

Dla umożliwienia użycia więcej niż jednej etykiety typu `accept-state` w pojedynczej instrukcji `proctype` lub żądaniu `never`, nazwa etykiety zostaje rozszerzona. Przykładowo, następujące wersje etykiety są przydatne: `accept`, `acceptance`, `accepting`. Powyższe uwagi dotyczące prawidłowego użycia poprawnych założeń, prowadzą do uznania równości pętli typów `non-progress` oraz `acceptace`. Spróbujmy np. zastąpić etykietę `progress`, w ostatnim przykładzie potwierdźmy, że weryfikator znajdzie zarówno poprawne jak i niepoprawne pętle w modelu wynikowym. Kroki działania weryfikatora będą następujące:

[95] Jeśli wszystko jest zgodne, będziesz w stanie potwierdzić, że w wyniku próby istnieją jedynie pętle typu *non-fair non-progress.*

```
$ spin -a fair_accept.pml
$ cc -o pan pan.c # note: no -DNP is used
$ ./pan -a              # all acceptance cycles
…
$ ./pan -a -f           # fair acceptance cycles only
```

5.4.5.28. Raport przebiegu

5.4.5.29. **Żądanie nigdy** (never claims). Do tego punktu mówiliśmy na temat specyfikacji kryteriów poprawności, ze względu wyrażenia stwierdzające w połączeniu z meta-etykietami. Silne kryteria poprawności, może oczywiście zostać sformułowane z pomocą tych narzędzi, jeszcze jak dotąd jedyną opcją, jest dodanie tego kryterium do indywidualnej deklaracji `proctype`. Nie możemy po prostu wyrazić naszego żądanie w postaci: „każdy stan systemu, w którym własność `p` jest *true*, prowadzi do stanu systemu w którym własność `q` jest również *true*. Przyczyną, dla której nie możemy sprawdzić tak sformułowanego żądania z pomocą opisanych dotychczas mechanizmów, dlatego że nie ma sposobu zdefiniowania badania, które może być wykonywane w każdym pojedynczym kroku działania systemu. Zauważmy, że nie możemy przyjąć założeń dotyczących względnej szybkości wykonywania procesu, co oznacza, że między dowolnymi dwoma wykonywanymi instrukcjami każdego standardowego procesu PROMELA, może wystąpić pauza, której długość jest wykorzystana do wykonania pewnej liczby kroków innych procesów systemu. Wyrażenia PROMELA typu `never claims`, są środkiem dającym możliwość określenia bardziej precyzyjnie badanie. Krótko mówiąc: żądanie `never` jest normalnie używane do specyfikacji zarówno skończonego, jak i nieskończonego zachowania systemu, które nigdy nie nastąpi.

5.4.5.30. **Jak działa żądanie nigdy** (*never claim*)**?** Rozważmy wykonanie następującego modelu *pots*, wcześniej już omawianego przykładu centrali telefonicznej (raport 5.4.5.31).

```
$ spin -c pots.pml | more
proc 0 = pots
proc 1 = subscriber
q     0     1
  2     .    line!offhook, 1
  2     line?ofhook, 1
  1     who!dialtone
  1     .  me?dialtone
  1     .  me!number
  1     who?number
  1     who!ringing
  1     .  me?ringing
  1     who!connected
  1     .  me?connected
timeout
  1     .  me!hangup
  1      who?hangup
  2      .  line!offhook, 1
  2     line?offhook, 1
  1     who!dialtone
  1     .  me?dialtone
  1     .  me!number
  1     who?number
  1     who!ringing
```

```
1      .  me?ringing
1      .  me!hangup
1      who?hangup
```

5.4.5.31. Raport przebiegu działania modelu centrali telefonicznej

Powyżej mamy dwadzieścia cztery kroki wykonywania modelu. Jedenaście kroków było wykonanych przez serwer *pot*, zaś trzydzieści przez proces *subscriber*. Ponieważ jest to system rozproszony, ważną jest nie tylko kolejność wykonywania poszczególnych procesów, ale również przeplatanie się wykonywania instrukcji w toku działania systemu.

Żądanie `never` daje możliwość sprawdzenia własności systemu tuż przed oraz tuż po – wykonaniu każdej instrukcji, bez względu, który z procesów wykonał daną instrukcję. Żądanie `never` pozwala na dokładne określenie, jaką z własności zamierzamy sprawdzić przy każdym kroku. Najprostszy typ żądania może być sprawdzony, jako prosta własność `p` w każdym (bez wyjątku) kroku, aby mieć pewność, że zawsze zachodzi. Łatwo jest sprawdzić niezmienniczość własności, w taki sposób.
Następujący przykład (kod 5.4.5.32) pokazuje jak można użyć zdalnego odwołania wewnątrz żądania `never`:

```
/*
 * Processes 1 and 2 cannot enter their
 * critical sections at the same time.
 */
never {
            do
            :: user[1]@critical && user[2]@critical ->
                  break /* implicitly accepting */
            :: else /* repeat */
            od
}
```

5.4.5.32. Kod programu - użycia zdalnego odwołania wewnątrz żądania `never`

Jeśli jest w systemie - tylko jedna instancja danego `proctype`, identyfikator tego procesu jest w rzeczywistości zbyteczny, oraz (w SPIN version 4.0 lub wyższej) referencją z ostatniego przykładu, może jest `user@critical`. Jeśli natomiast, przejdziemy do więcej niż jednej instancji typu procesu, ten typ referencji wybierze arbitralnie jedną z instancji. Symulator SPIN dostarczy ostrzeżenia w przypadku pojawienia się przypadku jak wyżej.

5.4.5.33. **Żądanie niezmienniczości własności** - można by zapisać na wiele sposobów. Oryginalnie, żądanie `never` oznacza jedynie zbadanie zachowania, które nigdy nie może zajść – w przypadku poprawnego działania modelu. Co oznacza, że system weryfikacji może oflagować pojawienie się błędu, który w toku działania systemu, nigdzie się nie pojawi. Najprostszy sposób zapisania żądania `never`, które sprawdzi niezmienniczość własności systemu `p`, może wyglądać następująco (kod 5.4.5.34):

```
never {
        do
        :: !p -> break
        :: else
        od
}
```

5.4.5.34. Kod programu sprawdzania niezmienniczości własności `p`

Proces sprawdzania jest wykonywany w każdym kroku systemu. Skoro tylko własność `p` przyjmie wartość *false*, proces przerwie pętlę i zakończy się, stwierdzając tym samym pojawienie się błędu. Jak długo wyrażenie `p` pozostanie *true*, żądanie pozostanie w stanie początkowym, a wszystko będzie w porządku.

Łatwo jest nadużyć stosowanie własność żądania `never` – pisząc bardziej intuicyjną wersję sprawdzania. Przykładowo (kod 5.4.5.35), uzyskamy ten sam efekt używając następujące wersji żądania `never`:

```
never {
        do
        :: assert(p)
        od
       }
```

5.4.5.35. Kod programu – intuicyjna wersja sprawdzania

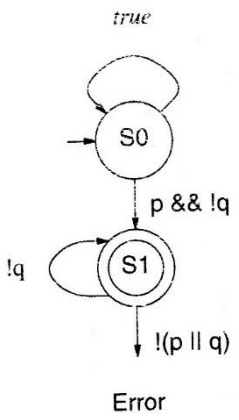
Teraz pętla zawiera stwierdzenie, że własność `p` jest zawsze *true*. Ta instrukcja stwierdzająca jest wykonywana zawsze. Pierwszy raz jest wykonana w stanie początkowym systemu. Po wykonaniu danego kroku, wykonywany jest następujący za nim krok. Jeśli nastąpi naruszenie własności określonej w stanie inicjacji, żądanie spowoduje natychmiastową sygnalizację naruszenia poprawności. Jasno mówiąc, obydwie instrukcje `!p` oraz `assert(p)` – są wolne od efektu ubocznego, dlatego też wersja żądania nie jest w stanie zmienić zachowania systemu. Ze względu na rozważaną własność prostej niezmienniczości, możemy ten sam efekt uzyskać na innej drodze. Możemy, uzyskać daną własność dodając proces, działający jako monitor wykonywania modelu (kod 5.4.5.36):

```
active proctype monitor()
{
      atomic { !p -> assert(false) }
}
```

5.4.5.36. Kod programu monitora

Taki monitor może zainicjować sekwencję `atomic`, w każdym kroki działania modelu, gdy tylko okaże się, że `p` przyjmie wartość *false*. Oznacza to, że jeśli istnieje stan osiągalny przez system, w którym niezmiennicza własność `p` przyjmie wartość *false*, monitor stwierdzi to natychmiast i wywiesi flagę pojawienia się błędu. Jeśli takie zachowanie modelu jest możliwe, to proces monitorujący jest rozwiązaniem.

