SZKOŁA JESIENNA PTI

WSPÓŁCZESNE KIERUNKI ROZWOJU INFORMATYKI

Organizowana przez POLSKIE TOWARZYSTWO INFORMATYCZNE

Do użytku
wewnętrznego

Mrągowo, 3 — 7 listopad 1986 r.

SZKOŁA JESIENNA PTI

WSPÓŁCZESNE KIERUNKI ROZWOJU INFORMATYKI

Organizowana przez POLSKIE TOWARZYSTWO INFORMATYCZNE

Mrągowo, 3 - 7 listopad 1986 r.

SPIS TREŚCI

Str.

1. Mgr Włodzimierz Grudziński (Uniwersytet Warszawski)
Zastosowanie Prologu w bazach danych 7

2. Prof. Cliff B. Jones (University of Manchester)
Specifications and Programs 33

3. Doc. Antoni Kreczmar (Uniwersytet Warszawski)
Języki obiektowo-zorientowane 49

4. Doc. Jan Madey (Uniwersytet Warszawski)
Problematyka systemów operacyjnych na przykładzie systemu UNIX 69

5. Dr Stefan Sokołowski (Uniwersytet Gdański)
Programowanie funkcjonalne 91

6. Doc. Stanisław Waligórski (Uniwersytet Warszawski)
Informatyka szkolna i jej problemy 109

7. Doc. Jan Zabrodzki (Politechnika Warszawska)
Cyfrowa generacja obrazów 123

SŁOWO WSTĘPNE

Głównym celem Jesiennych Szkół Polskiego Towarzystwa Informatycznego jest dokonywanie krytycznego przeglądu wybranych kierunków rozwoju informatyki i jej zastosowań na świecie. Oczywiście ze względu na ograniczony czas w programie Szkoły i niewielką objętość niniejszego tomu, nie jesteśmy w stanie przedstawić żadnego z omawianych tematów w sposób umożliwiający jego natychmiastowe podjęcie na gruncie zawodowym. Ale też szkoły PTI nie mają charakteru kursów dokształcających. Chcielibyśmy, aby nasze listopadowe spotkania odgrywały przede wszystkim rolę inspirującą, a także aby były forum, na którym spotykają się ludzie z różnych środowisk zawodowych i o różnych poglądach na informatykę. Aby były miejscem wymiany tych poglądów oraz nawiązywania kontaktów.

W trakcie trwania Pierwszej Szkoły PTI w Rydzynie niektórzy uczestnicy zwracali uwagę, że program Szkoły zbyt daleko odbiegał od zagadnień, z którymi informatycy w naszym kraju spotykają się na co dzień. I w pewnym sensie rzeczywiście tak było. W Rydzynie mówiono o systemach współbieżnych, o układach bardzo wysokiej skali integracji, o sformalizowanych środkach wspomagających produkcję oprogramowania, o systemach wielomikroprocesorowych. Czy jednak wspomnianej uwagi nie należało sformułować inaczej? Czy to nie nasza codzienność zbyt jest odległa od nurtów wyznaczających kierunki rozwoju informatyki?! Choć więc nie zawsze mamy na tę codzienność wystarczający wpływ, to przecież kropla drąży skałę i nie kto inny jak my informatycy powinniśmy temu drążeniu nadawać kierunek.

Wzorem ubiegłych Szkół również i tym razem wszystkie wykłady przygotowane zostały specjalnie na zaproszenie organizatorów. Jak zwykle prosiliśmy autorów, aby były to teksty o charakterze przeglądowym i popularyzacyjnym. Pozostawiamy ocenę czytelnikom, czy udało nam się ten cel osiągnąć. Co do wyboru tematyki, to kierowaliśmy się opiniami uczestników poprzednich spotkań wyrażanymi w ankietach oraz oczywiście dostępnością wykładowców. Większość wykładów stanowi pewną kontynuację tematyki obecnej już w programach poprzednich szkół. Trzy z nich natomiast, dotyczą tematyki nowej.

Zachęceni udanym eksperymentem zeszłorocznym zaprosiliśmy ponownie wykładowcę zagranicznego. Jest nim Cliff B. Jones, profesor Uniwersytetu w Manchester, znany w Polsce jako autor książki "Konstruowanie Oprogramowania Metodą Systematyczną", WNT 1984. Nasz gość, podobnie jak występujący w zeszłym roku Dines Bjørner, jest jednym z twórców i popularyzatorów formalnej metody projektowania systemów softwareowych VDM. Metodzie tej i jej zastosowaniom poświęcony też został jego wykład. Ponieważ wygłoszony on będzie w języku angielskim, w tym też języku publikujemy jego tekst.

Warszawa, wrzesień 1986 Andrzej Blikle

Jesienna Szkoła PTI
Mrągowo, listopad 1986

Zastosowanie Prologu w bazach danych

Włodzimierz Grudziński
Instytut Informatyki
Uniwersytetu Warszawskiego
PKiN, p. 850
00-901 Warszawa

1. Wstęp

W ostatnich latach nastąpił szybki rozwój komputeryzacji różnych obszarów działalności gospodarczej, administracyjnej, dydaktycznej, wojskowej i naukowej. Nowe potrzeby użytkowników oraz rozmaitość oferowanego sprzętu powodują wzrost wymagań stawianych oprogramowaniu. Dotyczy to także systemów baz danych coraz częściej wykorzystywanych w innych zastosowaniach niż tradycyjnie pojmowane przetwarzanie danych, np. wspomaganie projektowania, automatyzacja prac biurowych, tzw. systemy eksperckie (ang. expert systems), itp. Po pierwszym okresie, w którym w znacznej mierze poradzono sobie z problemami sprawnego zarządzania dużymi zbiorami danych, efektywnej realizacji żądań użytkowników, niezawodności oraz ochrony dużych baz danych, poszukuje się lepszych sposobów modelowania świata rzeczywistego oraz zwiększenia "inteligencji" systemów zarządzania bazami danych. Do nowych wymagań stawianych przed tymi systemami należą między innymi:

- zapewnienie obsługi obiektów o złożonej strukturze (szczególnie utrudnione w systemach relacyjnych),

- obsługa niesformatowanej informacji, takiej jak: teksty, obrazy, mapy, sygnały dźwiękowe itp.,
- przechowywanie pewnej wiedzy o modelowanym świecie wyrażonej zazwyczaj w postaci ogólnych reguł a nie jedynie faktów (danych),
- umiejętność choćby prostego wnioskowania opartego na takich regułach i faktach.

Wiele problemów związanych z dwoma ostatnimi zagadnieniami wchodzi w zakres badań nad sztuczną inteligencją. Wśród różnych podejść do ich rozwiązania jednym z najbardziej obiecujących jest wykorzystanie wyników oraz narzędzi stosowanych w nowej, powstałej w początkach lat 70 dziedzinie informatyki zwanej programowaniem w logice. Zainteresowanie nią wzrosło gwałtownie w ciągu ostatnich kilku lat, zwłaszcza po tym, gdy japoński projekt opracowania tzw. piątej generacji komputerów przyjął programowanie w logice jako podstawę ich oprogramowania systemowego.

Celem tej pracy jest przedstawienie dotychczas osiągniętych wyników przy zastosowaniu programowania w logice, a w szczególności Prologu, najpopularniejszego języka programowania w logice, do baz danych. Mówiąc o bazach danych mamy na myśli relacyjny model danych i relacyjne systemy baz danych aczkolwiek większość rozważań odnosi się w równej mierze do innych modeli i systemów. Praca ma charakter opisowy oraz przeglądowy, stąd niezbyt formalna prezentacja tematu. O wielu zagadnieniach jedynie wspomnimy, odsyłając zainteresowanego czytelnika do odpowiedniej literatury.

W rozdziałach 2 i 3 omówimy w zarysie idee leżące u podstaw programowania w logice oraz jego związki z bazami danych. W rozdziałach 4 i 5 w podobny sposób przedstawimy krótkie wprowadzenie do Prologu. Rozdziały 6 i 7 poświęcone są odpowiednio dwóm sposobom wykorzystania Prologu w bazach danych: jako języka zapytań i jako języka implementacji baz danych.

2. Programowanie w logice

Przez wiele lat logika pierwszego rzędu stosowana była w badaniach nad sztuczną inteligencją jako formalizm do reprezentacji wiedzy i rozwiązywania problemów. Również w bazach danych, a szczególnie w teorii relacyjnch baz danych, wykorzystywano logikę w pracach dotyczących języków zapytań, wyznaczania odpowiedzi na zapytania, projektowania schematów baz danych, tworzenia i utrzymywania więzów integralnościowych itp. [Ull 82], [GMN 84]. R.Kowalski [Kow 74] pokazał, że logikę można traktować jako język programowania. Programy mogą być wyrażone jako zdania logiki pierwszego rzędu uzupełnione odpowiednimi informacjami opisującymi sterowanie, to znaczy sposób wykonania programu. Obliczenie polega wówczas na kierowanym poszukiwaniu dowodu.

Można pokazać, że każde zdanie logiki pierwszego rzędu może być przekształcone na zbiór klauzul postaci

$$A_1 \vee \ldots \vee A_n \leftarrow B_1, \ldots, B_m \qquad (1)$$

gdzie v i , oznaczają odpowiednio alternatywę i koniunkcję a <-- implikację. A_i i B_j $(i=1,\ldots,n,\ j=1,\ldots m)$ są literałami postaci $p(t_1,\ldots,t_k)$, gdzie p jest nazwą predykatu o krotności k a każdy term t_i może być stałą, zmienną lub funktorem, po którym następuje lista termów (argumentów). Przykładami literałów są

para(1,2), xyz(X,Y,Z), lista(Pocz,Kon),
równanie(X, plus(1, minus(5,a))), ~ab(a,b)

~ oznacza zaprzeczenie.

O języku programowania w logice mówi się czasami jako o podzbiorze logiki pierwszego rzędu zwanym klauzulami Horna. Klauzule Horna mają postać

$$A \leftarrow B_1, \ldots, B_m \qquad (2)$$

B_i nazywane są przesłankami a A wnioskiem.

Można powiedzieć, że predykat odpowiada pewnej relacji zachodzącej między obiektami pewnego świata. Przyporządkowanie obiektom nazw predykatów, funktorów, stałych i zmiennych wyznacza interpretację zdania (klauzuli) logiki. Tylko wówczas można mówić o jego znaczeniu.

Zakłada się również, że wszystkie zmienne występujące w klauzulach są w zasięgu kwantyfikatora ogólnego. Na przykład klauzula

lata(X) v biega(X) <-- ptak(X) (3)

oznacza, że wszystkie ptaki potrafią latać lub biegać. Predykatom i zmiennej X nadaliśmy tutaj oczywistą interpretację. We wszystkich przykładach stosujemy przyjętą na ogół w Prologu konwencję rozpoczynania nazw zmiennych dużą literą a stałych, predykatów i funktorów małą.

Mając daną interpretację można stwierdzić czy spełnia ona zbiór klauzul wyznaczając ich wartość, tzn. sprawdzając ich prawdziwość lub fałszywość. Mówimy również, że klauzula w wynika ze zbioru klauzul W, jeśli każda interpretacja spełniająca W spełnia również w, co oznacza się W $\models$ w. Na przykład ze zbioru zdań

ptak(orzeł), ~biega(orzeł)

i klauzuli (3) wynika

lata(orzeł)

Do formalnego wnioskowania umożliwiającego wyprowadzanie zdań ze zbioru innych zdań jedynie na podstawie ich syntaktycznej postaci służą reguły wnioskowania. Do najbardziej znanych należą np. modus ponens czy prawa De Morgana. Języki programowania w logice wykorzystują zazwyczaj jedynie jedną regułę wnioskowania zwaną rezolucją [Rob 65]. Klauzule Horna mogą przybierać jedną z trzech postaci:

a) implikacji: A <-- $B_1, \ldots, B_n$

b) stwierdzenia: A
gdy nie ma przesłanek,

c) zaprzeczenia: $\sim(B_1, \ldots, B_n)$

gdy brak jest wniosku.

Jest to równoważne zapisowi $\leftarrow B_1, \ldots, B_n$.

Reguła rezolucji mówi, że z dwóch zdań postaci

$\sim X \vee Y$ (czyli $Y \leftarrow X$) i $X \vee Z$

wynika zdanie

$$Y \vee Z$$

Przyjrzyjmy się najprostszym i szczególnie interesującym nas przypadkom:

1. z dwóch klauzul

   zaprzeczenia S1: $\sim A$

   oraz implikacji S2: $A \leftarrow B$

   zastosowanie rezolucji wyprowadzi nową klauzulę

   s: $\sim B$

2. z klauzul

   zaprzeczenia S1: $\sim A$

   i stwierdzenia S2: A

   otrzymamy klauzulę pustą, co oznacza, że doprowadziliśmy do sprzeczności.

Oczywiście w obu przypadkach predykaty A w S1 i S2 muszą być identyczne.

Przypadek 2 znany jest w matematyce jako sprowadzenie do sprzeczności i tak właśnie stosuje się rezolucję w językach programowania w logice. Zatem chcąc wykazać $W \models w$ należy wziąć zaprzeczenie w i znaleźć w W klauzulę, której wniosek jest identyczny z w lub da się do takiego sprowadzić za pomocą odpowiedniego przekształcenia. Polega ono na znalezieniu właściwego podstawienia na zmienne, zwanego unifikatorem głównym. Istnieje algorytm, nazywany algorytmem unifikacji lub uzgadniania, który dla każdego skończonego zbioru wyrażeń znajduje jego unifikator główny lub sygnalizuje, że jest to niemożliwe. Za jego pomocą można również uzyskać ciąg kolejno znajdowanych unifikatorów głównych. Następnie należy zastosować regułę rezolucji do tej wybranej klauzuli i do $\sim w$. Otrzymana klauzula dodawana jest do zbioru W i powyższy proces powtarza się dopóty, dopóki nie otrzyma się klauzuli pustej lub stwierdzi niemożliwość znalezienia odpowiedniej klauzuli w W. Wynika stąd,

że ta metoda pozwala nie tylko wykazać sprzeczność pewnego zbioru klauzul, czyli odpowiedzieć na pytanie o prawdziwość w, lecz także znaleźć obiekty powodujące tę sprzeczność. Oczywiście należy przyjąć ustalony sposób wyboru klauzul ze zbioru W. W ten sposób można dokonywać formalnego wnioskowania. Załóżmy na przykład, że dane są klauzule

ojciec(jan,piotr), ojciec(piotr,anna), ojciec(jan,ewa)

Można wówczas zadawać następujące rodzaje pytań:

- czy Jan jest ojcem Piotra?

Aby uzyskać odpowiedź należy sprawdzić czy z powyższego zbioru klauzul wynika

~ojciec(jan,piotr)

Stosując metodę rezolucji otrzymuje się natychmiast klauzulę pustą. To znaczy, że powyższa klauzula jest sprzeczna ze zbiorem danych klauzul, czyli odpowiedzią na pytanie jest "tak";

- kto jest ojcem Piotra?

Tym razem badane jest wynikanie klauzuli

~ojciec(X,piotr)

gdzie X oznacza zmienną. Sprzeczność osiągnie się natychmiast, gdy przyjętym podstawieniem będzie X/jan co stanowi odpowiedź na powyższe pytanie. Jest to jedyne rozwiązanie;

- można również otrzymać imiona wszystkich ojców i dzieci zadając pytanie

<-- ojciec(X,Y)

co jak pamiętamy jest równoważne ~ojciec(X,Y).

Programowaniu w logice poświęcona jest książka [Hog 84] a bardziej teoretyczne podejście zawarte jest w [Llo 84].

3. Programowanie w logice a bazy danych

Relacyjny model danych oraz relacyjne języki zapytań stanowią w oczywisty sposób podzbiór logiki pierwszego rzędu. Dla baz danych charakterystyczne jest, że zawierają duży zbiór niezinterpretowanych danych, przechowywanych w rekordach i plikach wraz ze strukturami pomocniczymi, np. indeksami, oraz schemat bazy danych (na ogół znacznie mniejszy) reprezentujący pewną wiedzę o modelowanym świecie (tzw. metadane). Powodem tego rozdzielenia jest troska o niezależność danych od ich opisu oraz o efektywność wyszukiwania. Schemat zawiera pewien rodzaj reguł dedukcyjnych lub perspektyw (schematów zewnętrznych w nomenklaturze ANSI/SPARC), które pomagają użytkownikom interpretować dane i wyprowadzać z nich nowe informacje. Struktury danych i więzy integralnościowe schematu pojęciowego zapewniają właściwe sterowanie operacjami w bazie danych. Każda z funkcji systemu zarządzania bazami danych opisywana jest za pomocą innego języka (języki definicji danych, języki zapytań, itd.). Zaimplementowanie ogólnych reguł dedukcyjnych oraz sprawdzanie więzów integralnościowych wymaga pisania specjalnych programów w macierzystym języku programowania systemu. Języki zapytań (języki manipulacji danymi) są na to zbyt ubogie. Przyjmuje się również, że rozmiary schematu są niewielkie - niewiele metadanych zarządza dużymi zbiorami danych. W wielu współczesnych zastosowaniach liczba metadanych rośnie tak znacznie, że to założenie przestaje obowiązywać. Coraz częściej mechanizmy tradycyjnych systemów baz danych okazują się niewystarczające. Tymczasem używając logiki pierwszego rzędu dane, reguły dedukcyjne, więzy integralnościowe i programy działające na bazie danych można wyrazić jednolicie za pomocą tego samego języka. Co więcej jest to język deklaratywny, to znaczy opisuje się w nim to co chce się osiągnąć bez podawania sposobu wykonania żądania. Zanika formalny podział na schemat, dane i programy. Dużą część danych można zatem opisać za pomocą ogólnych reguł. Na przykład w genealogicznej bazie danych z poprzedniego rozdziału zawierającej dane o ojcach i dzieciach

można zdefiniować regułę

brat(X,Y) <-- ojciec(Z,X), ojciec(Z,Y) (4)

stwierdzającą, że X jest bratem Y, jeśli mają wspólnego ojca. Chcąc się dowiedzieć czyim bratem jest Piotr należy zadać pytanie

<-- brat(piotr,X)

Odpowiedzią jest: ewa, mimo iż w bazie danych nie występuje jawnie informacja

brat(piotr,ewa)

Tego typu reguły stanowią uogólnienie perspektyw z modelu relacyjnego, gdyż można korzystać w nich z rekursji, np. w regule

przodek(X,Y) <-- ojciec(X,Y)
przodek(X,Y) <-- ojciec(X,Z), przodek(Z,Y)

lub posługiwać się obiektami o złożonej strukturze, np. w regule znajdującej mieszkańców Opola o nazwisku identycznym z nazwą ulicy, na której mieszkają

imiennik(X) <-- opolanin(X, adres(X,Nr,Kod))

W ten sposób można zapisywać nie tylko perspektywy lecz także więzy integralnościowe, np. regułę

<-- ojciec(Y,X), ojciec(Z,X), Z=Y

mówiącą, że można mieć tylko jednego ojca. Reguły dedukcyjne wszelkiego typu mogą być również wykorzystywane do optymalizacji wykonania zapytań. Ten rodzaj optymalizacji, nazywany czasami optymalizacją semantyczną [HZ 80], [Kin 81], jest trudny do przeprowadzenia w tradycyjnych systemach baz danych.

Rekursja znacznie zwiększa siłę ekspresji w porównaniu z językami zapytań stosowanymi w systemach baz danych. Powstaje jednak nierozwiązany jak dotąd problem efektywnego wykonania takich zapytań. Z braku miejsca nie omawiamy tego zagadnienia.

Propozycje rozwiązań dla różnych klas zapytań rekurencyjnych można znaleźć np. w [HN 84], [Ull 85], [Ioa 85]. Przeglądu dotychczasowych wyników dokonano w [BR 86]. Innym źródłem bibliograficznym jest [GMN 84]. Posługując się językiem programowania w logice można również znacznie łatwiej badać i implementować bazy danych z niepełną informacją lub wartościami nijakimi (ang. null values).

Zastosowanie ogólnych reguł w miejsce pojedynczych danych zdecydowanie ułatwia ich wprowadzanie i zmianę oraz oszczędza miejsce w pamięci. Bazy danych zawierające takie reguły nazywa się często dedukcyjnymi bazami danych. Można w nich uzyskiwać nowe informacje z danych jawnie zapamiętanych w pamięci. Praca [GMN 84] stanowi doskonały przegląd tematyki związanej z zastosowaniem logiki w bazach danych i dedukcyjnymi bazami danych wraz z bogatą bibliografią. Teoretyczne podstawy takich systemów przedstawione są w [LT 85].

Potrzeba rozszerzenia możliwości systemów baz danych przestaje już budzić wątpliwości wśród ich twórców i użytkowników. Jeden ze sposobów polega na wbudowaniu nowych mechanizmów w już istniejące systemy. Nie omawiamy go tu, wspomnimy jedynie, że: propozycje rozszerzenia algebry relacji można znaleźć w [Zan 85], próby wbudowania mechanizmów wniskowania w system relacyjny INGRES w [ISW 84] i [Sto 85] a w [JLS 85] zaproponowano nową operację umożliwiającą stosowanie rekursji. W [DS 85] przedstawiono koncepcję systemu mającego zapewnić spełnienie wszystkich wymagań wymienionych we wstępie tej pracy. My skoncentrujemy się na zastosowaniu w tym celu Prologu, języka programowania w logice.

4. Wprowadzenie do Prologu

Prolog (Programmation en logique) opracowany został w Marsylii przez zespół pod kierunkiem A.Colmerauera w 1972r. Oparty jest na koncepcji programowania w logice i posługuje się metodą rezolucji. Od tego czasu powstało wiele różnych wersji i dialektów. Tworzone są również inne języki programowania w logice. Chociaż Prolog jest językiem dosyć rozpowszechnionym i ma

już sporo zastosowań nie można powiedzieć, że osiągnął swój kształt ostateczny. Dokonywane są coraz to nowe zmiany i udoskonalenia mające z jednej strony zwiększyć jego siłę wyrazu a z drugiej efektywność wykonania. Jednym ze źródeł zmian są wnioski i potrzeby wynikłe z prób stosowania go w bazach danych. Stosowana tu składnia oparta jest (za [KS 85]) na najbardziej popularnej wersji, znanej jako Prolog-10, zaimplementowanej przez D.H.D.Warrena dla komputera DEC-10. Podamy tutaj najważniejsze cechy Prologu. Pełny opis języka można znaleźć w [KS 83], gdzie omówiona jest oryginalna wersja marsylijska, lub [CM 81]. Bardziej zaawansowany materiał z nietrywialnymi przykładami i opisem implementacji Prologu zawarty jest w [KS 85].

W Prologu nie istnieje pojęcie typu danych. Można posługiwać się obiektami prostymi i złożonymi, których opisy nazywane są termami. Obiektom prostym, zazwyczaj utożsamianym ze stałymi, nie przypisuje się żadnej ustalonej interpretacji, o ile nie są użyte jako argumenty standardowych operacji arytmetycznych, porównania czy wejścia/wyjścia. Obiektami złożonymi są drzewa i listy. Termy reprzentujące takie obiekty zapisuje się w postaci nazwy, zwanej funktorem, po której w nawiasie następują termy opisujące składowe. Prolog umożliwia stosowanie dowolnych funktorów jako funktorów pozycyjnych (prefiksowych, infiksowych lub postfiksowych). Na przykład term [Głowa|Ogon] opisuje listę, gdzie Głowa odpowiada pierwszemu elementowi listy a Ogon reszcie. Elementy listy oddzielane są kropką. Lista pusta oznaczana jest []. Obiekty o nieznanej strukturze nazywane są zmiennymi a ich nazwy rozpoczynają się z dużej litery. W trakcie obliczeń zmienna może zostać ukonkretniona (związana z termem), tzn. lepiej określony staje się opis reprezentowanego przez nią obiektu. Zmienna związana z termem jest od niego nieodróżnialna. Ten term może zawierać inne zmienne. Oto kilka przykładów termów wraz z ich interpretacją:

X	wszystkie typy obiektów.
[G|O]	wszystkie niepuste listy.
zwierzur(X,zoo(wrocław,1986))	wszystkie zwierzęta urodzone we wrocławskim Zoo w 1986 r.
[A.x]	wszystkie dwuelementowe listy o drugim elemencie równym x.

[A.A|T] wszystkie co najmniej dwuelementowe listy z identycznymi pierwszymi dwoma elementami.

Program w Prologu jest to zbiór procedur składających się z klauzul. Klauzulę tworzy nagłówek i, być może pusta, treść. Procedurą nazywa się zbiór klauzul, których nagłówki mają tę samą nazwę i liczbę parametrów. Procedurą jest na przykład

```
brat(X,Y) :- ojciec(Z,X), ojciec(Z,Y).
brat(X,Y) :- matka(Z,X), matka(Z,Y).
```

Pierwsza z tych klauzul dokładnie odpowiada klauzuli (4) z poprzedniego rozdziału. Nagłówek od treści oddzielany jest symbolem :- a koniec klauzuli oznaczany jest kropką. Klauzule o tej samej nazwie i różnej liczbie parametrów należą do różnych procedur, np.

```
ojciec(X) :- ojciec(X,Y).
```

Każdą klauzulę postaci

```
p :- q,r,s.
```

można interpretować na dwa sposoby. Pierwsza interpretacja, zwana deklaratywną lub pragmatyczną, stwierdza, że zachodzi p, jeśli zachodzi q, r i s. Druga, nazywana proceduralną, mówi, że aby obliczyć p należy najpierw obliczyć q, r i s (czyli wywołać te procedury). Na przykład procedura

```
student(X) :- matura(X,_), pktegz(X,P), P>16.
```

definiuje studenta jako posiadacza matury, który otrzymał więcej niż 16 punktów egzaminacyjnych. Zakładamy, że drugi parametr procedury matura określa nazwę szkoły średniej. Podkreślenie oznacza zmienną anonimową. Stosuje się ją wówczas, gdy pewne obiekty nie są konieczne i nie ma potrzeby odwoływania się do nich. Można również powiedzieć, że aby stwierdzić czy X jest studentem należy sprawdzić, czy ukończył szkołę średnią, a następnie obliczyć liczbę otrzymanych przez niego punktów i sprawdzić czy jest większa od 16. Jeśli do tej procedury dołączymy klauzulę

```
student(X) :- matura(X,_), olimpijczyk(X).
```

wówczas do poprzedniej deklaratywnej interpretacji dodamy: "lub posiada maturę i jest finalistą olimpiady". Różne klauzule tej samej procedury oznaczają różne możliwe drogi obliczenia.

Z powyższego przykładu widoczna jest przyjęta w Prologu kolejność wykonywania procedur, od lewej do prawej w treści klauzuli. Klauzule wybierane są w kolejności ich pojawienia się w programie. Oczywiście do wykonania wybierana jest klauzula, której nagłówek można uzgodnić z wywołaniem procedury. Każda klauzula wybierana jest tylko raz dla jednego wywołania procedury. Gdy pewne wywołanie procedury zawiedzie, tzn. algorytm unifikacji nie znajdzie obiektów spełniających opisywaną przez nią relację (nie znajdzie unifikatora), to wykonywany jest nawrót. Polega on na anulowaniu części uzyskanych ostatnio wyników i odtworzeniu historii obliczeń aż do napotkania procedury mającej jeszcze nieuaktywnione klauzule o nagłówkach uzgadnialnych z jej wywołaniem. Prześledźmy to na przykładzie genealogicznej bazy danych. Przypuśćmy, że program w Prologu składa się z następujących procedur:

```
ojciec(jan,piotr).
ojciec(piotr,anna).
ojciec(jan,ewa).

mężczyzna(jan).
mężczyzna(piotr).

brat(X,Y) :- ojciec(Z,X), ojciec(Z,Y), mężczyzna(X).
brat(X,Y) :- matka(Z,X), matka(Z,Y), mężczyzna(X).
```

i załóżmy, że chcemy znaleźć brata Piotra. Piszemy zatem

```
:- brat(piotr,X), mężczyzna(X), write(X).
```

Najpierw wywołana zostanie pierwsza z klauzul procedury brat. Algorytm unifikacji przypisze parametrom formalnym odpowiednie termy: X/piotr, Y/X. Otrzymujemy wówczas

:- ojciec(Z,piotr), ojciec(Z,X), mężczyzna(piotr),
mężczyzna(X), write(X).

(Podkreśleniem wyróżniamy aktualnie wywoływaną procedurę.) Następuje pierwsze ukonkretnienie zmiennej Z, gdyż podstawiając Z/jan można dokonać unifikacji z pierwszą klauzulą procedury ojciec. Mamy zatem

:- ojciec(jan,piotr), ojciec(jan,X), mężczyzna(piotr),
mężczyzna(X), write(X).

Zmienna X wiązana jest teraz ze stałą ewa, gdyż druga klauzula procedury ojciec ewidentnie nie da się uzgodnić. Jednak po sprawdzeniu, że Piotr jest mężczyzną zawodzi wywołanie procedury mężczyzna(ewa). Wówczas następuje nawrót do wywołania ojciec(jan,X) anulujący poprzednie ukonkretnienie zmiennej X. To wywołanie również zawodzi (nie ma już niezbadanych klauzul procedury ojciec a klauzula może być wybrana tylko raz dla jednego wywołania procedury). Dokonuje się nawrotu do poprzedniego wywołania procedury ojciec i próbuje znaleźć innego ojca Piotra (czyli inne ukonkretnienie zmiennej Z w ojciec(Z,piotr)). Oczywiście nie ma takiego, a zatem zawodzi pierwsza klauzula procedury brat. Możliwe jest jednak uzgodnienie z wywołaniem brat(piotr,X) nagłówka drugiej klauzuli. Mamy wówczas

:- matka(Z,piotr), matka(Z,X), mężczyzna(piotr),
mężczyzna(X), write(X).

lecz algorytm unifikacji nie znajduje procedury matka, co w większości implementacji Prologu traktowane jest jako niepowodzenie. Próba nawrotu również kończy się niepowodzeniem, to była ostatnia klauzula procedury brat. Zatem w odpowiedzi otrzymujemy informację "nie ma".

Unieważnienie części dotychczasowych wyników w czasie nawrotu nie odnosi się do procedur standardowych powodujących efekty uboczne takie jak wypisanie na ekran czy też dynamiczne dodawanie i usuwanie klauzuli. Wyników działania takich procedur nie da się anulować.

Przyjęto na ogół, że w przypadku znalezienia odpowiedzi użytkownik może zażądać następnej wprowadzając znak ;. Jeśli jednak jego zamiarem jest znalezienie wszystkich odpowiedzi, np. wszystkich dzieci Jana, może napisać

```
:- ojciec(jan,X), write(X), fail.
```

fail jest standardowo przyjętą nazwą procedury powodującej niepowodzenie. Zatem zawsze po znalezieniu odpowiedzi nastąpi nawrót, a gdy wywołanie procedury ojciec zawiedzie otrzyma informację "nie ma więcej". Łatwo zauważyć, że automatyczne nawracanie po każdym niepowodzeniu może powodować znaczne marnotrawienie czasu i pamięci. Choćby w powyższym przykładzie, gdy próbuje się znaleźć drugiego ojca Piotra. Aby dać programiście możliwość pewnego sterowania nawrotami wprowadzono operator odcięcia oznaczany !. Jego wystąpienie powoduje niemożliwość dokonania nawrotu do procedur występujących przed nim i tym samym oznacza ostateczną akceptację wszystkich osiągniętych wcześniej wyników. Na przykład procedurę sprawdzającą czy dwa zbiory są rozłączne można zapisać jako

```
rozłączne(Zb1,Zb2) :- elzb(El,Zb1), elzb(El,Zb2), !, fail.
rozłączne(Zb1,Zb2).
```

Widać, że jeśli istnieje element należący do obu zbiorów Zb1 i Zb2 to nie ma potrzeby dalszego sprawdzania (zakładamy, że jest to robione przez procedurę elzb). Odcięcie zapobiega nawrotowi i procedura zawodzi. Bez odcięcia nawrót powodowałby niepotrzebne sprawdzenie wszystkich elementów zbioru Zb1. Jeśli takiego elementu nie ma to nie dojdzie do wywołania odcięcia i po nawrocie nastąpi wywołanie drugiej, zawsze spełnionej klauzuli. Odcięcie psuje jednak deklaratywną interpretację procedury i znacznie zmniejsza jej czytelność. W wielu przypadkach wystarczy zastosować standardową procedurę not, która kończy się sukcesem gdy zawodzi próba wywołania procedury będącej jej parametrem. Na przykład mając daną procedurę sprawdzającą czy dwa zbiory mają część wspólną, procedurę rozłączne można zdefiniować jako

```
rozłączne(Zb1,Zb2) :- not(mającześćwspólną(Zb1,Zb2)).
```

Jedną z charakterystycznych cech Prologu jest możliwość stosowania tej samej procedury w różnych celach. Typowym przykładem jest procedura append

```
append([], L, L).
append([Pocz|Kon], L, [Pocz|Konl]) :- append(Kon, L, Konl).
```

Mówi ona, że lista L powstaje z połączenia L z listą pustą a lista [Pocz|Konl] powstaje z połączenia list L i [Pocz|Kon], jeśli Konl jest złączeniem list L i Kon.
Teraz w zależności od sposobu wywołania append można listy łączyć lub rozdzielać. Na przykład

```
:- append([a.b], [c], L), write(L).
```

da w wyniku listę [a.b.c], a wywołanie

```
:- append(P, K, [a.b.c]), write(P), write('&'),
                write(K), nl, fail.
```

da w wyniku

```
[] & [a.b.c]
[a] & [b.c]
[a.b] & [c]
[a.b.c] & []
```

5. Baza danych w Prologu

W tym rozdziale chcemy pokazać możliwości i ograniczenia Prologu jako języka do definiowania i działania na bazach danych. Z poprzednich rozdziałów widać w jaki sposób można zaimplementować w Prologu prostą relacyjną bazę danych. Dane, czyli krotki, opisywane są przez klauzule unarne, tj. klauzule zawierające jedynie nagłówek, takie jak

```
ojciec(jan,piotr).
mężczyzna(jan).
```

Zauważmy, że nie zawierają one zmiennych chociaż, np. klauzula unarna ojciec(X,jan) może reprezentować informację o tym, że

Jan ma ojca. Tego typu klauzule wprowadzają do bazy danych niepełną informację i nie zajmujemy się tu nimi. Zauważmy jednak jak naturalnie można je zapisać, choć oczywiście nie zmniejsza to trudności w ich interpretacji.

W związku z brakiem typów danych Prolog nie daje zazwyczaj bezpośrednich możliwości definiowania dziedzin atrybutów relacji. Można to zrobić korzystając z dostępnych procedur standardowych, np. procedury integer(T) sprawdzającej czy T jest liczbą całkowitą. Zmusza to do bardziej kłopotliwego sposobu operowania bazą danych, gdyż trzeba napisać programy sprawdzające poprawność danych. Jednak istnieją już wersje Prologu, np. Turbo Prolog na IBM PC, w których wprowadzono proste typy danych. Definicja relacji i krotek w Turbo Prologu wygląda następująco:

```
domains
        osoba = symbol
        wiek = integer
predicates
        mężczyzna(osoba, wiek)
        ojciec(osoba,osoba)
        kobieta(osoba)
clauses
        mężczyzna(piotr,35).
        mężczyzna(jan,68).
        ojciec(jan,piotr).
        kobieta(maria).
        kobieta(X) :- not(mężczyzna(X,_).
```

Zwróćmy uwagę na definicję relacji kobieta. Jedynie jedna krotka została wprowadzona jawnie i tylko ją otrzyma sie w odpowiedzi na żądanie znalezienia wszystkich kobiet. Taka definicja wygodna jest wówczas, gdy chce się sprawdzać czy ktoś jest kobietą. Przyjmuje się przy tym założenie o tzw. zamkniętości świata bazy danych [Rei 78], że prawdziwe są jedynie fakty zapisane w bazie danych (klauzule występujące w programie) lub dające się z nich wyprowadzić. W ten właśnie sposób zaimplementowana jest w Prologu operacja zaprzeczenia. Wynika stąd, że na pytanie czy Antoni jest kobietą otrzymamy

odpowiedź twierdzącą. Nie jest wymieniony wśród mężczyzn, a zatem przez domniemanie przyjmuje się, że jest kobietą, zgodnie z definicją tej relacji.

Następne przykłady pokażą w jaki sposób można zapisać podstawowe operacje algebry relacji (jeszcze jedna interpretacja procedury). Załóżmy, że mamy zdefiniowane dwie dwuargumentowe relacje R(A,B) i S(C,D).

rzut	$\pi_A(R)$	R1(A) :- R(A,B).
selekcja i rzut	$\pi_B\, \sigma_{A=5}(R)$	R1(B) :- R(5,B).
selekcja	$\sigma_{C<D}(S)$	S1(C,D) :- S(C,D), C<D.
złączenie	$R \underset{A=D}{\bowtie} S$	RS(A,B,C) :- R(A,B), S(C,A).

