2-3 1986

informatyka

Prof. Niklaus Wirth o Moduli-2

Język Icon

Zaawansowane konstrukcje w C

Zmienny przecinek w Forthie

Komputery osobiste w zastosowaniach profesjonalnych

O normie IEEE języka asemblerowego

Wydawnictwo Naukowe PWN

Cena 180 zł

informatyka

2-3 1986

Nr 2-3
Miesięcznik Rok XXI
Luty-Marzec 1986

Organ Komitetu Informatyki MNSZWiT oraz Komitetu Naukowo-Technicznego NOT ds. Informatyki

Redakcja: 00-041 Warszawa, ul. Jasna 14/16, pok. 243 i 244, tel. 27-71-40 lub 26-82-61 w. 184

Zakł. Graf. „Tamka". Zam. 0187-1300/86. Obj. 4,0 ark. druk. Nakład 7200 egz. P-68.

ISSN 0542-9951, INDEKS 36124

Cena egzemplarza 120 zł
Prenumerata roczna 1200 zł

NACZELNA ORGANIZACJA TECHNICZNA
WYDAWNICTWO
SIGMA
CZASOPISM I KSIĄŻEK TECHNICZNYCH
00-950 Warszawa
skrytka pocztowa 1004
ul. Biała 4

W NUMERZE: Strona

W NASTĘPNYCH NUMERACH:

- Brian A. Wichmann i J.G.J. Meijerink o konwersji oprogramowania na język Ada
- Roman Faber i Michał Ostrowski o uniwersalnym procesorze do redagowania tekstów dla MERY 400
- Andrzej Janczura o rozwiązywaniu pasmowych równań liniowych na mikrokomputerach
- Zbigniew Szkaradnik o rekurencji w języku Forth
- Paweł Grzegorzewicz o mechanizmie półautomatycznego generowania poleceń dla systemu operacyjnego RSX-11M
- Piotr Zaskórski i Ewa Kasprzyk o technologii wytwarzania oprogramowania ze wspomaganiem komputerowym
- Krzysztof Perycz o minikomputerowym systemie operacyjnym S1
- Andrzej Macioł i Adam Stawowy o środkach przetwarzania tekstów
- Piotr Cofta o normie IEEE

NIKLAUS WIRTH
Institut fur Informatik
Eidgenossische Technische Hochschule
Zurych

Zasady wieloprogramowości i ich implementacja w Moduli-2

Wieloprogramowością nazywamy określenie w programie kilku (być może wielu) procesów sekwencyjnych, które są wykonywane współbieżnie. Celem wieloprogramowości jest osiągnięcie i zagwarantowanie zharmonizowanej współpracy między procesami. Wymaga to podania zasad i określenia operacji elementarnych do synchronizacji czynności współbieżnych i komunikacji między nimi. W literaturze proponowano wiele koncepcji rozwiązania tego zagadnienia, a kilka z nich zrealizowano w różnych językach programowania. Ostatnio — w pracach [1, 8, 9] — opublikowano kilka dokładnych porównań i ocen tych metod.

Wspólną cechą tych propozycji jest uznanie indywidualnych procesów za ciągi działań wykonywanych z dowolną szybkością. Dlatego nie występuje w nich pojęcie czasu, z wyjątkiem sytuacji, w których operacje synchronizacji umożliwiają opóźnienie określonego procesu, aż do spełnienia pewnego warunku przez inne procesy. Tego rodzaju synchronizacja występuje dość rzadko. W takich przypadkach mówi się o procesach słabo powiązanych, w przeciwieństwie do tablic procesów wykonywanych „w miarowych taktach".

Procesy muszą być synchronizowane, gdy planuje się ich współpracę. W procesach obliczeniowych współpraca jest równoznaczna z komunikacją. Komunikacja oznacza wymianę informacji, tj. danych. W tym artykule rozróżnia się dwa rodzaje komunikacji, tj. **komunikację przez współdzielenie zmiennych** (ang. sharing variables) i **przekazywanie komunikatów** (ang. passing messages). Odzwierciedlają one niejawne założenie co do mechanizmu wykonywania procesów, który przy pominięciu innych szczegółów jest różny dla procesorów dzielących wspólną pamięć (tj. połączonych przez wspólną magistralę o dużej przepustowości) i procesorów rzeczywiście rozłożonych, komunikujących się za pomocą przewodów. Choć można zrealizować każdy z tych rodzajów komunikacji przy użyciu drugiego, to nie jest celowe zbytnie rozwijanie abstrakcji, gdyż jej koszt spowodowany brakiem efektywności może okazać się za duży. Zasada przekazywania komuniktów jest stosowana dopiero od niedawna, ponieważ rozwój technologii spowodował, że tworzenie systemów o większej liczbie procesorów rozłożonych stało się ekonomicznie uzasadnione. Z tego względu nie należy się dziwić, że dotychczasowe języki programowania mają konstrukcje ukierunkowane głównie na komunikację przez zmienne współdzielone (np. Concurrent Pascal, Modula, Ada). Przesunięcie zainteresowań na komunikację opartą na przekazywaniu komunikatów jest widoczne w języku Occam [5].