Rozważmy, np. taką własność: *Każdy stan systemu w którym `p` jest prawdą (true) – prowadzi do stanu systemu w którym `q` jest prawdą (true), gdzie dalej `p` jest prawdą (true).* Weryfikator SPIN, nie jest zainteresowany w wykryciuu systemu spełniającego własność, ale wyszukuje przypadki naruszenia danej własności. Zauważmy, że naruszenie własności gdy `q` pozostaje *false*, może być pokazane tylko przy nieskończenie krotnym wykonaniu. Ponieważ zajmujemy się skończoną przestrzenią stanów, każda nieskończenie krotne wykonanie, tylko może być pętlą. Nie możemy więc użyć stwierdzenia dotyczącego wykrywania takiego rodzaju błędu. Nie możemy więc znaleźć odpowiedniego typu sprawdzenia, dla wykrycia żądania typu `never`. Ten typ żądania jest prosty do zapisania jako formuła w LTL (*Linear Temporal Logic*), jak pokazano na rysunku 5.4.5.37. Wykonanie żądania rozpoczyna się od etykiety S0, która zawiera pętle o dwóch opcjach (kod 5.4.5.38).



5.4.5.37. Przepływ sterowania w języku LTL.
Własność **¬□(p→(q→Uq)).**

```
never {
S0:         do
            :: p && !q -> break
            :: true
            od;
S1:         do
            :: !q
            :: !(p || q) -> break
            od
}
```

5.4.5.38. Kod programu realizacji formuły LTL

Druga z opcji zawiera pojedynczą instrukcję `true`. Jasno mówiąc, ta opcja jest zawsze wykonalna i nie powoduje żadnego oddziaływania - zachowuje żądanie w stanie początkowym. Pierwsza z opcji, jest wykonywana tylko wtedy kiedy wyrażenie `p` ma wartość `true`, zaś wyrażenie `q` ma wartość `false`. W tym przypadku stwierdzamy, że dopasowanie do poprzedniej własności: wartość `true` dla wyrażenia `p`, ale jeszcze nie `true` dla wyrażenia `q`. Wykonanie będzie kontynuowane, przez instrukcję z etykietą `accept`. Jak widać na omawianym rysunku, interpretacja instrukcji `break`, podobnie jak instrukcji `goto`, nie spowoduje wykonania kolejnego kroku. Działa jedynie jako specjalny typ separatora instrukcji, który zamaże domyślny wybór następnego stanu. Pojawienie się w tym miejscu instrukcja `break`, określa która z instrukcji zostanie wykonana po drugim wartowniku (znak ::).

Mówiąc jaśniej, żądanie nie może przebić się w pierwszym stanie, ponieważ najpierw trzeba wykonać wartujące wyrażenie `true`. Wykonując wartujące `true` jest równoważne ignorowania żądaniu systemu w pojedynczym kroku, odraczając sprawdzanie na później. W stanie `accept` żądania, mamy znowu dwie opcje wykonania. Jedyną drogą aby pozostać w tym stanie jest sytuacja w której wyrażenie q będzie mieć na stałe wartość `false`. Jeśli to jest możliwe, to w sposób ciągły będzie naruszać żądanie poprawności. Dlatego też, oznaczamy ten stan etykietą `accept`. Naruszenie poprawności, może teraz być wyłapane przez SPIN jako pętla akceptacji. Jedyna inna możliwa droga, na której nastąpi naruszenie poprawności, jest przypadek uzyskania wartości `false` przez wyrażenie `p`, zanim wyrażenie `q` przyjmie wartość `true`. Ten rodzaj naruszenia poprawności jest wyłapywany przez wymuszenie zakończenia żądania typu `never`. Zakończenie żądania jest interpretowane jako pełna zgodność specyfikowanego zachowania; zachowania, które z założenia przyjmujemy, że nigdy (`never`) nie będzie miało miejsca.

Co natomiast nastąpi, jeśli oba wyrażenia `p` oraz `q` przyjmą wartość `true`? Żadna z dwu opcji dla przypadku kontynuacji wykonywania procesu nie jest dopuszczalna: żądanie nie może się przebić. Kuriozalnie, to nie jest błąd, ale oczekiwany warunek. Jeśli żądanie w następnym kroku wykonywania systemu, nie pasuje do wykonywania procesu, wychwycone nieoczekiwane zachowanie przez żądanie `never`, nie może być w pełni dopasowane do wykonywania. Wszystko jest w porządku: żądanie zatrzymuje badanie systemu wzdłuż ścieżki wykonania i błędy nie są sygnalizowane. Użycie weryfikatora powoduje, że możemy zagwarantować efektywność poprawnego dobierania zachowań przez żądanie, nie tylko przy jednokrotnym wykonaniu, ale przy wszelkich możliwym wykonaniu. Dla jasności, trzeba powiedzieć: SPIN w trybie symulacji nie jest w stanie zagwarantować poprawności, ale w trybie weryfikacji – jest w stanie i to robi.

5.4.5.39. **Powiązania z LTL**. Żądanie `never` dostarcza silnego mechanizmu dla wyrażania własności systemów rozproszonych. Według powszechnej opinii, pozwala na trudne podejście z formalizacją własności systemu przez bezpośrednie kodowanie żądania `never`. Na szczęście, istnieje prostsza droga, aby to uzyskać. Jedną z takich metod jest użycie wbudowanego w SPIN translatora formuł liniowej logiki temporalnej (LTL) – dla zapisania żądania `never`. Przykładowo ostatnie żądanie koresponduje z formułą LTL:

```
![](p -> (p U q))
```

Można wygenerować żądanie `never` używając powyższej formuły (kod 5.4.5.40):

```
$ spin -f '![](p -> (p U q))'
never {     /* ![](p -> (p U q)) */
T0_init:
            if
            :: (! ((q)) && (p)) -> goto accept_S4
            :: (1) -> goto T0_init
            fi;
accept_S4:
            if
            :: (! ((q)) -> goto accept_S4
            :: (! ((p)) && ! ((q)) -> goto accept_all
            fi;
accept_all:
            skip
}
```

5.4.5.40. Kod programu w języku translatora SPIN - formuł liniowej logiki temporalnej (LTL)

Inną metodą jest użycie małego narzędzia graficznego SPIN, nazwanego *timeline editor*, dla przekonwertowania zależności czasowych w żądanie `never`. W wielu przypadkach, formuły LTL są prostsze do zrozumienia i użycia niż żądanie `never`, są one również mniej wyraziste.

5.4.5.41. **Stwierdzenie ścieżki** (trace assertion). Podobnie jak żądanie `never`, *stwierdzenie ścieżki* (`trace assertion`) – nie określa nowego zachowania, ale wyraża wymaganą poprawność zachowania w danym systemie. Stwierdzenie `trace` wyraża własność kanału komunikacji, oraz w szczególności jest formalną instrukcją dotyczącą dopuszczalności albo niedopuszczalności sekwencji operacji, które proces może wykonywać w kanale komunikacji. Wszystkie nazwy odnoszące się do `trace` muszą być zdefiniowane jako globalne, a wszystkie pola komunikatów muszą być globalnymi stałymi lub `mtype` symbolicznymi stałymi. Prosty przykład użycia *stwierdzenia ścieżki* (`trace assertion`) jest (kod 5.4.5.42):

```
trace {
            do
            :: q1!a; q2?b
            od
}
```

5.4.5.42. Kod programu – przykład użycia wyrażenia `trace assertion`

W tym przykładzie *stwierdzenie* specyfikuje poprawność wymagań, mówiąc że operacja wysłania (*send*) przez kanał `q1` następuje z operacją otrzymania (*receive*) w kanale `q2` i co więcej, że wszystkie operacje wysyłania w `q1` są wyłącznie komunikatami typu `a`, zaś wszystkie operacje otrzymania w kanale `q2` są wyłącznie komunikatami typu `b`.