Koniunkcje warunków selekcji zapisuje się w tej samej klauzuli a dysjunkcje jako kolejne klauzule. Inne podstawowe operacje przedstawia się równie prosto: sumę jako kolejne warianty procedur R i S a różnicę jako

różnica(X,Y) :- R(X,Y), not(S(X,Y)).

Warto zauważyć, że gdy dopuści się występowanie klauzul unarnych ze zmiennymi powyższa procedura da nieprawidłowe rezultaty.

Aktualizacji można dokonywać za pomocą standardowych procedur assert i retract odpowiednio wstawiającej i usuwającej dowolną klauzulę. Efekty tych procedur nie są anulowane w czasie nawrotu co utrudnia nieco ich stosowanie, zmienić bowiem trzeba sposób myślenia o programie. Maleje także czytelność programu. Istnieją propozycje wprowadzenia innych operacji umożliwiających aktualizację klauzul (bazy danych) i nienaruszających logicznej semantyki języka [War 84].

Z licznych przykładów omówionych w poprzednich rozdziałach widać, że perspektywy i zapytania definiuje się w ten sam sposób. Jednak tak zdefiniowane perspektywy nie spełniają swoich funkcji ochronnych, co jest ważne w systemach baz danych, ponieważ aktualizacja perspektywy nie wpływa na relacje, z których jest

zbudowana. Użytkownik bezpośrednio modyfikuje bazę danych. Na przykład aktualizacja relacji (perspektywy) brat

```
:-assert(brat(adam,anna)).
```

nie spowoduje dodania żadnej klauzuli do procedury ojciec, lecz pojawi się trzecia klauzula procedury brat. Z drugiej strony widać, że jest to dobry sposób definiowania wyjątków od reguł. Dużą zaletą jest możliwość wykorzystywania rekursji do tworzenia znacznie bardziej skomplikowanych perspektyw niż jest to możliwe w systemach baz danych

W Prologu można również w naturalny sposób zapisywać i utrzymywać więzy integralnościowe. Na przykład reguła o posiadaniu tylko jednego ojca

```
jedenojciec(X,Y) :- ojciec(Z,Y), !, fail.
jedenojciec(X,Y).
```

może wchodzić w skład procedury

```
poprojciec(X,Y) :- jedenojciec(X,Y), !, ......
```

która sprawdzana jest przy wprowadzaniu nowych elementów relacji ojciec chociażby w następujący sposób

```
insert(ojciec(X,Y)) :- poprojciec(X,Y), assert(ojciec(X,Y)).
```

Tego rodzaju więzy jak procedura jedenojciec opisują po prostu zależności funkcyjne. W [Men 85] pokazano jak można je zaimplementować w Prologu, aby odcięcie wstawiane było automatycznie przez system w sposób niewidoczny dla użytkownika.

Oczywiście nie należy zapominać, że Prolog jest normalnym, choć niekonwencjonalnym językiem programowania a nie wyspecjalizowanym językiem programowania baz danych, mimo iż ma wiele ich możliwości. Prolog znakomicie nadaje się na przykład do pisania translatorów, co dobrze ilustrują implementacje relacyjnych języków zapytań Query-by-Example [NAW 83] i Sequela. Ta ostatnia wykonana została w IIUW. Cały program (500 linii w

Prologu !) implementujący nieco okrojoną wersję Sequela, Toy-Sequel, zamieszczony jest w [KS 85]. Z drugiej strony dzięki swojej zwięzłości i mocy świetnie nadaje się do szybkiego tworzenia i badania prototypów baz danych oraz zastosowań i systemów interakcyjnych zawierających wiele reguł i niezbyt dużo danych. Z tego powodu coraz chętniej stosowany jest do tworzenia systemów eksperckich. Istnieją jednak dwie poważne niedogodności w stosowaniu Prologu do implementacji średnich nawet, np. rzędu kilku megabajtów, baz danych. Klauzule prologowe przechowywane są w pamięci operacyjnej, co znacznie ogranicza rozmiary bazy danych, a ponadto wyszukiwane są pojedynczo (jedna klauzula jest uzgadniana przy jednym odwołaniu do procedury), co czyni go bardzo wolnym. W wielu wersjach Prologu próbuje się złagodzić te ograniczenia. Wprowadzono operacje zwracające zbiory klauzul (krotek), np. setof i bagof, w MU-Prologu i Turbo Prologu można przechowywać dane na plikach, w [War 81] pokazano jak za pomocą prostego indeksowania można zmienić standardowy sposób wyszukiwania klauzul, itd. Oczywiście należy pamiętać o tym, że od systemu baz danych wymaga się między innymi zapewnienia odtwarzania i ochrony danych, współbieżnego dostępu wielu użytkowników, itp. Więcej o tego typu zmianach w Prologu ukierunkowanych na zastosowanie go jako języka implementacji baz danych powiemy w rozdz. 7.

6. Prolog jako język zapytań

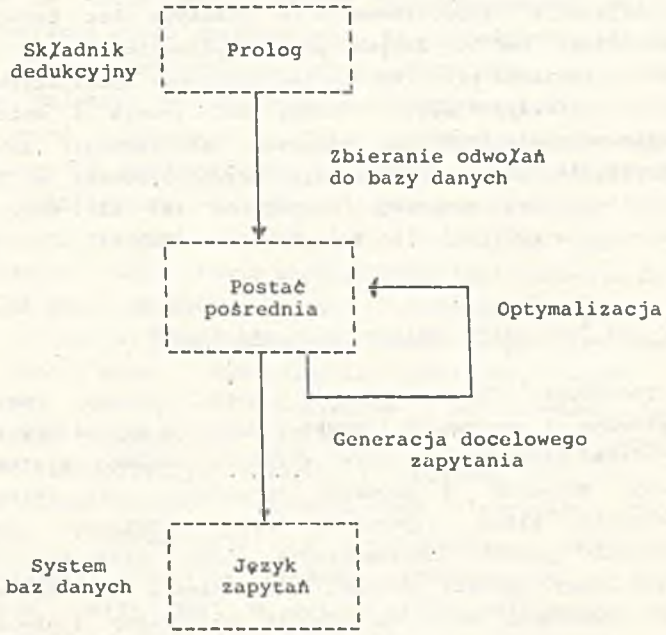
Dedukcyjne możliwości Prologu coraz częściej próbuje się wykorzystać w zastosowaniach wymagających większej liczby danych. Najpopularniejszą metodą jesy połączenie go, jako tzw. składnika dedukcyjnego, z istniejącym systemem zarządzania bazami danych. Składnik dedukcyjny (od tej pory oznaczany przez SD) zarządza regułami dedukcyjnymi odnoszącymi się do danych utrzymywanych przez system baz danych. Służy także do komunikacji z użytkownikami. Zwolennicy tego podejścia twierdzą, że oba łączone systemy mają tak różne cele i mechanizmy ich realizacji, iż próba stworzenia jednolitej całości nie może dać zadowalających rezultatów. Takie połączenie może być dokonane w sposób "luźny" lub "ścisły". Pierwszy z nich polega na wyekstrahowaniu odpowiedniego fragmentu bazy danych i przeniesieniu go do pamięci

zarządzanej przez SD (tzn. staje się częścią systemu napisanego w Prologu). Taką operację wykonuje się przed rozpoczęciem korzystania z SD. Wyszukanie odpowiednich danych zapewnia system bazy danych. Należy jednak dokonać ich konwersji do postaci wymaganej przez Prolog. Poważną wadą jest to, że nie wydaje się możliwe, aby proces wyboru fragmentu bazy danych do skopiowania mógł być, oprócz bardzo prostych zastosowań, zautomatyzowany. Inną wadą jest statyczny charakter danych. Każda zmiana dokonana w bazie danych wymaga powtórzenia operacji kopiowania. Niewątpliwą zaletą jest łatwość stworzenia takiego połączenia, ponieważ nie wymaga ono zmian w żadnej ze składowych. Oczywiście pod warunkiem, że kopiowane dane mieszczą się w pamięci operacyjnej. W przeciwnym przypadku należy zaimplementować w SD obsługę danych przechowywanych na plikach co znacznie zmniejsza atrakcyjność tego podejścia.

Sposób drugi polega na takim połączeniu obu składowych, aby baza danych stanowiła rozszerzenie SD. To znaczy krotki (rekordy) w bazie danych traktowane są jako klauzule unarne procedur (predykatów) z SD odpowiadających właściwym relacjom zdefiniowanym w schemacie pojęciowym. Konsekwencją naturalnego korzystania z takiego połączenia, często nazywanego podejściem interpretacyjnym, jest odwoływanie się do systemu baz danych za każdym razem, gdy nastąpi uaktywnienie takiej klauzuli unarnej. Oznacza to obciążenie go bardzo dużą liczbą niezależnych żądań poddawanych standardowemu procesowi kompilacji, optymalizacji, szeregowania, itp. Stwarza to olbrzymi narzut czasowy nawet w przypadku prostych zapytań. Innym problemem jest niemożliwość bezpośredniego przełożenia skomplikowanych pytań (procedur) wyrażonych w Prologu, np. rekurencyjnych, na język zapytań systemu baz danych. Proponowane rozwiązanie, tzw. podejście kompilacyjne, postuluje stworzenie języka pośredniego, służącego do komunikacji między Prologiem a systemem baz danych. Architekturę takiego połączenia przedstawiono na rys. 1.

Prolog wstrzymuje realizację odwołań do bazy danych tak długo jak to możliwe. Oznacza to, że procedury tworzące reguły dedukcyjne wykonywane są dopóty, dopóki nie pozostaną jedynie odwołania do bazy danych. Takie żądanie użytkownika wyrażone w Prologu przetwarzane jest na postać pośrednią, w której odwołania

do pojedynczych klauzul (krotek) grupowane są i zamieniane na żądania odnoszące się do ich zbiorów, np. relacji. Językiem pośrednim może być na przykład podzbiór Prologu (bez zmiennych i z odwołaniami tylko do procedur odpowiadających relacjom bazy danych) [JCV 84]. Następnie dokonuje się optymalizacji m.in. usuwając redundantne żądania. Wykorzystuje się przy tym więzy integralnościowe z SD a być może również ze schematu bazy danych. Decyduje się tu również czy wyniki zapytania mają być zapamiętane co jest bardzo ważne szczególnie przy przetwarzaniu zapytań rekurencyjnych. Ostatnim etapem jest przetłumaczenie otrzymanego zbioru żądań na język zapytań systemu baz danych.



Rys. 1

Takie podejście daje, dzięki własnościom Prologu, znacznie większe możliwości optymalizacji niż jest to robione w systemach baz danych. Szczególnie ważne, a także trudne, jest to w przypadku zapytań rekurencyjnych, ponieważ problemy, o

których wspomniano w rozdz. 3, nie zależą od sposobu połączenia Prologu z bazą danych. Dobra optymalizacja wymaga także pobrania odpowiedniej informacji ze schematu bazy danych, co nie jest wielkim problemem, gdy przyjmie się tradycyjne założenie o jego niewielkich rozmiarach. W przeciwnym przypadku a także, gdy otrzymane wyniki pośrednie zapytań są zbyt duże powstaje problem zarządzania tymi danymi. Można odsyłać je do systemu baz danych tworząc oddzielną bazę danych. Powoduje to jednak znaczne zwiększenie wzajemnych odwołań między obu składowymi. Innym rozwiązaniem jest odpowiednie rozszerzenie możliwości Prologu. Zaletą podejścia kompilacyjnego jest oparcie się o istniejące już systemy. Pozwala to, przy pewnych ograniczeniach, dosyć szybko tworzyć różnorodne zastosowania korzystające z eksploatowanych w praktyce baz danych, bez konieczności bardzo dużych prac implementacyjnych. Była to poważna przeszkoda przy tworzeniu systemów operujących bazami wiedzy, wykorzystujących metody modelowania i wnioskowania uzyskane w badaniach nad sztuczną inteligencją. Ten sposób wykorzystania Prologu do baz danych przyjęty został w japońskim projekcie piątej generacji komputerów [KY 82]. Omawiany jest także m.in. w [JCV 84], [Li 84], [JV 84], [Yok 84], [Mar 84].

7. Prolog jako język implementacji baz danych

Podejście łączeniowe nie jest jednak powszechnie akceptowane i ma swoich krytyków. Wskazują oni między innymi na zły podział pracy między obie składowe takiego systemu, duże narzuty związane z procesem komunikacji oraz niepotrzebne powtarzanie wielu czynności, np. zapytanie dwukrotnie przechodzi proces optymalizacji. Oba podsystemy wymagają również innej postaci danych, stąd częsta ich konwersja. W sumie, twierdzą, daje to produkt niezgrabny i nieefektywny. Dąży się zatem do stworzenia jednego, mającego wszystkie potrzebne cechy systemu opartego na Prologu. Byłby on używany jako jednolity język implementacji systemu baz danych i zastosowań. Nie jest to proste zadanie, wymaga bowiem rozwiązania wielu problemów a także dużo pracy implementacyjnej. Stąd nie należy spodziewać się w najbliższej przyszłości powstania takiego, pełnosprawnego systemu. Jednakże

poczyniono już pierwsze próby w tym kierunku i przedstawiono propozycje pewnych rozwiązań.

Podstawowym zadaniem jest zapewnienie sprawnego przechowywania i wyszukiwania dużej liczby danych. Jednym ze sposobów jest wykorzystanie techniki dynamicznego kodowania mieszającego do organizacji plików klauzul. Jest to wygodne przy częstym w Prologu stosowaniu wyszukiwania z częściową odpowiedniością (ang. partial match retrieval). Tak skonstruowany system obsługi bazy danych dla Prologu (napisany jednak w CDL-2) opisano w [CG 83]. Programista w Prologu dysponuje w nim nowymi procedurami systemowymi umożliwiającymi działanie na klauzulach unarnych przechowywanych na plikach. Jednak w dalszym ciągu przetwarzane są pojedyncze klauzule co znacznie wpływa, szczególnie przy nawrotach, na efektywność wykonania zadania. W innych pracach proponuje się zastosowanie znanych metod indeksowania. Zarys takiej implementacji w Prologu wraz z mechanizmem zarządzania buforami (również w Prologu) naszkicowano w [SW 84]. Jedną z zalet tej propozycji jest możliwość indeksowania wszystkich klauzul co jest istotne, gdy występuje wiele reguł dedukcyjnych (perspektyw). Poruszono tam również problemy optymalizacji i sterowania współbieżnością w takim systemie. Tym co różni go od tradycyjnych systemów baz danych jest m.in. występowanie nawrotów, inny rodzaj aktualizacji (zastąpionej usunięciem i wstawieniem), jednolity zapis reguł dedukcyjnych (schematu) i danych stwarzający możliwośc aktualizacji jednych i drugich. Nie można zatem skopiować dobrze znanych i sprawdzonych algorytmów. Propozycje mechanizmów odtwarzania i sterowania współbieżnością uwzględniających specyfikę Prologu przedstawiono w [CDG 84]. Odtwarzanie oparte jest o metodę plików różnicowych, nazywaną także hipotetyczną bazą danych [WS 83], rozszerzoną o obsługę efektów nawracania w czasie poszukiwania odpowiedzi na zapytanie. Przy omawianiu drugiego zagadnienia rozważana jest jednolita metoda współbieżnego wykonywania programów w Prologu sekwencyjnym i Concurrent Prologu [Sha 83]. Z kilku przedyskutowanych rozwiązań wybrano wersję dwufazowego blokowania [JBB 81], zmodyfikowaną odpowiednio do potrzeb Prologu.

Zagadnienia związane ze stosowaniem Prologu w bazach danych i stworzeniem bardziej "inteligentnych" systemów zarządzania bazami danych budzą coraz większe zainteresowanie. Wiele różnych propozycji wraz z argumentacją na rzecz jednego lub drugiego podejścia przedstawiono ostatnio na dwóch dużych konferencjach [Ker 84] i [Bro 85]. W wielu ośrodkach prowadzi się intensywne badania w tej dziedzinie. Należy się spodziewać, że wkrótce pojawią się także raporty opisujące praktyczne zastosowania.

Bibliografia

[BR 86] F.Bancilhon, R.Ramakrishnan: An Amateur's Introduction to Recursive Query Processing Strategies. Proc. of ACM SIGMOD Conf. on Management of Data, Washington 1986.

[Bro 85] M.L.Brodie(ed): Proc. of Workshop on Large Scale Knowledge Base and Reasoning Systems, Islamorada, 1985.

[CDG 84] M.Carey, D.J.DeWitt, G.Graete: Mechanisms for Concurrency Control and Recovery in Prolog - A Proposal. W [Ker 84].

[CG 83] J.Chomicki, W.Grudziński: A Database Support System for Prolog. Proc. of Logic Programming Workshop, Albufeira 1983.

[CM 81] W.F.Cloksin, C.S.Mellish: Programming in Prolog. Springer Verlag 1981.

[DS 85] U.Dayal, J.M.Smith: PROBE: A Knowledge-Oriented Database Management System. W [Bro 85].

[GMN 84] H.Gallaire, J.Minker, J-M.Nicolas: Logic and Databases: A Deductive Approach. Computing Surveys, V16, N2, 1984.

[HN 84] L.J.Henshen, S.A.Naqvi: On Compiling Queries in Recursive First-Order Databases. JACM V31, N1, 1984.

[Hog 84] C.J.Hogger: Introduction to Logic Programming, Academic Press 1984.

[HZ 80] M.Hammer, S.Zdonik: Knowledge-based Query Processing. Proc. of 6th Conf. of VLDB, Montreal 1980.

[Ioa 85] Y.E.Ioannidis: A Time Bound on The Materialization of Some Recursively Defined Views. Proc. of 11th Conf. of VLDB, Stockholm 1985.

[ISW 84] Y.E.Ioannidis, L.L.Shinkle, E.Wong: Enhancing INGRES with Deductive Power. W [Ker 84].

[JBB 81] J.Jordan, J.Banerjee, R.Batman: Precision Locks. Proc. of ACM SIGMOD Conf. on Management of Data 1981.

[JCV 84] M.Jarke, J.Clifford, Y.Vassiliou: An Optimizing Prolog Front-End to a Relational Query System. Proc. of ACM SIGMOD Conf. on Management of Data, Boston 1984.

[JLS 85] M.Jarke, V.Linnemann, J.W.Sdhmidt: Data Constructors: on the Integration of Rules and Relations. Proc. of 11th Conf. of VLDB, Stockholm 1985.

[JV 84] M.Jarke, Y.Vassiliou: Coupling Expert Systems with Database Management Systems, w W.Reitman(ed): Artificial Intelligence Applications for Business, Ablex, Norwood 1984.

[Ker 84] L.Kerschberg (ed): Proc. of 1th Int. Workshop on Expert Database Systems, Kiawah Island 1984.

[Kin 81] J.J.King: A System for Semantic Query Optimization in Relational Databases, Proc. of 7th Conf. of VLDB, Cannes 1981.

[Kow 74] R.A.Kowalski: Predicate Logic as Programming Language. Proc. of IFIP Congress, North Holland 1974.

[KS 83] F.Klużniak, S.Szpakowicz: Prolog, WNT, Warszawa 1983.

[KS 85] F.Klużniak, S.Szpakowicz: Prolog for Programmers, Academic Press 1985.

[KY 82] S.Kunifuji, H.Yokota: Prolog and Relational Database for Fifth Generation Computer Systems. W J-M.Nicolas (ed): Proc. of Workshop on Logical Bases for Data Bases, Toulouse 1982.

[Li 84] D.Li: A Prolog Database System, Research Institute Press, England 1984.

[Llo 84] J.W.Lloyd: Foundation of Logic Programming. Springer Verlag 1984.

[LT 85] J.W.Lloyd, R.W.Topor: A Basis for Deductive Database Systems. Journal of Logic Programming, V2, N2, 1985.

[Mar 84] G.Marque-Pucheu i inni: Interfacing Prolog and Relational Data Base Management Systems. W G.Gardarin, E.Gelenbe (eds): New Applications of Data Bases, Academic Press 1984.

[Men 85] A.O.Mendelzon: Funcional Dependencies in Logic Programs. Proc. of 11th Conf. of VLDB, Stockholm 1985

[NAW 83] J.C.Neves, S.O.Anderson, M.H.Williams: A Prolog Implementation of Query-by-Example. H.J.Schneider (ed): Proc. of 7th Int. Computing Symposium, Nurnberg 1983.

[Rei 78] R.Reiter: On Closed World Databases. W H.Gallaire, J.Minker (eds): Logic and Databases, Plenum Press 1978.

[Rob 65] J.A.Robinson: A Machine Oriented Logic Based on The Resolution Principle. JACM, V1, N4, 1965.

[Sha 83] E.Y.Shapiro: A Subset of Cocurrent Prolog and Its Interpreter. TR-003, ICOT, Tokyo 1983.

[Sto 85] M.Stonebraker: Inference in Database Systems Using Lazy Triggers. W [Bro 85].

[SW 84] E.Sciore, D.S.Warren: Towards an Integrated Database - Prolog System. W [Ker 84].

[Ull 82] J.D.Ullman: Principles of Database Systems. Computer Science Press 1982.

[Ull 85] J.D.Ullman: Implementation of Logical Query Language for Databases. ACM TODS, V10, N3, 1985.

[War 81] D.H.D.Warren: Efficient Processing of Interactive Relational Data Base Queries Expressed in Logic. Proc. of 7th Conf. of VLDB, Cannes 1981.

[War 84]. D.S.Warren: Database Updates in Pure Prolog. Proc. of Int. Conf. on Fifth Generation Computer Systems, Tokyo 1984.

[WS 83] J.Woodfill, M.Stonebraker: An Implementation of Hypothetical Relations. Proc. of 9th Conf. of VLDB, 1983.

[Yok 84] H.Yokota i inni: An Enhanced Inference Mechanism for Generating Relational Algebra Queries. Proc of ACM SIGACT/SIGMOD Conf. on Principles of Database Systems 1984.

[Zan 85] C.Zaniolo: The Representation and Deductive Retrieval of Complex Objects. Proc. of 11th Conf. of VLDB, Stockholm 1985.

Specifications and Programs

C.B. Jones
Department of Computer Science
University of Manchester
M13 9PL, United Kingdom

September 7, 1986

Abstract

A number of issues concerning the concepts needed in a specification language are discussed. The essential distinctions between specification and implementation must be borne in mind even where a design is recorded in a mixture of specification and implementation languages. The discussion here focusses on the *VDM* approach and its reliance on proof obligations in design steps of data reification and operation decomposition. This approach is contrasted to approaches in which the implementation and specification language are united into one.

1 Introduction

In some respects, this paper is an extended abstract for [19]: it presents an overview of the so-called *VDM* ("Vienna Development Method") as applied to program development. (*VDM* has also been used extensively on programming languages — see [4]).

There are three more or less distinct approaches to the formal development of programs. Each approach starts with a formal statement of the required function of the program and uses formally provable steps to link the final program to the specification. The approaches and their emphases are:

- specification/design/verification: the specification is written in a distinct specification language but the design (and eventual code) are written in a normal implementation language; correctness of the design is established by discharging defined proof obligations. *VDM* uses this approach.

- transformation: the "specification" is given as an executable function but one where clarity is considered to be paramount (the execution of the function is likely to be extremely

inefficient); an acceptable implementation is created by a series of syntactic transformations (with some of which are associated the obligation to prove applicability). The best known example of this approach is the CIP project (see [8]).

- constructive mathematics: here the specification is taken as the statement of a theorem from whose constructive proof, can be extracted a program (cf. [9]).

VDM uses mainly model-oriented specifications. These are contrasted to property-oriented specifications in the next section and the respective rôles of the two methods are discussed. Operations are specified by post-conditions of initial and final states as well as (separate) pre-conditions.

The section on Abstract Models shows how model-oriented specifications are useful in capturing the architecture of a system. An extended example is used in the section on Development Steps to illustrate the systematic design aspects of *VDM*. (The examples are given without formal proofs.) A final section contrasts *VDM* to other approaches.

2 Data Types

There are two different schools in the area of data type specification. They are referred to here as the ***property-oriented*** and ***model-oriented*** approaches. Both approaches have their uses and, if used appropriately, complement each other. Suppose that it is wished to describe a basic data type like finite sets of natural numbers. The signatures of the operators could be:

$empty: \rightarrow Set_{\mathbb{N}}$
$add: \mathbb{N} \times Set_{\mathbb{N}} \rightarrow Set_{\mathbb{N}}$
$is\text{-}empty: Set_{\mathbb{N}} \rightarrow \mathbb{B}$
$is\text{-}memb: \mathbb{N} \times Set_{\mathbb{N}} \rightarrow \mathbb{B}$

With these operators, it is possible to generate terms like:

$add(3, add(5, empty()))$

or propositions like:

$is\text{-}memb(5, (add(3, add(5, empty()))))$

It would clearly be more natural to introduce the standard infix operator symbol ($\in$) for

is-memb

and a minor extension to the way of presenting signatures would permit this. But the signature represents only the syntactic part of the description of the data type. The essential difference between property and model descriptions comes in the way that the semantics is presented. In a property-oriented description, the meaning of the operators is fixed by equations[1]. The key to the creation of these equations is to recognise — in the case in hand — that all finite sets can be *generated* by the operators *empty* and *add*. It is then straightforward to characterize those operators which deliver values in *externally visible* types (i.e. Natural numbers and Booleans) in terms of the generators. For example:

$is\text{-}empty(empty()) = \text{true}$
$is\text{-}empty(add(i, s)) = \text{false}$

captures, in some way, obvious properties of the

[1]The use of such algebraic equations gives rise to the more commonly used names for this approach: *equational specifications* or *algebraic presentations* or even *algebraic specifications*. This last is somewhat of a misnomer since it is as acceptable to present a model of rational numbers as it is to give the axioms of natural numbers in an algebraic text (cf. [22]).

is-empty

operator. Similarly, the

is-memb

operator can be described by the equations:

$is\text{-}memb(i, empty()) = \text{false}$
$is\text{-}memb(i, add(j, s)) = (i = j \vee is\text{-}memb(i, s))$

The example chosen here is very simple but it makes it possible to discuss the strengths and weaknesses of property-oriented specifications. The most obvious advantage of such a description is that it is presented without reference to any underlying, or pre-defined, data type. In fact, the rôle of a model is provided by the valid terms (*word algebra*) which can be built from the generators. A more subtle advantage derives from the fact that the whole concept is built on a branch of mathematics (i.e. Algebra) where notions relevant to data types have been studied. In particular *signatures*, *sorts*, *equations* and *models* are all of interest. The generalisation from a specific data type like Set_N to a set type which is parameterized by the type of its elements is most easily studied in the property-oriented approach.

The choice between a property and a model specification should be made on pragmatic considerations. There are, however, some technical difficulties with property-oriented descriptions which should be understood. In the Set_N example, all of the operators are *total*; had the task been to define sequences of natural numbers, the operator to take the first element of a list (*hd*) would have had to have been *partial*. Partial operators arise very often in computing and a treatment which fits the way in which they are used is essential. The first major approach to the handling of *error algebras* (cf. [12]) was less than satisfactory; more recent work (e.g. [6]) fits more closely the way in which partial operators are used.

A second area of difficulty is the question of *interpretations* of such equations[2]. The choice between *initial*, *loose* or *final* interpretations are too technical to pursue in detail here. Section 9.2 of [19] gives a brief overview of how either extra operators or equations are necessary in the final and initial approaches respectively in order to ensure the appropriate identifications. For a fuller treatment, the reader is referred to a textbook such as [3] or [11].

A far deeper problem comes from the fundamental limit on the expressive power of a specification by properties. It has long been known that certain data types cannot be characterized by finite sets of equations. This gives rise to the need for, so-called, *hidden functions*. The relationship between these functions and a model is an interesting topic for research. For now, it is more important to observe that the presence of such hidden functions weakens the main advantage of a property-oriented specification: the ideal that a data type can be understood solely in terms of its operators (or functions) and their relationship is clearly unattainable if new functions are introduced to describe the inter-relationship.

It is now time to turn to the pragmatic questions which are likely to govern the choice between a property and a model based description of a data type. A distinction can be made between data types like Set_N which possess no obvious state and those like a database where the concept of a state affecting, and being changed by, the execution of operations[3] is pervasive. In fact, even with the example of a *Stack* — almost the standard example of a data type specification — there is a natural place for a state. It is possible to disguise this fact by presenting a signature of the form:

[2] This is also connected with the question of what happens when a convenient set of generator operators is not present.

[3] The term *operations* is used in preference to "operators" in order to emphasize the rôle of side-effects.

$empty: \rightarrow Stack$
$push: X \times Stack \rightarrow Stack$
$top: Stack \rightarrow X$
$remove: Stack \rightarrow Stack$
$is\text{-}empty: Stack \rightarrow \mathbb{B}$

But this separates the two parts (*top* and *remove*) of what is naturally a single *POP* operation which changes the state by side effect and delivers the required value as a result. There is no basic reason why property based descriptions could not be extended to handle signatures with more than one result (see, for example, [23][4]). It does, however, remove some of the elegance with which the defining equations can be presented.

The alternative, model-oriented, specification style handles the operations separately. Each operation is characterized by pre- and post-conditions in which there is no difficulty in handling state-like objects. The obvious danger presented by basing a specification on a model is that of "over-specification". This problem has been characterized in [16] and [19] as *implementation bias*: a test is given there which establishes that the underlying state exhibits no bias.

A series of operations are specified with respect to a state; the state is constructed from combinations of known types. Picking up the example of a stack, the underlying state might be defined in terms of sequences of X. The *POP* operation could then be specified:

POP () $r: X$
ext wr st : seq of X
pre $st \neq [\,]$
post $r = \text{hd}\ \overleftarrow{st} \wedge st = \text{tl}\ \overleftarrow{st}$

The ext clause identifies the non-local objects to which the operation has access. In this case, there is only one variable to be considered since the state is so simple. In larger examples, listing only those variables required goes some way to solving the so-called "frame problem". Furthermore, distinguishing between read-only access (rd) and read-write access (wr) can also clarify the potential effect of an operation. The pre-condition is a predicate of a state and can be used to limit the cases in which the operation has to be applicable (here, the implementor of *POP* is invited to ignore states in which the initial st is an empty sequence). The post-condition is a predicate of two states: it specifies the relationship required between the states before and after execution of the operation. Here, then, it is necessary to distinguish two values of the same (external) variables. There are many possible conventions for doing this; in [19] the values in the old state are decorated with a hook (e.g. $\overleftarrow{st}$).

Such an operation specification could be translated into a more functional notation:

POP: seq of $X \rightarrow$ seq of $X \times X$

$$\begin{array}{l} \forall \overleftarrow{st} \in \text{seq of } X \cdot pre\text{-}OP(\overleftarrow{st}) \Rightarrow \\ \quad \exists st \in \text{seq of } X, r \in X \cdot OP(\overleftarrow{st}) = (st, r) \wedge \\ \quad \forall st \in \text{seq of } X, r \in X \cdot \\ \qquad OP(\overleftarrow{st}) = (st, r) \Rightarrow r = \text{hd}\ \overleftarrow{st} \wedge st = \text{tl}\ \overleftarrow{st} \end{array}$$

Notice, from this translation, that operation specifications require termination over the specified domain (i.e. *total correctness* is being handled). This paper uses the more schematic style of *VDM* operation specification throughout. The decision to separate the pre-condition in an operation specification is made on pragmatic grounds. Partial operations are very common in

[4] In the presence of non-determinism, the attempt to separate operations into functions which only deliver one result is invalid.

computing and the pre-condition focuses attention on the assumptions. The full power of the post-conditions becomes apparent on more complex examples. For now, the advantages are simply listed:

- the ability to specify non-deterministic (and thus under-determined) operations;
- results can often be conveniently specified by conjunctions of properties — this makes it far easier to specify, than it is to create, the result;
- the use of negation has a similar effect;
- it is often easy to specify an operation in terms of some inverse.

Both partiality and non-determinism[5] present problems in property-oriented specification techniques[6].

One disadvantage of model-oriented specifications by pre- and post-conditions is that it is possible to specify an operation which is unimplementable (e.g. producing an even prime number greater than 10). This gives rise to the first of many *proof obligations* which are an inherent part of *VDM*. An operation (e.g. *POP*) is *implementable* only if:

$$\forall \overleftarrow{st} \in \text{seq of } X \cdot pre\text{-}POP(\overleftarrow{st}) \Rightarrow$$
$$\exists st \in \text{seq of } X, r \in X \cdot post\text{-}POP(\overleftarrow{st}, st, r)$$

These particular proof obligations are not normally subject to formal proof but do provide a convenient reminder that type information, pre-condition and post-condition all combine to govern whether an operation is implementable.

3 Abstract Models

It is pointed out above that — in model-oriented specifications — operations can be considered separately. In this section, it is shown that the structure of a state can be used to study the *architecture* of a system even before the operations are considered.

Suppose that the task is to design and specify a file system. For this purpose, it is possible to ignore the internal structure of a *File* (it might be a sequence of bytes). In order to access the files, they are given names (*Name*). A trivial file system might be defined on states which contain only:

$$Trivial = \text{map } Name \text{ to } File$$

(The map objects, like sets and sequences, are basic ways of building composite objects in *VDM*.) On such a state, it would be possible to define operations to *CREATE*, *DELETE* and *COPY* files. But it is also necessary to observe what cannot be done. From the properties of maps, it follows that no two files can have the same name. Thus, in this trivial system, there is no support for different users to be given different name spaces. The system is not rich enough and this can be seen from the state even before operation specifications are written.

Separate name spaces could be created by nested directories. The state for such an enriched system could be defined:

$$Nestedfs = Directory$$

[5] An interesting approach of *partial interpretations* is described in [6]. This handles under-determined but not non-deterministic functions. The need for this latter, even in a deterministic implementation, comes from the change of equality at different levels of abstraction.

[6] See, however, [23].

$Directory = \text{map } Name \text{ to } Node$

$Node = Directory \cup File$

Such a system would allow different users to employ the same names. The directory structure is, in many respects, like that of Unix-like systems: the *Node* concept reflects the way in which files and directories can occur in the same directory.

Here again, it would be possible to define operations on *Nestedfs*. But it is also wise to check what cannot be done in this state. It is not possible to share the same *File* via different path names (sequences of names). If it were wished to permit a change via one path name to affect access via other path names, the state would again have to be extended.

There is a relatively standard way of introducing such sharing patterns into specifications. An intermediate link (in this case, *Fid*) is introduced[7]:

$$\begin{aligned} Sharedfs :: \quad & root : Directory \\ & filem : \text{map } Fid \text{ to } File \end{aligned}$$

$Directory = \text{map } Name \text{ to } Node$

$Node = Directory \cup Fid$

A single file can now be shared as in the following object:

$$\begin{aligned} & mk\text{-}Sharedfs(\{id_1 \mapsto fid_1, \\ & \quad id_2 \mapsto \{id_1 \mapsto fid_2, id_2 \mapsto fid_1\}\}, \\ & \quad \{fid_1 \mapsto file_a, fid_2 \mapsto file_b\}) \end{aligned}$$

It is now possible to define operations on this state. One operation can be defined to show the contents of a directory:

$Dirstatus = \text{map } Name \text{ to } \{FILE, DIR\}$

$$\begin{aligned} & SHOW \quad () \; r : Dirstatus \\ & \text{ext rd } d : Directory \\ & \text{post } r = \{nm \mapsto (\text{if } d(nm) \in Directory \text{ then } DIR \text{ else } FILE) \mid nm \in \text{dom } d\} \end{aligned}$$

Another operation might create a new directory within an existing one:

$$\begin{aligned} & MKDIR \quad (n : Name) \\ & \text{ext wr } d : Directory \\ & \text{pre } n \notin \text{dom } d \\ & \text{post } d = \overleftarrow{d} \cup \{n \mapsto \{\}\} \end{aligned}$$

An operation to insert a new file might be defined:

$$\begin{aligned} & MKFILE \quad (n : Name, f : File) \\ & \text{ext} \quad \text{wr } d : Directory \\ & \quad \text{wr } fm : \text{map } Fid \text{ to } File \\ & \text{post } \exists fid \in Fid \cdot \\ & \qquad fid \notin \text{dom } \overleftarrow{fm} \wedge d = \overleftarrow{d} \cup \{n \mapsto fid\} \wedge fm = \overleftarrow{fm} \cup \{fid \mapsto f\} \end{aligned}$$

---

[7] The *VDM* composite object notation is somewhat like Pascal records. In this case:

$$\begin{aligned} & Sharedfs = \\ & \quad \{mk\text{-}Sharedfs(root, filem) \mid root \in Directory \wedge filem \in \text{map } Fid \text{ to } File\} \end{aligned}$$

The interested reader should be able to define other operations (e.g. for deletion) at the level of a *Directory*.

One of the reasons that the work on property-oriented specifications has been important is that convenient ways of structuring specifications have been studied. Section 7.4 of [19] employs a technique for *promoting* operations from one data type to another by *operation qoutation*. This technique can be used to apply operations on a single directory to the whole directory structure. For this purpose, the state might be extended with components which — for example — contain the current path.

This directory example shows how the architecture of a system can be studied via its state. Other examples in [19] show how a system like virtual storage can be studied at different levels of abstraction in order to bring out different concepts at each level. Section 8.3 of the same reference shows how the same ideas can be used to describe interfaces. In [5] similar ideas are applied to problems of programming languages (see also [4]).