Choć implementacja obu zasad na pojedynczym procesorze jest bardzo podobna, należy pamiętać, że istotą przekazywania komunikatów jest transmisja informacji przez wartość i brak zmiennych współdzielonych. System, w którym tzw. komunikaty są wskaźnikami, tj. wskazują segmenty współdzielonego bufora, nie może być zaliczony do systemów przekazywania komunikatów. Sam fakt transmisji informacji przez wartość czyni tę zasadę pojęciowo prostą i dlatego atrakcyjną.

Szczególnym przypadkiem synchronizacji, który wynika z zasady komunikacji przez zmienne współdzielone, jest wykluczanie wzajemne. Umożliwia ono przyznanie procesowi wyłącznego dostępu do pewnych zmiennych, tj. wykluczenie dostępu innych procesów, aż do zrezygnowania procesu z tego przywileju. Wykluczanie wzajemne jest koniecznością praktyczną. Choć można je zaprogramować używając elementarnych operacji synchronizacyjnych, celowe jest przeznaczenie specjalnej konstrukcji językowej do wyrażenia wykluczania wzajemnego. Taka konstrukcja nazywa się monitorem [3]. Systemy przekazywania komunikatów nie wymagają wykluczania wzajemnego.

Innym podstawowym i ważnym pojęciem jest rzeczywista współbieżność procesów, która oznacza, że każdy proces jest wykonywany przez indywidualny procesor. W praktyce, dzięki założeniu, że nie uwzględnia się szybkości obliczeń, procesor może być użyty do wykonywania kilku procesów częściami. Dlatego należy odróżniać logiczne procesy od fizycznych procesorów, pamiętając że to rozróżnienie jest kwestią zastosowanej techniki implementacji. Jeżeli procesy są reprezentowane przez poszczególnych użytkowników komputera, to mówi się o systemie z podziałem czasu (ang. time-sharing system), jeżeli zaś procesy wyrażają różne czynności współbieżne tego samego użytkownika, to mówi się raczej o wieloprocesowości. Oba pojęcia wyrażają jednak tę samą myśl wykorzystania jednego procesora przez kilka procesów. Zamiast mówić o rzeczywistej współbieżności, mówi się wtedy o quasiwspółbieżności. Ważną sprawą jest takie rozumienie procesów, aby nie było istotne czy do ich wykonania używa się wielu procesorów, czy jednego, pracującego z podziałem czasu. Dzięki temu programista uzyskuje swobodę w optymalnym wykorzystaniu zasobów, a program jest niezależny od aktualnej konfiguracji systemu. Ten rodzaj abstrahowania od rzeczywistej konfiguracji procesorów jest istotą wieloprogramowości.

Prof. NIKLAUS WIRTH uzyskał doktorat na Uniwersytecie Kalifornijskim w Berkeley w roku 1963. Do roku 1967 pracował jako Assistant Professor na Uniwersytecie w Stanford (stan Kalifornia). Od roku 1968 jest profesorem w Instytucie Informatyki Politechniki Federalnej w Zurychu (ETH), którego dyrektorem był w latach 1982—1984. Najbardziej znane są jego osiągnięcia w dziedzinie języków programowania, a szczególnie opracowanie języków: Euler, Algol-W, Pascal, Modula i Modula-2.
Implementacji tego ostatniego języka dokonał na opracowanym przez siebie komputerze osobistym Lilith. Niklaus Wirth otrzymał w roku 1984 nagrodę im. Alana Turinga, przyznawaną przez amerykańskie stowarzyszenie Association for Computing Machinery. W chwili oddawania numeru do druku N. Wirth przebywał w Xerox Palo Alto Research Center w Kalifornii.
Artykuł jest tłumaczeniem — dokonanym za zgodą Autora — raportu opublikowanego w czerwcu 1984 roku, pt. „Schemes for Multiprogramming and Their Implementation in Modula-2" (ETH Institut für Informatik, No. 59).