Stwierdzenie ścieżki stosuje się tylko do operacji kanałowych wysyłania i otrzymywania wiadomości. Są one używane do specyfikowania względnej kolejności, w jakiej tego typu operacje zawsze muszą być wykonywane. Tylko nazwy kanałów wyspecyfikowanych w *stwierdzeniu* są rozważane, żeby zawierały się w przedziale sprawdzania. Wszystkie operacje wysyłania i otrzymywania wiadomości dotyczące innych kanałów są ignorowane. Stwierdzenie `trace` trywialnie definiuje automat który przebiega skończony zbiór stanów kontrolnych, monitorowanych w czasie wykonywania systemu. Ten automat zmienia stan tylko wtedy, gdy wykonywana operacja wysyłania lub otrzymywania mieści się w zadanym przedziale. Jeśli natomiast operacja jest wykonywana w dopuszczalnym przedziale, ale nie może być dopasowana przez przejście automatu `trace`, weryfikator natychmiast sygnalizuje błąd.

Jeśli co najmniej jedna operacja wysłania (otrzymania) na którymś kanale pojawi się przy *stwierdzeniu ścieżki*, to wszystkie operacje wysyłania (otrzymania) są podmiotem sprawdzania. Podobnie jak z żądaniem `never`, istnieją ograniczenia dotyczące typu instrukcji, które mogą pojawić się w przypadku użycia *stwierdzenia ścieżki*. Poza konstrukcją przepływu sterowania, *stwierdzenie ścieżki* może zawierać tylko proste operacje wysłania i otrzymania. Nie może używać wariacji, takich jak: losowe otrzymania, posortowane wysyłania lub operacja głosowania kanałów. Żadne obiekty danych nie mogą być deklarowane lub przywoływane wewnątrz *stwierdzenia ścieżki*.

Jeśli pola komunikatów muszą pasować w operacjach wysyłania i otrzymywania, które pojawiają się wewnątrz *stwierdzenia ścieżki*, to muszą być specyfikowane ze standardową nazwą `mtype` lub być wartością stałą. Operacje wysyłania i otrzymania, które pojawiają się nazywamy zdarzeniami monitorowanymi. Te zdarzenia nie generują nowych zachowań, ale one wymagają dopasowania zdarzeń w kanałach modelu z dopasowującymi parametrami wiadomości. Zdarzenia dotyczące wysyłania i otrzymania pojawiają się gdy instrukcje wysyłania i otrzymywania są wykonywane przez system, to jest gdy zdarzenia te pojawią się w czasie przejścia od jednego stanu do innego.

Deklarowanie ścieżki może zawierać etykiety `end-state`, `progress-state` oraz `accept-state` z typową interpretacją. Jest jednak, kilka ważnych różnic pomiędzy stwierdzeniem `trace` a żądaniem `never`:

- Przeciwnie niż przy żądaniu `never`, stwierdzenie `trace` zawsze musi być deterministyczne.
- Stwierdzenie `trace` może pasować do występowania zdarzeń, pojawiających się przy przejściu pomiędzy stanami systemu podczas gdy żądanie `never` tylko pasuje do stwierdzonej wartości stanu systemu.

- Stwierdzenie `trace` monitoruje tylko podzbiór zdarzeń systemowych - i tylko tych typów, które wymienione są w `trace` (tj. monitorowanych zdarzeń). Żądanie `never`, wszystkie stany osiągalne z drugiej strony są widziane w całym globalnym systemie, oraz muszą pasować do każdego osiągalnego stanu.

5.4.5.43. **Zdalne referencje.** W SPIN version 4.0 lub wyższej, inne typy referencji są również obsługiwane. Dodatkowe typy referencji pomijają standardowy zakres zasad umożliwiając każdemu procesowi PROMELA, jak również żądaniu `never`, odwoływać się do bieżących wartości lokalnych zmiennych przez inne procesy. Tej własności należy używać ostrożnie, ponieważ można doprowadzić do konfliktu z założonym w SPIN zakresem działania - zasad określonych w strategii redukcją częściowego uporządkowania.

5.4.5.44. **Kwantyfikacja przejść** (*Path Quantification*). Będziemy omawiawać algorytm weryfikacji użyty w SPIN z dodatkowymi szczegółami, na początek zwrócimy uwagę na kilka istotnych własności przyjętego podejścia. Wszystkie własności poprawności, weryfikowane przez Spin można interpretować jako formalne żądania wystąpienia albo nieobecności pewnych zachowań.

- Żądanie wyrażone instrukcją formalizującą stwierdzenie, że niemożliwym jest aby dane wyrażenie przyjmowało wartość *false*.
- Etykieta `end` stwierdza, że jest niemożliwym żeby system zakończył swoje działanie zanim wszystkie aktywne procesy: albo zakończą działanie albo zostaną zatrzymane przez jedną z etykiet stany `end`.
- Etykieta `progress` stwierdza, że jest niemożliwym dla systemu, żeby w nieskończoność działał bez przejścia nieskończenie często, przynajmniej przez jeden specjalnie etykietowany stan `progress`.
- Etykieta `accept` stwierdza, że jest niemożliwym dla systemu, żeby w nieskończoność działał bez przejścia nieskończenie często, przynajmniej przez jeden specjalnie etykietowany stan `accept`.
- Żądanie `never` stwierdza, że jest niemożliwym dla systemu żeby pokazywał zarówno nieskończone jak i skończone – zachowanie całkowicie dopasowane z działaniem sformalizowanym przez żądanie.
- Stwierdzenie ścieżki (`trace assertion`) – na koniec, że jest niemożliwym dla systemu, który narusza zasady sformalizowanego działania.

We wszystkich powyższych przypadkach, weryfikacja wykonywana przez SPIN została zaprojektowana tak, żeby dowieść, że użytkownik się myli: SPIN będzie próbować znaleźć kontrprzykład, dla każdego postawionego żądania. Tak więc SPIN nigdy nie dąży do udowodnienia poprawności programu. Próbuje coś odwrotnego. Polowanie na kontrprzykłady, ma zalety w stosunku do dowodu wprost. W szczególności, jeśli pozwala to weryfikatorowi na użycie efektywniejszych procedur wyszukiwania. Jeśli, np. błędne zachowanie - da się sformalizować do postaci żądania `never`, SPIN może ograniczyć przeszukiwanie do kontrprzykładu zachowania pasującego do żądania. Żądanie `never` – działa jako restrykcja przestrzeni wyszukiwania. Jeśli błędne zachowanie w rzeczywistości jest niemożliwe, tak jak żąda tego użytkownik, weryfikator ma niewiele pracy do wykonania. Jeśli natomiast błędne zachowanie jest *prawie* niemożliwe, może mieć trochę więcej pracy, ale ciągle niekoniecznie tak dużo, jak w przypadku przeszukania całej przestrzeni ze względu na wszelkie możliwe zachowania. To zaś się zdarza tylko w najgorszych przypadkach.

Piśmiennictwo: *Holzmann G.* H.3.1, H.3.2.

5.5. PRZYKŁADY WALIDACJI MODELI

5.5.0. UWAGI WSTĘPNE

Pierwszym krokiem biznesowym wykorzystania procesora SPIN do walidacji zadanego systemu, jest konstrukcja wiernego modelu napisanego w języku PROMELA (porównaj - *5.1.3. Tworzenie abstrakcji i problem eksplozji liczby osiąganych stanów*). Język ten pozwolił na sprawne napisanie zwięzłego modelu (abstrakcji). Celem budowy modelu, jest wychwycenie tych aspektów systemu, które jedynie odpowiadały badanym zadaniom koordynacji działania systemu. Wszystkie inne szczegóły zostały pominięte. Formalnie – ten model jest redukcją systemu do ekwiwalentu funkcjonalności systemu, ograniczonego do koordynację działań systemu - dotyczącego badanego problemu. Skonstruowany model (abstrakcja) jest podstawą czegoś - co nazwano zestawem walidacyjnym (*validation suites*). Zestawu, który zostaje następnie użyty do zbadania własności modelu. Żeby zbudować taki zestaw walidacji, musieliśmy uwzględnić w modelu relacje niezmienniczej dotyczącej zadanych wartości zmiennych lub dopuszczalnych sekwencji zdarzeń zachodzących w modelu.