4 Development Steps

A specification of a single operation contains a pre- and a post-condition. As pointed out above, this makes it possible to define partial and non-deterministic operations. Intuitively, it should be acceptable for an implementation to terminate on more inputs than are required. (i.e. have a bigger domain — or be defined on more input values) or to produce some subset of the permitted answers for any required input (i.e. be more determined). An implementation is written in some implementation language and it is therefore necessary to have some common way of discussing the meaning of both specifications and of programs. One convenient model is to define both in terms of the set of states over which termination is required and the meaning relation which defines the possible results:

$$(S, R)$$

where S is a subset of the set of states, say Σ, and R is a relation on Σ.

It is clear that a given *pre*, *post* pair can be translated into this form by:

$$\{\{\sigma \in \Sigma \mid pre(\sigma)\},$$
$$\{(\overleftarrow{\sigma}, \sigma) \in \Sigma \times \Sigma \mid post(\overleftarrow{\sigma}, \sigma)\})$$

The ideas of denotational semantics can be used to express programming language constructs in terms of the same model (see, for example, [17]).

In terms of the set/relation pairs, it is possible to define precisely the notion of *satisfaction* (sat), which is described intuitively above:

$$(S_1, R_1) \text{ sat } (S_2, R_2) \;\Leftrightarrow\; S_2 \subseteq S_1 \wedge S_2 \lhd R_1 \subseteq R_2$$

This sat relation provides the basis against which steps of development must be shown to be correct.

There are two things which make specifications far shorter than programs: they use data objects (e.g. maps) which are not present in most programming languages and they use post-conditions which do not show how to compute a result. Both of these specification "tricks" have to be removed in the design of a program. The realization of the data object is handled in steps of *data reification*; the development of control constructs to satisfy post-conditions is handled by steps of *operation decomposition*. In the development of any significant system, development will take place in many steps. Experience with *VDM* suggests that the early, or high-level, design stages concern data reification and the later, low-level, steps concern operation decomposition. Both of these sorts of steps are illustrated on the example below and the *proof obligations* necessary to establish correctness of such steps are explained in terms of the example.

It is important to see the rôle of *compositionality* in a development method. In outline, the idea is that a decision at one step of development cannot be affected by subsequent decisions. This is, perhaps, easiest to see in terms of steps of operation decomposition. Starting with a specification, say *OP*, this might be decomposed into two sub-operations *OPA* and *OPB* with the design decision that they are to be executed one after the other. The sub-operations can be specified and the design decision verified by discharging the relevant proof obligations. Subsequent development might result in a loop construct being used to realize *OPA*. In a compositional development method, this step of design can be verified without any reference to the context in which *OPA* is to be used: only its specification need be considered. Compositional development methods are not too difficult to find for sequential programs. For programs which permit interference of parallel processes, the challenge is much greater. Some work in the *VDM* framework is reported in [18]; more recent work on Temporal Logic is described in [2].

The remainder of this section illustrates the development steps via an example. The specification is a simple one which involves the storage of a set of objects of some given type — say, X. The initial state is the empty set:

$$s_0 = \{\}$$

An operation to add a new element to a set might be specified:

ADD $(e{:}X)$
ext wr s : set of X
pre $e \notin s$
post $s = \overleftarrow{s} \cup \{e\}$

To delete an element:

DELETE $(e{:}X)$
ext wr s : set of X
pre $e \in s$
post $s = \overleftarrow{s} - \{e\}$

An operation to test whether an element is present can be specified:

ISPRESENT $(e{:}X)\ r{:}\mathbb{B}$
ext rd s : set of X
post $r \Leftrightarrow e \in s$

This specification looks trivial because the chosen state objects match the problem exactly. Strictly, the implementability (see above) proof obligation should be considered for each operation. Here, the result follows immediately from totality of the set operators.

One way of building a representation for sets, which permits efficient access, is to store the elements in a *binary tree*. Such trees:

- have two (possibly nil) branches and an element (of X) at each node;
- are arranged so that all elements in the left branch of a node are less than[8] (and all elements in the right branch are greater than) the element in the node.

[8] For brevity, the ordering relation is written as $<$.

Finally, the *Setrep* data structure is defined:

$Setrep = [Node]$

$Node :: \quad lt : Setrep$
$\qquad\quad mv : X$
$\qquad\quad rt : Setrep$

where[9]:

$$inv\text{-}Node(mk\text{-}Node(lt, mv, rv)) \triangleq$$
$$(\forall lv \in retrns(lt) \cdot lv < mv) \wedge (\forall rv \in retrns(rt) \cdot mv < rv)$$

$retrns : Setrep \rightarrow \text{set of } X$

$retrns(sr) \triangleq \text{cases } sr \text{ of}$
$\qquad\qquad\qquad\qquad \text{nil} \rightarrow \{\}$
$\qquad\qquad mk\text{-}Node(lt, mv, rt) \rightarrow retrns(lt) \cup \{mv\} \cup retrns(rt)$
$\qquad\qquad \text{end}$

Defining such a representation is the first step in a *data reification*. The next step is to define the relationship between the abstract structure and the representation. For each abstract object (set of X) there are many possible representations (*Setrep*). This is a typical situation and therefore it is convenient to define the relationship between the abstraction and the representation by a function from the latter to the former (called, in *VDM*, a *retrieve function*). In this simple case, the function *retrns* is exactly what is needed. Notice that this function is total over *Setrep*. The next step in data reification is to establish the proof obligation known as *adequacy*. This requires that there is at least one representation for each element of the abstract states:

$$\forall s \in \text{set of } X \cdot \exists sr \in Setrep \cdot retrns(sr) = s$$

This could be proved formally by induction on the set generators; here an informal argument would suffice.

Having established the basic properties of the data representation, it is necessary to define each of the operations on *Setrep*. Thus, for example:

$ADD_1 \quad (e : X)$
$\text{ext wr } sr \ : \ Setrep$
$\text{pre } e \in retrns(sr)$
$\text{post } retrns(sr) = retrns(\overleftarrow{sr}) \cup \{e\}$

might be given as the specification of an operation which is intended to mirror the behaviour of *ADD*. Notice that this post-condition is non-deterministic in that there are — except in trivial cases — many possible results which would be acceptable. This illustrates the way in which non-determinism can be used to structure a design. Even if the final program would be deterministic, this post-condition makes it possible to record and justify the design decision to use binary trees whilst postponing the decision about the tree balancing algorithm.

For the sake of simplicity, a more definite post-condition is used here (the idea of *operation quotation* is employed for illustration):

$ADD_1 \quad (e : X)$
$\text{ext wr } sr \ : \ Setrep$

[9]For efficiency, such trees should also be *balanced*; for brevity, this requirement is not treated formally here.

pre $e \notin retrns(sr)$

post $\overleftarrow{sr} = \text{nil} \wedge sr = mk\text{-}Node(\text{nil}, e, \text{nil}) \vee$
$\quad \overleftarrow{sr} \in Node \wedge$
$\quad$ let $mk\text{-}Node(\overleftarrow{lt}, mv, \overleftarrow{rt}) = \overleftarrow{sr}$ in
$\quad e < mv \wedge$
$\quad\quad \exists lt \in Setrep \cdot post\text{-}ADD_1(e, \overleftarrow{lt}, lt) \wedge sr = mk\text{-}Node(lt, mv, \overleftarrow{rt}) \vee$
$\quad e > mv \wedge$
$\quad\quad \exists rt \in Setrep \cdot post\text{-}ADD_1(e, \overleftarrow{rt}, rt) \wedge sr = mk\text{-}Node(\overleftarrow{lt}, mv, rt)$

This records the essential recursion which could be used in ADD_1 but preserves the requirement that post-conditions are simply predicates (it is not possible to invoke an operation from within a predicate).

At this level of design, the obligation to prove implementability is non-trivial. That this algorithm yields an object which satisfies the data type invariant (

inv-Node

), should be proved. Thus:

$$\forall \overleftarrow{sr} \in Setrep, e \in X \cdot pre\text{-}ADD_1(e, \overleftarrow{sr}) \Rightarrow \exists sr \in Setrep \cdot post\text{-}ADD_1(e, \overleftarrow{sr}, sr)$$

Such proofs are shown formally in [19].

The model of the *DELETE* operation can be specified:

$DELETE_1$ $(e{:}X)$
ext wr sr : $Setrep$
pre $e \in retrns(sr)$
post $retrns(sr) = retrns(\overleftarrow{sr}) - \{e\}$

Here, the removal of the non-determinacy is less easy and the design of a specific recursive algorithm is left as an exercise.

The model of *ISPRESENT* can be specified as:

$ISPRESENT_1$ $(e{:}X)$ $r{:}\mathbb{B}$
ext rd sr : $Setrep$
post $\overleftarrow{sr} = \text{nil} \wedge \neg r \vee$
$\quad \overleftarrow{sr} \in Node \wedge$
$\quad$ let $mk\text{-}Node(\overleftarrow{lt}, mv, \overleftarrow{rt}) = \overleftarrow{sr}$ in
$\quad e = mv \vee$
$\quad e < mv \wedge post\text{-}ISPRESENT_1(e, lt(\overleftarrow{sr}), r) \vee$
$\quad e > mv \wedge post\text{-}ISPRESENT_1(e, rt(\overleftarrow{sr}), r)$

Each of these models must be justified with respect to the more abstract specification. The proof obligations for operation data reification are:

$$\forall sr \in Setrep \cdot pre\text{-}OP(retrns(sr)) \Rightarrow pre\text{-}OP_1(sr)$$

for the domain part of the rule; and:

$$\forall \overleftarrow{sr}, sr \in Setrep \cdot pre\text{-}OP(retrns(\overleftarrow{sr})) \wedge post\text{-}OP_1(\overleftarrow{sr}, sr) \Rightarrow$$
$$post\text{-}OP(retrns(\overleftarrow{sr}), retrns(sr))$$

for the result part. (These rules require obvious extensions to cope with arguments and results.)

In general, since data reification steps come earlier in the design process, such proofs should be undertaken rather formally. Illustrations of such proof are given in Section 8.3 of [19].

These tree-like data structures show the essential idea behind binary trees. The data structures are not, themselves, representable in most programming languages because of their recursive nature. In, for example, Pascal it would be necessary to represent such trees via pointers and objects allocated on the heap. Still within *VDM* notation, this could be described as:

$$root = [Ptr]$$

$$Heap = \text{map } Ptr \text{ to } Node_2$$

$$\begin{aligned} Node_2 :: \quad lt &: [Ptr] \\ mv &: X \\ rt &: [Ptr] \end{aligned}$$

where:

$$inv\text{-}Heap : [Ptr] \times Heap \to \mathbf{B}$$

$$inv\text{-}Heap(p, m) \quad \triangleq \quad \ldots$$

Following the pattern set above, the next step is to write a retrieve function:

$$retrsr : [Ptr] \times Heap \to Setrep$$

$$\begin{aligned} retrsr(p, h) \quad \triangleq \quad & \text{if } p = \text{nil} \\ & \text{then nil} \\ & \text{else let } mk\text{-}Node_2(lt, mv, rt) = h(p) \text{ in} \\ & \qquad mk\text{-}Node(retrsr(lt, h), mv, retrsr(rt, h)) \end{aligned}$$

At this stage, the specification of the relevant *ADD* operation is almost as detailed as code in a programming language:

$$\begin{aligned} & ADD_2 \ \ (e : X) \\ & \text{ext wr } h \ : \ Heap, \\ & \qquad \text{wr } p \ : \ Ptr \\ & \text{pre } e \notin retrns(retrsr(p, h)) \\ & \text{post } \overleftarrow{p} = \text{nil} \wedge \\ & \qquad p \notin \text{dom } h \wedge h = \overleftarrow{h} \cup \{p \mapsto mk\text{-}Node_2(\text{nil}, e, \text{nil})\} \vee \\ & \qquad \overleftarrow{p} \neq \text{nil} \wedge p = \overleftarrow{p} \wedge \\ & \qquad \text{let } mk\text{-}Node_2(\overleftarrow{lp}, mv, \overleftarrow{rp}) = \overleftarrow{h}(\overleftarrow{p}) \text{ in} \\ & \qquad e < mv \wedge \\ & \qquad (\exists h_i \in Heap, lp \in Ptr \cdot post\text{-}ADD_2(e, \overleftarrow{h}, \overleftarrow{lp}, h_i, lp_i) \wedge \\ & \qquad h = h_i \dagger \{\overleftarrow{p} \mapsto mk\text{-}Node_2(lp_i, mv, \overleftarrow{rp})\}) \vee \\ & \qquad mv < e \wedge \\ & \qquad (\exists h_i \in Heap, rp \in Ptr \cdot post\text{-}ADD_2(e, \overleftarrow{h}, \overleftarrow{rp}, h_i, rp_i) \wedge \\ & \qquad h = h_i \dagger \{\overleftarrow{p} \mapsto mk\text{-}Node_2(\overleftarrow{lp}, mv, rp_i)\})) \end{aligned}$$

The use of p as an external leaves open the possibility to pass this variable *by location* in a recursive implementation of *ADD*.

The *ISPRESENT* operation is simpler:

$$\begin{aligned} & ISPRESENT_2 \ \ (e : X, p : Ptr) \ r : \mathbf{B} \\ & \text{ext rd } h \ : \ Heap \end{aligned}$$

post $p = \text{nil} \wedge \neg r \vee$
$\quad p \neq \text{nil} \wedge$
$\quad$ let $mk\text{-}Node_2(\overleftarrow{lp}, mv, \overleftarrow{rp}) = \overleftarrow{h}(p)$ in
$\quad e = mv \wedge r \vee$
$\quad e < mv \wedge post\text{-}ISPRESENT_2(e, \overleftarrow{lp}, \overleftarrow{h}, r) \vee$
$\quad mv < e \wedge post\text{-}ISPRESENT_2(e, \overleftarrow{rp}, \overleftarrow{h}, r)$

Here, the pointer p could be passed *by value* (which is the assumed mode in a *VDM* operation specification).

Having made two steps of data reification, the representation is one which fits with Pascal. Notice that the second step did not rely in any way on the first — this is a manifestation of compositionality.

Although the post-conditions for these operations are very algorithmic, they are not themselves Pascal statements. In order to make this last step, *operation decomposition* is necessary (cf. Chapter 10 of [19] and, for a fuller account, [20]). Here, only an outline of the relevant proof obligations is given.

Writing P_i for predicates of one state and R_i for predicates of two states, the rules required are, for sequential composition:

$$\frac{\{P_1\}S_1\{P_2 \wedge R_1\},\ \{P_2\}S_2\{R_2\}}{\{P_1\}S_1; S_2\{R_1 \mid R_2\}}$$

to permit the inheritance of a pre-condition:

$$\frac{\{P\}S\{R\}}{\{P\}S\{\overleftarrow{P} \wedge R\}}$$

to permit the strengthening of a pre-condition or the weakening of a post-condition:

$$\frac{PP \Rightarrow P,\ \{P\}S\{RR\},\ RR \Rightarrow R}{\{PP\}S\{R\}}$$

and for the iterative construct:

$$\frac{\{P \wedge B\}S\{P \wedge R\}}{\{P\}\ \text{while}\ B\ \text{do}\ S\{P \wedge \neg B \wedge R^*\}}$$

with the requirement that R is well-founded and transitive (R^* is the reflexive closure of R).

These rules can be used as the proof obligations in stepwise operation decomposition. On such a small example as $ISPRESENT_2$, it is more convenient to use them as *annotations* to the final code:

$ISPRESENT\ (e{:}X)\ \mathbb{B}$
ext rd re : Ptr,
$\quad$ rd h : $Heap$

```
p: Ptr, b:B
p:= rt; b:= false
;
pre inv-Heap(p, h)
    while p ≠ 0 ∧ ¬b do
        inv inv-Heap(p, h)
            with p ↑ do
                if e = mv
                then b:= true
                else if e < mv
                    then p:= lp
                    else p:= rp
        rel ¬b ∧ (is-present(e, p, h) ⇔ is-present(e, p̄, h̄)) ∧ depth(p) < depth(p̄) ∨
            b ∧ is-present(e, p̄, h̄)
post b ⇔ is-present(e, p̄, h̄)
;
ISPRESENT:= b
```

Notice that, at various stages of development, a mixture of the programming language and the specification language are used to record the design. But that the task of specification is distinct from that of implementation.

5 Discussion

This closing section refers to various alternatives to the approach given in the body of the paper. The data refinement rule used above is incomplete in the sense that there are some things that one would like to view as representations which cannot be verified by its use. An alternative rule has been found and is described in [23,13] — a general motivation is included in [20]. The new rule is — in a sense made precise in the source papers — *complete*.

The logic used by this author since [1] caters for partial functions. Essentially, a natural deduction proof style has been provided for the symmetric "three valued" truth-tables of Lukasiewicz found in [21]. A forthcoming paper ([7]) will cover questions of the usability of various approaches to the problem of undefined terms in logical expressions. In [20] reference is made to:

- use of "existential equality" and the problem of its negation;
- use of "strong equality" and separation of definedness proofs;
- approaches due to Abrial, Blikle and Owe.

In this area, it is clear that there is need for further experimentation to establish a convenient proof style.

There are a number of important questions surrounding the relationship between specification and programming languages. In *VDM*'s steps of operation decomposition, for example, it is natural to mix programming language constructs and specifications of operations. Michel Sintzoff (cf. [24]) argues that specification and programming languages are part of a continuum. If this point is conceded, the result should not be to use a programming language as a specification language — the effect is likely to be, at best, a loss of clarity in the expression of partial and non-deterministic operations. More importantly, because of the lack of obvious algebraic properties like commutativity, it is difficult to reason about such texts and this is essential for

specifications. Experience with some attempts in this area also suggests that students who are presented with such a language will tend to over use the constructive features.

A more interesting approach is to use parts of a programming language as a subset of a specification language. One way to bring these onto a common semantic footing is to define the programming constructs by translation to predicates. Assignment statements can then be used to overcome the "frame problem" but predicates can be used where more appropriate. Both [15,14], explain approaches which have particularly elegant rules for decomposition. In neither case, however, are they compatible with the satisfaction ordering used here, nor can their notion of specification support as many distinctions concerning non-termination as here. (A very interesting approach is being developed by Jean-Raymond Abrial.) The translation to predicates provides a semantic basis but is likely to lose the link to design decisions. This could result in similar problems to those identified (for example, in [10]) in the use of "Verification Condition Generators". One of the most interesting parts of the work of [15,14] is the development of algebraic properties for the languages involved. This brings together the specification/design/verification and transformational approaches discussed in the introduction to this paper. Perhaps an alternative should be sought in which a restricted subset of predicates can be translated into programs.

Acknowledgements

The author would like to express his thanks to the organizers of the "Third Autumn School of the Polish Informatics Society" for their kind invitation to speak at Mragowo and for their permission to publish this same material elsewhere. The paper was typed, under considerable time pressure, by Mrs. Julie Hibbs. The author is also grateful to SERC for financial support.

References

[1] "A Logic Covering Undefinedness in Program Proofs", H. Barringer, J.H. Cheng and C.B. Jones, Acta Informatica, Vol 21, No. 3, pp. 251–269, 1984.

[2] "Now You May Compose Temporal Logic Splecifications", H. Barringer, R. Kuiper and A. Pnueli, Proceedings of the 16th ACM Symposium on the Theory of Computing, Washington DC, 1984.

[3] "Algorithmic Language and Program Development", F.L. Bauer and H. Wössner, Springer-Verlag, 1982.

[4] "Formal Specification and Software Development", D. Bjørner and C.B. Jones, Prentice-Hall International, 1982.

[5] "??", D. Bjørner, Proceedings of "The Second Autumn School of the Polish Informatics Society", 1985.

[6] "Partial Interpretations of Higher Order Algebraic Types", M. Broy, Preprint at Marktoberdorf Summer School, 1986.

[7] "On the Haldling of Partial Functions", J.H. Cheng and C.B. Jones, forthcoming.

[8] "The Munich Project CIP — Volume 1: The Wide Spectrum Language CIP-L", CIP Language Group, Springer-Verlag, Lecture Notes in Computer Science, Vol.183, 1985.

[9] "Implementing Mathematics with the Nuprl Proof Development System", R.L. Constable, S.F Allen, H.M. Bromley et al, Prentice-Hall Int., 1986.

[10] "A Technical Review of Four Verification Systems: Gypsy, Affirm, FDM and Revised Special", D. Craigen, August 1985.

[11] "Fundamentals fo Algebraic Specification 1: Equations and Initial Semantics", H. Ehrig and B. Mahr, in "EATCS Monographs on Theoretical Computer Science", Springer-Verlag, 1985.

[12] "Abstract Errors for Abstract Data Types", J.A. Goguen, in: "Formal Descriptions of Programming Concepts" (ed.) E.J. Neuhold, North-Holland, 1978.

[13] "Data Refinement Refined: Resume", J. He, C.A.R. Hoare and J.W. Sanders, ESOP '86, (eds.) B. Robinet and R. Wilhelm, LNCS Vol 213, Springer-Verlag, 1986.

[14] "The Logic of Programming", E.C.R. Hehner, Prentice-Hall International, 1984.

[15] "Laws of Programming: A Tutorial Paper", C.A.R. Hoare, He Jifeng, I.J. Hayes, C.C. Morgan, J.W. Sanders, I.H. Sørensen, J.M. Spivey, B.A. Sufrin and A.W. Roscoe, Oxford University Technical Monograph PRG-45, May 1985.

[16] "Implementation Bias in Constructive Specifications of Abstract Objects", C.B. Jones, 1977.

[17] "Development Methods for Computer Programs including a Notion of Interference", C.B. Jones, Oxford University, PRG-25, June 1981.

[18] "Specification and Design of (Parallel) Programs", C.B. Jones, Proceedings of IFIP '83, North-Holland Publishing Co., 1983.

[19] "Systematic Software Development using VDM", C.B. Jones, Prentice-Hall International, 1986.

[20] "Program Specification and Verification in VDM", C.B. Jones, (to be published in) Proceedings of 1986 Marktoberdorf Summer School, 1987.

[21] "Introduction to Metamathematics", S.C. Kleene, North-Holland Publishing Co. Amsterdam, 1967.

[22] "Algebra", (Second Edition), S. MacLane and G. Birkoff, Collier Macmillan International, 1979.

[23] "Non-Deterministic Data Types: Models and Implementations", T. Nipkow, Acta Informatica Vol. 22, pp. 629–661, 1986.

[24] "Expressing Program Design in a Design Calculus", M. Sintzoff, Preprint at Marktoberdorf Summer School, 1986.

Jesienna Szkoła PTI
Mrągowo, listopad 1986

JĘZYKI OBIEKTOWO-ZORIENTOWANE

doc. dr hab. Antoni Kreczmar
Instytut Informatyki UW
PKiN 8p. , 00-901 Warszawa

1. Wstęp

Profesor Andrzej Blikle zaproponował mi wygłoszenie cyku wykładów podczas jesiennej szkoły PTI. Poczułem się bardzo zaszczycony tą propozycją, nie miałem jednak żadnego ciekawego pomysłu na tematykę wykładów. Początkowo wydawało mi się, że mogę jedynie zaproponować wykład o języku programowania Loglan, którym zajmuję się jako współtwórca projektu, raportu i implementacji od około dziesięciu lat. Czy taki temat może jednak zainteresować szerokie grono informatyków? Bardzo wątpię. Po pierwsze wykłady o jednym języku programowania są z natury rzeczy bardzo nudne. Raczej przypominają szkółkę niedzielną, a nie poważne jesienne szkoły. Po drugie szczegółowy opis poszczególnych konstrukcji języka, które częstokroć różnią się nieznacznie od konstrukcji powszechnie znanych, znuży uczestników szkoły. Wreszcie wiele lat agitacji w kraju jak i zagranicą, mającej na celu upowszechnienie Loglanu, wyczerpało moją inwencję. Jednym słowem, zmuszony zaszczytnym zaproszeniem i zniechęcony Loglanem, postanowiłem poświęcić ten cykl wykładów tematyce bardziej ogólnej, jednakże niezbyt odległej od moich obecnych zainteresowań. Mianowicie, zaproponowałem profesorowi Bliklemu temat modny i zachęcający: Języki obiektowo-zorientowane. Został przyjęty. To bardzo brzydki termin, tłumaczenie bezpośrednio z angielskiego "object oriented languages". Ale niestety, my polscy informatycy cierpimy męki starając się trafić ładnym polskim terminem w odpowiednik an-

gielski (może jedynie profesor Władysław Turski, prezes PTI, nie cierpi mąk lecz znajduje zabawę intelektualno-lingwistyczną).

Wyjaśnienia wymaga zatem pojęcie języka obiektowo-zorientowanego. Historycznie pierwszym językiem obiektowo-zorientowanym była Simula-67, jakkolwiek w tamtych odległych czasach nie zdawano sobie sprawy z tej właściwości języka. Dopiero pojawienie się języka Smalltalk, który oferowany wraz z bogatym oprogramowaniem wspomagającym i na wyspecjalizowanym sprzęcie podbija świat, uświadomiło społeczności informatycznej znaczenie "obiektowości". Ostatnie lata przyniosły nowe wyniki w tej dziedzinie. W Polsce powstał język Loglan, który wzorując się na Simuli-67 istotnie wykorzystał pojęcie obiektu. Ten sam kierunek rozwoju reprezentuje Paragon, którego autorem jest Marek Sherman - co ciekawsze jest on także jednoosobowym wykonawcą całego cyku pracy, tj. od projektu do implementacji języka.

Co łączy te wszystkie języki programowania? Dlaczego mówimy, że są one obiektowo-zorientowane? Otóż ich wspólną cechą jest możliwość operowania na obiektach. Bardzo dobrze, powie uważny Czytelnik. Ale cóż to jest obiekt? Przecież ten termin nic nie mówi, tymbardziej że informatyka nie wprowadziła jeszcze na stałe definicji tego pojęcia do swego bogatego słownika. Postaramy się zatem nasz wykład rozpocząć od wyjaśnienia czym jest obiekt i jak można go użyć w językach programowania.

2. Obiekty

Obiekt jest egzemplarzem struktury utworzonej według pewnego wzorca. W informatyce przyjęto nazwać takie wzorce klasami. A zatem klasa daje wzorzec według którego można utworzyć dowolną (oczywiście skończoną) liczbę obiektów. Ich wspólną cechą jest to, że powstały według jednego wzorca, jednakże każdy taki obiekt jest unikalny, a więc inny niż pozostałe utworzone obiekty.

Związek pomiędzy obiektami a klasami przypomina świat Platona. Klasa reprezentuje wszystkie cechy podobnych przedmiotów. Klasy istniejące w idealnym świecie są wzorcami, według których powstają obiekty (przedmioty). Przypomnijmy, co pisze sam Platon w dialogu Parmenides (PWN, Warszawa 1961, 130 V-B).

" - A taką mi rzecz powiedz. Tobie się wydaje, jak mówisz, że istnieją postacie pewne, w których uczestniczą te tutaj rzeczy i stąd mają ich nazwy; na przykład te, które uczestniczą w podobieństwie, nazywają się podobne, w wielkości wielkie, a w piękności i sprawiedliwości są sprawiedliwe i piękne?

- Tak jest - mówi Sokrates.

- Nieprawdaż; albo w całej postaci, albo w jakieś części uczestniczy to, co uczestniczy? Czy może istnieć jakieś inne uczestniczenie poza tym?

- No, jakże? - powiada.

- Więc czy wydaje ci się, że cała postać jest w każdym z wielu przedmiotów, zostając jedną, czy jak?

- No, cóż przeszkadza, Parmenidesie - powiedział Sokrates - co jej przeszkadza być w nich całej?

- Więc ona, będąc czymś Jednem i tym samym w licznych przedmiotach oddzielonych od niej, będzie w nich cała tkwiła i w ten sposób gotowa być oddzielona od siebie.

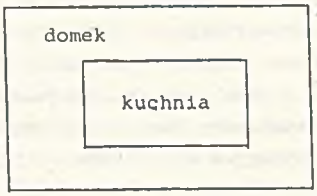
- No nie - powiada; - gdyby była taka, jak dzień, który będąc jednym i tym samym, w wielu miejscach jest równocześnie i zgoła nie jest dzięki temu oddzielony sam od siebie; może w ten sposób i każda postać może być jedną we wszystkich równocześnie i zostawać tą samą.

- Ty bardzo sympatycznie, Sokratesie - powiada - jedno i to samo równocześnie na wielu miejscach kładziesz; zupełnie jakbyś nad wieloma ludźmi jeden żagiel rozpinał i mówił, że oto jeden, a jest cały nad wieloma. Czy nie myślisz, że twierdzisz coś w tym rodzaju?"

Ta rozmowa pomiędzy Parmenidesem i Sokratesem uzmysławiła nam, jak trudno jest podać precyzyjną definicję obiektu i klasy - ponoć dialog "Parmenides" należy do najtrudniejszych dialogów Platona. Zamiast podawać zatem formalną definicję obiektu i klasy posługując się pojęciami pochodzącymi z logiki i algebry, postaramy się wprowadzić oba te pojęcia metodą przykładów i ich uogólnień. Jest to dobra klasyczna metoda, która ma tę zaletę, że nie wymaga podawania przykładów dla trudnych, formalnych definicji, których inaczej niż przez właściwie podane przykłady nie da się zrozumieć.

Spróbujmy od prostego przykładu, który duchowo nas podbuduje. Opiszmy klasę obiektów umownie nazwanych "domek". Słowo "domek" może nam się kojarzyć z wieloma pojęciami, ale każdy domek ma pewne

cechy wyróżniające go od innych przedmiotów, ma na przykład pewną liczbę izb, drzwi wejściowe, pewną liczbę okien, kuchnię, łazienkę itp. Jeżeli będziemy chcieli opisać formalnie klasę takich obiektów zwracając uwagę tylko na te cechy, które są nam potrzebne do opisu tej klasy oraz te których prawdopodobnie będziemy w przyszłości używać, to wystarczy podać w jakiejś kolejności listę takich cech wraz z nazwami (nazwy są konieczne, albowiem nazwy te pozwalają odwoływać się do pojęć). Ale co na takiej liście może się znajdować? Otóż mogą być to znowu inne klasy. Na przykład w każdym "domku" jest "kuchnia" (jest to znowu założenie umowne, wiemy że są domki bez kuchni, ale dla nich można przecież wprowadzić inną klasę). Zatem nasz "domek" będzie miał zawsze "kuchnię", a być może coś jeszcze, ale o tym później będziemy mówić. Taką klasę łatwo zilustrować na rysunku:



W języku programowania ta definicja może przybrać postać następującą:

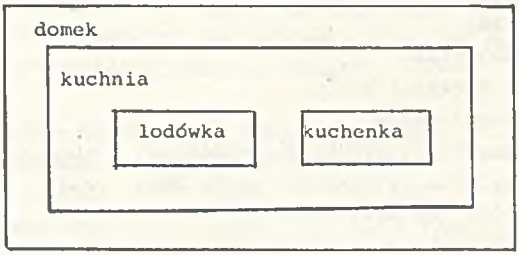
```
domek: class
  kuchnia: class
  end kuchnia;
end domek;
```

Pomiędzy słowami kluczowymi class i end umieszczamy właśnie listę cech przynależących do danej klasy. Cechy te będziemy nazywać, zgodnie z terminologią informatyczną, atrybutami. Nazwa klasy poprzedza jej definicję a kończy słowo kluczowe end, po którym znowu może pojawić się nazwa klasy (tego typu ortografia nie występuje we wszystkich wymienionych we wstępie językach programowania, jednakże znacznie zwiększa czytelność tekstu programu i zmniejsza liczbę nieporozumień). Będziemy się starali także systematycznie wzębiać kolejne wewnętrzne definicje, co nie jest już ortografią języka, ale również zwiększa znaczenie czytelności programu.

W kuchni mogą znajdować się przedmioty, które chcemy wprowadzić jako atrybuty tej klasy, np. lodówkę i kuchenkę. Nasza definicja byłaby wówczas następująca:

```
domek: class
  kuchnia: class
    lodówka: class
    end lodówka;
    kuchenka: class
    end kuchenka;
  end kuchnia;
end domek;
```

Takiej klasie odpowiadałby rysunek następujący:



Jak dotąd podaliśmy przykłady klas, których atrybutami są inne klasy. Z drugiej strony taka najbardziej wewnętrzna klasa, jak na przykład "kuchenka", nie ma w naszym przykładzie żadnych atrybutów. Oczywiście dwie różne klasy bez atrybutów nie różnią się strukturą wewnętrzną, a jedynie samą nazwą. Z taką sytuacją mamy bardzo rzadko do czynienia. Najczęściej klasy, poza atrybutami, które także są klasami, mają pewne inne atrybuty, które nie posiadają już żadnej struktury wewnętrznej. Takie atrybuty bez struktury wewnętrznej są atrybutami ilości lub jakości. Na przykład "lodówka" może mieć jako atrybuty liczby określające wymiary, napięcie znamionowe, pobór mocy, pojemność, a także kolor nazwę producenta, kierunek otwierania drzwi itp.

Wielkości liczbowe w informatyce są tzw. typami pierwotnymi "integer" i "real". Nie będziemy ich tu definiować (zgodnie z klasyczną metodą przykładów i ich uogólnień, jeden informatyk podał

drugiemu informatykowi przykład liczby "integer", a tamten to uogólnił i dokładnie zrozumiał - inny informatyk tak samo postąpił z liczbami "real" i popadł w straszne tarapaty, a dlaczego tak się stało wiedzą tylko wytrawni numerycy). Typy jakościowe, jak na przykład kolor, użytkownik może sam zdefiniować, tak jak w Pascalu, np.

```
kolor = (biały, niebieski, zielony, siny, szkarłatny)
```

Dysponując już dużym wachlarzem typów atrybutów, możemy podać nową definicję domku.

```
domek: class
  liczba_izb: integer;
  kubatura: real;
  izba: class
    powierzchnia: real
  end izba;
  kuchnia: class
    powierzchnia: real;
    lodówka: class
      szerokość, wysokość, głębokość: integer;
      napięcie, pojemność, pobór mocy: real;
      kolor_lodówki: kolor;
      kierunek_otwierania_drzwi: boolean;
      nazwa_producenta: text;
    end lodówka;
    kuchenka: class
    end kuchenka;
  end domek;
```

Możemy taką definicję rozbudowywać dalej, ale nie ma potrzeby. W przyszłości zobaczymy, jak można takie rozbudowywanie łatwo wykonać bez konieczności przepisywania klas już zbudowananych (zwiększa to przyjemność zwiększania wyposażenia naszego idealnego domku). Przypominam w tym miejscu, że mamy do czynienia z domkiem idealnym, takim bardziej platońskim. Ale na tym nie poprzestaniemy. Przecież chodzi nam o domki konkretne. Otóż takim domkom konkretnym będą odpowiadały w języku obiektowo zorientowanym właśnie obiekty.

Wyobraźmy sobie, że będziemy chcieli według wzorca klasy "do-

mek" utworzyć kilka, kilkanaście, kilkadziesiąt, kilkaset itd. domków. W świecie rzeczywistości wszystko zależy od naszych możliwości inwestycyjnych, w informatyce od naszych upodobań, albo od wymagań użytkownika. Ograniczmy się na razie do trzech domków. W programie, w którym występuje definicja klasy "domek" możemy zadeklarować trzy różne nazwy, które będą odpowiadały trzem różnym domkom:

```
domek_Tomka, domek_Romka, domek_Atomka: domek;
```

Taka deklaracja będzie mówiła, że te trzy nazwy będą mogły wskazywać na obiekty klasy "domek", i tylko na takię. Nie znaczy to, że od razu muszą wskazywać na obiekty klasy domek, czasem mogą wskazywać, a czasem nie. Otóż w momencie deklaracji nie wskazują na nic. Dopiero wówczas, gdy programista podejmie decyzję,, że dana nazwa ma wskazywać na dany obiekt, może taki obiekt utworzyć i związać go z tą nazwą. Do tego celu służy instrukcja generancji obiektu, która we wszystkich językach obiektowo-zorientowanych ma podobną postać. Na przykład poniższe trzy instrukcje generacji obiektu "domek":

```
domek_Tomka:=new domek; domek_Romka:=new domek;
        domek_Atomka:=new domek;
```

utworzą trzy egzemplarze domku, każdy związany z inną nazwą. Jest to bardzo ważny szczegół, który częstokroć niedoceniany przez programistów może prowadzić do wielu błędów. Otóż gdybyśmy próbowali w jednej instrukcji generacji związać te trzy nazwy z domkiem, jak na przykład:

```
domek_Tomka, domek_Romka, domek_Atomka:=new domek;
```

to otrzymalibyśmy jeden egzemplarz domku związany z trzema różnymi nazwami. To nie musi być błąd, czasem programista właśnie tak chce postąpić, jednakże trzeba na tę istotną różnicę zwrócić baczną uwagę - dotyczy to w szczególności początkujących obiektowych-programistów.