Wynika stąd, że funkcje zarządzania zasobami mogą być wywoływane tylko wtedy, gdy proces zażąda wykonania operacji synchronizacji lub komunikacji, które — jak stwierdzono — są ze sobą ściśle związane. Wskutek tego, przydział zasobów, tzn. czynność przełączania procesora, może być ukryta za instrukcjami synchronizacji i komunikacji. Te pierwsze implementuje się w modułach niskiego poziomu, a same instrukcje stanowią wywołania procedur tych modułów. Kluczową sprawą jest rozdzielenie pojęciowe tych poziomów, co stanowi nieodzowny warunek zdolności adaptacyjnej procedur zarządzania zasobami do różnych konfiguracji procesorów.

Poniżej zaproponowano kilka zasad wieloprogramowości. Każdej z nich odpowiada zbiór operacji elementarnych do synchronizacji i komunikacji, stanowiący zbiór procedur. Są one wyrażone przy użyciu modułów definicyjnych Moduli-2 [11]. Krótkie przykłady zastosowań przedstawiono w postaci klientów tych modułów definicyjnych. Omówiono także odpowiednie moduły implementacyjne, reprezentujące zarządzanie zasobami na niskim poziomie. Implementacje oparto na zestawie jednoprocesorowym, przy czym cechy szczególne użytego komputera występują w nich dość sporadycznie.

WSPÓŁPROGRAMY

Podstawowym narzędziem wszystkich systemów, które umożliwia przełączenie procesora z wykonywania jednego procesu na drugi, jest współprogram. Proces implementuje się jako współprogram. Dla programisty jest oczywiste, że procesy są wykonywane na przemian, a w programie podaje się jawnie miejsca, w których musi nastąpić przełączenie. Elementarną instrukcją przełączania współprogramu **p** na współprogram **q** jest instrukcja **Transfer (p, q).** Współprogram odbiorczy **q**, kontynuowany po wykonaniu instrukcji **Transfer**, zastaje obliczenia dokładnie w tym samym stanie, w jakim pozostawił je współprogram nadawczy **p** przed wykonaniem tej instrukcji. Tak więc, instrukcja **Transfer** nie jest niczym więcej jak jawnym operatorem szeregowania procesów.

Współprogram składa się z segmentu programu określającego czynności i na ogół ze zbioru zmiennych lokalnych. W Moduli-2 współprogram jest wyrażony za pomocą procedury i tzw. przestrzeni roboczej, służącej do przechowywania danych lokalnych podczas wywoływania procedury. Współprogram powołuje się za pomocą procedury elementarnej **InitCoroutine.** Jej parametrami są: procedura **P,** która tworzy program, adres przestrzeni roboczej **wsp** i jej rozmiar. Wywołanie procedury **InitCoroutine** nie uaktywnia współprogramu, tzn. nie powoduje przełączenia procesora. Jej działanie polega na zainicjowaniu deskryptora umieszczonego w przestrzeni roboczej, tak aby późniejsza instrukcja **Transfer** spowodowała rozpoczęcie wykonywania nowego współprogramu od pierwszej instrukcji procedury **P.** Adres przestrzeni roboczej jest parametrem instrukcji **Transfer.** Użycie typu **ADDRESS** świadczy o tym, że współprogram jest obiektem niskiego poziomu. Dwie procedury elementarne, **Transfer** i **IntiCoroutine,** zdefiniowano w poniższym module definicyjnym:

```
DEFINITION MODULE Coroutines;
  FROM SYSTEM IMPORT ADDRESS;
  PROCEDURE Transfer(VAR from, to: ADDRESS);
  PROCEDURE InitCoroutine(P: PROC; wsp: ADDRESS; size: CARDINAL);
END Coroutines.
```

Wydruk 1

Implementacja tego modułu jest specyficzna dla mikrokomputera Lilith [12]. Dla innych komputerów zazwyczaj pisze się ją w kodzie asemblerowym, ponieważ ciała rzeczywistych procedur składają się zaledwie z kilku instrukcji. Typ **Coroutine** definiuje strukturę deskryptora współprogramu umieszczonego w nagłówku przestrzeni roboczej, i reprezentuje stan współprogramu w chwili jego zawieszenia. Zmienne **G, L, PC, M, S i H** oznaczają rejestry procesora (wydruk 2).