Piśmiennictwo: *Holzmann G.* H.3.1, H.3.2.

5.5.1. MODEL H. HYMAN'A OBSŁUGI SEKCJI KRYTYCZNEJ SYSTEMU

Pierwszym przykładem zastosowania praktycznego procesora SPIN, była weryfikacja algorytmu obsługi sekcji krytycznej systemu, opublikowanego w 1966 roku przez *H. Hyman'a* w czasopiśmie *Communications of the ACM*. Algorytm ten był napisany w tzw. pseudo Algolu i miał postać jak niżej (kod 5.5.1.00):

```
 1 Boolean array b(0;1) integer k, i,
 2 comment process i, with i either 0 or 1, and k = 1-i;
 3 C0:  b(i) := false;
 4 C1:  if k != i then begin;
 5 C2:  if not b(1-i) then go to C2;
 6      else k := i; go to C1 end;
 7      else critical section;
 8      b(i) := true;
 9      remainder of program;
10      go to C0;
11      end
```

5.5.1.00. Oryginalny kod programu realizacji algorytmu Hyman'a

Rozwiązanie zaproponowane przez *Hyman'a*, zawiera podobnie jak wcześniejsze rozwiązanie *Dekker'*a, dwa procesy, numerowane 0 oraz 1. Założono, że celem jest dowiedzenie, iż rozwiązanie zaproponowane przez *Hyman'a*, jest rozwiązaniem gwarantującym w pełni wzajemnie wykluczający się dostęp do sekcji krytycznej. Pierwszym zadaniem, było napisanie modelu zaproponowanego rozwiązania w języku PROMELA. Starano się, zachować nazwy użyte przez *Hyman'a* w jego wersji algorytmu (kod 5.5.1.01).

```
 1  bool want[2];      /* Bool array b */
 2  bool turn; /* integer    k */
 3
 4  proctype P(bool i)
 5  {
 6     want[i] = 1;
 7     do
 8     :: (turn != i) ->
 9             (!want[1-i]);
10             turn = i
11     :: (turn == i) ->
```

```
12              break
13      od;
14      skip; /* critical section */
15      want[i] = 0
16  }
17
18  init { run P(0); run P(1) }
```

5.5.1.01. Kod programu napisanego w PROMELA realizacji algorytmu Hyman'a

Po wygenerowaniu plików analizatora, skompilowaniu - wykonano przebieg weryfikacji, w pierwszej kolejności w celu sprawdzenia, czy nie pojawiło się globalne zakleszczenie modelu (raport 5.5.1.02).

```
$ spin -a hyman0
$ gcc -o pan pan.c
$ ./pan
full statespace search for:
assertion violations and invalid endstates
vector 20 byte, depth reached 19, errors: 0
      79 states, stored
       0 states, linked
      38 states, matched      total: 117
hash conflicts: 4 (resolved)
(size 2^18 states, stack frames: 3/0)

unreached code _init (proc 0):
        reached all 3 states
unreached code P (proc 1):
        reached all 12 states
```

5.5.1.02. Raport przebiegu symulacji modelu algorytmu Hyman'a

W ten sposób model algorytmu przeszedł pierwszy test. Natomiast nadal nie mamy pewności, że algorytm gwarantuje poprawne działanie wzajemnie wykluczającego dostępu, jednego z dwóch procesów, do sekcji krytycznej systemu. Jest szereg dróg dokonania takiej weryfikacji. Najprostsze podejście, to dodanie do modelu niezbędnej informacji, wyrażającej stwierdzenie (w rozumieniu języka PROMELA) poprawnego działania algorytmu - zgodnie z wymaganiami (kod 5.5.1.10).

```
 1  bool want[2];
 2  bool turn;
 3  byte cnt;
 4  proctype P(bool i)
 5  {
 6    want[i] = 1;
 7    do
 8    :: (turn != i) ->
 9            (!want[1-i]);
10            turn = i
11    :: (turn == i) ->
12            break
13    od;
14    skip; /* critical section */
15    cnt = cnt+1;
16    assert(cnt == 1);
17    cnt = cnt-1;
18    want[i] = 0
19  }
20  init { run P(0); run P(1) }
```

5.5.1.10. Kod programu algorytmu Hyman'a z rozszeżeniami

Dodaliśmy globalną zmienną `cnt`, której wartość jest powiększana przy każdym dostępie do sekcji krytycznej I zmniejszana przy każdym opuszczeniu tejże sekcji krytycznej. Maksymalna wartość tej zmiennej zawsze powinna być 1, a wartość ta zawsze jest osiągana, jeśli któryś z dwu procesów jest wewnątrz sekcji krytycznej (raport 5.5.1.11).

```
$ spin -a hyman1
$ gcc -o pan pan.c
$ ./pan
assertion violated (cnt==1)
pan: aborted (at depth 15)
pan: wrote pan.trail
full statespace search for:
assertion violations and invalid endstates
search was not completed
vector 20 byte, depth reached 25, errors: 1
     123 states, stored
       0 states, linked
      55 states, matched      total: 178
hash conflicts: 42 (resolved)
(size 2^18 states, stack frames: 3/0)
```

5.5.1.11. Raport przebiegu modelu algorytmu Hyman'a z rozszeżeniami

Walidator stwierdził, że stwierdzenie mogło być naruszone. Używamy (raport 5.5.1.12), więc ścieżkę błędów (*error trail*), do sprawdzenia z pomocą opcji SPIN `-t`:

```
$ spin -t -p hyman1
proc  0 (_init) line 24 (state 2)
proc  0 (_init) line 24 (state 3)
proc  2 (P)     line 8 (state 7)
proc  2 (P)     line 9 (state 2)
proc  2 (P)     line 10 (state 3)
proc  2 (P)     line 11 (state 4)
proc  1 (P)     line 8 (state 7)
proc  1 (P)     line 12 (state 5)
proc  1 (P)     line 15 (state 10)
proc  2 (P)     line 8 (state 7)
proc  2 (P)     line 12 (state 5)
proc  2 (P)     line 15 (state 10)
proc  2 (P)     line 16 (state 11)
proc  2 (P)     line 17 (state 12)
proc  2 (P)     line 18 (state 13)
proc  1 (P)     line 16 (state 11)
proc  1 (P)     line 17 (state 12)
spin: "hyman1" line 17: assertion violated
step 17, #processes: 3
                want[0] = 1
                _p[0] = 12
                turn[0] = 1
                cnt[0] = 2

proc  2 (P)     line 18 (state 13)
proc  1 (P)     line 17 (state 12)
proc  0 (_init) line 24 (state 3)
3 processes created
```

5.5.1.12. Raport przebiegu sprawdzającego modelu algorytmu Hyman'a z rozszeżeniami

Tutaj mamy inną drogę złapania błędu. Dodatkowo wyposażmy model w informację pozwalające policzyć liczbę procesów równocześnie znajdujących się w sekcji krytycznej (kod 5.5.1.13).

```
 1  bool want[2];
 2  bool turn;
 3  byte cnt;
 4
 5  proctype P(bool i)
 6  {
 7    want[i] = 1;
 8    do
 9    :: (turn != i) ->
10            (!want[1-i]);
11            turn = i
12    :: (turn == i) ->
13            break
14    od;
15    cnt = cnt+1;
16    skip;   /* critical section */
17    cnt = cnt-1;
18    want[i] = 0
19  }
20
21  proctype monitor()
22  {
23    assert(cnt == 0 || cnt == 1)
24  }
25
26  init {
27    run P(0); run P(1); run monitor()
28  }
```