Te trzy obiekty klasy "domek", które już poprzednio wygenerowaliśmy, mają nieokreślone wartości atrybutów. (Tutaj mamy do czynienia dokładnie z tym samym zjawiskiem, co z nazwami obiektów jedynie zadeklarowanymi, a nie wygenerowanymi). Możemy teraz przystąpić do określenia atrybutów, na przykład:

```
domek_Tomka.liczba_izb:=5; domek_Tomka.kubatura:=3000.5;
  domek_Romka.liczba_izb:=3;
```

itd. Czyli atrybuty nieklasowe, tzn. typu nie będącego klasą, określamy za pomocą zwykłej instrukcji podstawienia. Warto w tym miejscu wspomnieć, że dostęp do atrybutu obiektu uzyskujemy poprzez nazwę, po której następuje kropka. Nazwa wskazuje na obiekt, a kropka jest znakiem interpunkcyjnym oddzielającym tę nazwę od nazwy atrybutu. Takich kropek może być zresztą w jednym wyrażeniu wiele (gdy chcemy się dostać do bardzo zagnieżdżonego atrybutu).

Aby poprawnie wykonać dostęp do atrybutu klasowego, musimy najpierw przygotować nazwę, która będzie wskazywać na taki nowoutworzony obiekt. Inaczej utworzymy obiekt bez możliwości odwołanie się do niego. Zatem w klasie "domek" możemy na przykład umieścić deklaracje nazw obiektowych:

```
domek:class
  liczba_izb:integer;
  kubatura:real;
  izby:array[ ] of izba;
  kuchnia_moja:kuchnia;
  kuchnia:class
        ...
  end kuchnia
end domek;
```

a następnie generować odpowiednie obiekty w sposób zdalny:

```
domek_Tomka.kuchnia_moja:=domek_Tomka.new kuchnia;
domek_Tomka.izby [1]:=domek_Tomka.new izba;
domek_Tomka.izby [2]:=domek Tomka.new izba;
```

itp.

Klasy mogą mieć parametry. Sposoby przekazywania parametrów w klasach są takie same jak w procedurach, nie będziemy się zatem rozpisywać na ten temat. Dla uproszczenia załóżmy, że będziemy mieli do czynienia tylko z parametrami przekazywanymi przez wartość (parametry wejściowe).

Jeżeli klasa ma parametr formalny wołany przez wartość, to w momencie generacji odpowiedni parametr aktualny określa jego wartość, tak jak ma to miejsce w przypadku wołania procedury. Przykładowo, dla naszej klasy "domek" atrybuty "liczba_izb" oraz "kubatura" możemy umieścić na liście parametrów:

```
domek:class (liczba_izb:integer, kubatura:real);
  izby:array [ ] of izba;
  ...
  end domek;
```

Wówczas generacja obiektu takiej klasy pozwala jednocześnie określić wartości tych atrybutów:

```
domek_Tomka:=new domek(5,3000,5);
domek_Romka:=new domek(3, domek_Tomka.kubatura * 2);
```

Jak dotąd klasa dawała możliwość tworzenia obiektów i wykonywania na tych obiektach pewnych czynności z zewnątrz, tj. za pomocą dostępu kropkowanego (zwanego dostępem zdalnym). To jeszcze nie wszystko, na co pozwalają języki obiektowo-zorientowane. Otóż klasa może mieć także zdefiniowany pewien ciąg akcji, które są wykonywane w momencie generacji obiektu.

Wróćmy do naszego przykładu, modnego ostatnio budownictwa jednorodzinnego. Klasę "domek" napiszemy tak, aby generacja "kuchni" i "izb" wykonywała się samoczynnie dla każdego obiektu, bez konieczności wykonywania tego z zewnątrz. Taka deklaracja może mieć następującą postać:

```
domek:class(liczba_izb:integer, kubatura:real);
  izby:array[1:liczba_izb] of izba;
  izba:class
  end izba;
  kuchnia_moja:kuchnia;
  kuchnia:class
  end kuchnia;
begin
  for i:=1 to liczba_izb do izby  i :=new izba;
  kuchnia_moja:=new kuchnia;
```

```
end domek;
```

Jeżeli teraz wygenerujemy obiekt klasy "domek", np.:

```
domek_Tomka:=new domek(5,3000.5);
```

wówczas ta instrukcja spowoduje utworzenie obiektu, przesłanie parametrów i wykonanie ciągu instrukcji wyznaczonych przez deklarację klasy. A zatem zostaną utworzone "izby" oraz "kuchnia _moja". Takie same czynności zostaną także wykonane przy każdym innym wygenerowanym obiekcie klasy "domek".

Atrybutami mogą być także procedury i funkcje. Na przykłady takich atrybutów natrafimy w dalszej części wykładu i będą one w miarę naturalne, natomiast sztuczne rozbudowywanie klasy "domek" w celu zilustrowania tego pojęcia nie będzie przekonywujące. Kończymy zatem pierwszy punkt naszego wykładu, w którym przedstawiliśmy pojęcie klasy i obiektu.

3. Dziedziczenie

Tak jak klasy i obiekty, dziedziczenie występuje we wszystkich językach obiektowo-zorientowanych. Jest to niezwykle ważne i ciekawe narzędzie. Pozwala tworzyć hierarchię klas (a więc hierarchię w świecie wzorców). Taka hierarchia ze świata wzorców przenosi się automatycznie na hierarchię w świecie obiektów.

Rozpoczniemy jak zwykle od przykładu. Niech poniższa klasa reprezentuje wzorzec dla typu pojazd:

```
vehicle:class
  dead_weight:real;
end vehicle;
```

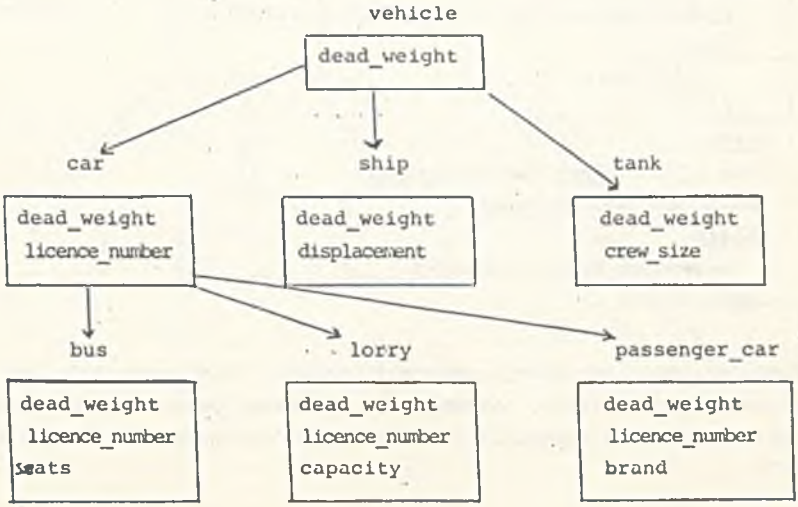
Jeżeli chcielibyśmy rozwinąć tę klasę w naturalny sposób na klasy "car", "ship", "tank" itp., nie musimy powtarzać w każdym z z tych wzorców za każdym razem atrybutów klasy "vehicle". Wystarczy skorzystać właśnie z dziedziczenia. Klasy "car", "ship", "tank" mogą dziedziczyć wszystkie cechy klasy "vehicle". Przykładowo, definicję tych klas mogą wyglądać następująco:

```
car of vehicle:class (licence_number:integer);
end car;
ship of vehicle:class
  displacement:integer;
end ship;
tank of vehicle:class
  crew_size:integer;
end tank;
```

Możemy tak postępować dalej. Mianowicie klasę "car" rozbudujemy tworząc klasy "bus", "lorry", "pass enger_car":

```
bus of car:class
  seats:integer;
end bus;
lorry of car:class
  capacity:real;
end lorry;
passenger_car of car:class (brand:text);
end passenger_car;
```

Ponieważ nasza hierarchia trochę się już skomplikowała, podsumujmy jakie atrybuty mają poszczególne klasy. Najlepiej przedstawić to na rysunku:



Strzałki ilustrują kierunek dziedziczenia. Jak widać więc atrybuty są dziedziczone i nie ma potrzeby ich definicji powtarzać w klasie dziedziczącej. Obiekty klas dziedzicznych generujemy w taki sposób, jak obiekty klas zwykłych. Przypuśćmy , że mamy następujące deklaracje:

```
T_32:tank, Mercedes_Benz:bus, Ford:passenger_car;
```

wówczas możemy utworzyć obiekty oraz określić ich atrybuty w sposób następujący:

```
T_32:=new tank;  T_32.dead_weight:=10_000; T_32.car_seize:=8;

Mercedes_Benz:=new bus (19876);
Mercedes_Benz.dead_weight:=10_000; Mercedes_Benz.seats:=89;
Ford:=new passenger_car (1111, "Granada");
```

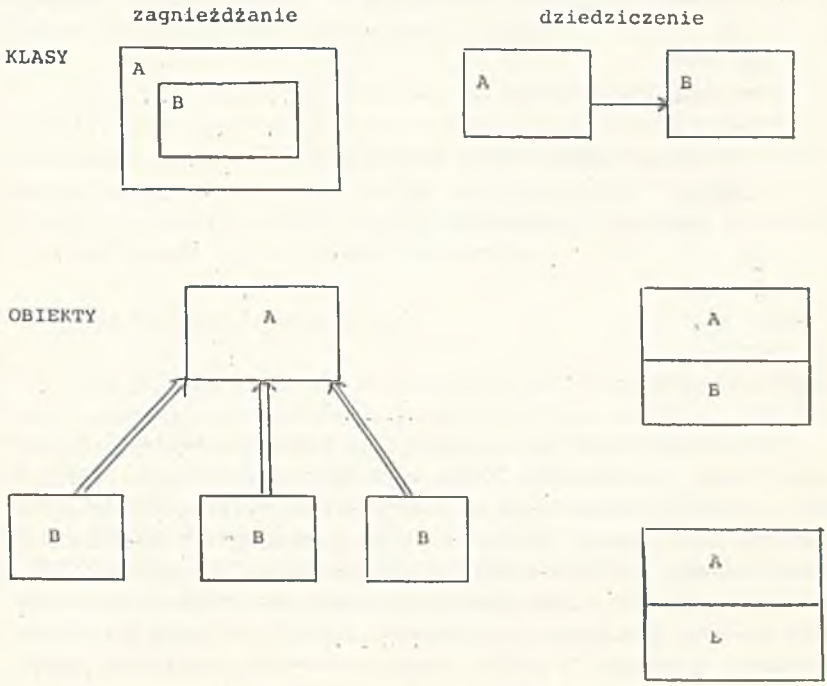
Jak sobie przypominamy, klasa może mieć określone pewne akcje. Jeżeli klasa dziedziczy klasę z akcjami, to akcje te wykonują się w trakcie generacji obiektu klasy przed wykonamiem akcji klasy właściwej. Przypuśćmy, że przed chwilą określone klasy będą miały pewne akcje (jednocześnie przerobimy trochę te deklaracje):

```
vehicle:class (dead_weight:real);
begin
  if dead_weight<0 or dead_weight > 1.0E20
  then
    call alarm
  fi;
end;
bus of car:class (seats:integer);
passenger_seats:integer;
begin
  passenger_seats:=seats-1;
end;
```

Wykonamie teraz instrukcji generacji obiektu "bus" spowoduje, po przekazaniu parametrów, sprawdzenie, czy "dead_weight" spełnia warunki brzegowe, a następnie określi wartość zmiennej "passenger_seats":

Mercedes_Benz:=new bus (10_000,19876,89);

Dziedziczenie jest formą strukturalną zupełnie inną niż zagnieżdżanie. Zagnieżdżając klasę B w klasie A decydujemy, że po wygenerowniu obiektu klasy A można wygenerować wiele obiektów klasy B zależnych od tego wygenerowanego obiektu klasy A (vide przykłady klasy "domek", gdzie generujemy wiele obiektów klasy "izba"). Z dziedziczeniem jest całkiem inaczej. Jeżeli klasa B dziedziczy klasę A, to decydujemy, że każde wygenerowanie obiektu klasy B automatycznie utworzy stowarzyszony obiekt klasy A tylko na potrzeby tego obiektu klasy B. Tworzenie takiego obiektu jest automatyczne i nie musimy dla niego stosować oddzielnie instrukcji new (vide przykład ostatni). Tę różnicę pomiędzy dziedziczeniem i zagnieżdżaniem dobrze ilustruje następujący diagram:



4. Klasy i struktury danych

Jedną z ciekawszych właściwości klasy jest możliwość wyposażenia jej w atrybuty proceduralne. Atrybuty takie mogą być wywoływane w obiektach klasy w sposób zdalny. Daje to programiście narzędzie wyśmienicie nadające się do implementowania struktur danych. Przytoczymy oklepany już przykład struktury stosu:

```
push_down:class (size:integer);
  stack:array [1:size] of object;
  top:integer;
  push:procedure (x:objest);
  begin
    if top >  size then write ("stack_overflow")
    else
      stack [top] :=x;  top:=top+1
    fi
  end push;
  pop:function:object;
  begin
    if top <=1 then  write ("empy stack")
    else
      top:=top-1; pop:=stack [top]
  fi
 end  pop;
begin
  top:=1
end push_down;
```

Klasa "push_down" ma tak zwane pole zmiennych wspólnych tj. tablicę "stack" oraz zmienną "top", oraz dwie procedury tj. "push" i "pop". Procedury te działają na wspólnym polu danych. Jeżeli teraz utworzymy obiekt klasy "push_down" możemy wywoływać w sposób zdalny obie procedury, uzyskując efekt przez nas zamierzony, tzn. możemy wstawiać i wyjmować z tak utworzonego stosu potrzebne dla naszych celów obiekty. Nie troszczymy się przy tym o wewnętrzną strukturę tej klasy. Zamknięta "w pudle" użycza nam swoich możliwości tylko poprzez "wstawione na zewnątrz" procedury. Obiektów stosu możemy utworzyć tyle, ile w programie potrzebujemy i każdy z nich będzie miał swoje niezależne wspólne pole danych:

```
store1, store2, store3: push_down; ob1, ob2, obj3:object;
n,m:integer;
  ...
  store1:=new push_down (1000);
  store2:=new push_down (200);
  store3:=new push_down (n+m);
  ...
  store1.push(ob1); store2.push(ob2);
  ob3:=store2.pop; store3.push(ob3);
  ...
```

Ze względu na specyfikę korzystania z atrybutów proceduralnych język Smalltalk przyjął dla nich nietradycyjną nazwę, mianowicie nazywają się one metodami, w odróżnieniu od zmiennych, które są zwykłymi atrybutami. Klasa wraz z metodami daje opis struktury danych, atrybuty nieproceduralne są natomiast pomocniczymi jednostkami służącymi do prawidłowego zrealizowania operacji.

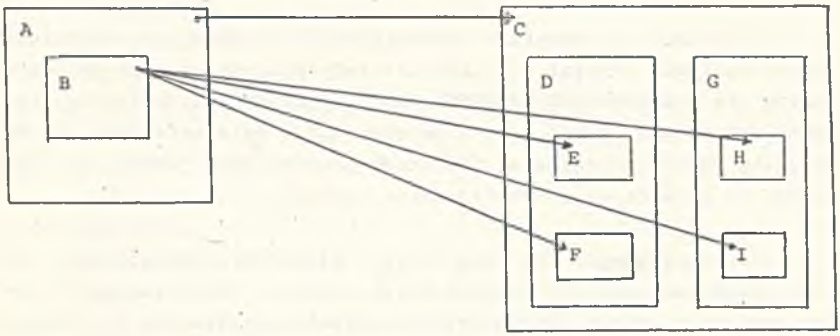
Struktury danych też mogą tworzyć hierarchię. W tworzeniu takiej hierarchi może być pomocne dziedziczenie. Jednakże nie w każdym obiektowo-zorientowanym języku wszystkie możliwości dziedziczenia są jednakowo wykorzystane. Ma to istotny wpływ właśnie na tworzenie hierarchii struktur danych. Postaramy się krótko w ostatnim punkcie naszego wykładu omówić te różnice.

5. Dziedziczenie a zagnieżdżanie

Jak już wspomnieliśmy w poprzednich punktach, dziedziczenie jest zupełnie inną formą strukturalną niż zagnieżdżanie. Jednakże muszą te dwie formy jakoś współistnieć. Niestety, nietrudno jest wprowadzić do języka wiele konstrukcji, ale znacznie trudniej jest znaleźć dla nich wszystkich efektywną implementację. Ponadto może się zdarzyć, że wprowadzone bardzo swobodne pomysły nie dają jednolitej, niesprzecznej semantyki - a to już może prowadzić do okropnych nieszczęść. Z dziedziczeniem mamy właśnie taką sytuację. Jak w tych czterech obiektowo-zorientowanych językach wybrnięto z powyższych kłopotów?

W językach Simula-67 jest zagnieżdżanie i dziedziczenie. Jednakże na dziedziczenie, właśnie ze względów implementacyjnych, na-

łożono wiele ograniczeń, Najistotniejsze dotyczy związku pomiędzy poziomem zagnieżdżenia w klasie dziedziczącej i dziedziczonej. Otóż definicja języka mówi, że poziom zagnieżdżenia obu tych klas musi być ten sam, tzn. klasa z "płytszego" poziomu zagnieżdżenia nie może być dziedziczona w "głębszym" poziomie zagnieżdżenia, jakkolwiek jest widoczna. Takie dziedziczenie nazywa się jednopoziomowym. Poniższy diagram ilustruje sytuację, gdy dziedziczenie nie spełnia takiego warunku:



Klasa A reprezentuje strukturę danych działającą na obiektach klasy B, zagnieżdżonej w kasie w klasie A (jest to dosyć typowa sytuacja). Klasa C jest rozszerzeniem klasy A, a więc ją dziedziczy. Klasy E i F są rozszerzeniem klasy B, zatem ją dziedziczą. Natomiast ze względów organizacyjnych, w klasie C chcemy powiązać ze sobą klasy E i F poprzez zagnieżdżenie w pomocniczej klasie (czy procedurze) D. Podobnie chcemy postąpić z klasami H i I, zagnieżdżonymi w klasie G. Wówczas poziom klsy B musi być inny niż poziom klas E, F, H oraz I, gdyż pomiędzy tymi poziomami musi wystąpić poziom klas D i G.

W Simuli-67 jest jeszcze inne nieprzyjemne ograniczenie. Mianowicie klasa, w której zagnieżdżono inną klasę, nie może być używana bezpośrednio do tworzenia obiektów, a jedynie jako pomocnicza klasa, którą można dziedziczyć w bloku (nie będziemy tego technicznego zjawiska szerzej wyjaśniać , bo to nie ma większego teoretycznego znaczenia). Ale konstrukcja, którą przedstawiliśmy na samym początku, z "domkiem" w którym zagnieżdżono "kuchnię", nie jest verbatim wyrażalna w Simuli.

W języku Smalltalk można zagnieżdżać jedynie procedury i funkcje (czyli metody). Język ma strukturę płaską, są klasy i dziedziczenie, nie ma natomiast możliwości zagnieżdżenia klas.

W języku Loglan zniesiono ograniczenie Simuli tak dotyczące jednopoziomowości dziedziczenia, jak również ograniczonej używalności klasy, w której zagnieżdżono inną klasę. Wszystkie możliwości dziedziczenia i zagnieżdżenia są wykorzystane.

Język Paragon poszedł jeszcze dalej. Mianowicie dziedziczenie przedstawiliśmy jako formę strukturalną pozwalającą na kolejne rozszerzenia klasy, a więc definiowana klasa może dziedziczyć bezpośrednio co najwyżej jedną klasę. Oczywiście klasa dziedziczona może być także klasą dziedziczącą, niemniej w jednym kroku dziedziczenia zakładaliśmy istnienie tylko jednego przodka. Wyobraźmy sobie bardziej uniwersalną formę strukturalną, pozwalającą na dziedziczenie jednoczesne wielu klas. Taki mechanizm dopuszcza Paragon.

Aby zilustrować taki sposób dziedziczenia, wróćmy na chwilę do przykładu z punktu 3. Otóż w przykładzie tym występują klasy "vehicle", "car", "ship", "tank" itd. Klasy "car", "ship" i "tank" dziedziczą "vehicle". Przypuśćmy teraz, że chcemy wprowadzić klasę dziedziczącą "ship" i "tank" np.:

```
monitor of ship, tank:class
  cannon:integer;
end monitor;
```

W klasie "monitor" klasa "vehicle" jest dwukrotnie dziedziczona, raz za pośrednictwem "ship", a drugi raz za pośrednictwem "tank". A zatem obiekt tej klasy może posiadać dwa razy pole danych klasy "vehicle", ale nie musi. Sprawa pozostaje do dyspozycji projektanta języka. Widzimy więc pewne niejednoznaczności w semantyce języka, które muszą być rozstrzygnięte. Z drugiej strony nieuprzedzony użytkownik może mieć wątpliwości w zrozumieniu takiej formy strukturalnej, co więcej wydaje się, że w pewnych sytuacjach wygodniej byłoby mieć semantykę z powtarzalnym polem danych, a czasem z jednym. Autor języka Paragon przyjął, z konieczności, semantykę z jednym polem danych, jeżeli z powodu dziedziczenia jedna klasa będzie wielokrotnie dziedziczona. Wprowadzone w Paragonie dziedziczenie będzie-

my nazywać umownie wielo-dziedziczeniem, chociaż tym terminem będziemy obejmować nie jedną, ale wszystkie dopuszczalne semantyki.

Dziedziczenie w Simuli i Smalltalku jest jednopoziomowe. W Loglanie dziedziczenie dopuszcza wielopoziomowość. W Paragonie mamy możliwość wielo-dziedziczenia wielopoziomowego. Nie miejsce tu na analizę trudności semantycznych , które pojawiają się przy tak skomplikowanych konstrukcjach. To zadanie dla projektantów języka i realizatorów jego implementacji. Trzeba jednak zaznaczyć, że pojawienie się języków obiektowo-zorientowanych wraz z różnymi formami dziedziczenia, stworzyło nową dziedzinę informatyki w której tak doświadczenie praktyczne jak i zdolności teoretyczne odgrywają niebagatelną rolę.

Czytelnik mógłby się zapytać, czy języki obiektowo-zorientowane i całe to okropne wielo-dziedziczenie wielo-poziomowe nie jest zbyt trudne w zwykłym programowaniu. Wydaje się, że nie powinien mieć takich obaw. Klasa, obiekt i dziedziczenie są bardzo naturalnymi pojęciami i należy je w sposób naturalny rozumieć. Dziedziczenie wielopoziomowe może być przez użytkownika nawet niedostrzeżone, bo polega na tym, że wolno dziedziczyć to co jest widoczne - powiedziałbym nawet przekornie, że dziedziczenie jednopoziomowe powinno drażnić użytkownika, bo dlaczego takie dziwne ograniczenie. Wielodziedziczenie też naturalnym rozszerzeniem jedno-dziedziczenia i jeżeli użytkownik nie natrafi na jakąś skomplikowaną pułapkę semantyczną, nie będzie się zastanawiał, i słusznie, co może się kryć dziwnego za tymi pojęciami.

Mądre używanie języka obiektowo-zorientowanego przynosi użytkownikowi wiele korzyści. Otrzymuje on produkt szybciej i szybciej działający. Co więcej, umożliwia w elastyczny sposób dopasowanie świata pojęć informatycznych do świata opisywanego. Dziedziczenie ułatwia hierarchizację. A na tym opiera się przecież właściwa organizacja wszelkich systemów, także komputerowych.

BIBLIOGRAFIA

Bartol W.M. and others: "Report on the Loglan 82 programming language", PWN 1984

Bartol W.M., Kreczmar A., Litwiniuk A., Oktaba H.: "Semantics and

implementation of prefixing at many levels", LNCS 148, 1980, pp. 45-80

Dahl O.J., Myhrahaus B., Nygaard K.: "Common base langue", NCC Report, 1970

Ingalls D.: "The Smalltalk 76 programming system design and implementation", Proc. POPL, 1978, pp. 9-16

Krause M., Kreczmar A., Langmaack H., Warpechowski M.: "Concatention of program modules, an algebraic approach to the semantics and implementation problems", LNCS 208, pp. 134-156,

Salwicki A.: "On the algoritmic thoery of stacks", LNCS 64, 1978, pp. 452-461

Sherman M.S.: "Paragon", LNCS 189, 1985

Jesienna Szkoła PTI
Mrągowo, listopad 1986

PROBLEMATYKA SYSTEMÓW OPERACYJNYCH NA PRZYKŁADZIE SYSTEMU UNIX

Jan Madey
Instytut Informatyki
Uniwersytet Warszawski
PKiN pok.850
00-901 Warszawa
tel. 268-258

1. Wprowadzenie

Niezbędnym elementem oprogramowania podstawowego każdego komputera (w tym i mikrokomputera) jest *system operacyjny*. Jego zadaniem jest tworzenie środowiska operacyjnego, w którym użytkownik komputera może wygodnie i bezpiecznie opracowywać, uruchamiać lub eksploatować programy. System operacyjny ukrywa przed użytkownikiem pewne cechy i funkcje sprzętowe, a dostarcza mu w zamian narzędzi wygodniejszych do rozwiązywania problemów. W szczególności system operacyjny umożliwia:

- przechowywanie informacji przez dłuższy okres czasu w pamięciach zewnętrznych,
- wspólne wykorzystywanie niektórych urządzeń komputerowych przez grupy osób,
- jednoczesne wykonywanie różnych czynności przez system komputerowy,
- korzystanie z różnorodnych pakietów programistycznych i języków programowania.

W zależności od typu komputera, trybu jego wykorzystywania oraz od rodzaju urządzeń wchodzących w skład instalacji, systemy operacyjne istotnie różnią się od siebie stopniem złożoności i zakresem oferowanych usług. Znane są systemy operacyjne, które były konstruowane przez kilka lat w wieloosobowych zespołach specjalistów i mają bardzo złą reputację (np. OS/360 dla komputerów typu IBM 360/370). Znane są również systemy, które powstały bardzo szybko, w małych zespołach, a mimo to przebiły się i wzbudziły powszechne zainteresowanie. Najważnieszym reprezentantem tej grupy jest system *Unix*. Chociaż Unix ma już ponad piętnastoletnią historię, jego popularność ciągle rośnie, a niektóre z przyjętych w nim rozwiązań są naśladowane w nowszych systemach.

System Unix jest podobnym zjawiskiem wśród systemów operacyjnym, jak język Pascal wśród języków programowania. W obu wypadkach był to produkt autorstwa małej grupy osób, tworzony bez reklamy i specjalnego poparcia. W obu wypadkach hasłem naczelnym była *prostota*, a twórcy świetnie wyczuli przy tym potrzeby odbiorcy. W efekcie ich dzieła odniosły prawdziwy sukces, przebiły się bez instytucjonalnych sponsorów. Co ciekawsze, zarówno Pascal, jak i Unix, stają się obecnie standardami dla mikrokomputerów, mimo że nie były projektowane z myślą o tym typie sprzętu.

Pierwszą wersję systemu operacyjnego Unix opracowano na przełomie lat sześćdziesiątych i siedemdziesiątych w Bell Laboratories, New Jersey, USA, na wewnętrzny użytek tej firmy. Najważniejsze dwie osoby z tego okresu czasu, to Ken Thompson i Dennis Ritchie. Unix, napisany początkowo w asemblerze na komputer DEC PDP-7, tworzył środowisko ułatwiające programowanie. Głównymi odbiorcami i krytykami tej wersji systemu byli sami jego twórcy. Ze względu na potrzebę przeniesienia systemu na inny typ komputera został on ponownie zaprogramowany w języku wysokiego poziomu (język ten, o nazwie *C*, został opracowany do tego właśnie celu) i zainstalowany na PDP-11. Przy tej okazji wprowadzono do systemu pewne rozszerzenia i modyfikacje. W 1973 roku Unix został udostępniony ośrodkom akademickim, co bardzo szybko przyniosło mu dużą popularność. Z czasem różne instytucje zajęły się rozwijaniem i dystrybucją systemu Unix, a w szczególności przenoszeniem

Unixa na inne komputery. Powstało przy tym wiele różnych wersji i mutacji systemu, często dość istotnie różniących się między sobą. Do najbardziej zaawansowanych należą implementacje na komputerach VAX opracowane przez oddział Uniwersytetu Kalifornijskiego w Berkeley. Jedna z ostatnich wersji, to *Unix 4.2BSD* (skrót od *Berkeley Software Distribution*). Interesujący jest również zwrot ku mikrokomputerom. Już obecnie istnieją instalacje Unixa na systemy 16 bitowe z procesorami firm Motorola, Intel i Zilog. Powstają również mutacje Unixa, noszące odmienne nazwy. Na szczególną uwagę zasługują dwie z nich: *Xenix* oraz *Tunis*. Pierwszy z tych systemów jest produktem firmy Microsoft i został wprowadzony na rynek w 1980 roku jako komercyjna wersja Unixa dla mikroprocesorów Intel 8086/88. Zmodyfikowano przy tym nieznacznie strukturę Unixa oraz wprowadzono pewne ulepszenia (Xenix bazował na wersji *Unix V.7* z 1979 roku). System Tunis powstał natomiast jako produkt uboczny pewnego projektu na Uniwersytecie w Toronto. Celem tego projektu było wykazanie, że jeżeli dysponuje się właściwymi narzędziami (odpowiednim językiem i metodyką programowania), to zadania programowania systemowego znacznie się upraszczają. W Toronto opracowano specjalny język o nazwie *Concurrent Euclid* i w tym języku zrealizowano - głównie siłami studentów - nową wersję Unixa na komputer VAX, w której została od nowa zaprojektowana najistotniejsza część systemu (tzw. jądro). Tunis jest obecnie przenoszony także na mikrokomputery szesnastobitowe. Również inne systemy operacyjne mikrokomputerowe (np. MS-DOS) mają wiele cech przejętych z Unixa. Wszystko więc wskazuje na to, że Unix staje się standardowym systemem operacyjnym dla komputerów 16 i 32 bitowych, a dominującą w tym kontekście wersją jest obecnie tzw. *Unix System V edycja 2*, z 1984 roku. Dotyczy to także sieci -- powstają wersje "rozproszone" Unixa. Należy przypuszczać, że dopiero systemy komputerowe o istotnie nowej architekturze mogą spowodować zmierzch Unixa.

W niniejszym opracowaniu przedstawimy krótko wybrane aspekty problematyki systemów operacyjnych i zilustrujemy je na przykładzie systemu Unix. Ze względu na złożoność zagadnienia, prezentacja ta będzie bardzo zwięzła i daleka od kompletności.

2. Struktura systemu operacyjnego

2.1. Podział systemu na warstwy

Każdy system operacyjny ma wyodrębnioną część, która realizuje jego najbardziej podstawowe funkcje. Część tę nazywa się zwykle *jądrem* systemu, albo *warstwą wewnętrzną*. Pozostała część systemu tworzy *warstwę zewnętrzną*. Podział taki nie jest jednoznacznie ustalony dla wszystkich systemów. Wczesne systemy operacyjne miały niekiedy warstwę zewnętrzną pustą, czyli innymi słowy system tworzył jedną całość. Niekiedy znowu autorzy systemu nie prezentowali jego struktury w sposób klarowny lub też stosowali odmienną od naszej terminologię. Niemniej jednak wydaje się, że poza patologicznymi przypadkami można zawsze rozróżnić te dwie warstwy, a dla celów dydaktycznych wyodrębnienie ich ma duże znaczenie. W jądrze systemu implementuje się te wszystkie funkcje, które z racji swojej wagi i charakteru muszą być traktowane w uprzywilejowany sposób. W szczególności więc jądro jest odpowiedzialne za:

- obsługę przerwań,
- przydział procesora,
- operacje wejścia/wyjścia.

Ponadto w jądrze są implementowane mechanizmy, dzięki którym system komputerowy wraz z jądrem tworzy bardziej atrakcyjną dla użytkownika maszynę, zwaną często *maszyną wirtualną* lub *maszyną rozszerzoną*. Na przykład warstwa zewnętrzna może składać się z *procesów współbieżnych* -- wówczas jądro systemu musi dostarczyć narzędzi m.in. do tworzenia i usuwania procesów i do ich synchronizowania.

Jakie funkcje realizuje warstwa zewnętrzna? Trudno na to pytanie odpowiedzieć jednoznacznie. Zwykle warstwa zewnętrzna obejmuje procesy użytkowników i te moduły systemu operacyjnego, które mogą współzawodniczyć z użytkownikami o zasoby w podobny sposób, jak użytkownicy współzawodniczą między sobą. Innymi słowy w warstwie zewnętrznej umieszcza się te funkcje systemu, które mogą "trochę poczekać" na procesor, pamięć operacyjną lub inne

zasoby. Przykładem jest tu moduł szeregujący zadania (ang. *Job Scheduler*) lub moduły odpowiedzialne za wczytywanie zadań i wydrukowywanie wyników (ang. *Spooler*). Warstwa zewnętrzna ma niekiedy postać rodziny procesów współbieżnych zorganizowanych w *strukturę hierarchiczną*. Relacją określającą tę strukturę jest zwykle zależność zleceniodawca - zleceniobiorca.

2.2. Struktura systemu Unix

Unix jest wielodostępnym systemem operacyjnym wyposażonym ponadto w różne narzędzia wspomagające projektownie, tworzenie i uruchamianie programów. Narzędzia takie nie stanowią części systemu operacyjnego przy tradycyjnej definicji tego pojęcia. Dlatego też termin "Unix" utożsamiany jest powszechnie nie tyle z samym systemem operacyjnym, co z całym bogaty środowiskiem programistycznym. Wśród takich ułatwień programistycznych są znane edytory (np. *ed, vi, emacs*), programy wspomagające redagowanie tekstów (np. *troff, scribe*), kompilatory popularnych języków programowania (np. *Pascal, C, Fortran*), programy porównujące teksty (*cmp, diff*), wyszukujące określone sekwencje znaków (*grep, awk*), sortujące (*sort*), obsługujące pocztę elektroniczną (*mail*), itd.

Podział na warstwy jest w Unixie bardzo klarowny. Jądro systemu, poza czynnościami wspomnianymi w p.2.1. powyżej (obsługa przerwań, przydział procesora, wejście/wyjście), implementuje bardzo ważne w Unixie pojęcie *procesu*. Ponadto jądro prowadzi gospodarkę pamięcią operacyjną i pomocniczą, a w szczególności implementuje interesujący *system plików*. Każda akcja realizowana poza jądrem, to pewien proces. Warstwa zewnętrzna Unixa jest więc zestawem procesów, które odpowiadają wykonywaniu zarówno wspomnianych uprzednio programów systemowych, jak i programów użytkownika. Wśród procesów warstwy zewnętrznej specjalną rolę odgrywa tzw. *powłoka* (ang. *shell*). Jest to proces odpowiedzialny za konwersację z użytkownikiem, czyli interpretuje on komendy wprowadzane z klawiatury terminala. Przy tym powłoka jest wymienialna -- każdy użytkownik Unixa może napisać swój własny

interpretator komend i wstawić go w miejsce standardowego. W powszechnym użyciu są obecnie dwa interpretatory: tzw. *Bourne shell* (od nazwiska jego twórcy) i *C shell*.

3. Współbieżność w systemach operacyjnych

3.1. Wstęp

Różne składowe instalacji komputerowej mogą w pewnych okresach pracować równolegle. Podobnie, różne programy użytkowników, a także niektóre programy systemowe mogą być jednocześnie wykonywane, o ile pozwalają na to warunki sprzętowe i organizacyjne. Co więcej, w ramach pojedynczego programu dają się często wyodrębnić fragmenty, które możnaby także wykonywać równolegle. Spostrzeżenia te doprowadziły do powstania i rozwoju bardzo ważnej dyscypliny -- *programowania współbieżnego*. Dyscyplina ta zajmuje się całokształtem problemów związanych z notacją i technikami służącymi do wyrażania *potencjalnej równoległości*. W szczególności, chodzi o rozwiązywanie pojawiających się w tych sytuacjach problemów *synchronizacyjnych* i *komunikacyjnych*.

3.2. Procesy współbieżne

Ścisłe określenie pojęcia "proces" jest trudne. Trzeba w tym celu posłużyć się precyzyjnym aparatem matematycznym. Dlatego też w podręcznikach poświęconych systemom operacyjnym oraz w wielu pracach na ten temat stosuje się zwykle intuicyjną, nieformalną definicję. Podobnie i my postąpimy.

Proces sekwencyjny jest to realizacja programu sekwencyjnego lub też wstrzymana realizacja w oczekiwaniu na zdarzenie, które umożliwi jej kontynuację. Proces odpowiada więc programowi, ale jest obiektem (potencjalnie) aktywnym, któremu przydzielane są zasoby komputerowe, takie jak pamięć operacyjna oraz procesor. System operacyjny przekształcając program na proces tworzy pewne

struktury danych. Służą one do przechowywania informacji o *atrybutach* procesu. Do atrybutów takich zalicza się m.in. opis zasobów przydzielonych danemu procesowi oraz *stan* procesu: wykonywany, gotowy lub zawieszony. Mówimy, że proces jest *wykonywany*, jeżeli jest mu w danej chwili przydzielony procesor centralny. Jeżeli natomiast proces oczekuje wyłącznie na procesor, to jest on w stanie *gotowy*. Mówimy wówczas (por. p.4.1), że proces ten ma przydzielony *procesor wirtualny*. Jeżeli natomiast do kontynuacji procesu jest niezbędne zajście pewnego zdarzenia (np. zakończenie operacji wejścia), to jest on w stanie *zawieszony*.