Na ogół nie zaleca się używania współprogramów w tak bezpośredni sposób. Nie pozostawia on systemowi możliwości na zarządzanie własnymi zasobami i obciąża program szczegółami przydziału procesora, co ma ujemny wpływ na wykonanie właściwego zadania. Jednak w przypadkach, gdy strategia zarządzania jest prosta i oczywista, a ponadto podstawowe znaczenie ma efektywność, użycie udogodnienia tak niskiego poziomu jest uzasadnione. Takim przypadkiem jest obsługa współpracy koprocesorów lub innych urządzeń komunikujących się za pomocą przerwań.

```
IMPLEMENTATION MODULE Coroutines;
  (*implementation for Lilith system*)
  FROM SYSTEM IMPORT ADDRESS, ADR;

  TYPE CorPtr = POINTER TO Coroutine;

    Coroutine =
      RECORD
        G:     ADDRESS;
        L:     ADDRESS;
        PC:    ADDRESS;
        M:     BITSET;
        S:     ADDRESS;
        H:     ADDRESS;
        err:   CARDINAL;
        trapMask: BITSET;
        start: PROC;      (*start of workspace*)
        scnt:  CARDINAL
      END ;

  PROCEDURE GlobalBase(): ADDRESS;
    CODE 25B; 0   (*LGA 0*)
  END GlobalBase;

  PROCEDURE CALL;
    CODE 357B   (*call procedure variable*)
  END CALL;

  PROCEDURE TRA(VAR from, to: ADDRESS);
    CODE 256B; 0   (*transfer*)
  END TRA;

  PROCEDURE Transfer(VAR from, to: ADDRESS);
  BEGIN TRA(from, to)
  END Transfer;

  PROCEDURE InitCoroutine(P: PROC; wspa: ADDRESS; size: CARDINAL);
    VAR cor: CorPtr;

    PROCEDURE SetPC;
      PROCEDURE pc(): CARDINAL;
        CODE 40B; 2  (*LLW 2*)
      END pc;
    BEGIN cor^.PC := pc() + 1
    END SetPC;

  BEGIN cor := wspa;
    WITH cor^ DO
      G := GlobalBase(); L := 0;
      M := {};           S := ADR(scnt)+1;

        H := wspa + size;  err := 0;
        trapMask := {};
        start := P;        scnt := 0
      END ;
      SetPC;
      RETURN;
      CALL; HALT
    END InitCoroutine;
  END Coroutines.
```

Wydruk 2

Uznajemy takie urządzenie (np. drukarkę) za procesor. Ponieważ zazwyczaj z procesem drukowania związane są pewne czynności, jak obsługa buforów lub sprawdzanie stanu, niezależnie od możliwości urządzenia, to cały proces dzieli się na dwie części. Pierwsza część dotyczy nieprogramowalnych czynności urządzenia, takich jak rzeczywiste drukowanie, a druga — programowalnych, wymagających użycia procesora. Z tego względu konieczne jest przełączanie procesora. Gdy zakończone zostanie wykonywanie części programowalnej, związanej z uaktywnieniem urządzenia, wtedy procesor jest przełączany jawną instrukcją **Transfer** na inną czynność wymagającą kontynuacji. Gdy urządzenie zakończy wykonywanie swojej części, wykazuje ten fakt sygnałem przerwania. Następuje wtedy niejawne przekazanie sterowania, powodujące przełączenie procesora z powrotem do punktu kontynuacji, określonego przez jawną instrukcję **Transfer.** Zasadę tę zilustrowano w poniższym przykładzie drukarki laserowej. Część programowalną nazywa się zwykle podprogramem obsługi przerwań. Część wykonywana przez drukarkę jest reprezentowana jawną instrukcją **Transfer.** Uważa się, że kluczową sprawą jest traktowanie podprogramu obsługi przerwań jak części całego cyklicznego procesu, a samych przerwań — jak nieuszeregowanego przekazywania sterowania współprogramu. Takie podejście znacznie ułatwia zrozumienie mechanizmu współpracy. Polega ono na przeniesieniu koncepcji przerwań z poziomu maszyny do języka strukturalnego i umożliwia ich implementację bez poświęcenia efektywności, co ma w tym wypadku istotne znaczenie.

```
MODULE PrinterDriver;
  IMPORT ADDRESS, WORD, ADR, InitCoroutine, Transfer;
  EXPORT out;
  CONST size = 100;
  VAR printer, main: ADDRESS; (*wskaźniki współprogramów*)
    buffer: ...
    wsp: ARRAY [0..size-1] OF WORD;
  PROCEDURE P;
  BEGIN
    LOOP
      (*jeżeli bufor jest niepusty, pobierz z niego dane, wyślij je
        do rejestru drukarki i uaktywnij ją*)
      Transfer(printer, main)
    END
  END P;
  PROCEDURE out(data: Type);
  BEGIN (*wpisz dane do bufora*)
    IF „drukarka wolna" THEN Transfer(main, printer) END
  END out;
```

```
BEGIN (*inicjowanie bufora*)
  InitCoroutine(P, wsp, size); printer := ADR(wsp)
END PrinterDriver.
```