5.5.1.13. Kod programu dodatkowo zliczającego procesy

Warunek niezmienniczości wartości zmiennej licznika `cnt`, zostaje umieszczony w oddzielnym procesie `monitor()` (ta nazwa jest nieistotna). Ten dodatkowy proces jest wykonywany razem z dwoma pozostałymi. Proces kończy swoje działanie po wykonaniu jednego kroku, ale krok ten może być wykonywany w dowolnym momencie. Modelowany w języku PROMELA - system i weryfikowany z pomocą SPIN, jest kompletnie asynchroniczny. Oznacza to, weryfikacja SPIN bierze pod uwagę wszelki możliwe względne przebiegi czasowe wszystkich trzech procesów. Przy pełnej weryfikacji, wyliczanie stwierdzenia, może być wykonane w dowolnym momencie życia pozostałych dwóch procesów. Jeśli raport weryfikacji nie zawiera naruszenia stwierdzenia, można wnioskować, że niema takiej sekwencji wykonywalnej wszystkich trzech procesów (przy każdej szybkości wyboru jednego - z pośród trzech procesów), w której stwierdzenie zostanie naruszone. Wstawienie do modelu procesu monitora, jest eleganckim sposobem walidowania niezmiennika systemu. A oto procedura weryfikacji (raport 5.5.1.14):

```
$ spin -a hyman2
        $ gcc -o pan pan.c
        $ ./pan
        assertion violated ((cnt==0)||(cnt==1))
        pan: aborted (at depth 15)
        pan: wrote pan.trail
        full statespace search for:
        assertion violations and invalid endstates
        search was not completed
        vector 24 byte, depth reached 26, errors: 1
             368 states, stored
```

```
          0 states, linked
        379 states, matched      total: 747
  hash conflicts: 180 (resolved)
  (size 2^18 states, stack frames: 4/0)
```

5.5.1.14. Raport przebiegu weryfikacji

W wyniku powstania dodatkowych oddziaływań dwóch wcześniej wprowadzonych w modelu procesów, liczba stanów systemu, które zostały zbadane, istotnie wzrosłą, ale błąd jest nadal poprawnie zaraportowany.

Piśmiennictwo: *Holzmann G.* H.3.1, H.3.2.

5.5.2. NASTĘPNY WAŻNY PRZYKŁAD

Nie zawsze jednak wymaganie poprawności, może zostać wstawione, jako globalny niezmiennik systemu. Poniżej pokazany jest przykład ilustrujący taki brak możliwości. Mamy dany prosty protokół zmiany bitu, modelujący możliwość straty komunikatu i zniekształcenie sytuacji - wynikającego z negatywnego potwierdzenia (kod 5.5.2.00).

```
 1  #define MAX      5
 2
 3  mtype = { mesg, ack, nak, err };
 4
 5  proctype sender(chan in, out)
 6  { byte o, s, r;
 7
 8    o=MAX-1;
 9    do
10    :: o = (o+1)%MAX; /* next msg */
11  again: if
12         :: out!mesg(o,s) /* send */
13         :: out!err(0,0)  /* distort */
14         :: skip       /* or lose */
15         fi;
16         if
17         :: timeout   -> goto again
18         :: in?err(0,0) -> goto again
19         :: in?nak(r,0) -> goto again
20         :: in?ack(r,0) ->
21           if
22           :: (r == s) -> goto progress
23           :: (r != s) -> goto again
24           fi
25         fi;
26  progress: s = 1-s /* toggle seqno */
27    od
28  }
29
30  proctype receiver(chan in, out)
31  { byte i;         /* actual input   */
32    byte s;         /* actual seqno   */
33    byte es;        /* expected seqno */
34    byte ei;        /* expected input */
35
36    do
37    :: in?mesg(i, s) ->
38           if
39           :: (s == es) ->
40                   assert(i == ei);
```

```
41  progress:        es = 1 - es;
42                   ei = (ei + 1)%MAX;
43                   if
44    /* send,   */  :: out!ack(s,0)
45    /* distort */  :: out!err(0,0)
46    /* or lose */  :: skip
47                   fi
48           :: (s != es) ->
49                   if
50    /* send,   */  :: out!nak(s,0)
51    /* distort */  :: out!err(0,0)
52    /* or lose */  :: skip
53                   fi
54           fi
55    :: in?err ->
56           out!nak(s,0)
57    od
58  }
59
60  init {
61    chan s_r = [1] of { mtype,byte,byte };
62    chan r_s = [1] of { mtype,byte,byte };
63    atomic {
64           run sender(r_s, s_r);
65           run receiver(s_r, r_s)
66    }
67  }
```

5.5.2.00. Kod programu prostego protokołu komunikacyjnego

Testu poprawności transferu danych powyższego protokołu, można dokonać już w pierwszym przebiegu weryfikacji. W tym celu wysyłający zostaje ustawiony na przesłanie nieskończonej serii liczb całkowitych jako komunikatu, gdzie wartość liczb całkowitych jest zwiększana modulo `MAX`. Sama zaś wartość `MAX` nie jest ze względu na nasze rozważania zbyt interesująca, jak długo jest ona większa niż rząd sekwencji liczb w protokole. Chcemy weryfikacji wysyłanych danych, ze względu na dostarczanie ich do odbiorcy bez żadnego opuszczania pozycji lub zmiany kolejności, pomimo możliwości niezależnego tracenia komunikatów. Stwierdzeni podane w wierszu 40-tym weryfikuje pojawienie się takiej sytuacji. Zauważmy, że jeśli protokół nie spełni powyższego wymagania, to wówczas stwierdzenie nie będzie spełnione.

Pierwszy przebieg zapewni nas, że sytuacja taka nie może powstać (raport 5.5.2.01).

```
$ spin -a ABP0
$ gcc -o pan pan.c
$ ./pan
full statespace search for:
assertion violations and invalid endstates
vector 40 byte, depth reached 131, errors: 0
     346 states, stored
       1 states, linked
     125 states, matched       total: 472
hash conflicts: 17 (resolved)
(size 2^18 states, stack frames: 0/25)

unreached code _init (proc 0):
        reached all 4 states
unreached code receiver (proc 1):
        line 58 (state 24)
        reached: 23 of 24 states
```

```
unreached code sender (proc 2):
        line 28 (state 27)
        reached: 26 of 27 states
```

5.5.2.01. Raport przebiegu SPIN

Musimy jednak zachować ostrożność. Raportowany wynik oznacza, że wszystkie dane zostały dostarczone w pożądanej kolejności bez strat. Nie sprawdziliśmy jednak, czy dane na pewno zostały dostarczone. Żeby to sprawdzić, trzeba wprowadzić dodatkowe stany dla procesów nadawcy i odbiorcy, które niewątpliwie będą oznaczone, jako „`progress states`".

Wówczas, jesteśmy w stanie zademonstrować nieobecność wykonywanych w nieskończoność pętli, które nie przechodzą przez żaden z tzw. „`progress states`". Nie możemy użyć takiego samego ustawienia przebiegu jak poprzednio (raport 5.5.2.01), ale nie jest łatwo ustawić weryfikator na wykrywanie pętli typu „`non-progress`":

```
$ gcc -DNP -o pan pan.c
$ ./pan -l
pan: non-progress cycle (at depth 6)
pan: wrote pan.trail
full statespace search for:
assertion violations and non-progress loops
search was not completed
vector 44 byte, depth reached 8, loops: 1
      12 states, stored
       1 states, linked
       0 states, matched      total: 13
hash conflicts: 0 (resolved)
(size 2^18 states, stack frames: 0/1)
```

5.5.2.03. Raport przebiegu walidacji

Jak wynika z raportu przebiegu walidacji, ma miejsce wystąpienie przynajmniej jednej pętli typu „`non-progress`". Pierwszy z nich został umieszczony w pliku ścieżki błędu, przez weryfikator i właśnie potwierdziliśmy ten fakt. Ten wynik jest pokazany w pierwszej połowie raportu 5.5.2.06. Kanał może przeinaczyć lub zagubić dowolnie często. Sam jednak scenariusz nie jest zbyt interesujący. Żeby zobaczyć liczbę pętli typu „`non-progress`", musimy użyć opcji `-c`. Jeżeli ustawimy numeryczny argument tej opcji równy zero, to dostaniemy jedynie całkowitą liczbę powtarzania się tegoż błędu.

```
$ pan -l -c0
full statespace search for:
assertion violations and non-progress loops
vector 44 byte, depth reached 137, loops: 92
     671 states, stored
       2 states, linked
     521 states, matched       total: 1194
hash conflicts: 39 (resolved)
(size 2^18 states, stack frames: 0/26)
```

5.5.2.04. Raport przebiegu walidacji z użyciem opcji `-c`

Łącznie mamy, więc 92 przypadki, z których każdy indywidualnie możemy ocenić, używają kolejnych wartości towarzyszących opcji `-c` (`-c1`, `-c2`, `-c3`, … itd.). Możemy wykonać to zadanie znacznie prościej, na drodze odfiltrowania błędów powodowanych nieskończonymi stratami komunikacji. W tym celu, wystarczy wszystkie zdarzenia tracenia (linie 13, 43 i 48), jako