Dwa procesy sekwencyjne są *współbieżne*, jeżeli wykonywanie jednego z nich zaczęło się po rozpoczęciu, ale przed zakończeniem wykonywania drugiego. Innymi słowy, przedziały czasu w których te procesy istnieją nie są rozłączne. Procesy współbieżne mogą, ale nie muszą być wykonywane równolegle -- zależy to zarówno od dostępnych środków technicznych, jak i od sposobu współpracy tych procesów ze sobą. Dlatego też mówi się często, że współbieżność jest *abstrakcją równoległości*. Powyższe określenie rozszerza się w naturalny sposób na układ więcej niż dwóch procesów -- mówimy więc, że mamy *układ procesów współbieżnych*, jeżeli każdy z nich jest współbieżny z jakimś (różnym od siebie) procesem z tego układu. Program, który specyfikuje układ procesów współbieżnych nazywamy *programem współbieżnym*.

Procesy współbieżne mogą operować na rozłącznych obiektach i ze sobą w ogóle nie współpracować. Mówimy wówczas o procesach *rozłącznych*. Jest to jednak sytuacja mało interesująca. Problemy pojawiają się dopiero wówczas, gdy procesy współbieżne są *interakcyjne*, tzn. albo konkurują między sobą w dostępie do zasobów, albo współpracują ze sobą komunikując się i wymieniając informacje. W obu tych wypadkach jest konieczna *synchronizacja* procesów, a więc czasowe wstrzymywanie jednego lub kilku z nich, dopóki nie zajdzie określone zdarzenie. Właśnie sprawa synchronizacji i komunikacji jest najtrudniejszym problemem do rozwiązania w programowaniu współbieżnym. Może się tu pojawić w szczególności bardzo niebezpieczna sytuacja -- tzw. blokada (ang. *deadlock*). Proces jest w stanie *blokady*, jeżeli oczekuje na zdarzenie, które

nigdy nie może nastąpić. Typowym przykładem blokady jest sytuacja, w której procesy *P1* i *P2* potrzebują zasobów *A* i *B*, a system operacyjny przydzielił już procesowi *P1* zasób *A*, zaś procesowi *P2* zasób *B*. Wówczas *P1* jest zawieszony nieskończenie w oczekiwaniu na przydział zasobu *B*, a *P2* w oczekiwaniu na *A*. Unikanie blokady jest bardzo trudnym i ważnym zadaniem przy projektowaniu systemów operacyjnych stanowiących układ procesów współbieżnych.

3.3. Synchronizacja

Jednym z najbardziej typowych reguł synchronizacyjnych jest *wzajemne wykluczanie* (ang. *mutual exclusion*). Jeżeli dwie operacje nie mogą być nigdy wykonywane jednocześnie, to mówimy, że muszą się one wzajemnie wykluczać w czasie. Z koniecznością wzajemnego wykluczania stykamy się bardzo często -- w każdym wypadku, gdy jakiś proces musi mieć zagwarantowany wyłączny dostęp do wspólnego, dzielonego zasobu w celu wykonania na nim pewnych operacji. Przypuśćmy więc, że mamy układ *n* procesów współbieżnych, z których każdy ma taką samą strukturę: w nieskończonej pętli na początku wykonuje pewne operacje nieistotne z punktu widzenia pozostałych procesów, a następnie przechodzi do operacji wymagających wyłączności. Tę pierwszą część nazwiemy *operacjami lokalnymi*, a tę drugą: *rejonem krytycznym*. Instrukcje rejonu krytycznego muszą być ujęte w jakieś "ogrodzenie", gwarantujące spełnienie warunku wzajemnego wykluczania. Problem polega teraz na zaproponowaniu mechanizmu oraz notacji dla tego właśnie "ogrodzenia". Dosyć szybko okazało się, że standardowe mechanizmy programowania sekwencyjnego mogą być do tego celu użyte, ale otrzymuje się wówczas rozwiązanie bardzo skomplikowane i podatne na błędy. Pojawiły się więc różne propozycje nowych konstrukcji, z których pewne zostały użyte praktycznie w projektowaniu i implementacji systemów operacyjnych. Przedstawimy krótko idee dwóch takich konstrukcji: *semaforów* oraz *monitorów*.

Semafory wprowadził Dijkstra ponad 20 lat temu. Był to pierwszy ogólnie zaakceptowany mechanizm synchronizacyjny. Na użytek niniejszej pracy przyjmiemy następującą definicję, dość

bliską oryginalnej:
Semafor jest to zmienna całkowita nieujemna, dla której są określone tylko trzy operacje: inicjacja, *wait* oraz *signal*. Pierwsza z nich służy do nadania zmiennej semaforowej *s* wartości początkowej. Może być ona wykonana tylko jeden raz dla danej zmiennej i poza procesami. Dwie następne operacje są jednostkowe, niepodzielne i wzajemnie wykluczające się. Ich znaczenie jest następujące:

wait(s): gdy *s* dodatnie, to *s := s - 1*,
(w przeciwnym razie proces jest zawieszany
w oczekiwaniu na ten warunek)

signal(s): *s := s + 1*

Z pomocą semafora można w następujący sposób przedstawić schemat każdego z procesów z omawianego powyżej układu. Niech *mutex* będzie zmienną semaforową o wartości początkowej równej 1. Wówczas:

```
repeat
    operacje lokalne;
    wait(mutex);
    rejon krytyczny;
    signal(mutex)
until false
```

Nie wchodząc dalej w rozważania o semaforach zauważmy tylko, że jest to mechanizm niestrukturalny i w przypadku złożonego systemu procesów można względnie łatwo popełnić omyłkę niewykrywalną w czasie kompilacji programu współbieżnego opisującego ten system. Dlatego też kilkanaście lat temu podjęto próby opracowania lepszego, strukturalnego mechanizmu synchronizacyjnego. W połowie lat siedemdziesiątych ukazały się prace proponujące nową konstrukcję, zwaną "monitorem". Przyczynili się do tego trzej światowej sławy informatycy: Brinch Hansen, Dijkstra oraz Hoare. Pod pojęciem monitora rozumie się moduł grupujący zarówno struktury danych reprezentujące wspólne, dzielone przez procesy obiekty, jak i procedury opisujące jedyne wykonalne na tych zasobach

operacje. Co więcej, procedury te z definicji wzajemnie wykluczają się w czasie. Tak więc monitor stanowi owo "ogrodzenie", w którym umieszcza się operacje rejonu krytycznego. Monitor poza swoim zasadniczym celem - synchronizacja dostępu do zmiennych dzielonych gwarantująca wzajemne wykluczanie - może także służyć do definiowania abstrakcyjnych typów danych oraz jako narzędzie modularyzacji programu. Zakres niniejszej pracy nie pozwala niestety nawet na przytoczenie przykładu -- musielibyśmy wprowadzić jeszcze wiele pojęć, definicji oraz konwencji notacyjnych. Zauważmy więc tylko na koniec, że monitor występuje jako konstrukcja w kilku językach programowania współbieżnego (np. Concurrent Pascal, Pascal-Plus, Concurrent Euclid, Mesa, Chill) i był użyty do implementacji niektórych systemów operacyjnych. Dotyczy to w szczególności wspomnianego w p.1. niniejszego opracowania systemu Tunis, będącego strukturalną wersją Unixa zrealizowaną w języku Concurrent Euclid.

3.4. Współbieżność w systemie Unix

W systemie Unix, jak wiadomo (por. p.2.2.), każda czynność realizowana poza jądrem, to proces. Jądro implementuje więc mechanizmy niezbędne do tworzenia i niszczenia procesów oraz do manipulowania nimi. Każdy istniejący proces może zażądać utworzenia nowego procesu, realizując operację jądra o nazwie *fork*:

processNo = fork()

Jądro tworzy wówczas nowy proces, "syna", który jest prawie identyczny z "ojcem". Otrzymuje on oddzielną przestrzeń adresową, ale z tym samym programem, zmiennymi i z tymi samymi wartościami zmiennych. Jedyna różnica, to wartość zmiennej *processNo* identyfikującej proces. U ojca zmienna ta uzyskuje wartość niezerową, jednoznacznie identyfikującą syna, natomiast u syna jest ona zerowana. Następnie ojciec i syn, jako procesy współbieżne, kontynuują realizację tego samego programu od instrukcji następującej po wywołaniu *fork*. Zwykle jednak proces syn został utworzony w celu wykonywania innego programu. Jest to możliwe dzięki operacji systemowej *execve*. W programie procesu powinna wystąpić wówczas instrukcja warunkowa badająca wartość

processNo i w wypadku syna wywołująca *execve* ze wskazaniem programu, który ma zastąpić bieżąco wykonywany program. Ojciec z kolei może czekać na zakończenie syna:

processNo = wait(status)

W wyniku powyższego wywołania operacji *wait* jądro wstrzymuje ojca aż do chwili, gdy jeden z jego synów się zakończy. Identyfikacja tego syna jest przypisywana zmiennej *processNo*, zaś powód zakończenia jest opisany w słowie wskazywanym przez *status*. Każdy proces może się sam zniszczyć (zakończyć) wywołując operację *exit*, lub też zniszczyć inny proces operacją *kill*.

Zilustrujmy obecnie dla przykładu strukturę procesu interpretującego komendy użytkownika (jest to, jak pamiętamy, proces systemowy o nazwie *powłoka* -- *shell*). Proces ten w chwili odczytania komendy z klawiatury tworzy syna, który jest odpowiedzialny za realizację danej komendy. Standardowo ojciec (czyli powłoka) oczekuje na zakończenie się syna i wówczas pobiera kolejną komendę. Jednakże jest możliwe wskazanie przez użytkownika by dana komenda wykonywała się "w tle" -- wówczas powłoka nie czekając na zakończenie danego syna pobiera kolejną komendę i tworzy nowego syna (odpowiedzialnego za tę nową komendę). Czynności te są realizowane z wykorzystaniem omówionych operacji *fork*, *execve*, *exit* oraz *wait*.

Procesy mogą się ze sobą komunikować. Najprostszym aparatem komunikacyjnym jest konstrukcja zwana *pipe*, będąca anonimowym jednokierunkowym, ograniczonym *buforem*. Procesy połączone ze sobą takim buforem współpracują na zasadzie *producent - konsument*. Próba zapisania do pełnego bufora lub odczytania z pustego kończy się zawieszeniem procesu aż do chwili, gdy stan danego bufora się odpowiednio zmieni.

Normalny użytkownik siedzący przy terminalu nie korzysta w sposób jawny z omówionych powyżej operacji. Forma podawania przez niego zleceń jest znacznie wygodniejsza -- interpretator komend określa bowiem pewien język wysokiego poziomu do tego celu. Język ten jest bogaty i elastyczny, niemniej jednak przyjęty zwyczaj stosowania zwięzłych skrótów bywa przedmiotem krytyki, zwłaszcza

wśród nowicjuszy. Podamy teraz kilka prostych przykładów formułowania komend w języku powłoki. Przy tym symbol "*" będzie tu oznaczał zgłoszenie się powłoki w oczekiwaniu na kolejną komendę (w rzeczywistości używany jest w tym celu najczęściej znak "procent", ale z pewnych względów w niniejszej pracy zmieniono tę konwencję). Niech *prog* będzie nazwą pliku z programem, który chcemy wykonać. Wówczas zapisujemy to po prostu:

* *prog*

Przypuśćmy teraz, że *prog* oczekuje danych z klawiatury, a wyprowadza wyniki na ekran. My natomiast chcielibyśmy, aby teraz dane były pobrane z pliku o nazwie *dane*, a nie z klawiatury. Można ten efekt uzyskać bardzo prosto, bez ingerencji w treść programu:

* *prog* < *dane*

Podobnie, możemy zmienić miejsce wyprowadzania wyników na np. plik o nazwie *wyniki*:

* *prog* > *wyniki*

Jest też możliwe połączenie tych obu żądań w jedną komendę:

* *prog* < *dane* > *wyniki*

Przypuśćmy teraz z kolei, że chcemy aby wyniki *prog* były danymi wejściowymi dla programu *filtr*, który np. zastępuje wielokrotne spacje pojedynczymi, a po tej operacji chcemy owe wyniki wydrukować na drukarce. Polecenie takie zapisuje się bardzo prosto:

* *prog* | *filtr* | *lpr*

Powłoka zleci wówczas utworzenie dwóch buforów do komunikacji pomiędzy procesami; mówimy często, że powstaje *rurociąg* (ang. *pipeline*). Program o nazwie *lpr* jest standardowym programem systemowym drukowania na drukarce.

Język interpretatora komend (wersja *Bourne shell* i *C shell*) ma wiele konstrukcji programistycznych spotykanych w normalnych językach programowania wysokiego poziomu, co pozwala na odejście od trybu interakcyjnego. Użytkownik może skomponować program w tym języku, zapisać go na jakimś pliku, a następnie zlecić powłoce wykonanie tego pliku. Mamy wówczas analogię do przetwarzania w trybie wsadowym.

4. Zarządzanie zasobami

4.1. Wstęp

W systemach komputerowych występują różnego rodzaju zasoby, którymi zarządza system operacyjny. Zarządzanie to polega na takim rozdzielaniu zasobów pomiędzy użytkowników, aby każdemu z nich stwarzać iluzję, że pracuje on na swoim własnym komputerze. Taki zindywidualizowany komputer nazywamy często *wirtualnym*. Jest on dostosowany do bieżących potrzeb użytkownika, a różni się od rzeczywistego komputera przede wszystkim wolniejszą pracą (bo procesor jest dzielony pomiędzy wiele osób). Zasoby komputera wirtualnego nazywamy *zasobami wirtualnymi*. Komputer wirtualny daje użytkownikowi niekiedy większe możliwości od rzeczywistego. Na przykład *pamięć wirtualna* jest dużo większa od rzeczywistej pamięci operacyjnej. Efekt ten uzyskuje się dzięki temu, że odpowiedni moduł systemu operacyjnego nie tylko bezkolizyjnie dzieli pamięć operacyjną pomiędzy użytkowników, ale również zapewnia automatyczne przesyłanie informacji pomiędzy pamięcią operacyjną a pomocniczą. Stwarza to każdemu użytkownikowi złudzenie posiadania do własnej dyspozycji prawie nieograniczonej pamięci operacyjnej (limitowanej wielkością przydzielonej pamięci pomocniczej).

Zasoby komputerowe są to obiekty dzielone (współużywane) przez użytkowników oraz przez sam system operacyjny. Tradycyjnie wyróżnia się przy tym *zasoby sprzętowe* oraz *zasoby programowe*. Do pierwszej grupy zaliczamy podstawowe składowe sprzętowe komputera, tj. *procesory* (procesor centralny, kanały, urządzenia wejścia/wyjścia) i *pamięci* (obszary na różnego rodzaju nośnikach informacji). Drugą grupę tworzą programy usługowe oraz struktury danych kreowane przez użytkownika lub przez system (*pliki*). Ponadto w niektórych systemach do zasobów programowych zalicza się jeszcze pewne bufory, służące do komunikacji między procesami. Niezależnie od tej klasyfikacji, zasoby dzielą się na *stałe* (niezużywalne) i *tymczasowe* (zużywalne). Przykładem stałego zasobu jest urządzenie fizyczne, przykładem tymczasowego - komunikat.

Obsługując użytkowników system operacyjny dla każdego z nich tworzy, utrzymuje, a następnie likwiduje komputer wirtualny. Przy jednoczesnej obsłudze wielu klientów system musi więc bezkolizyjnie i bezpiecznie dzielić zasoby. W tym celu określone fragmenty systemu (tzw. *zarządcy zasobów*) prowadzą i realizują politykę przydziału zasobów oraz śledzą ich stan i stopień wykorzystania. Do zadań zarządcy zasobów należy zatem:

- śledzenie na bieżąco stanu danego zasobu bądź jego części ("wolny", "zajęty") oraz ewentualne śledzenie stopnia wykorzystania tego zasobu w określonym przedziale czasu;
- ustalanie strategii przydziału danego zasobu: kto, kiedy, jak często, na jak długo i ile otrzyma;
- przydzielanie zasobu według przyjętej strategii;
- odzyskiwanie zasobu.

W dalszych dwóch podpunktach przedstawimy zwięźle wybrane fakty o zarządzaniu dwoma ważnymi zasobami: *procesorem centralnym* oraz *pamięcią operacyjną*. Omówienie choćby skrótowe pozostałych typów zasobów przekracza zakres niniejszego opracowania. Niemniej jednak pewne uwagi na temat systemu plików znajdą się w p.4.4. poświęconym Unixowi.

4.2. Procesor

Zarządzanie procesorem jest realizowane w jądrze systemu. Każdy z procesów, który ubiega się w danej chwili o procesor jest umieszczany w *kolejce procesów gotowych*. Procesy systemowe mają przy tym zawsze wyższy priorytet od procesów użytkowników, czyli im jest przede wszystkim przydzielany procesor (w kolejności ustalonej zwykle dla danego systemu). Natomiast konkurujące między sobą procesy użytkowników są obsługiwane zgodnie z pewną strategią.

Strategie przydziału procesora dzielą się na dwie podstawowe grupy: *z wywłaszczaniem* (ang. *preemptive scheduling*) oraz *bez wywłaszczania*. W pierwszym wypadku procesor może być odebrany procesowi, a w drugim -- nie. Przykładem strategii bez wywła-

szczania jest obsługa procesów według kolejności ich pojawiania się w kolejce procesów gotowych. Strategia ta jest często nazywana *FIFO* (skrót od *First-In-First-Out*). Przykładem strategii z wywłaszczaniem jest tzw. *karuzelowa obsługa procesów*, zwana *RR* (skrót od *Round-Robin*). Dla jej realizacji niezbędne jest urządzenie fizyczne zwane *zegarem interwałowym*, który to zegar powoduje przerwania co określony kwant czasu. Po każdym takim przerwaniu procesor jest przydzielany kolejnemu gotowemu procesowi, a proces wywłaszczony jest umieszczany na końcu kolejki. Wielkość kwantu czasu jest zwykle ustalona dla danej intalacji, ale są też systemy ze zmiennym kwantem, którego wielkość jest uzależniona np. od obciążenia systemu.

4.3. Pamięć operacyjna

Metody przydziału pamięci operacyjnej są przedmiotem wszechstronnych badań od ponad 20 lat i uzyskano już wiele ciekawych wyników teoretycznych oraz praktycznych. Organizacja pamięci, jej parametry (wielkość i szybkość dostępu) oraz środki techniczne realizacji uległy bardzo istotnej ewolucji. Podobnie, ewolucji uległy strategie przydziału pamięci. Zauważmy jednak, że niezmienna pozostała zasada wiążąca ze sobą pamięć z procesorem: *nie ma sensu przechowywać w pamięci operacyjnej procesów, które nie są bezpośrednimi kandydatami do przydziału procesora, i odwrotnie, nie ma sensu przydzielanie procesora takiemu procesowi, którego przestrzeń adresowa nie jest w pamięci operacyjnej.*

Początkowo były rozwijane metody, które zakładały obecność w spójnym obszarze pamięci operacyjnej całej przestrzeni adresowej wykonywanego procesu. Ograniczając się do systemów wieloprogramowych, należy wspomnieć o następujących ważnych strategiach:

- statyczny podział na strefy,
- dynamiczny podział na strefy,
- wymiana (ang. *swapping*).

W pierwszym wypadku pamięć operacyjna jest podzielona w czasie inicjacji systemu operacyjnego na *stałe strefy*, tzn. rozłączne i spójne obszary. Procesowi jest przydzielana jedna strefa, w

ramach której są wykonywane wszystkie obliczenia tego procesu (liczba stref limituje więc stopień wieloprogramowości). Łatwo jest przy tym zagwarantować *ochronę pamięci* -- wystarczą dwa rejestry graniczne, których wartości określają zakres dopuszczalnej zmienności adresów dla danego procesu. Strategia ta, prosta w realizacji, ma istotną wadę: często duże obszary pamięci są niewykorzystane. To niekorzystne zjawisko nazywamy *fragmentacją pamięci*. Wyróżniamy przy tym *fragmentację zewnętrzną* (pozostają niewykorzystane wolne strefy, gdyż są za małe na potrzeby procesów gotowych) oraz *fragmentację wewnętrzną* (część strefy nie jest wykorzystywana przez realizowany w niej proces). W celu wyeliminowania fragmentacji wewnętrznej opracowano drugą z wymienionych wyżej strategii. Strefy są w niej tworzone dynamicznie, w chwili pojawienia się nowego procesu. Otrzymuje on wówczas tylko tyle pamięci, ile potrzebuje (stąd przyjęło się mówić w tym wypadku o *zmiennych strefach*). Pojawiają się jednak dwa nowe problemy. Po pierwsze, trzeba znaleźć w pamięci odpowiedni obszar, a po drugie, pamięć po pewnym okresie czasu jest bardzo "poszatkowana" -- są w niej przemieszane obszary zajęte z obszarami wolnymi, przy czym te ostatnie są często nieduże i przez to pozostają niewykorzystane. Jest kilka metod wyszukiwania wolnego obszaru. Najczęściej stosowane algorytmy, to *pierwszy zdatny* (ang. *first-fit*) i *najlepszy zdatny* (ang. *best-fit*). W pierwszym wypadku przydziela się procesowi pierwszy znaleziony wolny obszar, w którym proces się zmieści. W drugim wypadku przydziela się najmniejszy z takich obszarów (w celu zminimalizowania wielkości odrzucanego kawałka wolnego obszaru). Jeżeli natomiast chodzi o "poszatkowanie" pamięci, to stosuje się co pewien czas tzw. *scalanie* (ang. *compaction*), czyli przemieszcza się tak przestrzenie adresowe procesów w pamięci operacyjnej, aby wszystkie wolne obszary połączyły się w jeden. Jest to operacja bardzo kosztowna. Trzecia z wymienionych strategii, *wymiana* (ang. *swapping*), polega na czasowym usuwaniu całej przestrzeni adresowej pewnego procesu z pamięci operacyjnej i sprowadzaniu w to miejsce innego procesu. Strategia ta bywa stosowana zarówno samodzielnie, jak i w połączeniu z podziałem na strefy. Jest ona szczególnie przydatna w wielodostępnych systemach operacyjnych opartych o podział czasu (ang. *time-sharing*).

Gdy zauważono w latach sześćdziesiątych, że nie ma potrzeby przechowywania w pamięci operacyjnej przez cały czas całej przestrzeni adresowej wykonywanego procesu, zaczęły się rozwijać nowe metody alokacji pamięci. Doprowadziło to (po rozwiązaniu także pewnych problemów sprzętowych) do wypracowania techniki zwanej *pamięcią wirtualną*. Idea polega na zautomatyzowaniu przesyłania fragmentów przestrzeni adresowej procesów pomiędzy pamięcią pomocniczą a pamięcią operacyjną i ukryciu faktu istnienia dwóch poziomów pamięci przed użytkownikiem. Najczęściej stosowana metoda realizacji tej techniki, to *stronicowanie na żądanie* (ang. *demand paging*). Zarówno pamięć operacyjna, jak i pomocnicza są podzielone na jednakowej wielkości obszary zwane *ramkami stron*. Przestrzeń adresowa procesu użytkownika jest umieszczana w pamięci pomocniczej i dzielona na *strony* o rozmiarach odpowiadających ramkom. Następnie do pamięci operacyjnej sprowadzane są tylko te strony, które są rzeczywiście potrzebne w danej chwili. Każda strona może być przy tym umieszczona w dowolnej ramce. Problem pojawia się wtedy, gdy brakuje wolnych ramek -- moduł zarządzający pamięcią musi wówczas usunąć jedną ze stron z pamięci operacyjnej do pamięci pomocniczej. Wybór takiej "ofiary" do usunięcia odbywa się według przyjętej w systemie metody. Najczęściej stosuje się algorytm zwany *LRU* (od ang. *Least-Recently-Used*) -- usuwa się stronę najdłużej nie używaną.

Realizacja pamięci wirtualnej nastręcza wiele problemów, zarówno natury technicznej, jak i systemowej. Zły dobór algorytmu usuwania stron, nieodpowiednia wielkość strony, niewłaściwe urządzenie dla pamięci pomocniczej, brak wspomagania sprzętowego do efektywnego wyliczania adresów, wszystko to może znacznie zmniejszyć korzyści z wirtualizacji pamięci. W przypadkach patologicznych może nawet dojść do zjawiska zwanego *wigotaniem stron* (ang. *thrashing*), kiedy to system nie robi zasadniczo nic innego, tylko zajmuje się przesyłaniem stron pomiędzy obydwoma poziomami pamięci. Niemniej jednak, pamięć wirtualna jest powszechnie stosowana, co świadczy o rozwiązaniu podstawowych problemów.

Moduł zarządzający pamięcią, w zależności od tego jaką realizuje strategię, musi współpracować z różnymi innymi modułami

systemu operacyjnego. Jak już było wspomniane, jest ścisły związek między zarządzaniem pamięcią, a zarządzaniem procesorem. Ponadto moduł ten musi współpracować z modułem zarządzającym pamięcią zewnętrzną. Jest on więc często umieszczany w jądrze systemu operacyjnego, choć nie jest to niezbędne.

4.4. Zarządzanie zasobami w Unixie

System Unix, jak już wspominaliśmy, ma wiele wersji, edycji i mutacji. Oczywistym więc jest, że zachodziły zmiany także w modułach zarządzających zasobami, zwłaszcza przy istotnie różnych konfiguracjach sprzętowych. Niezmienny pozostał jednak *system plików* (przynajmniej z punktu widzenia użytkownika), który przyczynił się w dużym stopniu do popularności Unixa. Przedstawimy więc bardzo skrótowo podstawowe cechy tego systemu, przedtem jednak podamy kilka uwag o zarządzaniu procesorem i pamięcią operacyjną.

Zarządzanie procesorem w Unixie jest ukierunkowane na faworyzowanie procesów interakcyjnych (konwersacyjnych). Każdy proces ma przyporządkowany przez system priorytet, który jest uaktualniany dynamicznie (co T1 sekund). Ponadto jest ustalony kwant czasu T2 (zwykle mniejszy o rząd wielkości od T1), co który jest przydzielany procesor. Im więcej czasu procesora wykorzystał proces, tym ma mniejszy priorytet -- dla procesów obliczeniowych (ang. *cpu-bound*) przyjęta metoda redukuje się do karuzelowej obsługi procesów (*RR*) z kwantem T2. Dla przykładu, w wersji *Unix 4.2BSD* przyjęto: T1 = 1 sekunda, T2 = 0.1 sekundy.

Zarządzanie pamięcią uległo istotnym zmianom w miarę powstawania nowych wersji systemu. Początkowo stosowana była metoda wymiany (*swapping*) w połączeniu ze strefami, a następnie stronicowanie na żądanie. Nie były to "czyste" strategie, ale uwzględniały wiele specyficznych cech i rozwiązań przyjętych w Unixie, a w szczególności strukturę przestrzeni adresowej procesu (której nie omawialiśmy z braku miejsca).

Przejdźmy teraz do prezentacji głównych cech systemu plików. Przede wszystkim autorzy Unixa podjęli pewne strategiczne decyzje, które zdeterminowały charakter systemu plików:

- *plik* jest ciągiem bajtów, bez żadnej struktury wewnętrznej,
- pliki są grupowane w *katalogi* (ang. *directory*), które same są także traktowane jako pliki,
- oprócz plików tekstowych i katalogów są także *pliki specjalne*, reprezentujące urządzenia zewnętrzne,
- nie ma wstępnej alokacji miejsca na dysku dla pliku, więc może on być dowolnej długości,
- biblioteka plików może być instalowana na kilku pakietach dyskowych,
- system ochrony plików jest prosty i elastyczny, pozwalający wyróżniać poziomy w prawach dostępu.

System plików tworzy drzewiastą strukturę hierarchiczną. Jest mianowicie wyróżniony katalog nadrzędny, zwany *korzeniem* (lub *katalogiem pierwotnym*, ang. *root*), który zawiera informacje o swoich bezpośrednich podkatalogach. Z kolei każdy z tych katalogów jest początkiem poddrzewa, składającego się znowu z katalogów i/lub innych plików. Do obsługi tej struktury użytkownik ma do dyspozycji bogaty zestaw prostych narzędzi. Można łatwo tworzyć nowe pliki (i dołączać je do dowolnego katalogu), usuwać, kopiować, przesuwać. Podobnie prosto można operować na katalogach, a więc manipulować poddrzewami. Plik jest identyfikowany przez *ścieżkę* prowadzącą do niego albo od katalogu pierwotnego (mówimy wtedy o *ścieżce bezwzględnej*), albo od bieżącego katalogu, którym operuje w danej chwili użytkownik (mamy wówczas *ścieżkę względną*). Syntaktycznie, jest to ciąg nazw plików-katalogów rozdzielanych kreską ukośną "/". Na przykład:

JM/Referaty/R12/roz4

identyfikuje plik *roz4* będący elementem katalogu *R12*, który jest elementem katalogu *Referaty*, a ten z kolei jest podkatalogiem katalogu *JM*. Ścieżka ta jest względna, bo zaczyna się od nazwy *JM*. Natomiast zapis:

/usr/JM/poczta

oznacza ścieżkę bezwzględną, identyfikującą plik *poczta* w katalogu *JM*, który jest podkatalogiem *usr*. Początkowa kreska ukośna nie

jest rozdzielaczem, ale nazwą katalogu pierwotnego. W typowej instalacji Unixa istnieje zwykle dosyć rozbudowana biblioteka plików, o pewnej liczbie standardowych katalogów (np. *usr, lib, bin, dev*) i o dużej liczbie katalogów i wychodzących z nich poddrzew tworzonych i dynamicznie zmienianych przez użytkowników. W szczególności jeden plik może być znany pod różnymi nazwami w jednym lub więcej katalogach -- dzięki temu można uniknąć kosztownego kopiowania oraz ułatwić współpracę między użytkownikami.

Powyższe zwięzłe omówienie było zrobione z perspektywy użytkownika. Operacje dostępne na tym poziomie są przekształcane na wywołania jądra systemu, które dysponuje odpowiednim zestawem bardziej prymitywnych narzędzi. Należą do nich takie operacje, jak *open, close, read, write, seek, create, link, unlink.*

Na zakończenie jeszcze kilka zdań na temat ochrony plików, czyli *praw dostępu.* Dla każdego pliku jest zapisana informacja, czy dany użytkownik może ten plik odczytać (atrybut *r*), na ten plik zapisać (atrybut *w*) lub ten plik wykonać, gdy jest to program (atrybut *x*). Użytkownicy są podzieleni przy tym na trzy typy: właściciel pliku, wcześniej określona grupa osób i wszyscy. Z każdym plikiem jest więc stowarzyszone 9 bitów do zapisania tej informacji. I tak, prawo wykonania wszystkich czynności przez wszystkich na danym pliku jest zapisywane zewnętrznie następująco:

rwx rwx rwx

Jeżeli natomiast właściciel może z plikiem robić wszystko (ale nie jest to plik z programem), członek grupy może tylko plik odczytywać, a pozostałe osoby nie mają w ogóle dostępu, to opis wyglądą następująco:

rw- r-- ---

Mechanizm ten jest prosty i efektywny. Można przy tym łatwo modyfikować prawa dostępu dla danego (własnego) pliku. Na przykład przypisanie plikowi *test* bezpośrednio powyżej określonych praw dostępu uzyskuje się komendą:

* *chmod 640 test*

Dlaczego jest podana tutaj wartość *640*? Niech Czytelnik zechce sam się nad tym zastanowić.

5. Uwagi końcowe

Problematyka systemów operacyjnych jest bardzo bogata. Przedmiot ten jest prezentowany na uczelniach w ramach całorocznego wykładu z ćwiczeniami. Niektóre podręczniki z systemów operacyjnych mają po kilkaset stron. W niniejszym opracowaniu można więc było zaledwie naszkicować wybrane aspekty tej problematyki. Nie poruszyłem w ogóle wielu istotnych kwestii.

Podobnie, system Unix ma wiele wersji i każdej z nich można poświęcić wiele godzin wykładowych oraz wiele stron tekstu. Musiałem się więc z konieczności ograniczyć do pobieżnego przedstawienia tylko niektórych faktów o tym systemie. Trzeba przy tym pamiętać, że na Unix nadal nie ma licencji eksportowej do Polski, stąd i rozważania o tym systemie mają akademicki charakter.

Bardzo trudno również wskazać literaturę. Zarówno na temat systemów operacyjnych, jak i na temat Unixa ukazało się na świecie bardzo dużo podręczników, prac, raportów, dokumentacji. Kilka wartościowych pozycji (ale niestety już przestarzałych) z zakresu systemów operacyjnych zostało przetłumaczonych na język polski. Prawdę mówiąc jednak, autor niniejszego opracowania od wielu już lat poszukuje "idealnego" podręcznika i "idealnej" metodyki nauczania tego przedmiotu, Jak dotąd, bezskutecznie. Dlatego też zrezygnowałem z Bibliografii w niniejszym opracowaniu. Nie korzystałem przy jego opracowywaniu z żadnego konkretnego zestawu materiałów, poza własnym tekstem pod tytułem *Systemy operacyjne dla mikrokomputerów* z zeszłorocznej Jesiennej Szkoły PTI. Musiałbym więc przytoczyć albo bardzo wiele tytułów, albo nic. Wybrałem to drugie rozwiązanie.

Jesienna Szkoła PTI
Mrągowo, listopad 1986

PROGRAMOWANIE FUNKCJONALNE

Stefan Sokołowki
Instytut Matematyki, Uniwersytet Gdański
ul. Wita Stwosza 57, 80-952 Gdańsk

Mówiąc o "rewolucji informatycznej" najczęściej mamy na myśli poprawianie technicznych parametrów komputerów, takich jak prędkość działania, pojemność pamięci, zaawansowanie techniczne urządzeń zewnętrznych; a z drugiej strony ich rozpowszechnienie i wpływ na coraz to nowe dziedziny działalności człowieka. Niejako w cieniu tej technicznej oraz społecznej rewolucji dokonuje się przewrót w sposobach używania komputera: programujemy dzisiaj zupełnie inaczej niż 20 lat temu. W początkach swego istnienia programowanie polegało na instruowaniu maszyny co ma robić krok za krokiem; pełna odpowiedzialność za efekty spadała na programistę. Obecnie w coraz większym stopniu zadaniem programisty jest ścisłe sformułowanie problemu; odpowiedzialność za jego rozwiązanie przenosi się na komputer.

Z pewną przesadą można stwierdzić, że programowanie staje się sztuką precyzyjnego definiowania pojęć, które rozumiemy intuicyjnie. Temu trendowi towarzyszy, lub może stanowi jego podstawę, postępująca matematyzacja wiodącego stylu programowania. Historia rozwoju języków programowania jest historią żmudnego ponownego odkrywania rzeczywistości znanej od dawna matematykom. Do tej kategorii zjawisk należy przebój ostatnich lat -- programowanie funkcjonalne.

Programowanie funkcjonalne nie opiera się na żadnej konkretnej dziedzinie matematyki, natomiast eksploatuje do końca pojęcie funkcji stanowiące podstawę całej matematyki. Oprócz funkcji działających na elementy jednostkowe, takie jak liczby lub punkty płaszczyzny, matematycy używają tak zwanych funkcjonałów, lub funkcji wysokiego rzędu, których argumenty lub wyniki są znowu funkcjami. Typowym funkcjonałem jest pochodna:

Pochodna : (Real --> Real) --> (Real --> Real)

Powyższy zapis ustala typ funkcjonału Pochodna : działa ona na argumencie, który jest funkcją z Real do Real i w wyniku daje znowu funkcję z Real do Real . Jak dobrze wiedzą matematycy, każdy precyzyjny opis świata da się wyrazić za pomocą funkcji odpowiednio wysokich rzędów.

Termin "programowanie funkcjonalne" wiąże się ze słowem "funkcjonał" w taki sam sposób, jak "analiza funkcjonalna" -- dziedzina matematyki zajmująca się przestrzeniami, w których pojedynczymi punktami są całe funkcje. Stosowane czasem tłumacze-

nie angielskiego terminu "functional programming" na "programowanie funkcyjne" nie wydaje mi się dobrym pomysłem. To prawda, że słowo "funkcjonalny" ma w języju polskim inne znaczenie (użyteczny), ale słowo "funkcyjny" nie jest pod tym względem lepsze (por. dodatek funkcyjny do pensji). Tymczasem prawdziwy smak programowania funkcjonalnego przychodzi dopiero z funkcjami wysokich rzędów, czyli funkcjonałami. Dla porządku należy zauważyć, że w terminologii angielskiej istnieją dwa synonimy tego terminu: "applicative programming" od "application" -- zastosowanie, w tym wypadku funkcji do argumentu (nie mylić z "application programming" -- programowanie użytkowe, w przeciwieństwie do systemowego); oraz "declarative programming" -- od deklaracji, stawiający nacisk na konstruowanie deklaracji (czyli definicji) funkcji jako na najważniejszą czynność programisty. W odróżnieniu od omawianego w tej pracy programowania funkcjonalnego tradycyjny styl programowania nazywa się imperatywnym, a języki programowania takie jak Pascal czy Fortran -- językami imperatywnymi.