SYGNAŁY

Jeżeli trzeba definitywnie abstrahować od obsługi procesorów fizycznych i przyjąć, że każdy proces jest wykonywany przez indywidualnego pośrednika, to pojęcie współprogramu staje się bezużyteczne. Istotą przejścia na wyższy poziom abstrakcji jest wprowadzenie anonimowości procesów, polegającej na tym, że nie określają one jawnie wzajemnego zawieszenia ani dokończenia. Ich synchronizację uzyskuje się stosując inne operacje elementarne. Do odpowiednich konstrukcji należą semafory [2] i warunki [3]. Poniżej omówiono sygnały, które są równoważne warunkom [10].

Sygnał deklaruje się jak zmienną, choć nie ma on wartości i dlatego nie można mu niczego przypisywać ani go kopiować. Można go jedynie wysyłać lub odbierać. Wysyłanie sygnału **s** oznacza, że spełniony został pewien warunek $\mathbf{P_s}$ (nałożony na zmienne). Dlatego proces, który odebrał sygnał **s**, może kontynuować działanie przy założeniu, że ten warunek jest spełniony. Warunek $\mathbf{P_s}$ jest warunkiem wstępnym (ang. precondition) operacji **Send(s)** i warunkiem końcowym (ang. postcondition) operacji **Receive(s).** Sygnał **s** jest komunikatem oznaczającym spełnienie tego warunku. Operacje są zdefiniowane w module definicyjnym **Signals,** zawierającym również procedurę **StartProcess(P)** i funkcję logiczną **Expected(s).** Bezparametrowa procedura **P** tworzy program procesu, a wartość **TRUE** funkcji **Expected(s)** oznacza, że przynajmniej jeden proces oczekuje na odbiór sygnału **s.**

```
DEFINITION MODULE Signals;
  TYPE Signal;
  PROCEDURE StartProcess(P: PROC);
    (*rozpoczęcie programu współbieżnego przez program P*)
  PROCEDURE Send(VAR s: Signal);
    (*dokończenie procesu oczekującego na sygnał s*)
  PROCEDURE Receive(VAR s: Signal);
    (*oczekiwanie na odbiór sygnału s*)
  PROCEDURE Expected(s: Signal): BOOLEAN;
  PROCEDURE IntiSignal(VAR s: Signal);
END Signals.
```

Użycie sygnałów zilustrowano dobrze znanym przykładem pary procesów współpracujących jako producent i konsument danych wymienianych przez bufor. Bufor i skojarzone z nim zmienne, tj. liczba elementów **n** oraz indeksy oznaczające następne wolne miejsce i następny pobierany element, tworzą łącznie sprzężenie (ang. interface) między obydwoma procesami. To sprzężenie jest zdefiniowane jako lokalny moduł i stanowi monitor bufora. Zawiera on sygnały: **nonempty,** dla warunku $\mathbf{n>0}$, i **nonfull,** dla warunku $\mathbf{n<N}$.

Sprzężenie (monitor) zawiera zazwyczaj również te zmienne lokalne, które są współdzielone przez procesy. Ponieważ sygnały są z natury współdzielone, powinny występować tylko w sprzężeniach. Reguła deklarowania obiektów współdzielonych obowiązkowo wewnątrz sprzężeń stanowi ważną zasadę wieloprogramowości, postulowaną przez Hoare'a i Brinch Hansena.