„progress states”, używając etykiet, z 8-znakowym prefiksem „progress”, oglądając pozostałe pętle (etykiety wstawiamy za czterokropkiem „: :”).

```
$ spin -a ABP1
$ gcc -DNP -o pan pan.c

$ ./pan -l
pan: non-progress cycle (at depth 133)
pan: wrote pan.trail

full statespace search for:
assertion violations and non-progress loops
search was not completed

vector 44 byte, depth reached 136, loops: 1

     148 states, stored
       2 states, linked
       2 states, matched       total: 152

hash conflicts: 0 (resolved)
(size 2^18 states, stack frames: 0/26)
```

5.5.2.05. Raport przebiegu wskazujący na błąd protokołu

Tym razem, pojawi się znaczący i poważny błąd protokołu. W drugiej połowie raportu 5.6.2.06, pokazana zostanie ścieżka prowadząca do błędu.

```
$ spin -t -r -s ABP0
<<<<>>>>
proc  1 (sender)   line  13, Send err,0,0 -> queue 2 (out)
proc  2 (receiver) line  55, Recv err,0,0 <- queue 2 (in)
proc  2 (receiver) line  56, Send nak,0,0 -> queue 1 (out)
proc  1 (sender)   line  19, Recv nak,0,0 <- queue 1 (in)
spin: trail ends after 12 steps
step 12, #processes: 3
         _p[0] = 6
proc  2 (receiver)       line 36 (state 21)
proc  1 (sender) line 11 (state 6)
proc  0 (_init)  line 67 (state 4)
3 processes created
$
$ spin -t -r -s ABP1
...
proc  2 (receiver) line  39, Recv mesg,0,0 <- queue 2 (in)
proc  2 (receiver) line  47, Send err,0,0  -> queue 1 (out)
proc  1 (sender)   line  20, Recv err,1,0  <- queue 1 (in)
proc  1 (sender)   line  12, Send mesg,0,0 -> queue 2 (out)
proc  2 (receiver) line  39, Recv mesg,0,0 <- queue 2 (in)
proc  2 (receiver) line  52, Send nak,0,0  -> queue 1 (out)
proc  1 (sender)   line  21, Recv nak,0,0  <- queue 1 (in)
proc  1 (sender)   line  12, Send mesg,0,0 -> queue 2 (out)
proc  2 (receiver) line  39, Recv mesg,0,0 <- queue 2 (in)
proc  2 (receiver) line  52, Send nak,0,0  -> queue 1 (out)
<<<<>>>>
proc  1 (sender)   line  21, Recv nak,0,0  <- queue 1 (in)
proc  1 (sender)   line  12, Send mesg,0,0 -> queue 2 (out)
proc  2 (receiver) line  39, Recv mesg,0,0 <- queue 2 (in)
proc  2 (receiver) line  52, Send nak,0,0  -> queue 1 (out)
spin: trail ends after 226 steps
...
```

Raport 5.5.2.06 Ścieżka prowadząca do błędu w protokole

Wychodząc z pojedynczej widomości umieszczonej na ścieżce błędu i przesłanego, jako komunikatu `err` (wysyłającego oraz odbierającego), wykryta została nieskończona pętla, w której wysyłający z uporem godnym lepszej sprawy – powtarza ostatni komunikat, na który nie otrzymał potwierdzenia, ponieważ odbiorca, również z uporem godnym lepszej sprawy odrzucał komunikat z negatywnym potwierdzeniem.

Piśmiennictwo: *Holzmann G.* H.3.1, H.3.2.

Piśmiennictwo

A.1.1. Abadi Martin, Cardelli Luca, *A Theory of Objects* (Monographs in Computer Science), Springer Verlag, Berlin 1998.

A.2.1. Ajdukiewicz Kazimierz, *Główne zasady metodologii nauk i logiki formalnej* – skrypt autoryzowany, zred. M. Presburger, Warszawa 1928.

A.2.2. Ajdukiewicz Kazimierz, *Logika, jej zadania i potrzeby w Polsce współczesnej*, „Myśl Filozoficzna" 1951, nr 1-2, s. 50 - 67.

B.1.1. Banachowski Lech, Diks Krzysztof, Rytter Wojciech, *Algorytmy i struktury danych* – wydanie piąte, Wydawnictwa Naukowo-Techniczne, Warszawa 2001.

B.2.1. Ben-Ari Mordechai, *Mathematical Logic for Computer Science* – Third Edition, Springer Verlag, London, Heidenberg, New York, Dordrecht 2012.

B.2.2. Ben-Ari Mordechai, *A Primer on Model Checking*, ACM Inroads, 2010 March, Vol. 1, No 1.

B.3.1. Büchi, J.R., *On a decision method in restricted second order arithmetic*,. Proc. International Congress on Logic, Method, and Philosophy of Science. 1960 (Stanford: Stanford University Press): 1–12.

B.4.1. de Berg M., van Kreveld M., Overmars M., Schwarzkopf O., *Geometria obliczeniowa – algorytmy I zastosowania* (tłumaczenie z angielskiego) , Wydawnictwa Naukowo-Techniczne, Warszawa 2007.

B.5.1. BPMN - *Business Process Model Notation*, www.bpmn.org.

B.6.1. Booch Grady, *Object oriented design with applications*, The Benjamin/Cumming Publishing Company, Redwood City, California 1990.

B.7.1. Booch Grady, Jacobson Ivar, Rumbaugh James, *OMG Unified Modeling Language Specification, Version 1.3 First Edition:* March 2000, www.omg.org.

C.1.1. Chappell David, *Zrozumieć platformę .NET* (tłumaczenie z angielskiego), Wydanie II, Wydawnictwo Helion, Gliwice 2007.

C.2.1. Chellas Brian F., *Modal Logic - an Introduction*, Cambridge 1980

C.3.1. Clarke Edmund M., Emerson E. Allen, Sifakis Joseph, *Model Checking: Algorithmic Verification and Debugging*, Communiction of The ACM, November 2009, Vol. 52, No 11, pages 74- 84.

C.4.1. Codd Eduard F., *A relational model of data for large shared data banks*, Communication of the ACM, Vol. 13, No 6, June 1970.

C.5.1. Cormen Thomas, Leiserson Charles E., Stein Clifford, *Wprowadzenie do algorytmów* – nowe wydanie (tłumaczenie z angielskiego), Wydawnictwo Naukowe PWN, Warszawa 2012.

D.1.1. Dahl Ole-Johan, Nygaard Kristen, *Class and subclass declarations*, In Proceedings from IFIP TC2 Conference on Simulation Programming Languages, Lysebu, Oslo, ed.: J. N. Buxton, pages 158-174. North Holland, May 1967.

D.2.1. Dijkstra, E.W., *Guarded commands, non-determinacy and formal derivation of programs. CACM* 18, 8 (1975), 453-457.

D.3.1. Dobosiewicz Stanisław, Tokarski Jan, Wieczorkiewicz Bronisław, *Kultura języka*, Warszawa 1953.

D.4.1. D'Souza Desmond Francis, Wills Alan Cameron, *Objects, components, and frameworks with UML*, Addison-Wesley, New York 1998.

E.1.1. Euclides, *Początków jeometryi xiąg ośmioro, to jest sześć pierwszych, iedenasta i dwunasta z dodanemi przypisami i trygonometrią dla pożytku młodzi akademickiey* - tłumaczone i wydane przez Józefa Czecha, Wilno 1807.

F.1.1. Fowler Martin, *UML w kropelce* – wersja 2.0 (tłumaczenie z angielskiego), Wydawnictwo LPT, Warszawa 2005.

G.1.1. Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John, *Wzorce projektowe* (tłumaczenie z angielskiego), Wydawnictwa Naukowo-Techniczne, Warszawa 2005.

G.2.1. Greniewski Henryk, *Elementy logiki formalnej*, Wydawnictwo PWN, Warszawa 1955.

G.2.2. Greniewski Henryk, *Logique et cybernetique*, Actes du 1-er Congres International de Cybernetique, Namur 1956.

G.2.3. Greniewski Henryk, *Cybernetyka nie-matematyczna*, Państwowe Wydawnictwo Naukowe, Warszawa 1969.