1. Przykład z życia

Większość programistów na całym świecie zajmuje się na codzień rozmaitymi zastosowaniami ekonomicznymi (materiałówki, listy płac itp.). Ale i dla nich bardziej matematyczne spojrzenie na swoją pracę może okazać się korzystne. Żeby to uzasadnić przedstawię w tym rozdziale przykład programu o właśnie takiej niematematycznej motywacji, który dopiero po przetłumaczeniu na odpowiednio funkcjonały staje się prosty, elegancki i łatwy do zaprogramowania: listę płac przedsiębiorstwa.

Komputerowy system obsługi list płac powinien umożliwiać tworzenie nowych list (na przykład dla nowopowstających przedsiębiorstw), uaktualnianie starych list oraz zasięganie informacji ile zarabiają poszczególne osoby. To ostatnie wymaganie wskazuje, że mamy do czynienia z zależnością funkcyjną między nazwiskami osób a ich zarobkami, czyli że taka lista płac lp jest funkcją przyporządkowującą osobie liczbę oznaczającą jej zarobki:

lp : Nazwiska --> Nat

gdzie

Nazwiska -- typ wszystkich możliwych nazwisk, na przykład napisów o długości nie większej niż 100

Nat -- typ liczb naturalnych

są typami zdefiniowanymi już wcześniej. A więc typ list płac jest typem, do którego należą funkcje takie jak lp ; zapisujemy to następująco:

(1) Listypłac = Nazwiska --> Nat

Załóżmy, że

lp : Listypłac -- czyli lp jest jakąś listą płac
nz : Nazwiska -- czyli nz jest jakimś nazwiskiem

wtedy lp(nz) , czyli wynik zastosowania funkcji lp do argumentu nz , jest na podstawie (1) typu Nat , czyli jest liczbą naturalną. Przyjmijmy, że lp(nz) ma wartość 0 jeśli osoba o nazwisku nz nie figuruje na liście płac lp .

Pierwszy interesujący nas funkcjonał będzie podawał odpowiedź na pytanie ile zarabiają podane osoby. Dokładniej, funkcjonał Zarobki będzie miał jako pierwszy argument listę płac a jako drugi argument skończony ciąg nazwisk, na przykład

```
["Kowalski" ; "Nowak" ; "Wiśniewski"]
```

natomiast w wyniku będzie dawał ciąg par, na przykład

```
[("Kowalski" , 19800) ;
 ("Nowak" , 17200) ;
 ("Wiśniewski" , 23700)]
```

(Tu i niżej używam następujących oznaczeń:

1. Konkretny ciąg ujmuję w nawiasy kwadratowe, a jego elementy oddzielam średnikami
2. [] oznacza ciąg pusty, czyli nie zawierający ani jednego elementu
3. [a] oznacza ciąg zawierający dokładnie jeden element
4. ...^... jest tak zwaną konkatenacją ciągów, to znaczy działaniem na ciągach określonym tak:

   [a1;...;an] ^ [b1;...;bk] = [a1;...;an;b1;...;bk]

5. Typ ciągów skończonych o elementach typu A oznaczam przez A*
6. (a,b) oznacza parę złożoną z elementów a i b
7. Iloczyn kartezjański typów A i B , czyli typ wszystkich par (a,b) gdzie a:A i b:B oznaczam przez A x B

)

A więc typ funkcjonału Zarobki jest następujący:

(2) Zarobki : Listypłac x Nazwiska* --> (Nazwiska x Nat)*

a sam funkcjonał można zdefiniować rekursywnie:

```
(3)     LETREC  Zarobki (lp , ciągnazwisk)  =
                  IF  ciągnazwisk == []  THEN  []
                  ELSE
                    [(hd (ciągnazwisk) , lp (hd (ciągnazwisk)))]
                    ^ Zarobki (lp , tl (ciągnazwisk))
```

(Dalszy ciąg oznaczeń:

8. Słowo LETREC oznacza, że następuje definicja rekursywna, w tym wypadku jest to rekursywna definicja funkcji Zarobki

9. Konstrukcja IF ... THEN ... ELSE ... tworzy zwykłe wyrażenia warunkowe; należy zwrócić uwagę, że w przeciwieństwie do instrukcji warunkowych w imperatywnych językach programowania nie można w niej opuścić części ELSE ... (por. rozróżnienie między wyrażeniem warunkowym a instrukcją warunkową np. w Algolu 60)
10. Czynimy rozróżnienie między znakiem = , który oznacza "jest zdefiniowane przez", a znakiem == , który oznacza równość
11. Funkcje hd i tl działające na ciągach uważamy za standardowe, przy czym

```
hd : A* --> A    --   głowa (head) ciągu:
                        hd [a1;a2;...;an] = a1
tl : A* --> A*   --   ogon (tail) ciągu:
                        tl [a1;a2;...;an] = [a2;...;an]
```

)

Definicję (3) można uważać za opis algorytmu, ponieważ jest ona przepisem, jak prowadzić obliczenia:

```
Zarobki (lp, ["Kowalski" ; "Nowak"])  =

     (z części  ELSE ...  definicji (3))

=  [(hd ["Kowalski" ; "Nowak"] ,
     lp (hd ["Kowalski" ; "Nowak"]))]
   ^ Zarobki (lp , tl ["Kowalski" ; "Nowak"])  =

     (ponieważ  hd ["Kowalski" ; "Nowak"] = "Kowalski" ,
      a  tl ["Kowalski" ; "Nowak"] = "Nowak" )

=  [("Kowalski" , lp ("Kowalski"))]
   ^ Zarobki (lp , ["Nowak"])  =

     (załóżmy, że  lp ("Kowalski") = 19800 )

=  [("Kowalski" , 19800)] ^ Zarobki (lp , ["Nowak"])  =

     (z części  ELSE ...  definicji (3), ponieważ
      hd ["Nowak"] = "Nowak"  a  tl ["Nowak"] = [] )

=  [("Kowalski",19800)] ^ [("Nowak" , lp ("Nowak"))]
   ^ Zarobki (lp , [])  =

     (załóżmy, że  lp ("Nowak") = 17200 ; dalej z części
      THEN ..  definicji (3))

=  [("Kowalski",19800)] ^ [("Nowak",17200)] ^ []  =

=  [("Kowalski",19800) ; ("Nowak",17200)]
```

Następnym potrzebnym funkcjonałem jest uaktualnianie list płac: mając daną listę płac oraz ciąg poprawek tworzymy nową listę płac. Każda poprawka to para (nazwisko , kwota) :

```
(4)     Uaktualnij : Listypłac x (Nazwiska x Nat)* --> Listypłac

(5)     LETREC  Uaktualnij (lp , ciągpoprawek)  =
                  IF  ciągpoprawek == []  THEN  lp
                  ELSE
                    Uaktualnij (Zmień (lp , hd (ciągpoprawek)) ,
                                tl (ciągpoprawek)
                               )
```

gdzie

```
(6)     Zmień : Listypłac x (Nazwiska x Nat) --> Listypłac

(7)     LET  ( Zmień (lp , (nz,n)) ) (nz')  =
               IF  nz' == nz  THEN  n
               ELSE  lp (nz')
```

(Jeszcze jedno oznaczenie:

12. Słowo LET oznacza, że następuje definicja nierekursywna; dalej będą podane przykłady, że rozróżnianie między LET i LETREC jest konieczne

)

Zgodnie z powyższą definicją wyrażenie Zmień (lp , (nz,n)) oznacza funkcję lp uaktualnioną przez wartość n dla argumentu nz .

Potrzebujemy jeszcze jednej funkcji: tworzenie pustej listy płac:

```
(8)     Pusta : Listypłac
(9)     LET  Pusta (nz)  =  0
```

Dotychczas napisane definicje (3), (5), (7) i (9) w pełni określają interesujący nas system obsługi list płac. Towarzyszące im deklaracje typów: (2), (4), (6) i (8) mogą służyć jako komentarze lub jako zabezpieczenie przed błędami. Jasne jest, że konstruując definicje rekursywne jakieś komentarze tej postaci powinniśmy mieć przed oczyma.

Następujące przykłady pokazują, że te cztery niewielkie definicje dają wszystko, o czym moglibyśmy zamarzyć, konstruując listy płac. Załóżmy, że dysponujemy translatorem języka funkcjonalnego i że ten translator przyjął już definicje (3), (5), (7) i (9). W dalszym ciągu interakcyjnej sesji z komputerem możemy na przykład wykonać następujące kroki:

```
LET  Lista1  =  Uaktualnij (Pusta ,
                            [("Kowalski" , 19800) ;
                             ("Nowak" , 17200) ;
                             ("Wiśniewski" , 23700)]
                           )
```

W ten sposób utworzyliśmy nową listę płac o nazwie Lista1 zawierającą trzech pracowników.

```
LET Lista1 = Uaktualnij (Lista1 ,
                         [("Malinowicz",12500)]
                        )
```

Dopisaliśmy do listy płac nowego pracownika przyznając mu na początek raczej niskie pobory, stara Lista1 przestała istnieć. W powyższej definicji nie można zastąpić słowa LET słowem LETREC , ponieważ spowodowałoby to zapętlenie w czasie obliczeń:

```
Lista1 ("Nowak") =

= Uaktualnij(Lista1,[("Malinowicz",12500)]) ("Nowak") =

= Zmień(Lista1 , ("Malinowicz",12500)) ("Nowak") =

= Lista1 ("Nowak") = ...
```

Możemy system zapytać:

```
Zarobki (Lista1 , ["Nowak" ; "Malinowicz" ; "Marucha"])
```

na co powinniśmy otrzymać odpowiedź:

```
[("Nowak",17200) ; ("Malinowicz",12500) ; ("Marucha",0)]
```

Możemy usunąć kogoś z listy płac:

```
LET Lista1 = Uaktualnij (Lista1 , [("Kowalski",0)])
```

albo wszystkim dać podwyżkę 15%:

```
LET Lista1 (nz) = (Lista1 (nz) * 115) DIV 100
```

(Oznaczenia działań na liczbach naturalnych:

13. * oznacza mnożenie, a DIV -- dzielenie całkowite
)

Możemy też połączyć naszą listę z inną istniejącą listą płac:

```
LET Lista3 (nz) =
      IF Lista1 (nz) == 0 THEN Lista2 (nz)
      ELSE Lista1 (nz)
```

Który z klasycznych języków programowania dostarczyłby tak silnego i elastycznego aparatu kosztem tak niewielkiego wysiłku programisty (patrz definicje (3), (5), (7) i (9)) ? Jeśli Czytelnik nie czuje się jeszcze przekonany o wyższości programowania funkcjonalnego nad imperatywnym, niech spróbuje stworzyć analogiczny system w Pascalu lub Fortranie. Jestem pewien, że po zapisaniu samym kodem takiej ilości papieru, jaką zajął cały ten rozdział, jeszcze nie osiągnie tego, co umożliwiły cztery niepozorne definicje.

Odpowiedni program napisany klasycznie musiałby opisywać sposób implementacji listy płac za pomocą tablic haszowych, drzew binarnych lub innych struktur. Jeśli, jak to zakładałem powyżej, mamy możliwość implementowania ogólnych funkcji i funkcjonałów,

to możemy się tego typu problemami komputerowymi nie zajmować dopóki nie zależy nam na szybkości działania. Ale ponieważ czasem nam na niej zależy, więc z nadejściem programowania funkcjonalnego tablice haszowe i drzewa binarne zapewne nie wymrą, tak samo, jak z nadejściem języków imperatywnych wysokiego poziomu nie wymarły asemblery i mikroprogramowanie.

2. Na początku była lambda

Jaka jest wartość funkcji x+1 dla argumentu 5 ?

Na pierwszy rzut oka 6 . Ale ściśle biorąc należałoby raczej powiedzieć, że pytanie jest źle sformułowane, przecież wyrażenie x+1 wcale nie określa funkcji. Należałoby poprawniej zapytać:

Jaka jest wartość funkcji, która dowolnemu x -owi przyporządkowuje x+1 , dla argumentu 5 ?

Jako skrót wyrażenia "funkcja, która dowolnemu x -owi przyporządkowuje x+1 " przyjęło się pisać:

\x.x+1

przy czym znak \ czytamy "lambda". Ogólniej, jeśli x jest zmienną, a e wyrażeniem, w którym występuje x , to

\x.e

oznacza: "funkcja, która dowolnemu x -owi przyporządkowuje e ".

Funkcje opisane lambda-wyrażeniami można w prosty sposób stosować do innych wyrażeń, np.

(\x.x+1) 5 = 5+1

I ogólniej:

(1) (\x.e) e1 = e[e1/x]

gdzie e[e1/x] oznacza wyrażenie powstałe z e przez "ostrożne" zastąpienie zmiennej x przez wyrażenie e1 . Żeby nie przeładowywać popularnego opracowania szczegółami technicznymi, nie zamierzam ściślej definiować słowa "ostrożne", ale mam nadzieję, że poniższe przykłady wyjaśnią o co chodzi.

Przykład 1: f = \x.x
Dla dowolnego y :

f y =
= (\x.x) y = (na podstawie (1))
= y

czyli funkcja f jest identycznością.

Przykład 2: g = \x.c dla pewnej stałej c
Dla dowolnego y :

```
g y   =
=   (\x.c) y   =                          (na podstawie (1))
=   c
```

czyli g jest funkcją stałą.

Przykład 3: h = \x.f dla f z przykładu 1
Dla dowolnego y : h y = f , czyli jest to funkcjonał stały dający w wyniku funkcję identycznościową. Zauważmy, że

```
h y   =                                   (z przykładu 1)
=   (\x.(\x.x)) y   =
=   y
```

-- w ostatnim przejściu wewnętrzny x nie został zastąpiony przez y ; byłoby to "nieostrożne", ponieważ wewnętrzny x jest związany inną lambdą.

Przykład 4: k = \x.(\y.x+y)
Jest to funkcjonał, który dowolnemu x -owi przyporządkowuje funkcję, która dowolnemu y -owi przyporządkowuje wartość x+y . Odpowiada to funkcji dwóch zmiennych. Oto jak funkcja k działa na wyrażeniach 3*y oraz 2*y :

```
k (3*y) (2*y)   =   (z definicji k )
=   (\x.(\y.x+y)) (3*y) (2*y)   =
                    (przenazwowanie wewnętrznej zmiennej y
                     dla "ostrożności")
=   (\x.(\y'.x+y')) (3*y) (2*y)   =          (reguła (1))
=   (\y'. 3*y + y') (2*y)   =                (reguła (1))
=   3*y + 2*y
```

Zwróćmy uwagę, że przy "nieostrożnym" zastępowaniu, czyli bez przenazwowania, wynik byłby błędny:

```
...   =
=   (\y. 3*y + y) (2*y)   =
=   3*(2*y) + 2*y
```

Zasada (1) przekształcania lambda-wyrażeń nosi nazwę beta-konwersji. Towarzyszy jej druga zasada zwana eta-konwersją lub ekstensjonalnością:

(2) \x. e (x) = e

którą można stosować o ile w wyrażeniu e nie występuje zmienna x .

Lambda-rachunek stanowi teoretyczną podstawę języków funkcjonalnych w tym samym sensie, w jakim maszyny Turinga stanowią teoretyczną podstawę języków imperatywnych. W najprostszej postaci wyrażenia lambda-rachunku mają postać:

```
<wyrażenie>  ::=  <zmienna>
                ;  <wyrażenie> <wyrażenie>
                ;  \ <zmienna> . <wyrażenie>
                ;  ( <wyrażenie> )
```

Druga z powyższych klauzul oznacza zastosowanie funkcji do argumentu. Trzecia oznacza lambda-abstrakcję.

Najprostszy język funkcjonalny powinien zawierać jeszcze możliwość wprowadzania definicji:

```
<definicja>  ::=  LET <zmienna> = <wyrażenie>
```

I to już cała składnia języka. Za reguły obliczeniowe przyjmuje się beta- i eta-konwersję. I to już (prawie) cała semantyka.

Najprostszy translator lambda-wyrażeń, to interpreter, który przyjmuje do wiadomości podane mu definicje, a potem używa ich razem z regułami konwersji do upraszczania podanych mu wyrażeń. Na przykład reakcją translatora na program:

```
LET  f = \x. \y. x y
LET  g = \x. x
f g g
```

(przyjmuje się, że w braku nawiasów lambda-abstrakcja wiąże w prawo a zastosowanie funkcji do argumentu wiąże w lewo i silniej niż lambda-abstrakcja, tak więc \x. \y. x y oznacza (\x. (\y.(x y))) , natomiast f g g oznacza (f g) g) powinno być wykonanie obliczenia:

```
f g g   =   (definicja funkcji  f )
=   (\x. \y. x y) g g   =   (beta-konwersja)
=   (\y. g y) g   =   (beta-konwersja)
=   g g   =   (definicja funkcji  g )
=   (\x.x) g   =   (beta-konwersja)
=   g
```

i wydrukowanie wyrażenia g .

Jest oczywiste, że taki program upraszczający wyrażenia jest łatwy do napisania. Zaskakujące natomiast może być, że w ten sposób otrzymujemy translator języka posiadającego pełną moc obliczeniową maszyn Turinga, czyli również wszystkich języków programowania. Oznacza to, że wszystko, co można zaprogramować w jakimś języku programowania, daje się również zaprogramować, choć może w bardziej skomplikowany sposób, w opisanym wyżej języczku. W lambda-rachunku można bowiem bez trudu zdefiniować wszystko, czego potrzebuje programista: wartości logiczne, liczby naturalne, wyrażenia warunkowe, pary kartezjańskie, ciągi, wreszcie rekursję, która bynajmniej nie jest tu cechą wbudowaną (pierwotną). Podanie odpowiednich definicji wykraczałoby poza ramy tej pracy, zresztą byłoby nudne, dlatego podam jeszcze tylko jeden przykład wskazujący, że program w czystym lambda-rachunku może się zapętlić. Powinno to uspokoić wszystkich, którzy są świadomi tego, iż prawdziwemu programowaniu zawsze towarzyszy niebezpieczeństwo ślepej pętli:

```
LET  F   =  \f. \x. f (x x)
LET  Fix  =  \f. (F f) (F f)
LET  Id  =  \x. x
Fix Id
```

Obliczenie wygląda następująco:

```
Fix Id   =   (def. funkcji  Fix  oraz beta-konwersja)
=   (F Id) (F Id)   =
                (def. funkcji  F  oraz beta-konwersja)
=   (\x. Id (x x)) (F Id)   =   (beta-konwersja)
=   Id ((F Id) (F Id))   =
                (def. funkcji  Id  oraz beta-konwersja)
=   (F Id) (F Id)   =
=   ...
```

i dalej wszystko będzie się powtarzać. Zdefiniowany powyżej funkcjonał Fix jest t.zw. funkcjonałem najmniejszego punktu stałego mogącym służyć do wprowadzenia rekursji.

Wszystkie języki funkcjonalne, również ten, którego użyłem w przykładzie w rozdziale 1, są rozszerzeniami lambda-rachunku o pewną ilość konwencji notacyjnych oraz stałych i funkcji standardowych. Istotne jest, że żadne z tych rozszerzeń nie wprowadzają do języka przypisań ani pętli (chociaż na siłę dałoby się to zrobić). Zamiast przypisań mamy możliwość definiowania coraz to nowych stałych (ewentualnie o tej samej nazwie, por. Lista1 w rozdziale 1). Zamiast pętli używamy rekursję.

Ważną cechą programów funkcjonalnych jest niemożliwość istnienia efektów ubocznych. Wszystko, co chcemy zrobić, musimy zrobić wprost. Redukuje to poważnie szansę popełnienia błędu. Po nabraniu pewnej wprawy w operowaniu funkcjonałami programowanie staje się prostsze niż w językach imperatywnych a programy są krótsze. Również translatory języków funkcjonalnych są mniejsze i łatwiejsze do napisania. Natomiast problemem jest efektywność programów.

3. Ile kosztuje programowanie funkcjonalne

Najczęściej krytykowanym aspektem programowania funkcjonalnego jest wysoki czas działania programów. W tym rozdziale omówię pewne okoliczności wpływające na czas obliczania wyrażeń w językach funkcjonalnych.

Pierwszą taką okolicznością jest architektura istniejących komputerów. Posiadają one adresowaną pamięć zorganizowaną w linię prostą, w której drzewa potrzebne do upraszczania wyrażeń implementuje się bardzo nieefektywnie. Istniejące procesory zaopatrzone są w specjalne rozkazy przyspieszające działania na adresach (np. dodaj 1 do rejestru adresowego), oraz działania na spójnych odcinkach pamięci (np. przepisz pole bajtów). Wszystko to faworyzuje tradycyjne programy imperatywne. Ostatnio są prowadzone badania nad architekturą dającą większe szanse programom funkcjonalnym, np. nad bezadresową pamięcią zorganizowaną w drzewo. Wydaje się, że na tej drodze będzie można osiągnąć istotne przyspieszenie. Przy okazji zwróćmy uwagę, że na poziomie

mikrooperacji typowa pamięć komputerowa jest i tak zorganizowana w drzewo binarne a adres jest ciągiem bitów oznaczających wybór lewej (zero) lub prawej (jeden) gałęzi; dopiero mikroprogramowane rozkazy języka wewnętrznego ukrywają tą drzewiastą strukturę.

Drugą okolicznością wywierającą zasadniczy wpływ na sprawność liczenia jest wybór właściwej kolejności działań. Rozważmy następującą parę funkcji:

```
        f : Nat --> Nat*

(1)     LETREC  f (n)  =
          IF  n == 0  THEN  []
          ELSE  [n] ^ f(n-1)
```

która oblicza ciąg [n ; n-1 ;...; 1] ; oraz

```
        g : Nat* x Nat --> Nat

(2)     LETREC  g (ciąg , n)  =
          IF  hd(ciąg) == n  THEN  n
          ELSE  hd(ciąg) * g (tl(ciąg) , n)
```

która oblicza iloczyn wyrazów ciągu aż do pierwszego wystąpienia liczby n . Przypatrzmy się dwóm różnym sposobom obliczenia wartości wyrażenia g (f(6) , 4) (łatwo pokazać, że g(f(n) , k) = (n!) DIV (k!) dla dowolnych n,k : Nat). Pierwszy sposób polega na tym, że wszystkie obliczenia prowadzimy od najgłębiej zagnieżdżonych wyrażeń; tak więc będziemy musieli obliczyć do końca ciąg f(6) według definicji (1) zanim przystąpimy do obliczania g (... , 4) według definicji (2):

```
      g (f(6) , 4)   =                                  (definicja (1))
      =   g ([6]^f(5) , 4)   =                          (definicja (1))
      =   g ([6;5]^f(4) , 4)   =
            (definicja (1) zastosowana jeszcze 4 razy)
      =   g ([6;5;4;3;2;1]^f(0) , 4)   =                (definicja (1))
      =   g ([6;5;4;3;2;1] , 4)   =                     (definicja (2))
      =   6 * g ([5;4;3;2;1] , 4)   =                   (definicja (2))
      =   6 * 5 * g ([4;3;2;1] , 4)   =                 (definicja (2))
      =   6 * 5 * 1
```

Drugi sposób polega na tym, żeby zawsze liczyć od najbardziej zewnętrznych funkcji w nadziei, że ostatecznie się okaże, że pewnych podwyrażeń można w ogóle nie obliczać:

```
      g (f(6) , 4)   =                                  (definicja (2))

      =   IF  hd (f(6)) == 4  THEN  1
          ELSE  hd(f(6)) * g (tl(f(6)) , 4)
```

A więc aby pójść dalej należy policzyć hd(f(6)) oraz tl(f(6)) -- oczywiście nie do końca, a tylko tyle ile niezbędne. Z definicji (1):

```
      hd(f(6))   =   hd ([6]^f(5))   =   6
      tl(f(6))   =   tl ([6]^f(5))   =   f(5)
```

przy czym wartości f(5) na razie jeszcze nie obliczamy, bo nie wiemy, czy będzie konieczna. Wobec tego

```
g (f(6) , 4)   =
=   6 * g (f(5) , 4)
```

żeby móc iść dalej trzeba policzyć hd(f(5)) oraz tl(f(5)) :

```
hd(f(5))   =   5
tl(f(5))   =   f(4)
```

przy czym znowu nie obliczamy wartości f(4) . Teraz

```
g (f(6) , 4)   =
=   ...   =
=   6 * 5 * g (f(4) , 4)
```

Teraz wystarczy policzyć hd(f(4)) : ponieważ wynosi on 4 , więc z definicji (2):

```
g (f(6) , 4)   =
=   ...   =
=   6 * 5 * 1
```

W ten sposób zaoszczędziliśmy sobie w ogóle liczenia f(4) , co za pierwszym razem wymagało pięciokrotnego zastosowania definicji (1).

Pierwszy sposób liczenia nosi nazwę obliczenia chętnego, lub z zapałem (eager evaluation) albo też obliczenia od wnętrza (innermost-first evaluation). Odpowiada mu wołanie parametrów przez wartość w językach imperatywnych. Drugi sposób liczenia, to obliczenie leniwe (lazy evaluation) albo obliczenie od zewnątrz (outermost-first evaluation). Odpowiada mu wołanie parametrów przez nazwę. Wołanie przez wartość stosuje się w językach imperatywnych dla ochrony argumentu przed ubocznymi efektami użycia funkcji. W językach funkcjonalnych nie ma żadnych efektów ubocznych, więc tą kosztowną ochronę można sobie darować. W wielu sytuacjach stosowanie obliczenia leniwego znacznie skraca czas działania programów. W wypadkach skrajnych jest to skrócenie od nieskończoności do wartości skończonej, to znaczy usunięcie ślepej pętli. Na przykład niech

```
f : Nat --> Nat

LETREC  f(n)  =
          IF  (n DIV 2) * 2 == n  THEN  1 + f (n DIV 2)
          ELSE  0
```

(czyli maksymalna potęga 2 , przez którą dzieli się n) oraz

```
g : Nat x Nat --> Nat

LET  g (k,l)  =
       IF  l == 0  THEN  0
       ELSE  k
```

(czyli 0 o ile 1 == 0 oraz k w przeciwnym razie). Teraz obliczając wartość g (f(0) , 0) otrzymujemy ślepą pętlę przy obliczeniu chętnym:

```
g (f(0) , 0)    =
=   g (1+f(0) , 0)    =
=   g (1+1+f(0) , 0)    =
=   ...
```

oraz wartość 0 przy obliczeniu leniwym:

```
g (f(0) , 0)    =
=   IF  0 == 0  THEN  0  ELSE  f(0)    =
=   0
```

Zwolennicy programowania funkcjonalnego uważają, że nie ma żadnych zasadniczych przyczyn, dla których programy funkcjonalne miałyby działać wolniej niż ich imperatywne odpowiedniki, choć przyznają, że dużo jest jeszcze w tej sprawie do zbadania. Samo zastąpienie obliczenia chętnego przez leniwe nie wystarcza. Problemem natomiast pozostaje duża ilość potrzebnej pamięci operacyjnej.

Trzecią okolicznością spowalniającą obliczenia funkcjonalne jest stosowanie funkcji bardzo wysokiego poziomu. Są to z samej swej natury obiekty dość kosztowne; aby się o tym przekonać, niech Czytelnik spróbuje na przykład prześledzić krok za krokiem działanie programu

```
LET  Lista1 =
        Uaktualnij (Pusta ,
                    [("Kowalski",19800) ; ("Nowak",17200)]
                   )
LET  Lista2 =
        Uaktualnij (Lista1 , [("Malinowicz",12500)])
Zarobki (Lista2 , ["Nowak"])
```

Korzystając z definicji funkcji Zarobki , Uaktualnij , Zmień i Pusta z rozdziału 1. Widać, że przy obliczeniu zarobków Nowaka należy przejść przez całą historię tworzenia listy płac Lista2 mimo, iż ostateczne ustalenie jego pensji nastąpiło już w pierwszym kroku tej historii. Ale zauważmy, że chodzi tu o operacje na funkcjonałach w ogóle niedostępne w językach niższego poziomu. Kto nie chce płacić ich wysokiego kosztu, może z nich zrezygnować i wtedy zarówno jego styl programowania jak i efektywność będą zbliżone do tradycyjnych programów imperatywnych. W szczególności może się to sprowadzić do operowania w języku funkcjonalnym na tablicach haszowych lub na uporządkowanych drzewach zbalansowanych.

Na zakończenie tego rozdziału wspomnę jeszcze o perspektywie przyspieszenia obliczeń funkcjonalnych rysującą się z nadejściem komputerów wieloprocesorowych. Z tej rewolucji niewiele wynika dla języków imperatywnych bo obarczanie programisty zadaniem koordynowania akcji wielu równolegle działających procesorów wydaje się nierealne; z drugiej strony nie ma algorytmów potrafiących automatycznie rozbić program na fragmenty niezależne, które można by powierzyć niezależnym procesorom.

Tymczasem dla języków funkcjonalnych jest to bardzo proste. Rozpatrzmy na przykład następujący funkcjonał obliczający maksimum danej funkcji na danym przedziale liczb całkowitych:

```
        Max : (Int --> Int) x Int x Int --> Int

(4)     LETREC  Max (f,n,k)  =
          IF  n+1 == k  THEN  f(k)
          ELSE  m (Max (f , n , (n+k) DIV 2) ,
                   Max (f , (n+k) DIV 2 , k)
                 )
```

(czyli maksimum funkcji f na przedziale (n+1)..k), gdzie m jest określone następująco:

```
        m : Int x Int --> Int

        LET  m (n,k)  =
               IF  n > k  THEN  n
               ELSE  k
```

(czyli większa z dwóch liczb). Jest oczywiste, że dwa wywołania rekursywne funkcji Max w definicji (4) są całkowicie niezależne, więc translator może powierzyć ich wykonanie niezależnym procesorom. Spowoduje to, iż czas obliczenia wyrażenia Max (f,n,k) stanie się proporcjonalny do log (k-n) zamiast klasycznego k-n .

Zrobiłem, co mogłem, aby wykazać, że nie jest tak tragicznie z efektywnością programowania funkcjonalnego, jak mogłoby się wydawać. Dla osób ciągle nieprzekonanych na miejscu będzie następująca uwaga. To prawda, że programowanie funkcjonalne jest kosztowne, ale coraz bardziej nas stać na zapłacenie tych kosztów. Tak jak wysokie koszty nie powstrzymały zapanowania języków imperatywnych wysokiego poziomu, tak samo nie od kosztów będą zależeć losy programowania funkcjonalnego.

4. Programowanie bez translatora

Porównajmy następujące dwa fragmenty programów w Pascalu:

```
c:= 0 ;                          c := 0 ;
WHILE  b > 0  DO                 WHILE  b > 0  DO
BEGIN                            BEGIN
  b := b - 1 ;                     IF  b MOD 2 = 1
  c := c + a                         THEN  c := c + a ;
END                                b := b DIV 2 ; a := a + a
                                 END
```

Oba robią to samo: obliczają (w c) iloczyn a * b . Intuicyjnie jest to dosyć jasne, można tego również dowieść (zauważywszy, że w obu przypadkach przejście przez ciało pętli nie zmienia wartości wyrażenia a*b+c mimo, iż zmienia wartości zmiennych a , b i c). Ale nie są one bynajmniej równoważne, na przykład pierwszy z nich nie zmienia wartości zmiennej a podczas gdy drugi ją zmienia. Fakt, że oba programy "robią dla nas to samo"

jest trudny nie tylko do udowodnienia, ale nawet do ścisłego sformułowania.

Przyjrzyjmy się teraz funkcjonalnym odpowiednikom tych programów:

```
LETREC  f (a,b)  =                LETREC  g (a,b)  =
  IF  b == 0  THEN  0               IF  b == 0  THEN  0
  ELSE  f (a , b-1) + a             ELSE
                                      IF  b MOD 2 == 1
                                      THEN
                                        g (a+a , b DIV 2) + a
                                      ELSE  g (a+a , b DIV 2)
```

Sformułowanie twierdzenia o równoważności jest teraz proste:

(1) dla dowolnych a,b : Nat zachodzi f(a,b) == g(a,b)

Dowód (1) prowadzi się przez indukcję matematyczną po b według definicji funkcji f stosując następujący lemat:

(2) g(a,b-1) + a == g(a,b) dla dowolnych a,b : Nat , b > 0

którego dowodzi się przez indukcję po b według definicji funkcji g rozpatrując przypadki b MOD 2 == 1 oraz b MOD 2 == 0 .

Programy imperatywne są obiektami bardzo statycznymi i nieruchawymi. Z trudnością dają się porównywać, przekształcać lub w inny sposób manipulować. Programiści wykonują te czynności opierając się na zawodnej intuicji. Przeciwnie, na rekursywnych definicjach funkcji można działać wprost, a metoda dowodowa jest zawsze ta sama: indukcja i rozpatrywanie przypadków. Co więcej, próba wykazania, że programy pascalowe z początku tego rozdziału wyliczają tą samą wartość c , musi doprowadzić do stosowania własności działań arytmetycznych na liczbach naturalnych, które dowodzi się analogicznie do (1) i (2). A więc dowodzenie czegokolwiek o programach imperatywnych i tak sprowadza się do dowodu (być może niejawnego) dla odpowiadających programów funkcjonalnych.

Łatwość manipulowania definicjami rekursywnymi powoduje, że języki funkcjonalne są dobrymi językami specyfikacji problemów. Nawet jeśli ostatecznie nasz program ma działać w języku imperatywnym, warto napisać wpierw jego pierwowzór w języku funkcjonalnym, a następnie przetranskrybować go na żądany język imperatywny. Wielu programistów zaczyna tworzenie programu w asemblerze od napisania go w jakimś języku wysokiego poziomu, lub przynajmniej umieszcza obok komentarze przypominające język wysokiego poziomu; posiadanie lub nieposiadanie translatora tego języka komentarzy nie ma dla nich żadnego znaczenia. Języki funkcjonalne są językami jeszcze wyższego poziomu i wydają się jakby stworzone do bardziej zorganizowanego programowania w językach imperatywnych.

5. Przewodnik po literaturze

Czytelnikowi zainteresowanemu językami funkcjonalnymi doradzam sięgnięcie po pozycję [Fun82]. Jest to zbiór artykułów ujmujących sprawę z wielu różnorodnych punktów widzenia, od architektury komputerów do zastosowań. Podane są tam przykłady konkretnych języków funkcjonalnych oraz ich podstawy teoretyczne. Znajdują się tam również liczne odsyłacze do literatury. Należy również polecić książkę [Hen80].

Historycznie programowanie funkcjonalne wywodzi się od lambda-rachunku [Chu41]. Ale czytelnikowi współczesnego bardziej przyda się chyba nowszy wykład lambda-rachunku w książce [Sto77].

Manifestem programowym, od którego należy liczyć erę programowania funkcjonalnego, jest [Bac78].

Pierwszym historycznie pra-językiem funkcjonalnym był Lisp ([McC62] , [Mar80]) mimo, że ostatnio uważa się go raczej za język z pogranicza niż funkcjonalny. Do bardziej popularnych języków funkcjonalnych należą Hope [Bur80], KRC [Fun82], ML [Gor79] i OBJ [Gog79]. Wyrosły one z różnych potrzeb i różnice między nimi są większe niż można by wnosić z niniejszej pracy.

Na korzyści ze stosowania obliczenia leniwego zwrócił uwagę artykuł [Fri76].

W końcu z [Boy75] i [Dar77] wywodzą się pierwsze próby przekształcania i dowodzenia własności programów funkcjonalnych.