Zmienne współdzielone powinny być chronione przez wykluczanie wzajemne. Oznacza to, że wewnątrz sprzężenia stosuje się zwykle reguły programowania sekwencyjnego, ponieważ działania w sprzężeniu nie mogą być wykonywane przez dwa procesy jednocześnie. W poniższym przykładzie pominięto specyfikację wykluczania wzajemnego. Można tak postąpić przy założeniach, że program jest wykonywany na pojedynczym (współdzielonym) procesorze, i że przełączenie procesora następuje tylko w celu nadania lub odebrania sygnału.

Należy podkreślić, że w tym przykładzie jest oczywiste, który proces odbiera nadany sygnał, ponieważ istnieją tylko dwa procesy. Nie jest to jednak przypadek ogólny. Odbioru może dokonać dowolny z procesów, oczekujących na określony sygnał. Jednakże, pojedyncza operacja nadania sygnału powoduje wykonanie co najwyżej jednej operacji odbioru, tzn. nie ma rozgłaszania (ang. broadcast) — wydruk 3.

Weryfikacja poprawności modułu jest możliwa bez rozważenia kolejności współdziałania procesów. Należy ją zacząć od ustalenia niezmiennika sprzężenia, co w tym wypadku polega na stwierdzeniu, że bufor nie może być bardziej pusty od pustego i pełniejszy od pełniejszego, tzn. $\mathbf{0 \leqslant n \leqslant N}$. Z warunków końcowych **Receive(nonfull),** tzn. $\mathbf{n<N}$, i **Receive(nonempty),** tzn. $\mathbf{n>0}$, wynika, że element jest pobierany tylko wtedy, gdy $\mathbf{0<n\leqslant N}$ i umieszczany w buforze, gdy $\mathbf{0\leqslant n<N}$. Zauważmy, że rozważania dotyczące weryfikacji można wykonać dla sprzężenia dlatego, iż odnoszą się one tylko do obiektów lokalnych.

```
MODULE ProdCons;
  FROM Terminal IMPORT Read, Write;
  FROM Signals IMPORT
    Signal, StartProcess, Send, Receive, InitSignal;

  MODULE Interface;
    IMPORT Signal, StartProcess, Send, Receive, InitSignal;
    EXPORT get, put;
    CONST N = 8;

    VAR n, in, out: CARDINAL;
      nonfull, nonempty: Signal;
      buf: ARRAY [0 .. N-1] OF CHAR;

    PROCEDURE put(ch: CHAR);
    BEGIN
      IF n = N THEN Receive(nonfull) END ;
      n := n+1; buf[in] := ch; in := (in+1) MOD N;
      Send(nonempty)
    END put;

    PROCEDURE get(VAR ch: CHAR);
    BEGIN
      IF n = 0 THEN Receive(nonempty) END ;
      n := n-1; ch := buf[out]; out := (out+1) MOD N;
      Send(nonfull)
    END get;

  BEGIN n := 0; in := 0; out := 0;
    InitSignal(nonfull); InitSignal(nonempty)
  END Interface;


  PROCEDURE Producer;
    VAR i: CARDINAL; ch: CHAR;
      text: ARRAY [0..99] OF CHAR;
  BEGIN Write("(");
    i := 0; text := "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    WHILE text[i] > 0C DO
      Write("!"); Write(text[i]); put(text[i]); i := i+1
    END ;
    Write("!"); Write(")"); put(0C)
  END Producer;

  PROCEDURE Consumer;
    VAR ch: CHAR;
  BEGIN Write("[");
    Write("?"); get(ch);
    WHILE ch > 0C DO
      Write(ch); Write("?"); get(ch)
    END ;
    Write("]")
  END Consumer;

BEGIN
  StartProcess(Producer); Consumer; Write("$"); Write(36C)
END ProdCons.
```