G.3.1. Greniewski Marek J. red. ,*Technologia procesów przetwarzania danych dla zarządzania* (praca zespołowa), Państwowe Wydawnictwo Ekonomiczne, Warszawa 1972.

G.3.2. Greniewski Marek J., *Selected MRP II Standard System requirements presented in Z-notation*, Kybernetes, vol. 38 No. 7/8, Emerald publisher, Bingley 2009.

G.3.3. Greniewski Marek J., *The engine of MRP II Standard System requirements presented in Z-notation* (part 1), Studia I Materiały (Zeszyt 1), Oficyna Wydawnicza EWSIE, Warszawa 2011, s. 17-30.

G.4.1. Gruber Martin, *SQL – wydanie drugie*, Wydawnictwo Helion, Gliwice 2000.

G.5.1. Guttag John V., Horning James J., *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.

H.1.1. Harel David, Kozen Dexter, Tiuryn Jerzy, *Dynamic Logic*, Massachutts Institut of Technology, The MIT Press, Cambridge, Massachusetts, London, England 2000.

H.2.1. Harold Elliotte Rusty, Means W. Scott, *XML – Almanach* (tłumaczenie z angielskiego), Wydawnictwo Helion, Gliwice 2002.

H.3.1. Holzmann Gerard, *The SPIN Model Checker – Primer and Reference Manual*, Addison-Wesley, Boston, San Francisko, New York, Toronto, Montreal, London, Munich, Paris, Madrid, Capetown, Sydney, Tokyo, Singapore, Mexico City, 2004.

H.3.2. Holzmann Gerard, *Basic SPIN Manual*, http://www.spinroot.com

H.4.1. Hoare, C.A.R., *Communicating Sequential Processes*. *CACM* 21, 8 (1978), 666-677.

H.5.1. Hyde Randall, *Profesionalne programowanie* – część 1 i 2 (tłumaczenie z angielskiego), Wydawnictwo Helion, Gliwice 2005.

J.1.1. Jacobson Ivar, Ericsson Maria, Jacobson Agneta, *The Object Advantage*, Addison-Wesley, New York, 1994

J.2.1. Jakubowski Arkadiusz, *Podstawy SQL – ćwiczenia praktyczne*, Wydawnictwo Helion, Gliwice 2001.

J.3.1. Jaśkowski Stanisław, *Trzy przyczynki do dwuwartościowego rachunku zdań*, Toruń 1948.

J.4.1. Jordan Zbigniew, *Platon odkrywcą metody aksjomatycznej*, „Przegląd Filozoficzny", 1937, rocz. 40, z. 1, s. 57-67.

K.1.1. Kisielewicz Andrzej, *Sztuczna inteligencja i logika*, Wydawnictwa Naukowo-Techniczne, Warszawa 2011.

K.2.1. Knaster Bronisław, *O aplikacieh matematicke logiky na matematike*, „Casopis pro pestovani matematiku", Praha 1951, s. 3 - 22.

K.3.1. Kołmogorow A., *Algebres de Boole metriques completes traduction*, VI Zjazd Matematyków Polskich Warszawa 20 – 23 IX 1948, „Dodatek do Rocznika Polskiego Towarzystwa Matematycznego", t. 22, Kraków 1950.

K.4.1. Kotarbiński Tadeusz, *Elementy teorii poznania, logiki formalnej i metodologii nauk*, Lwów 1929.,

K.4.2. Kotarbiński Tadeusz, *Kurs logiki dla prawników*, Warszawa 1953.

K.4.3. Kotarbiński Tadeusz, *Myśl przewodnia metodologii Franciszka Bacona*, „Przegląd Filozoficzny", 1926, rocz. 29, s. 133 - 135.

K.5.1. Kozanecka Anna, *O rodzajach logik temporalnych*, Katolicki Uniwersytet Lubelski, Roczniki Filozoficzne, Tom IV, Nr 1, 2007.

K.6.1. Kuratowski Kazimierz, Mostowski Andrzej, *Teoria mnogości*, Warszawa 1952.

K.7.1. Kurose Jim, Ross Keith, *Sieci komputerowe* – IV wydanie (tłumaczenie z angielskiego), Wydawnictwo Helion, Gliwice 2010.

L.1.1. Liscov Barbara, *Three notation of proof*, *Knowing and the mystique of logic and rules*, Studies in Cognitive Systems Vol. 18, 1995 p.165-169.

Ł.2.1. Łoś Jerzy, *Logiki wielowartościowe a formalizacja funkcji intensjonalnych*, Kraków 1948.

Ł.2.2. Łoś Jerzy, *Podstawy analizy metodologicznej kanonów Milla*, „Annales Universitatis Mariae Curie-Skłodowska" 1947 [druk, 1948], sectio F, vol. 2, nr 5.

Ł.2.3, Łoś Jerzy, *Próba aksjomatyzacji logiki tradycyjnej*, „Annale Universitatis Mariae Curie-Skłodowska" 1946, sectio F, vol. 1, nr 3.

Ł.3.1. Łukasiewicz Jan, *Dowód zupełności dwuwartościowości rachunku zdań*, Sprawozdania z posiedzeń Towarzystwa Naukowego Warszawskiego Wydział III, Warszawa 1931.

Ł.3.2. Łukasiewicz Jan, *Elementy logiki matematycznej - skrypt autoryzowany*, oprac. M. Presburger, Warszawa 1929

Ł.3.3. Łukasiewicz Jan, *Logika dwuwartościowa*, „Przegląd Filozoficzny" 1921, rocz. 23 [druk. 1920], s. 189 – 205.

Ł.3.4. Łukasiewicz Jan, *Logika trójwartościowa*, „Ruch Filozoficzny" 1920, rocz. 5, nr 9, s. 170.

Ł.3.5. Łukasiewicz Jan, *O sylogistyce Arystotelesa*, Sprawozdania z czynności i posiedzeń Polskiej Akademii Umiejętności, t.44, nr 6, Kraków 1939.

Ł.3.6. Łukasiewicz Jan, *O zasadzie sprzeczności Arystotelesa*, Kraków 1910.

Ł.3.7. Łukasiewicz Jan, *Philosophische Bemerkungen zurmehrwertigen Systemen des Aussagen kalkuls*, Sprawozdania z posiedzeń Towarzystwa Naukowego Warszawskiego Wydzia lll, Warszawa 1930, s. 51-77.

Ł.4.1. Łukasiewicz Jan, Tarski Alfred, *Untersuchungenuber den Aussagenkalkul*, Sprawozdania z posiedzeń Towarzystwa Naukowego Warszawskiego Wydział III, Warszawa 1930.

M.1.1. Mano M. Morris, *Architektura komputerów*, (tłumaczenie z angielskiego) Wydawnictwa Naukowo-Techniczne, Warszawa 1980.

M.2.1. Mayer-Schonberger Victor, Cuker Kenneth, *Big Data Rewolucja, która zmieni nasze myślenie, pracę i życie,* (tłumaczenie z angielskiego), Wydawnictwo MT Biznes sp. z o.o., Warszawa 2014.

M.3.1. Mercer Dave, *XML kurs podstawowy* (tłumaczenie z angielskiego), Wydawnictwo Edition 2000, Kraków 2001.

M.5.1. Mostowski Andrzej, *Logika matematyczna*, Warszawa 1948.

N.1.1. Neyman Jerzy, *On the Two Different Aspects of the Representation Method: The Method of Stratified Sampling and the Method of Purposive Selection*, Journal of the Royal Statistical Society 97, No 4, 1934, pp. 558-625.

N.2.1. Nilsson Ulf, Maluszynski Jan, *Logic, Programming and Prolog (2ed)*, Linköping University - Department of Computer and Information Science, 2012 (*Internet*)

P.1.1. Pacholski Leszek, *Logika dla informatyków* (Uniwersytet Wrocławski: Materiały do zajęć - Studia magisterskie na kierunku Informatyka), Wrocław 2004.

P.2.1. Pawlak Zdzisław, *Systemy informacyjne - podstawy teoretyczne*, Wydawnictwa Naukowo-Techniczne, Warszawa 1983.

P.2.2. Pawlak Zdzisław, *Rought sets - theoretical aspects of resoning about data*, Theory and Decision Library / Series D: System theory, Knowledge Engineering and Problem Solution, Vol 9, Dordreht Kluwer Academy 1991.