Bibliografia

[Bac78] Backus J. "Can programming be liberated from the von Neumann style? A Functional style and its algebra of programs" Communications of the ACM 21, 613-641, 1978

[Boy75] Boyer R., Moore J. "Proving theorems about LISP functions" Journal of the ACM 22, 129-144, 1975

[Bur80] Burstall R.M., MacQueen D.B., Sannella D.T. "Hope: an experimental applicative language" Proceedings of 1980 LISP Conference, 136-143, Stanford, California, 1980

[Chu41] Church A. "The calculi of LAMBDA-conversion" Princeton University Press, Princeton, New Jersey, 1941

[Dar77] Darlington J., Burstall R. "A transformation system for developping recursive programs" Journal of the ACM 24, 44-67, 1977

[Fri76] Friedman D.P., Wise D.S. "CONS should not evaluate its arguments" Automata, Languages, Programming (ed. S. Michaelson, R.Milner), 257-284, Edinburgh, 1976

[Fun82] "Functional programming and its applications -- an advanced course" (ed. J.Darlington, P.Henderson and D.A. Turner) Cambridge University Press, 1982

[Gog79] Goguen J.A., Tardo J.J. "An introduction to OBJ: a language for writing and testing formal algebraic program specifications" Specifications of Reliable Software Conference Proceedings, Cambridge, Massachusetts, 1979

[Gor79] Gordon M.J., Milner R., Wadsworth C. "Edinburgh LCF -- a mechanised logic of computation" Lecture Notes in Computer Science 78, Springer-Verlag 1979

[Hen80] Henderson P. "Functional programming, its application and implementation" Prentice-Hall, Englewood Cliffs, New Jersey, 1980

[Mar80] Martinek J. "LISP -- opis, realizacja i zastosowanie" WNT, Warszawa, 1980

[McC62] McCarthy J. i inni "LISP 1.5 Programmer's Manual" MIT Press, Cambridge, Massachusetts, 1962

[Sto77] Stoy J. "Denotational semantics: the Scott-Strachey approach to programming language theory" MIT Press, Cambridge, Massachusetts, 1977

Jesienna Szkoła PTI
Mrągowo, listopad 1986

INFORMATYKA SZKOLNA I JEJ PROBLEMY

doc. dr hab. Stanisław Waligórski
Instytut Informatyki
Uniwersytetu Warszawskiego
00-901 Warszawa, PKiN p.850
tel. 268258.

Początek bieżącego roku szkolnego spowodował lawinę wypowiedzi i artykułów prasowych na temat szkolnej informatyki. Nic dziwnego: jest to pierwszy rok, w którym Ministerstwo Oświaty zaczyna wprowadzać, bardzo powoli i ostrożnie, nowy przedmiot nadobowiązkowy w liceach: elementy informatyki. Tylko niewielka ilość szkół będzie mogła w tym roku spełnić warunki określone przez Ministerstwo jako niezbędne dla prowadzenia tych zajęć. Wprowadzenie informatyki do szkół na szerszą skalę wymaga jeszcze wykonania wielu prac przygotowawczych. Ale zainteresowanie komputerami i informatyką jest już duże, zwłaszcza wśród młodzieży. Mikrokomputery trafiają do szkół, domów i klubów tam, gdzie dotychczas nie było żadnej możliwości zetknięcia się z informatyką, a o jej zastosowaniach można było mieć tylko dość mgliste pojęcie.

Przysłuchując się dyskusjom na temat informatyzacji szkół moż-

na stwierdzić, jak wiele narosło wokół tej sprawy nieporozumień. Jest to zrozumiałe: przy tak dużym zainteresowaniu tylko niewielka część osób angażujących się w działania i dyskusje może wykazać się odpowiednim doświadczeniem i znajomością rzeczywistych problemów informatyzacji kształcenia. A właśnie tu dobre rozumienie istniejącej sytuacji i możliwości ma decydujące znaczenie dla wyboru właściwej strategii postępowania. Tutaj też, w tej dziedzinie, pokusy działań pozornych, na pokaz, oraz znajdowania cudownych a łatwych rozwiązań istniejących trudności są szczególnie silne. Błędy są za to kosztowne i trudne do naprawienia.

Problemy informatyki szkolnej można z grubsza podzielić na trzy kategorie. Pierwsza obejmuje wszystkie sprawy związane z wprowadzaniem informatyki do szkół: organizacyjne, techniczne, finansowe itd. Druga wiąże się z zapewnieniem prawidłowego korzystania ze środków informatycznych, komputerów i oprogramowania, gdy one już znajdą się w szkołach. O trzeciej można powiedzieć, że zawiera najwięcej zagadnień badawczych, typowych dla informatyki: są to problemy związane z tworzeniem i doskonaleniem narzędzi informatycznych, oprogramowania i sprzętu, tak aby zapewnić najlepszą (osiągalną w istniejących warunkach zaopatrzenia i pracy szkół) jakość i wydajność.

Jest to podział dość zgrubny, ale jeśli już zgodzimy się, że można go przeprowadzić, to warto także zauważyć, że problemów tych trzech kategorii nie można rozważać oddzielnie, gdyż sposób rozwiązania jednych determinuje możliwości rozwiązywania innych. Ale wprowadzenie takiego podziału pozwoli usystematyzować dalszy ciąg tych rozważań.

Trzeba tu wyraźnie zaznaczyć, że mówiąc o wprowadzaniu informatyki do szkół nie mam na myśli tylko przedmiotu o odpowiedniej nazwie, ale użycie komputerów (i ich oprogramowania, co jest nie-

rozłączne) w jakichkolwiek zajęciach prowadzonych przez szkołę. No i oczywiście nie chodzi o to, by z młodzieży szkolnej robić na przykład programistów czy operatorów komputerów, ale by dać jej elementarne wiadomości o ich użytkowaniu i stosowaniu, które przydadzą się jej w przyszłości, gdy nieuchronna informatyzacja życia poczyni dalsze postępy.

Polskie Towarzystwo Informatyczne od początku swego istnienia zwracało wiele uwagi na sprawy informatyzacji szkół. Ci, którzy czytują Biuletyn PTI, znają kolejne wypowiedzi i uchwały władz Towarzystwa w tej sprawie. To właśnie PTI wyraźnie powiedziało, że wszelkie próby nauczania informatyki bez dostępu do komputerów, werbalnie, oraz przez osoby [illegible] szkodliwe. To PTI zalecało ostrożność w planowaniu masowego wprowadzania [illegible] informatyki do szkół, wskazywało skutki możliwych błędów i wy[illegible] dało się w sprawie rozwiązań organizacyjnych. Niektóre wypowiedzi PTI, jak np. ta o konieczności zachowania języka p[illegible] cych naleciałości, w nauczaniu informatyki, były niegdyś powodem ostrych polemik, ale wydaje się, że praktyka potwierdziła [illegible] ność tego stanowiska. Na zlecenie Ministerstwa Oświaty i Wychowania zespoły PTI opracowały projekt programu w zakresie wiedzy informatycznej, którego spora część została uwzględniona w ostatecznych propozycjach Ministerstwa, oraz program przedmiotu "Elementy informatyki", który jako fakultatywny jest wprowadzany do szkół właśnie od bieżącego roku, stopniowo w miarę powstawania możliwości w coraz większej, w następnych latach, liczbie szkół. PTI zainicjowało również dyskusję nad polskimi wersjami języków programowania, której wynikiem było stworzenie polskiej wersji języka Logo.

Wprowadzenie informatyki do szkół na większą skalę jest utrudnione brakiem dostatecznej liczby komputerów, które nadawałyby się do użycia w tych szkołach. Zwykle ta właśnie kwestia zwraca najbar-

dziej uwagę, do tego stopnia, że czasem paraliżuje myślenie o czymkolwiek innym. To jasne, że bez komputerów w ogóle nie ma co zabierać się do informatyzacji, ale nie jest to problem najważniejszy ani - w obecnych warunkach - najtrudniejszy. Konieczne jest

1. Przygotowanie nauczycieli,
2. Przygotowanie pomocy naukowych, podręczników, oprogramowania,
3. Dostarczenie szkołom komputerów odpowiadających wymogom kształcenia

Jak świadczą przykłady innych krajów, które już przez to przechodziły wcześniej, próby zignorowania lub ominięcia sprawy przygotowania nauczycieli z reguły kończą się niepowodzeniem tak prowadzonej "informatyzacji". W naszym przypadku najważniejsza trudność polega na tym, że dotychczas przeciętny absolwent studiów wyższych, zwłaszcza taki, który zostawał nauczycielem, miał z reguły zbyt słabe przygotowanie informatyczne. Jego możliwości pracy na jakimkolwiek komputerze, nawet w trybie wsadowym, były na ogół bardzo ograniczone. Nie mógł więc opanować w dostatecznym stopniu nie tylko współczesnych metod pracy z komputerem, czyli tego, czego miałby uczyć teraz w szkole, ale czasem nawet jakichkolwiek.

Słabe wyposażenie uczelni (zwłaszcza tych, które kształcą nauczycieli) w sprzęt informatyczny łączy się z ostrym brakiem nauczycieli akademickich informatyki o pełnych kwalifikacjach. O zakresie umiejętności programowania i czynnego posługiwania się środkami i metodami informatyki, przekazywanych studentom kierunków nie informatycznych, decydują często (w sensie negatywnym) ograniczone możliwości kadrowe i sprzętowe wydziału i uczelni, a nie, tak jak być powinno, potrzeby ich dyscypliny naukowej ani poziom wiedzy osiągniętej w skali światowej. Zbyt często zajęcia na tych kierunkach, prowadzone pod różnymi nazwami, jak ETC, Programowanie i Maszyny cyfrowe, Programowanie i Metody Numeryczne, sprowadzają się

tylko do pobieżnej nauki jednego z języków programowania. Często nie są one prowadzone przez informatyków. Często studenci uzyskują w ten sposób wiadomości przestarzałe, dające dość skrzywiony pogląd na metody i możliwości informatyki współczesnej.

Konieczne jest więc nie tylko przygotowanie i właściwe przeszkolenie informatyczne nauczycieli czynnych zawodowo, ale także stworzenie warunków i możliwości prawidłowego kształcenia informatycznego przyszłych nauczycieli przez uczelnie. O powodzeniu nauczania informatyki w szkole i w ogóle wszelkich kroków zmierzających do wprowadzenia komputerów i metod informatycznych do szkół zadecydują w ostatecznych rachunku nauczyciele. O przygotowaniu informatycznym tych nauczycieli zadecydują z kolei ci, którzy ich tego będą uczyć, a więc nauczyciele akademiccy w uczelniach i instruktorzy kursów dokształcających. Ci z kolei muszą być właściwie przygotowani do tych zadań, zwłaszcza jeśli dotychczas ich działalność miała inne cele i była prowadzona w innym zakresie, co z reguły miało miejsce. Jak widać, mówimy o wielopiętrowej strukturze, obejmującej nie tylko tych nauczycieli, którzy w szkole mają zajmować się informatyką, ale także nauczycieli tych nauczycieli i tak dalej. Wąski szczyt tej piramidy powinni zajmować informatycy, znający środki i metody informatyki współczesnej i doświadczenia innych krajów w rozwoju informatyki szkolnej, umiejący przekazać tę wiedzę innym.

Widać jasno, że w naszych warunkach o skali prawdziwej, rzetelnie traktowanej informatyzacji kształcenia zadecyduje nie liczba zainstalowanych mikrokomputerów, ale właśnie liczba właściwie przygotowanych nauczycieli. Brak literatury, pomocy naukowych, oprogramowania, komputerów może co najwyżej tę sytuację tylko dodatkowo pogorszyć. Brakujące wyposażenie powinno być kierowane w pierwszej kolejności do ośrodków kształcących nauczycieli, czyli zaangażowanych w to uczelni i instytucji oświatowych. W drugiej kolejności i stop-

niowo do szkół.

Aby nie przedłużać tych ogólnych wywodów pominą kwestię literatury, oprogramowania i innych pomocy naukowych i metodycznych i przejdę do kwestii, która, choć trudna, jest najłatwiejsza z tych trzech: sprzętu.

Na pytanie, do czego ma w szkole służyć mikrokomputer, odpowiada - pośrednio - program przedmiotu "Elementy informatyki" (zob. np Matematyka nr 6, 1985, str. 284-295, albo Program Liceum Ogólnokształcącego (profil podstawowy i matematyczno-fizyczny) Elementy informatyki (uzupełniający przedmiot nauczania), Wydawnictwa Szkolne i Pedagogiczne, Warszawa 1985):

"Zasadniczym celem zajęć z elementów informatyki jest nauczenie metod rozwiązywania z pomocą komputera prostych problemów ... dostosowanych do wiedzy i umiejętności uczniów. W trakcie tych zajęć uczniowie powinni ... zdobyć praktyczne umiejętności posługiwania się szkolnym sprzętem informatycznym i jego oprogramowaniem". A ogólne warunki realizacji tego programu mówią: "Zajęcia z tego przedmiotu mogą być prowadzone wyłącznie w tych szkołach, które mogą zapewnić wszystkim uczestnikom dostęp do mikrokomputera z oprogramowaniem dostosowanym do realizacji tego programu. Pracę uczniów z komputerem należy traktować jako główny sposób przyswajania wiadomości. Trzeba także zapewnić uczniom dostęp do komputera poza formalnymi godzinami zajęć".

W podobny sposób można powiedzieć o warunkach i trybie korzystania z komputera na zajęciach z innych przedmiotów. Ponieważ zainteresowanie młodzieży jest duże, a komputerów stosunkowo mało, na inne ich zastosowanie w szkołach, poza tu opisanym, nie będzie już miejsca.

Jaki powinien być mikrokomputer szkolny, aby mógł prawidłowo spełniać takie zadania? Oto odpowiedź, sformułowana przez PTI i nas-

tępnie przyjęta przez Ministerstwo Oświaty i Wychowania oraz inne zainteresowane instytucje jako zasada oceny sprzętu:

"Mikrokomputer szkolny powinien spełniać następujące warunki:

- pamięć wewnętrzna nie mniej niż 64 kB,
- pamięć zewnętrzna na dyskach elastycznych,
- monitor ekranowy z grafiką (możliwością rysowania kresek),
- klawiatura odpowiadająca polskim normom i polski alfabet na wszystkich urządzeniach wyjściowych,
- możliwość przyłączenia drukarki,
- możliwość przyłączenia magnetofonu kasetowego,
- struktura otwarta, umożliwiająca dalszą rozbudowę konfiguracji, łączenie z innymi urządzeniami peryferyjnymi, przyłączanie do sieci mikrokomputerowej.

Pożądane byłoby również:

- urządzenie do lokalizacji punktu na ekranie: "mysz", manipulator albo pióro świetlne.

Jednocześnie mikrokomputer szkolny powinien umożliwiać przenoszenie na niego oprogramowania dydaktycznego pochodzącego z innych krajów. Nie spełnienie tego warunku będzie oznaczało konieczność tworzenia całego oprogramowania od podstaw."

Jest to rzeczywiście minimum, poniżej którego zajęć nie można bez obniżenia poziomu i efektywności zajęć prowadzonych na takich mikrokomputerach. Pamięć wewnętrzna 64 kB umożliwia korzystanie z większości języków programowania i programów użytkowych niezbędnych w szkołach. Teraz, gdy coraz popularniejsze stają się mikrokomputery z pamięcią wewnętrzną 128 kB i większą, ten wymóg zdaje się być dość skromny, choć w czasie, gdy był sformułowany, budził pewne protesty. Potrzebę posiadania dysków elastycznych przy każdym mikrokomputerze mogą zdecydowanie potwierdzić wszyscy ci, którzy próbowali używać na zajęciach mikrokomputery wyposażone w magnetofony, urządze-

nia bardzo zawodne i powodujące niewspółmiernie duże straty czasu przy przepisywaniu czegokolwiek z nich do pamięci komputera lub odwrotnie. Brak dysków uniemożliwiłby również korzystanie z jakiegokolwiek prawdziwego systemu operacyjnego.

Przyjęto - dość skromnie, że w czasie zajęć w szkolnej pracowni informatycznej nie może być więcej niż 3 uczniów na jeden komputer, chociaż z praktyki wiadomo, że przy samodzielnym rozwiązywaniu trudniejszych zadań trzeba by tę granicę obniżyć do 2 lub nawet 1. Liczebność grupy biorącej udział w zajęciach jest więc ograniczona przez liczbę komputerów. Typowa pracownia szkolna powinna mieć 10 mikrokomputerów i dwie drukarki.

Pomnożenie tego przez liczbę szkół, które mają takie laboratoria otrzymać w tej pięciolatce da nam ogólne oszacowanie zapotrzebowania na sprzęt dla szkół. Od razu widać, że nie udałoby się w tym czasie wyposażyć w takie laboratoria szkół podstawowych. Jest to jeden z głównych powodów, dla których podjęto decyzję o wprowadzaniu komputerów i nauczaniu podstaw wiedzy informatycznej (w tym przedmiotu "Elementy informatyki") w pierwszej kolejności w liceach ogólnokształcących i wybranych szkołach zawodowych. Szkoły podstawowe mogłyby otrzymać takie laboratoria tylko wyjątkowo. Ale również w tych warunkach laboratoria w szkołach będą instalowane stopniowo, w ciągu całego pięciolecia. Najbardziej optymalnym rozwiązaniem byłoby powiązanie tego z akcją przygotowania nauczycieli. Jeśli dodamy do tak szacowanego zapotrzebowania szkół jeszcze potrzeby uczelni i instytucji oświatowych zajmujących się kształceniem i dokształcaniem informatycznym nauczycieli, to bez wchodzenia w szczegóły można stwierdzić, że chodzi o dziesiątki tysięcy mikrokomputerów w ciągu najbliższych pięciu lat. O dostawach poniżej 10 tys. do końca pięciolecia w ogóle, jak się wydaje, nie warto mówić. Na szczęście nie trzeba: z opublikowanych ostatnio wprawie oświadczeń

wynika, że rząd przewiduje dostawy w skali odpowiadającej rzeczywistym potrzebom szkolnictwa. Podjęcie prac nad Elwro 800 Junior pozwala żywić nadzieję, że wymagania jakościowe także będą spełnione.

Przejdźmy teraz do innej kategorii problemów. Załóżmy, że mówimy tu o jednej szkole, wzorcowej w tym sensie, że dla niej wszystkie problemy ogólne zostały już pozytywnie rozwiązane. Jest laboratorium mikrokomputerowe z właściwym sprzętem i oprogramowaniem. Są zapewnione naprawy sprzętu w razie potrzeby i dostawy niezbędnych do działania laboratorium materiałów, jak dyskietki, papier, taśmy do drukarek itd. Jest nauczyciel, znający dobrze, a nawet, załóżmy, bardzo dobrze metody współczesnej informatyki w zakresie niezbędnym do pracy w szkole, umiejący sprawnie posługiwać się mikrokomputerem i przekazywać swą wiedzę uczniom. Ma do swej dyspozycji wystarczającą literaturę i umie ją czytać w językach, w jakich jest napisana. Jak on powinien prowadzić zajęcia z uczniami? Dla uproszczenia przyjmijmy, że chodzi o przedmiot "Elementy informatyki".

(Tym, którzy chcieliby powiedzieć, że takiej szkoły raczej tu nie ma i wobec tego to wszystko jest mało realne, odpowiedziałbym, że od odpowiedzi na takie pytania można by zacząć rozważania na temat, czego należy uczyć przyszłych albo już czynnych nauczycieli. Oczywiście zaraz potem trzeba by z kolei zapytać, czy realne utrudnienia wprowadzają jakieś modyfikacje i w którym kierunku.)

Opinie dydaktyków są raczej zgodne. Uczeń powinien mieć możność samodzielnego działania i wykazywania swej aktywności od jak najwcześniejszego momentu; praktycznie od samego początku zajęć. Powinien od razu widzieć skutki swego działania w wyraźnej i czytelnej formie, pozwalającej łatwo ocenić poprawność uzyskanych wyników i ich zgodność ze swoimi zamierzeniami. Błędy nie powinny powodować sytuacji stresowych ani powodować utrudnień w dalszej pracy nad tym samym zagadnieniem - przeciwnie, powinny być zachętą do

dalszego eksperymentowania i samodzielnego poszukiwania właściwych rozwiązań. Kolejne porcje wiedzy powinny rozszerzać nie tylko zasób praktycznych umiejętności ucznia, ale także zwiększać jego możliwości samodzielnego dochodzenia do coraz to nowych wyników, nawet połączonego z całkowicie samodzielnym odkrywaniem tych możliwości i wiadomości, które jeszcze nie były pokazane i omówione przez nauczyciela. Od samego początku należy zwracać uwagę na prawidłowe opanowanie procesu rozwiązywania zadań i problemów z pomocą komputera, zaczynając już od najprostszych zadań i kończąc na takich, których bez stosowania właściwych metod postępowania nie dałoby się łatwo rozwiązać. Uczeń powinien przyzwyczaić się do tego, że próba rozwiązania jakiegokolwiek poważniejszego zadania na komputerze powinna być poprzedzona analizą tego zadania i doborem właściwego algorytmu, powinien także umieć rozkładać zadania złożone na bardziej elementarne, dla których dobór algorytmów będzie prostszy, rozumiejąc strukturę zadań, do której taka metoda zstępująca doprowadzi oraz powiązań algorytmów rozwiązań zadań elementarnych i ich komputerowej realizacji, określonych przez tę strukturę. Nie należy jednak uczyć opracowania algorytmów wyłącznie teoretycznie, to znaczy bez następującego zaraz potem sprawdzenia ich na komputerze. Pozwoli to uniknąć rozwijania szkodliwej skłonności zastępowania opracowań konkretnych i sprawdzonych metod rozwiązania zadań opowiadaniami o tym, jak to można czy należałoby zrobić.

Dydaktycy zwracają także uwagę, że źle prowadzone zajęcia lub niewłaściwie użyty komputer mogą służyć ograniczaniu inicjatywy ucznia, odbierając mu samodzielność i spontaniczność działania. Trzeba zdawać sobie sprawę że użycie komputera na źle prowadzonych zajęciach może wzmocnić i utrwalić negatywne skutki użycia złej metody dydaktycznej lub koncepcji nauczania. Używanie komputerów w szkole w taki sposób i w takich sytuacjach jest zdecydowanie szkodliwe i należy

tego unikać.

Przytoczone tu uwagi i wskazówki metodyczne zostały wybrane dla przykładu. Ich pełniejszy i bardziej systematyczny wykład można znaleźć w fachowej literaturze, zajmującej się metodyką nauczania informatyki w szkołach. Ale nawet te fragmentarycznie ujęte zasady pozwalają zauważyć, jak bardzo różni się to podejście od metod poprzednio stosowanych, które zresztą jeszcze dotychczas są dość powszechne w ośrodkach o bardziej tradycyjnym nastawieniu lub słabym wyposażeniu, a m.in. w uczelniach kształcących nauczycieli.

Dawniej powszechnie stosowaną zasadą było, że zanim się przystąpiło do ćwiczeń praktycznych, trzeba było wysłuchać wykładu , poprzedzonego jeszcze czasami wstępem "teoretycznym". Grafika i dźwięk były stosowane rzadko, więc nie było mowy o tym, by móc ocenić wyniki pracy bezpośrednio po ich uzyskaniu i praktycznie na pierwszy rzut oka. Rozwiązywanie zadań, dających rezultaty w postaci liczb albo tekstów, jedynie możliwe z braku grafiki, stwarza studentowi zupełnie inne warunki pracy. Ograniczone możliwości pracy konwersacyjnej z komputerem, tak typowe do niedawna dla większości naszych uczelni, oraz przestarzałe oprogramowanie nie dawały możliwości nabycia właściwych nawyków pracy konwersacyjnej, swobodnego rozwiązywania zadań w dialogu z komputerem z wykorzystaniem wszystkich istotnych jego możliwości. Uświadomienie sobie tych i innych trudności jest ważne nie tylko ze względu na konieczność prawidłowego opracowania programów informatycznego kształcenia i dokształcania nauczycieli, ale także ze względu na to, że wykształceni przy pomocy takich starych metod mogą mieć tendencję do propagowania ich dalej i przekazywania ich młodszym, ze szkodą dla tych ostatnich. Prawie całkowite odcięcie uczelni i szkolnictwa od literatury światowej sprawia ponadto, że tacy "tradycjonaliści", nie mający możności zapoznania się z nowszymi wynikami i tendencjami rozwoju nauczania

mogą stanowić silny a konserwatywny czynnik opiniotwórczy.

Dobry kontakt z tym, co się dzieje na świecie, jest niezbędny również dla prawidłowego planowania i wykonywania prac nad oprogramowaniem i sprzętem własnym, albo nad udoskonalaniem oprogramowania które już jest używane w kształceniu. Pominę tu konstrukcję i produkcję sprzętu; zainteresowani znają zapewne dość dobrze już uzyskane wyniki, sytuację i rezultaty pracy konstruktorów. Dzieje się tu wiele interesujących rzeczy, ale z punktu widzenia wyposażania szkolnictwa wszystko to są rozwiązania jednostkowe, gdy tymczasem wynikiem znaczącym byłoby dopiero przekroczenie liczby 10 tys.sztuk. Dlatego użyłem tu słowa "produkcja".

O ile można się orientować, w zakresie produkcji i doskonalenia oprogramowania ruch jest znacznie słabszy. Pomijając nieznaczące statystycznie przypadki cudownych dzieci trzeba powiedzieć, że ta działalność wymaga bardzo wysokich kwalifikacji programistycznych oraz, co nie mniej ważne w przypadku oprogramowania użytkowego przeznaczonego do wspomagania zajęć w szkole, dobrej wiedzy dydaktycznej i pedagogicznej. Oczywiście nie muszą to być talenty jednych i tych samych osób.

Wróćmy do naszego przykładu idealnej szkoły. Wskazówki metodyczne, które byliśmy skłonni przekazać nauczycielowi, można teraz odwrócić. Oprogramowanie powinno być takie, aby maksymalnie ułatwić mu postępowanie w podany sposób i osiągnięcie założonych celów dydaktycznych. Powinno umożliwiać aktywne i samodzielne działanie ucznia. Powinno być konwersacyjne. Rezultaty działania programów powinny być czytelne , łatwo zrozumiałe i podane w atrakcyjnej formie. Błędy nie powinny powodować zagubienia ucznia, sytuacji stresowych ani powodować utrudnień w dalszej pracy z tym samym programem. Uczeń powinien mieć możność nie tylko samodzielnej pracy z programem, ale także, jeśli zechce, stwarzanie rozmaitych nowych

sytuacji w jego dialogu z komputerem, pozwalających na głębsze i samodzielne opanowanie przekazywanej przy pomocy tego programu wiedzy. Nie jest to oczywiście pełna lista wymagań, jakie można stawiać konkretnym programom szkolnym, ale wydaje się, że przynajmniej te warunki powinny być zawsze spełnione. Przeciwnie, oprogramowanie oparte na błędnych koncepcjach dydaktycznych, nie rozwijające jego uzdolnień i zainteresowań, nie stwarzające możliwości własnych dociekań lub w ogóle pozostawiające ucznia wyłącznie w biernej roli odbiorcy nie spełni swej roli, do której miało być stworzone, a często jego użycie może przynieść więcej szkody niż pożytku. Należy więc go raczej unikać. W pierwszych momentach fascynacji komputerem nawet produkty złej jakości mogą cieszyć się powodzeniem, gdy będzie jeszcze trwał urok nowości sytuacji i użytych środków, ale nie powinno to osłabiać trzeźwego i krytycznego spojrzenia na te sprawy.

Ostatnio towarzystwa naukowe zwracają uwagę na rozszerzający się zalew szmiry naukowej w postaci najrozmaitszych publikacji, pisanych bez odpowiedniej wiedzy i podstaw naukowych ale traktujących - z pozoru - o ważnych naukowych problemach współczesności. Nie ma pełnej analogii z przekazem komputerowym, oczywiście, ale warto zwrócić uwagę, że tak atrakcyjna rzecz, jak komputer, też mógłby być wykorzystany do przekazywania różnych pseudonaukowych pomysłów.

Z punktu widzenia informatyka warunki opracowania oprogramowania szkolnego różnią się od jego zwykłych warunków pracy. Chodzi o to, że komputery wykorzystywane przez zawodowców mają większą moc i możliwości, niż komputery szkolne, które zawsze będą musiały być tańsze i wobec tego mniej sprawne od innych. Programista musi więc dostosowywać się do takich ograniczeń, jak stosunkowo niska szybkość i pojemność pamięci, dysponowanie tylko określonymi typami urządzeń peryferyjnych itd. Jak pokazuje praktyka, zawodowcy potrafią sobie z tym radzić.

Jesienna Szkoła PTI
listopad 1986

CYFROWA GENERACJA OBRAZÓW

Jan Zabrodzki
Politechnika Warszawska
Instytut Informatyki
Nowowiejska 15/19
00-665 Warszawa

1. Wstęp

Rozwój technologii układów scalonych pociąga za sobą rozwój wielu innych dziedzin. Dotyczy to m.inn. rastrowej techniki graficznej. Współczesne układy VLSI oraz monitory graficzne umożliwiają konstrukcję stosunkowo tanich urządzeń dla cyfrowej generacji i wyświetlania obrazów o dobrej jakości i w stosunkowo krótkim czasie. Obserwuje się rozwój konstrukcji urządzeń typu stacja graficzna zdolnych do samodzielnej pracy, bądź przy współpracy z większym systemem, do odciążania go od obliczeń typowo "graficznych".

Celem pracy jest przedstawienie podstawowych problemów związanych z cyfrową generacją obrazów przy wykorzystaniu techniki rastrowej, zwłaszcza z punktu widzenia konstrukcji sprzętu.

Po wprowadzeniu w zagadnienia techniki rastrowej przedstawione są kolejno problemy związane z: konstrukcją pamięci obrazu, wyświetlaniem na monitorach kolorowych (łącznie z omówieniem ich podstawowych cech) i z generacją realistycznych obrazów

w czasie rzeczywistym. Dalsza część poświęcona jest omówieniu współczesnych konstrukcji stacji graficznych oraz prezentacji najnowszych układów VLSI projektowanych dla potrzeb konstrukcji urządzeń grafiki komputerowej.

W końcowej części przedstawiono wybraną metodę rozwiązywania zagadnień związanych z wyznaczaniem obiektów widocznych i ich oświetleniem - tzw. metodę śledzenia promieni. Metoda ta rokuje duże nadzieje jeśli chodzi o możliwość cyfrowej generacji realistycznych obrazów, jednak jej złożoność obliczeniowa przekracza możliwości obecnie produkowanych specjalizowanych urządzeń i jej stosowanie wymaga dużych systemów komputerowych. Przykłady takich systemów są przedstawione w zakończeniu pracy.

2. Technika rastrowa

W technice rastrowej obraz jest tworzony w postaci prostokątnego zbioru punktów. Każdemu z tych punktów przypisany jest odpowiedni odcień szarości lub kolor. Efekt spójności obrazu uzyskuje się dzięki skończonej rozdzielczości oka ludzkiego. Jakość obrazu zależy od liczby punktów i jest tym większa im większa jest rozdzielczość obrazu określona przez wymiary tablicy punktów.

Cyfrowa generacja obrazów wiąże się z koniecznością rozwiązania wielu problemów, zwłaszcza wtedy gdy celem jest generacja obrazów jak najwierniej odzwierciedlających obiekty rzeczywiste. Problemy te wynikają głównie z konieczności reprezentowania obrazu w postaci skwantowanej i za pomocą sprzętu o określonych wartościach parametrów funkcjonalnych. Rozwój techniki rastrowej związany jest więc z jednej strony z poszukiwaniem coraz to doskonalszych algorytmów dla generacji obrazów a z drugiej strony ze stałym ulepszaniem parametrów sprzętu obliczeniowego i wyświetlającego generowane obrazy. Obie te grupy zagadnień wzajemnie się przenikają i uzupełniają. Opracowywane są algorytmy, które pozwalają w sposób optymalny wykorzystać istniejący sprzęt jak też stosowane są rozwiązania sprzęto-

we optymalizowane z punktu widzenia realizacji określonych algorytmów.

W największym uproszczeniu, każdy system grafiki rastrowej można przedstawić za pomocą schematu blokowego jak na rys.1. Centralnym elementem systemu jest pamięć obrazu, w której zapisana jest informacja o każdym punkcie (pikselu) obrazu. Infor-

Rys.1. Schemat blokowy systemu rastrowego

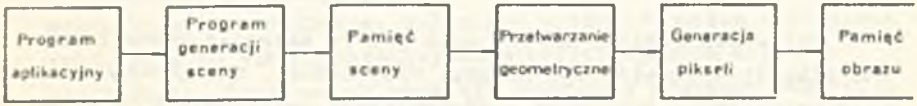
macje te są wytwarzane przez blok cyfrowej generacji obrazu. Natomiast za prezentację obrazu zawartego w pamięci obrazu odpowiedzialny jest blok wyświetlania.

Pojemność pamięci obrazu musi być co najmniej równa liczbie pikseli wyświetlanego obrazu. Przykładowo, dla ekranu o rozdzielczości 1024x1024 i ośmiobitowej informacji o kolorze piksela minimalna pojemność pamięci obrazu wynosi 1 MB. O ile uzyskanie pamięci o odpowiedniej pojemności nie stanowi obecnie istotnego problemu, to trudności związane są z zapewnieniem dostępu do tej pamięci przez oba współpracujące z nią bloki.

Dla uniknięcia efektu migotania, obraz powinien być wyświetlany z częstotliwością rzędu 60 Hz. Oznacza to w krańcowym przypadku konieczność zapisywania do pamięci 60 obrazów w ciągu

sekundy, czyli generowania informacji o kolejnych pikselach (co wiąże się z wykonaniem pewnej liczby operacji arytmetycznych) i zapisywaniu ich do pamięci co około 16 ns. (Dla porównania przypomnijmy, że czas wykonania pojedynczej instrukcji przez mikroprocesor 8086 wynosi kilkaset ns.) Równocześnie, z tą samą częstotliwością 60 obrazów na sekundę muszą być wyprowadzane informacje na ekran. Uwzględniając ograniczenia wynikające z konstrukcji monitorów okazuje się, że informacja o kolejnych pikselach powinna być odczytywana z pamięci co około 11 ns. Biorąc oba aspekty pod uwagę łącznie, czas dostępu do pamięci powinien być na poziomie 5 ns. Jak dotychczas nie są dostępne pamięci o takich czasach dostępu i wymaganej pojemności.

Na rys.2 przedstawiono zestaw zadań realizowanych przez blok cyfrowej generacji obrazu. Informacje wyjściowe dla procesu generacji obrazu są dostarczane przez program aplikacyjny.



Rys.2. Zadania realizowane przez blok cyfrowej generacji obrazu

Informacje te pozwalają określić poszczególne obiekty sceny a następnie skomponować scenę z tych obiektów. Opis sceny, np. w postaci zbioru wielokątów z odpowiedniami atrybutami, jest pamiętany w pamięci sceny. Opis trójwymiarowej sceny służy z kolei za podstawę dla utworzenia obrazu sceny w dwuwymiarowej

płaszczyźnie ekranu. Dokonują tego odpowiednie procedury geometryczne (zmiana układu współrzędnych, obroty, przesunięcia, skalowanie, rzuty perspektywiczne, obcinanie itd.). Często, dla uproszczenia obliczeń, operacji tych dokonuje się na ograniczonym zbiorze punktów np. na zbiorze wierzchołków wielokątów tworzących scenę. W takiej sytuacji po zakończeniu obliczeń konieczne jest generowania informacji o pozostałych punktach obrazu np. poprzez wypełnianie wielokątów odpowiednimi kolorami. Zależnie od stopnia złożoności sceny i wymaganej wierności odtworzenia sceny rzeczywistej wykonuje się różne zestawy algorytmów o różnej złożoności obliczeniowej.

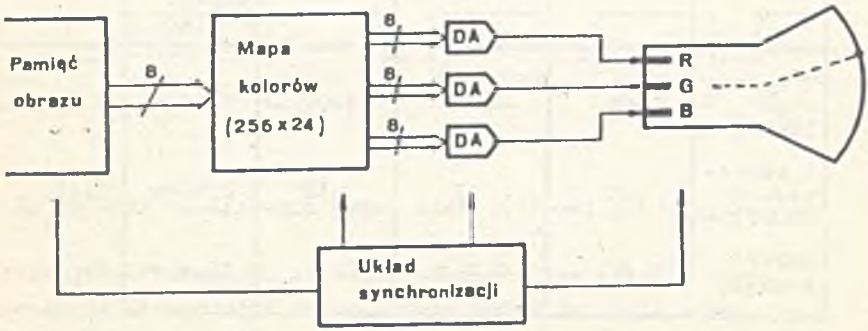
Dla zilustrowania problemu rozważmy operacje związane tylko z transformacjami geometrycznymi. Dla wykonania operacji skalowania, przesunięcia, obrotu i rzutu perspektywicznego jednego punktu o trzech wymiarach trzeba wykonać 20 operacji zmiennoprzecinkowych (9 dodawań, 9 mnożeń i 2 dzielenia). W tablicy podane są przykładowe czasy transformacji obrazu o 2000 wierzchołków za pomocą programu w systemie z mikroprocesorem 8086, przy wykorzystaniu koprocesora arytmetycznego 8087 i specjalnych układów mnożących (np. Am29325 lub Weitek 1032) [3]:

Metoda	Czas mnożenia μs	Czas dodawania μs	Czas dzielenia μs	Czas transformacji 2000 wierzchołków	Liczba obrazów /sekundę
Programowa (8086)	1600	1600	3200	70,4 s	0,015
Z koprocesorem (8087)	19	17	39	0,804s	1,25
Układ mnożący	0,2	0,2	0,8	10,4 ms	96

Blok wyświetlania obrazu zawiera układy, które umożliwiają przesłanie z odpowiednią szybkością informacji pobieranych

z pamięci obrazu do monitora. Układy te muszą zapewnić konwersję sygnałów cyfrowych na postać analogową wymaganą przez układy wejściowe monitora oraz wytworzenie sygnałów synchronizujących proces wyświetlania. Ponadto często stosowanym elementem w bloku wyświetlania jest tablica kolorów. Jest to pomocnicza pamięć, na której wejścia adresowe podawane jest z pamięci obrazu informacja o kolorze piksela, np. 8 bitowa, i na której trzech wyjściach, 4 lub 8 bitowych, pojawiają się kody składowych koloru R.G.B. Pozwala to na korzystanie z dużej gamy kolorów z pomocą krótkich słów niosących informację o kolorze piksela.