P.2.3. Pawlak Zdzisław, http://www.informatik.uni-trier.de/~ley/pers/hd/p/.

P.3.1. Pnueli Amir, *The temporal logic of programs*, SFCS '77 Proceedings of the 18th Annual Symposium on Foundations of Computer Science pp.46-57, IEEE Computer Society Washington, DC, USA 1977.

P.4.1. Prior Arthur Norman, *Time and Modality* (based on his 1956 John Locke lectures), Oxford University Press, 1957.

P.5.1. Pollack S. L., *DETAB: An improved business-oriented computer management*, Rand Corp. Memo RM-3273-PR, 1962.

R.1.1. Ross Kenneth A., Wright Charles R. B., *Matematyka dyskretna* (tłumaczenie z angielskiego), Wydawnictwo Naukowe PWN, Warszawa 1999.

S.1.1. Shannon C.E., *A Symbolic analysis of relay and switching circuits*, Transaction of the American Institute of Electrical Engineers 57, 1938, s. 1-11.

S.2.1. Shaw Alan C., *Projektowanie logiczne systemów operacyjnych* (tłumaczenie z angielskiego), Wydawnictwa Naukowo-Tecniczne, Warszawa 1980.

S.4.1. Shell Sam, Bayley Ian, *A Translation-Facilited Comparison Between the Common Language Runtime and the Java Virtual Machine*, Electronic Notes in Theoretical Computer Science, No 141 (2005) pp. 35-52.

S.5.1. Shirley P., *Fundamentals of Computer Graphics, sec. ed.*A K Peters, 2005.

S.6.1. Silberschatz A., Galdvin P. B., *Podstawy systemów operacyjnych* (tłumaczenie z angielskiego), Wydawnictwa Naukowo-Techniczne, Warszawa 2000.

S.7.1. Sipser Michael, *Wprowadzenie do teorii obliczeń* (tłumaczenie z angielskiego), Wydawnictwa Naukowo-Techniczne, Warszawa 2009.

S.8.1. Słupecki Jerzy, *Dowód aksjomatyzowalności pewnych systemów... – pełny trójwartościowy rachunek zdań*, „AnnalesUniversitatisMariaeCurie – Skłodowska"1946, sectio F, vol. 1, nr 3, s. 193 – 209.

S.8.2. Słupecki Jerzy, *O sylogistyce Arystotelesa*, „Annales Universitatis Mariae Curie – Skłodowska" 1946, sectio F, vol. 1, nr 3, s. 187- 209.

S.8.3. Słupecki Jerzy, *Z badań nad sylogistyką Arystotelesa*, Wrocław 1948.

S.9.1. Sosińsky Barrie, *Sieci komputerowe – Biblia* (tłumaczenie z angielskiego), Wydawnictwo Helion, Gliwice 2011.

S.10.1. Spivey J.M. (praca zespołowa), *The Z Notation: A Reference Manual*, 2nd Edition, Prentice Hall International (UK) Ltd., Oxford 1992.

S.11.1. Stallings William, *Organizacja i architektura systemu komputerowego* (tłumaczenie z angielskiego), Wydawnictwo Naukowo-Techniczne, Warszawa 2003.

S.12.1. Stencel Krzysztof, *Systemy operacyjne*, Wydawnictwo PIWSTK, Warszawa 2004.

S.13.1. Swift Jonathan, *Podróże Guliwera*, Warszawa 1951.

T.2.1.Tarski Alfred, *O pojęciu wynikania logicznego*, „Przegląd Filozoficzny"1936, rocz.39,z. 1, s. 58 – 68.

T.2.2. Tarski Alfred, *A Lattice-Theoretical Fixpoint Theorem and Its Applications,* Pacific J. Math. 5, 285-309, 1955.

T.3.1. Trzęsicki Kazimierz, *Logika temporalna w informatyce*, *www.kul.pl/files/57/wydzial/ logika_modalna/* 2009.

T.3.2. Trzęsicki Kazimierz, *Temporal Logic Model Checkers as Applied In Computer Science*, Studies In Logic and Rhetoric 17 (30) 2009.

T.4.1. Tiuryn Jerzy, Tyszkiewicz Jerzy, Urzyczyn Paweł, *Logika dla informatyków*, Uniwersytet Warszawski, 2006.

T.5.1. Tłuczek Marek, *Programowanie w języku C*, Wydanie II, Wydawnictwo Helion, Gliwice, 2011.

T.6.1. Turing Alan Mathison, *On Computable Numbers,* Proceedings of the London Mathematical Society, Ser. 2, Vol. 43, 1937.

T.8.1. Turkinton Garry, *Hadoop – Beginner's Guide*, Pact Publishing, Birmingham, UK 2013.

U.1.1. *Unified Modeling Language* (*UML*) – defined by The Object Management Group (OMG) - www.omg.org .

W.2.1. Wikipedia, *Algol 60*.

W.2.2. Wikipedia, *Algorytm.*

W.2.3. Wikipedia, *AMD.*

W.2.4. Wikipedia, *Architektura von Neumanna.*

W.2.5. Wikipedia, *Babbage Charles.*

W.2.6. Wikipedia, *Język C*.

W.2.7. Wikipedia, *EDVAC - Electronic Discrete Variable Automatic Computer.*

W.2.8. Wikipedia, *EDSAC - Electronic Delay Storage Automatic Calculator.*

W.2.9. Wikipedia, *ENIAC- Electronic Numerical Integrator And Computer.*

W.2.10. Wikipedia, *Enigma.*

W.2.11. Wikipedia, *Hollerith Herman.*

W.2.12. Wikipedia, *Grafika rastrowa.*

W.2.13. Wikipedia, *Grafika wektorowa.*

W.2.14. Wikipedia, *Intel.*

W.2.15. Wikipedia, *Jednostka zarządzania pamięcią – MMU Memory Management Unit.*

W.2.16. Wikipedia, Logika temporalna.

W.2.17. Wikipedia, *Magistrala – Bus.*

W.2.18. Wikipedia, *Modele barw*.

W.2.19. Wikipedia, Neumann John.

W.2.20. Wikipedia, *RAM - Random Access Memory.*

W.2.21. Wikipedia, *Pamięć masowa - Mass storage memory.*

W.2.22. Wikipedia, *Pamięć wirtualna.*

W.2.23. Wikipedia, *PNG.*

W.2.24. Wikipedia, *Procesor–procesor.*

W.2.25. Wikipedia, *Procesor wielordzeniowy – Multi-core procesor.*

W.2.26. Wikipedia, *Przetwarzanie potokowe.*

W.2.27. Wikipedia, Język *Simula*.

W.2.28. Wikipedia, Język Smalltalk.

W.2.29. Wikipedia, Grafika *SVG.*

W.2.30. Wikipedia, *Turing Alan Mathison.*

W.2.31. Wikipedia, *Zortrax.*

W.2.32. Wikipedia, Paradygmat programowania.

W.2.33. Wikipedia, Zastosowania technologii informatycznych.

W.3.1. Wilkosz Witold, *Człowiek stwarza naukę*, Kraków 1946.

W.3.2. Wilkosz Witold, *Liczę i myślę*, Warszawa 1951.

W.4.1. Wilkes Maurice, Wheeler David, Gill Stanley, *The Preparation of Programs for an Electronic Digital Computer* (original 1951); reprinted with new introduction by Martin Campbell-Kelly; 198 pp.; illus; biblio; bios; index; ISBN 0-262-23118-2. Available through Charles Babbage Institute.

W.5.1. Wirth Niklaus, *Algorytmy + struktury danych = programy* – wydanie 6 (tłumaczenie z angielskiego), Wydawnictwa Naukowo-Techniczne, Warszawa 2002.

W.6.1. Wojtuszkiewicz Krzysztof, *Urządzenia techniki komputerowej* - część 1 i 2, Wydawnictwo Naukowe PWN, Warszawa 2009.

W.7.1. Woodcock Jim, Davies Jim, *Using Z – specification, refinement, and proof*, Internet 1999.

W.8.1. W3C –*World Wide Web Consorcium*, www.w3.org .

Z.1.1. Zabrodzki J. i inni, *Grafika komputerowa, metody i narzędzia*, Wydawnictwo Naukowo-Techniczne, Warszawa 1994.