Dla przykładu pamięć kolorów o 256 słowach 24 bitowych pozwala korzystać z palety 16 milionów kolorów. Pamięć kolorów musi być pamięcią bardzo szybką. W czasie przeznaczonym na wyświetlenie jednego piksela (w podanym wyżej przykładzie 11 ns) konieczne jest dokonanie zarówno konwersji kolorów jak i konwersji cyfrowo-analogowej. Rys.3 poglądowo przedstawia omówione wyżej funkcje bloku wyświetlania.



Rys.3. Blok wyświetlania z mapą kolorów

3. Monitory kolorowe

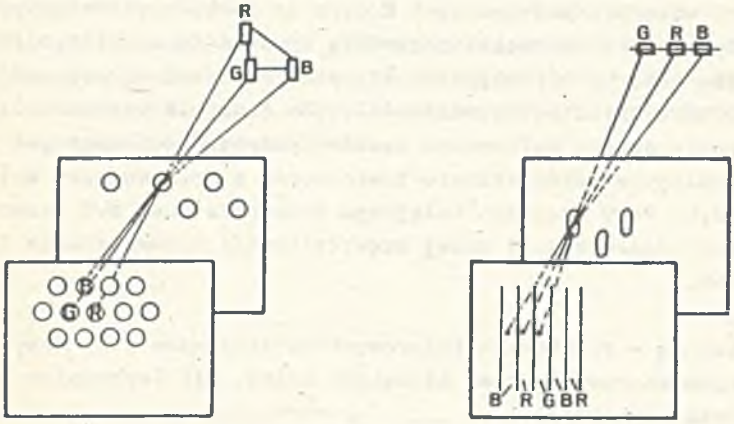
Podstawowym elementem monitora kolorowego jest kineskop kolorowy. Pozostałe układy monitora służą do sterowania kineskopem. Monitory kolorowe wysokiej jakości przyjmują informację przeznaczoną do wyświetlania w postaci napięć podanych na wejścia sterujące działami elektronowymi związanymi z kolorami czerwonym (R), zielonym (G) i niebieskim (B).

Przy odtwarzaniu obrazu kolorowego wykorzystuje się właściwości oka ludzkiego i jego zdolność widzenia barw. Oko ludzkie nie rozróżnia poszczególnych elementów kolorowych jeżeli powierzchnie elementów barwnych oglądane z pewnej odległości są odpowiednio małe. Ekran kineskopu kolorowego powinien świecić w trzech kolorach podstawowych R,G,B. Są to kolory, które poprzez odpowiednie sumowanie pozwalają uzyskać inne kolory. (Dla przykładu kolor biały uzyskuje się przy sumowaniu w proporcjach 0.3R, 0.59G, 0.11B .) Bezwładność oka w widzeniu barw umożliwia odtwarzanie obrazu kolorowego zarówno przy jednoczesnym jak i przy kolejnym rozświetlaniu luminoforów o podstawowych kolorach R,G,B. Przy zasadzie kolejnego świecenia musi być zachowany warunek dostatecznie dużej częstotliwości rozświetlania luminoforów.

Obecnie w monitorach kolorowych są stosowane trzy typy kineskopów maskowych, tzw. kineskopy delta, PIL (precision in line) oraz trynitron.

W kineskopach typu dekta luminofory są nałożone na ekran w postaci okrągłych "pastylek" o średnicy rzędu ułamka milimetra. "Pastylki" te są ułożone wzdłuż linii poziomych w kolejności R,G,B,R,G,B,R ... . W sąsiednich liniach "pastylki" są przesunięte względem siebie tak, żeby trzy sąsiednie "pastylki" wyznaczały wierzchołki trójkąta równobocznego. Liczba takich trójek (triad) zależy od rozmiarów ekranu. W kineskopie znajdują się trzy wyrzutnie elektronów, każda przeznaczona do pobudzania "pastylek" innego koloru, rozmieszczone w wierzchołkach trójkąta równobocznego. W odległości kilkunastu milimetrów

przed ekranem znajduje się cienka blacha z dużą liczbą otworów tzw. maskownica. W celu uzyskania obrazu o dobrej jakości trzy strumienie elektronowe powinny przecinać się w środku każdego otworu maskownicy a następnie "rozchodzić się" padając na odpowiednie "pastylki" luminoforów. Dla danego strumienia, przy dowolnym jego odchyleniu, "pastylki" dwóch innych luminoforów powinny być zasłonięte przez maskownicę. Przedstawioną zasadę poglądowo ilustruje rys.4a.



Rys.4. Zasada pracy kineskopu typu: (a) delta, (b) PIL

Strumienie elektronów z trzech wyrzutni R,G,B są odchylane w kierunkach pionowym i poziomym za pomocą jednego wspólnego zespołu odchylającego.

W kineskopach typu PIL ekran pokryty jest wąskimi paskami luminoforów ułożonymi pionowo. Otwory w maskownicy mają kształt

pionowych szczelin o jednakowej długości. Działa elektronowe są umieszczone obok siebie w płaszczyźnie poziomej. Zasadę działania kineskopu ilustruje rys.4b. W porównaniu z metodą delta, w technice PIL upraszcza się problem zbieżności strumieni elektronowych oraz upraszczają się w związku z tym układy sterowania kineskopem. Zaletą metody delta jest większa rozdzielczość.

Kineskopy typu trynitron są stosowane w zasadzie wyłącznie w monitorach firmy Sony. Podobnie jak w kineskopach typu PIL luminofor jest naniesiony w postaci wąskich pionowych pasków i na maskownicy znajdują się pionowe szczeliny. Zasadnicza różnica polega na odmiennej konstrukcji działa. W systemie tym stosowane jest jedno działo o trzech katodach umieszczonych poziomo. Rozwiązanie to pozwala uzyskać bardzo dobrą rozdzielczość, jest jednak trudne technologicznie.

Rozwój monitorów kolorowych do zastosowań w technice graficznej zmierza w kierunku uzyskania jak największej rozdzielczości. Wśród produkowanych obecnie monitorów i dostępnych handlowo największa rozdzielczość wynosi 1280x1024 (przy odległości między triadami - 0,31 mm). Są to monitory 19 calowe o częstotliwości przeszukiwania poziomego 64 kHz i paśmie wizyjnym rzędu 100 MHz.

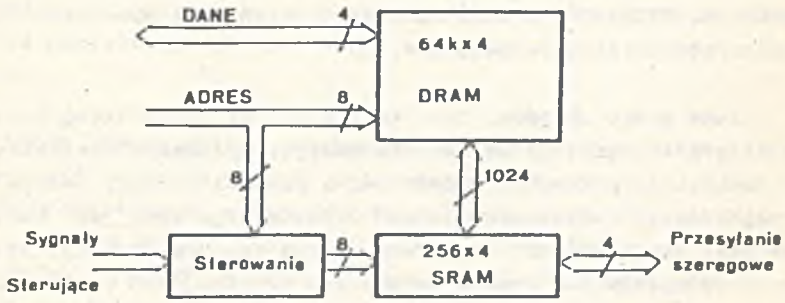
Sterowanie ruchem wiązek elektronów w kineskopie jest następujące. Poczynając od górnego lewego rogu strumień przesuwa się poziomo w prawo wybierając wszystkie punkty należące do pierwszej linii obrazu. Z kolei następuje przesunięcie strumienia do początku drugiej linii i rozpoczyna się wybieranie punktów należących do tej linii itd. W czasie każdego powrotu strumienia do początku linii nie jest wyświetlana żadna informacja (promień jest wygaszany). Z tego punktu widzenia czas powrotu jest czasem martwym. Podobnie czasem martwym jest czas powrotu plamki z punktu końcowego dla ostatniej linii do początku ekranu (przy wyświetlaniu z tzw. wybieraniem międzyliniowym w pierwszej fazie wybierane są linie parzyste a w drugiej linie nieparzyste).

Dla monitora o rozdzielczości 1280x1024 przy odświeżaniu obrazu 60 razy na sekundę przyjmując, że czas powrotu pionowego plamki jest około 600 μs otrzymuje się, że częstotliwość wybierania linii poziomych jest około 64 kHz. Stąd czas na wyświetlanie jednej linii wynosi około 13 μs (po odjęciu czasu powrotu poziomego plamki 4 μs). Przy 1280 punktach wyświetlanych w linii oznacza to, iż czas wyświetlania punktu jest rzędu 10 ns. Stąd wymagane pasmo wizyjne jest rzędu 100 MHz. (Podane parametry posiada np. monitor HM 4619 firmy Hitachi.)

4. Układy VLSI dla zastosowań graficznych

W ostatnich kilku latach opracowano wiele specjalizowanych układów scalonych przeznaczonych do konstrukcji urządzeń grafiki komputerowej. Dzięki tym układom stało się możliwe opracowanie urządzeń o szybkości i jakości tworzenia obrazów porównywalnych z parametrami osiąganymi do niedawna jedynie z pomocą dużych systemów budowanych z wykorzystaniem układów segmentowych i techniki ECL. Wśród nowych układów wyróżnić można kilka grup.

W grupie pamięci, obok różnego typu układów pamięci uniwersalnych DRAM o pojemności 1 Mb, pojawiły się układy pamięci VRAM (video-RAM). Są to pamięci o dwóch rodzajach dostępu: klasycznym - równoległym i nowym - szeregowym. Schemat blokowy wyjaśniający konstrukcję tego typu pamięci podany jest na rys.5 (jest to schemat układu Am 90C644 firmy AMD). Obok bloku pamięci dynamicznej o pojemności 64k x 4 (i organizacji wewnętrznej 256x256x4) w strukturze znajduje się rejestr szeregowy o pojemności 256 słów 4 bitowych. Poza normalnym trybem zapisu i odczytu pamięci (przy czasie dostępu 190 ns) możliwe jest równoczesne przepisanie w czasie jednego cyklu dostępu do pamięci 256 słów 4 bitowych do rejestru szeregowego. Od tej chwili informacja może być wyprowadzana z rejestru szeregowo niezależnie od pracy pamięci w trybie równoległego dostępu (z dużą częstotliwością rzędu 25 MHz). Podobnie, w razie potrzeby można wprowadzić informację szeregowo a następnie w czasie jednego cyklu



Rys.5. Schemat blokowy pamięci z dostępem równoległym i szeregowym

zapisać ją do matrycy pamięci DRAM. Zasadniczą zaletą tego typu pamięci jest wzrost procentowego udziału czasu dostępu do pamięci dla celów wprowadzania nowej informacji. Zaleta ta jest mniej istotna w przypadku dublowania pamięci obrazu. Przy takiej konstrukcji, do jednego bloku pamięci jest wprowadzany nowy obraz podczas gdy w tym czasie z drugiego bloku obraz jest wyprowadzany dla celów wyświetlania. Po zakończeniu pełnego cyklu role bloków pamięci zmieniają się.

Dla ułatwienia konstrukcji bloku wyświetlania opracowano różnego rodzaju zestawy układów przetworników wizyjnych cyfrowo-analogowych. Wiele z nich obok przetworników zawiera inne pomocnicze układy np. rejestry czy tablicę kolorów. Dostępne są układy z pojedynczymi przetwornikami jak też z zestawami trzech przetworników DA 4 lub 8 bitowych i częstotliwościach konwersji do 200 MHz (np. TEK 8120 firmy Tektronix, lub AH 8308 firmy Analogic - potrójny zestaw, 150 MHz).

Dla celów generacji sygnałów sterujących wyświetlaniem informacji są produkowane specjalne układy, zarówno dla zastosowań znakowych jak i graficznych. Jednym z bardziej popularnych układów tego typu jest układ 6845 firmy Motorola stosowany m.inn. w mikrokomputerach IBM PC. Bardziej zaawansowane układy są przystosowane do pracy w systemach typowo graficznych. Można tu wymienić układy MC 68486/487, 82716, TMS34061 oraz Am 8150. Umożliwiają one sterowanie wyświetlaniem przy rozdzielczości do 640x480 pikseli. Niektóre z nich mają wbudowane funkcje zarządzania systemem okien.

Inna grupa układów, tzw. kontrolery ze stałą listą instrukcji graficznych, ma za zadanie odciążyć procesor centralny od funkcji związanych z wypełnianiem pamięci obrazu. Jednym z najbardziej rozpowszechnionych układów tego typu jest kontroler graficzny µPD 7220 firmy NEC. Układ ten obok funkcji sterowania wyświetlaniem oraz odświeżaniem pamięci DRAM realizuje funkcje rysowania linii, łuków, prostokątów, znaków 8x8, zapełnianie pól, powiększanie fragmentów obrazu oraz przesuwanie obiektów po ekranie. Najszybsze wersje tego układu pozwalają rysować z szybkością 500 ns/piksel. Podstawową wadą tego układu jest rozłączność w czasie funkcji tworzenia obrazu i wyprowadzania go na ekran. Próbę rozwiązania tego konfliktu podjęli projektanci układu kontrolera HD 63484-8 firmy Hitachi. W układzie tym w podstawowym cyklu 1000 ns przewidziano czas zarówno na zapis do pamięci obrazu jak i na odczyt dla celów wyświetlania. Układ wykonuje 38 rozkazów rysowania. Może on sterować pamięcią obrazu o pojemności do 2 MB z 1,2,4,8 lub 16 bitami na piksel.

Do współpracy z pamięciami video RAM przystosowany jest układ kontrolera Am 95C60 firmy AMD. Jest to układ o strukturze 16 bitowej realizujący 60 instrukcji graficznych. Maksymalna rozdzielczość sterowanego układu wynosi 4096x4096. Układ ma wbudowany algorytm wygładzania linii z wykorzystaniem dwóch lub czterech poziomów cieniowania.

Najbardziej złożonymi układami projektowanymi z myślą o zastosowaniach graficznych są układy procesorów graficznych. Przedstawicielem tej grupy układów jest procesor TMS 34010 firmy Texas Instruments. Jest to układ 32 bitowego mikroprocesora ze zredukowaną listą instrukcji i o szybkości 6 MIPS. Układ wykonany jest w technologii CMOS (1,8 μm) i zawiera około 180000 tranzystorów. Układ pozwala na wykonywanie operacji przesyłania bloków pikseli (PixBlt). W układzie TMS 34010 wbudowana jest pamięć typu cache o pojemności 256 bajtów. Umożliwia to pamiętanie 128 instrukcji i przyspieszanie wykonywania pętli.

Obok samych układów scalonych możliwe jest korzystanie z gotowych pakietów grafiki kolorowej przystosowanych do współpracy z określonymi systemami. Przykładowo firma Imgraph produkuje pakiet AGC-1024-8 przystosowany do współpracy z systemami IBM PC XT/AT/RT. W pakiecie tym wykorzystano następujące układy graficzne: kontroler HD-63484, pamięć video RAM oraz układ przetworników wizyjnych DA AMD 8151. Pakiet umożliwia wyświetlanie z częstotliwością 60 Hz obrazów o rozdzielczości 1024x1024. Każdy piksel jest reprezentowany z pomocą 8 bitów co daje możliwość wyświetlania 256 kolorów z palety 16,8 miliona kolorów. Pakiet realizuje podstawowe instrukcje graficzne oraz operacje na blokach bitów (Bit Blt), sprzętowe obcinanie, podział ekranu. Maksymalna szybkość wprowadzania informacji do pamięci obrazu wynosi 5 mil. pikseli/s a częstotliwość wyprowadzania na ekran 87 MHz.

5. Stacje graficzne

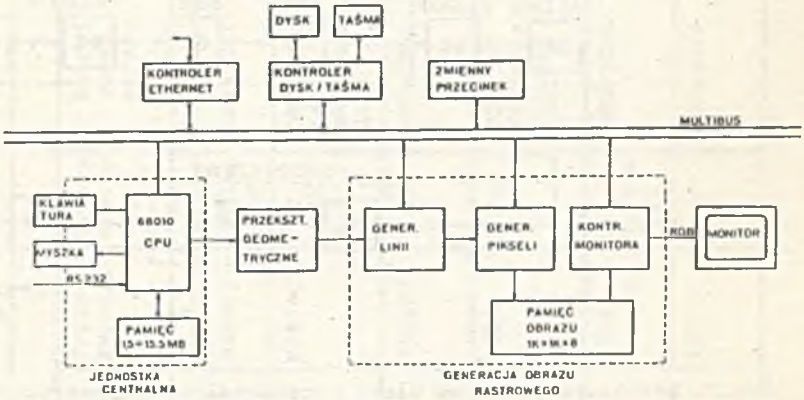
Do niedawna jedyną możliwość korzystania z techniki graficznej wysokiej jakości dawały duże systemy komputerowe połączone z drogimi terminalami graficznymi. Komputer obciążony był zarówno rozwiązywaniem określonego zadania jak i przetwarzaniem informacji dla utworzenia obrazu wyświetlanego na terminalu. W miarę postępu technologii pojawiła się tendencja do rozszerzania funkcji spełnianych przez terminal o zadania

związane z przygotowaniem wyświetlanego obrazu. Tendencja ta doprowadziła do powstania stacji graficznych. Są to urządzenia, które współpracując z dużym komputerem przejmują wszystkie funkcje graficzne. Równocześnie stacje takie często są wyposażone w mechanizmy konieczne dla pracy interakcyjnej i są zdolne do samodzielnej pracy. Niżej podane jest charakterystyka stacji graficznej IRIS 2400 firmy Silicon Graphics.

Jednostka centralna stacji jest zbudowana z wykorzystaniem mikroprocesora 68010. Stacja wyposażona jest w pamięć RAM o pojemności 1,5 ÷ 15,5 MB. Stacja wykorzystuje system operacyjny UNIX. Pojemność pamięci dyskowej typu Winchester wynosi od 72 MB do 440 MB. Rozdzielczość 19 calowego monitora kolorowego wynosi 1024x768 pikseli (przy wyświetlaniu z częstotliwością 60 Hz). Stacja może pracować niezależnie bądź też może współpracować z siecią Ethernet. Dostępna jest szyna Multibus. Stacja wyposażona jest w system okien.

Na rys.6 przedstawiony jest schemat blokowy stacji. W systemie wykorzystywana jest idea przetwarzania potokowego zarówno na poziomie trzech podstawowych bloków: jednostki centralnej, bloku przekształceń geometrycznych i bloku generacji obrazu rastrowego jak też i wewnątrz tych bloków.

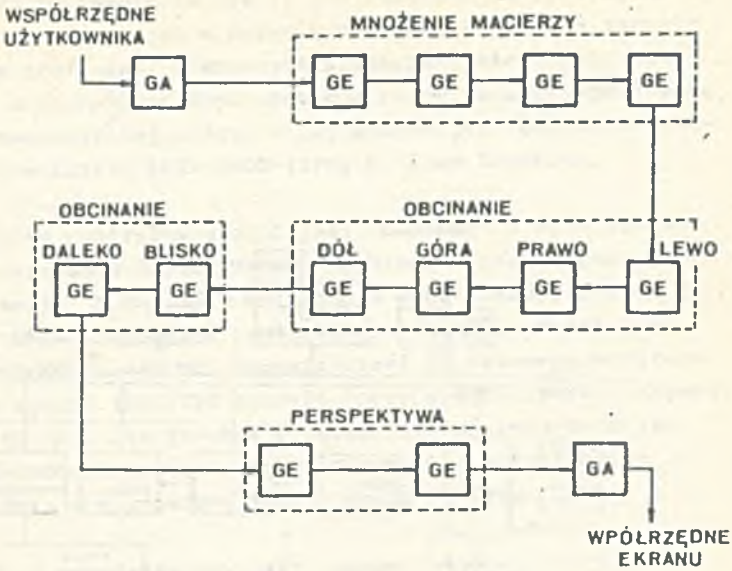
Najciekawszym elementem stacji jest blok przekształceń geometrycznych zbudowany w oparciu o dwa typy układów VLSI: układ przekształceń geometrycznych (geometry engine - GE) oraz układ przyspieszający (geometry accelerator - GA). Sposób połączenia układu w strukturze potokowej wyjaśnia rys.7. W skład układu GE wchodzą m.inn. cztery 32 bitowe, zmiennoprzecinkowe jednostki arytmetyczne oraz pamięć sterująca. W strukturze potokowej wykorzystuje się 10 lub 12 takich układów. Każdy z nich spełnia jedną z trzech funkcji określanych programowo. Pierwsze cztery układy wykonują operacje na macierzach współrzędnych jednorodnych (4x4) dla uzyskania obrotu, przesunięcia i skalowania. Następne cztery lub sześć układów GP mają za zadanie realizację funkcji obcinania dla określenia części obrazu należącej do ostrosłupa widzenia (z uwzględnieniem lub



Rys.6. Stacja graficzna IRIS 2400

nie głębokości obrazu). Ostatnie trzy układy GP wykonują operacje dzielenia dla uzyskania perspektywy i przejścia na dwuwymiarowe współrzędne ekranu. Układy przyspieszające GA mają do spełnienia dwa zadania: buforowanie (na zasadzie FIFO, konieczne ze względu na różnice szybkości pracy poszczególnych bloków) i konwersję na lub z postaci zmiennoprzecinkowej wykorzystywanej w układach GE. Efektem zastosowanych rozwiązań jest szybkość przekształceń geometrycznych rzędu 65000 transformacji na sekundę.

Współrzędne punktów uzyskiwane na wyjściu bloku przekształceń geometrycznych są wykorzystywane w bloku generacji



Rys.7. Struktura potokowa bloku przekształceń geometrycznych

obrazu rastrowego do utworzenia obrazu w pamięci obrazu. W pierwszej kolejności 16 bitowy procesor segmentowy generuje linie i wielokąty. Z kolei blok generacji zawartości pamięci obrazu realizuje algorytmy związane z wypełnianiem wielokątów. Wypełnianie wielokątów wypukłych odbywa się z szybkością 44 milionów pikseli/s. Realizowane są tu również algorytmy cieniowania Gouraud oraz określania intensywności linii i punktów w funkcji odległości od obserwatora. Operacje te mogą być realizowane z szybkością do 3 milionów pikseli/s. Układ kontrolera monitora zapewnia generację sygnałów R,G,B.

W stacji IRIS 2400 i innych stacjach firmy Silicon Graphics, konstruktorzy zdecydowali się na wykorzystanie dwóch specjalizowanych układów VLSI, własnej konstrukcji. Inne firmy na ogół dążą do korzystania z układów dostępnych handlowo. W tablicy 1 podano podstawowe parametry kilku wybranych stacji graficznych.

Tablica 1. Parametry wybranych stacji graficznych

Producent	Możliwości obliczeniowe						Grafika		
	Procesor	Pamięć cache (bytów)	Pamięć RAM (bytów)	Pamięć wirtualna (bytów)	Procesor zm.prz.	System operacyjny	Rozdzielczość ekranu	Szybkość rysowania (pixeli/s)	Kolor, liczba bitów/pixel
1	2	3	4	5	6	7	8	9	10
Apollo Computers	Segmentowy	4k inst. 16k dane	4M	256M	własny	Aegis Unix	1024x1024	1,5M	8 lub 24
DEC VAX station 520	VAX-780 chip	-	9M	4G	własny	VMS UNIX 4.2	1280x1024 (terminal Tektronix 6125)	500K	2 lub 8
Gould Inc.	32-bitowy własny	32k instr. 32k dane	16M	16M	własny	Unix 4.2	1280x1024	1M	8,16 lub 24
Hewlett Packard HP 9000 seria 300	68010 lub 68020	16k	7,5M	4G	68881	HP-UX	1024x768 (zewnętrzny monitor)	2,5M	
Jupiter Systems	68010	-	4M	4M	IEEE format	Unix System III	1280x1024	30M	4 lub 8
Mosaic Technology	32032	16k	8M	16M	32082	Unix 4.2	1280x1024	1,6M	4,8,16 lub 32

1	2	3	4	5	6	7	8	9	10
Ridge Computers	32 bit RISC procesor	-	8M	4G	Tak (w OPU)	Unix System V	1024x800	37M	czarno-biały
Silicon Graphics	68010	-	15M	16M	W kon-trole-rze gra-ficznym	Unix System V	1024x1024	3M	8 lub 32
Sun Microsystems	68010 lub 68020	-	8M	16M	68881	Unix 4.2	1152x900	1,5M	8
Tektronix	32016 lub 32032	-	10M	16M	32081	Unix 4.2	1024x768	2,7M	4 lub 8

Większość stacji pozwala wyświetlać obrazy z rozdzielczością 1k x 1k i wykonywać od 0,4 do 1,5 mln operacji/s. Stacje te są przystosowane do pracy w sieciach lokalnych. Odpowiednią moc obliczeniową uzyskuje się jednym z trzech sposobów: stosowanie wielomikroprocesorowych systemów 16 bitowych, wykorzystywanie mikroprocesorów 32 bitowych oraz konstruowanie procesorów segmentowych. Standardem jest stosowanie systemu Unix. Większość systemów wykorzystuje własne szyny wewnętrzne. Jednak prawie wszystkie stacje mają interfejs z szyną Multibus. Również prawie wszystkie stacje są wyposażone w system okien.

6. Metoda śledzenia promieni

Dążąc do uzyskania obrazów jak najbardziej zbliżonych do rzeczywistych obiektów opracowano wiele różnych metod rozwiązywania poszczególnych problemów, takich jak eliminacja linii i powierzchni niewidocznych, symulacja efektów oświetlenia, odbić, cieni, zmiany jasności w funkcji odległości itp. Dla realizacji niektórych z tych metod opracowano specjalne rozwiązania sprzętowe. Jednak w wielu przypadkach urządzenia dobrze nadające się do realizacji jednego rodzaju algorytmu gorzej radzą sobie z innymi algorytmami.

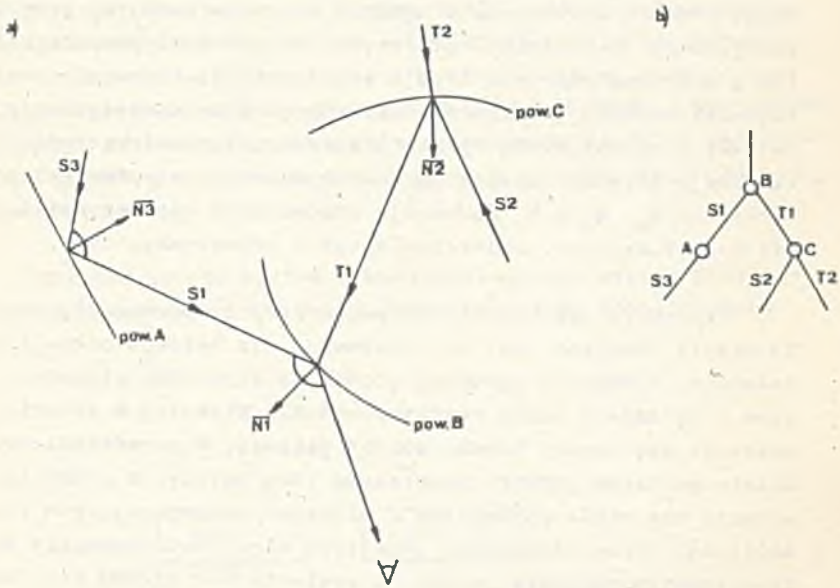
Metodą w pewnym sensie uniwersalną jest metoda śledzenia promieni. Jest to jednak metoda wymagająca bardzo dużej ilości obliczeń i jak dotychczas jej realizacja jest możliwa za pomocą dużych komputerów. Metodę tę cechuje pewna regularność obliczeń i możliwość zrównoleglania obliczeń co rokuje nadzieje na opracowanie specjalizowanych układów VLSI, bądź wykorzystanie innych urządzeń do przetwarzania równoległego.

Idea metody jest następująca. Analizuje się bieg każdego promienia, który przechodząc przez piksel obrazu dociera do oka obserwatora. Dla każdego takiego promienia wyznacza się przecięcie ze wszystkimi obiektami oglądanej sceny. Punkt przecięcia leżący najbliżej oka należy do obiektu widzialnego.

Po określeniu dla danego promienia powierzchni widocznej generowany jest drugi promień, który podlega odbiciu od tej powierzchni zgodnie z prawami optyki. W przypadku gdy odbity promień przecina inne obiekty sceny, to pierwszy taki obiekt jest widziany jako obraz odbity. Bieg promienia jest śledzony dalej tak długo, aż przestanie przecinać obiekty należące do sceny. Teraz możliwe jest ustalenie intensywności i koloru dla rozpatrywanego piksela poprzez wsteczne śledzenie promienia i rekursyjne stosowanie w każdym punkcie odbicia przyjętego modelu oświetlenia. Uwzględnia się przy tym funkcję tłumienia zależną od odległości pomiędzy kolejnymi punktami odbicia. Po zakończeniu procesu cofania uzyskuje się wartość intensywności lub kolor dla rozpatrywanego piksela z uwzględnieniem wpływu oświetlenia pochodzącego od wszystkich istotnych obiektów należących do analizowanej sceny.

W przypadku gdy w scenie występują obiekty przezroczyste, w każdym punkcie przecięcia śledzonego promienia z powierzchnią generuje się dwa promienie: jeden odbity i jeden załamany. Kierunek promienia załamanego określa się zgodnie z prawem Snelliusa. Również i dla tego promienia uwzględnia się efekt tłumienia w funkcji odległości i współczynnika przezroczystości. Zasadę generowania promieni ilustruje rys.8a. Natomiast na rys. 8b pokazany jest fragment drzewa, które jest tworzone dla każdego piksela. Każdy węzeł drzewa odpowiada punktowi przecięcia promienia z powierzchnią. Promienie S_i są promieniami odbijanymi zwierciadlanie, natomiast promienie T_i reprezentują promienie załamane. W ogólnym przypadku gałęzie kończą się gdy ich udział w końcowym bilansie intensywności piksela staje się na tyle mały, że można go pominąć (albo też wyczerpuje się dostępna pamięć).

W przypadku gdy uwzględnia się problem cieni rzucanych przez obiekty, w każdym punkcie przecięcia promienia z powierzchnią generuje się kolejne promienie skierowane do poszczególnych źródeł światła. O ile promień taki przetnie jakiś obiekt zanim dotrze do źródła światła, to punkt w którym ten promień został wytworzony znajduje się w obszarze cienia



Rys.8. Promienie generowane dla jednego piksela i odpowiednie drzewo promieni. Powierzchnia A jest nieprzezroczysta

związanego z danym źródłem światła. Możliwe jest również uwzględnianie przypadku obiektów przezroczystych znajdujących się między analizowanym punktem a źródłem światła. Uwzględnienie wpływu cieni na końcową intensywność piksela jest możliwe poprzez przyjęcie odpowiedniego modelu oświetlenia wykorzystywanego sekwencyjnie przy analizie każdego węzła drzewa.

W pracy [15] przyjęty jest następujący model oświetlenia

$$I = I_a + k_d \sum_{j=1}^{j=l_s} (\overline{N}.\overline{L}_j) + k_s S + k_t T$$

Intensywność I w danym punkcie powierzchni jest sumą czterech składników. Czynnik I_a określa wpływ światła otoczenia. Drugi czynnik pozwala uwzględnić wpływ rozproszonego światła odbijanego, którego intensywność zgodnie z prawem Lamberta jest proporcjonalna do cosinusa kąta między normalną do powierzchni ($\overline{N}$) a wektorem kierunku źródła światła ($\overline{L}_j$). Pozostałe dwa czynniki określają wpływ światła odbijanego zwierciadlanie oraz światła przepuszczanego przez przezroczystą powierzchnię (S i T oznaczają intensywności związane z odpowiednimi promieniami). Wielkości k_d, k_s i k_t oznaczają odpowiednio współczynniki odbicia: dyfuzyjnego, zwierciadlanego i przezroczystości.

Procedura wyznaczania intensywności za pomocą algorytmów śledzenia promieni musi być stosowana dla każdego piksela niezależnie. Wymaga to ogromnej ilości obliczeń dla złożonych scen i obrazów o dużej rozdzielczości. Niemniej w efekcie uzyskuje się obrazy bardzo dobrej jakości. W przedstawionym opisie pokazano jedynie podstawową ideę metody. W praktyce stosuje się wiele uproszczeń i ułatwień zmniejszających ilość obliczeń. Próby stosowania tego typu algorytmów wymagały obliczeń trwających kilka godzin na systemie VAX 11/780 przy rozdzielczości obrazu 640x480. Autor pracy [15] podaje, że 75% czasu obliczeń zajmowały procedury wyznaczania przecięć a 12% procedury związane z analizą oświetlenia.

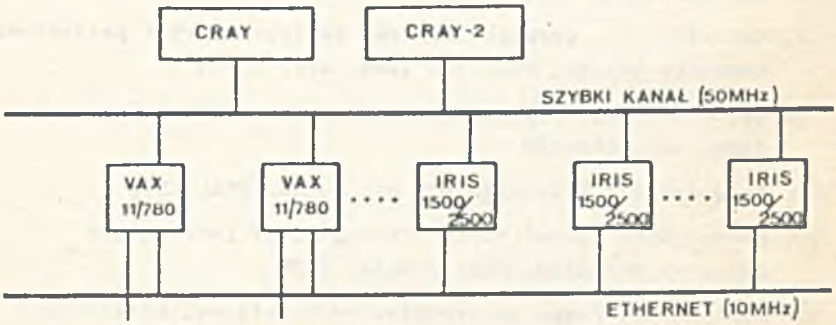
7. Zakończenie

W pracy przedstawiono wybrane problemy związane z konstrukcją urządzeń do cyfrowej generacji obrazów i ich wyświetlania. Przegląd ten pokazuje tendencję rozwoju tego typu urządzeń. Specjalizowane układy o coraz większym stopniu scalenia umożliwiają budowę urządzeń graficznych o coraz lepszych parametrach i coraz powszechniej dostępnych. Jednak, jak pokazuje choćby

przykład metody śledzenia promieni, nie wszystkie problemy grafiki komputerowej dają się rozwiązywać za pomocą dostępnych obecnie specjalizowanych urządzeń graficznych. Dla niektórych problemów nadal jedyną możliwość rozwiązania stanowią złożone systemy superkomputerowe.

W ośrodku firmy Digital Production [4] zainstalowana jest konfiguracja wykorzystywana do generacji obrazów animowanych i specjalnych efektów o bardzo wysokiej jakości. Dla zobrazowania złożoności problemu warto przytoczyć kilka liczb. Konfiguracja w skład której wchodzą: Cray X-MP/2230 i VAX 11/782 potrzebuje od 3s do 10g pracy dla wytworzenia 1s filmu animowanego. Daje to w rezultacie możliwość wytworzenia około 144 minut filmu rocznie.

Przykład innego systemu, wykorzystującego stacje graficzne, jest pokazany na rys.9 [7]. Jest to system zainstalowany



Rys.9. System ze stacjami graficznymi

w jednym z ośrodków NASA do celów symulacji efektów aerodynamicznych. W systemie tym przewidziano wykorzystanie 25 opisanych wyżej stacji firmy Silicon Graphics.

BIBLIOGRAFIA

1. Berk T., Realistic image generation techniques and issus. Fourth symposium on microcomputer and microprocessors applications, Budapeszt, 1985, str. 1-13
2. Clark J., The geometry engine: a VLSI geometry system for graphics, Computer Graphics, July 1982, str. 127-133
3. Demetrescu D., Moving pictures, Byte, November 1985, str. 207-217
4. Demos G., i inn., Digital scene simulation: the synergy of computer technology and human creativity, Proc. of the IEEE, January 1984, str. 22-31
5. Foley J.D., van Dam A., Fundamentals of interactive computer graphics. Addison - Wesley, 1982
6. Greenberg D., i inn., The computer image: applications of computer graphics, Addison - Wesley, 1982
7. Gustafson P., Graphics systems deliver upfront performance, Computer Design, July 15, 1985, str. 61-65
8. Leibson S.H., Graphics-controller ICs, EDN, February 6, 1986, str. 104-118
9. Masewicz T., Telewizja dla praktyków, WKiŁ 1982
10. Newman W.M. Sproul R.F., Principles of interactive computer graphics, McGraw-Hill, 1979
11. Panasuk C., Focus on graphics workstations, Electronics Design, August 22, 1985 str. 157-164
12. Price S.M., CMOS 256-k bit video RAM with wide two-way bus, picks up speed, drops power, Electronics Design, September 19, 1985, str. 171-177

13. Whitton M.C., Memory design for raster graphics displays, IEEE CG&A, March 1984, str. 48-64

14. Wiliams T., Graphics processing migrates from host to workstation, Computer Design, July 15, 1985 str. 49-57

15. Witted T., An improved illumination model for shaded display, Comm. ACM, June 1980, str. 343-349

16. Wright M., Color monitors, EDN May 31, 1984, str. 107-122.

Druk.Wyd. "ALFA" Nakł. 300 Zam. 3109/86