Wydruk 3

Jak zauważył Dijkstra, to klasyczne rozwiązanie ma jedną wadę — sygnały są wysyłane częściej niż potrzeba. Przykładowo, sygnał **nonempty** jest wysyłany zawsze wtedy, gdy producent umieścił kolejny element, choć wywiera skutek tylko wtedy, gdy konsument go oczekiwał. Można temu zaradzić poddając operację **send(nonempty)** warunkowi zwanemu dozorem (ang. guard) **Expected(s).** Dijkstra zaproponował uwzględnienie tej informacji w liczniku **n** i nazwał to rozwiązanie „śpiącym golarzem". Wartości $\mathbf{n>0}$ oznaczają liczbę elementów w buforze (klientów w poczekalni), a wartości $\mathbf{n<0}$ — liczbę oczekujących konsumentów (wolnych golarzy). Jak widać, metodę rozszerza się łatwo na wypadek kilku konsumentów i producentów, nie wpływając na rozumowanie konieczne do zachowania poprawności. Poniżej przedstawiono kod sprzężenia zmodyfikowanego w celu uwzględnienia „śpiącego golarza" (wydruk 4).

```
MODULE Interface;
  IMPORT Signal, StartProcess, Send, Receive, InitSignal;
  EXPORT get, put;
  CONST N = 8;

  VAR n: INTEGER; in, out: CARDINAL;
    nonfull, nonempty: Signal;
    buf: ARRAY [0 .. N-1] OF CHAR;

  PROCEDURE put(ch: CHAR);
  BEGIN n := n+1;
    IF n > N THEN Receive(nonfull) END ;
    buf[in] := ch; in := (in+1) MOD N;
    IF n = 0 THEN Send(nonempty) END
  END put;

  PROCEDURE get(VAR ch: CHAR);
  BEGIN n := n-1;
    IF n < 0 THEN Receive(nonempty) END ;
    ch := buf[out]; out := (out+1) MOD N;
    IF n = N THEN Send(nonfull) END
  END get;

BEGIN n := 0; in := 0; out := 0;
  InitSignal(nonfull); InitSignal(nonempty)
END Interface
```

Wydruk 4

IMPLEMENTACJA SYGNAŁÓW

Można podejrzewać, że implementacja mechanizmu sygnalizacji i związane z tym zarządzanie procesorami mogą być dość złożone, a zatem nieefektywne. W rzeczywistości, powody do takich podejrzeń daje większość dostępnych wielozadaniowych systemów operacyjnych. Na szczęście, w tym wypadku implementacja może być bezpośrednia i efektywna, co wykazano poniższym rozwiązaniem.

Oparto je na założeniu, że każdy generowany proces jest reprezentowany przez deskryptor. Wszystkie deskryptory są połączone w strukturę pierścieniową. Aktualnie wykonywa-

ny proces jest oznaczony zmienną wskaźnikową **cp**. W wypadku systemu wieloprocesorowego każdy procesor ma swój prywatny wskaźnik cp. Nowy deskryptor (typu **RingNode**) jest przydzielany przez procedurę **StartProcess(P)** i włączany do pierścienia. Ta sama procedura przydziela procesowi przestrzeń roboczą i inicjuje jego deskryptor. Szczegóły inicjowania przestrzeni roboczej są takie same, jak w module Coroutines, co świadczy o tym, że zasady używania sygnałów oparto o technikę współprogramów.

Szczególnie godna uwagi jest reprezentacja sygnałów. Procesy nie określają sygnału, na który oczekują, lecz to sygnał określa procesy oczekujące na wysłanie go przez inny proces. Dlatego sygnał jest wskaźnikiem nagłówka listy (kolejki) oczekujących procesów. Procedura **Send(s)** powoduje przekazanie sterowania procesowi z listy s i usunięcie z niej odpowiedniego deskryptora. Łatwo można zagwarantować równorzędność szeregowania procesów, ponieważ procesy są zawsze dołączane do tytułu listy i usuwane z jej nagłówka. Procedura **Receive(s)** jest nieco bardziej skomplikowana. Po przebiegnięciu listy i dołączeniu do niej deskryptora bieżącego procesu, następuje przeszukiwanie pierścienia, w celu znalezienia procesu zdolnego do zakończenia (gotowego). Jeżeli nie ma takiego procesu, to system procesów wpada w zakleszczenie (ang. deadlock).

Bardziej złożone rozwiązanie może polegać na usunięciu oczekujących procesów z pierścienia, tzn. na faktycznym przesunięciu ich z pierścienia do listy sygnałów. Wymaga to ich przesunięcia z powrotem do pierścienia, po odebraniu nadanego sygnału. Wydaje się, że takie zwiększenie efektywności, w porównaniu z metodą przeszukiwania pierścienia, nie kompensuje dodatkowej złożoności operowania wskaźnikami, chyba że liczba procesów oczekujących przekracza znacznie liczbę procesów gotowych (wydruk 5).

Jeżeli proces osiąga koniec procedury tworzącej jego ciało, to jego wykonywanie można uznać za zakończone. Wtedy sterowanie powraca do miejsca za instrukcją wywołania, która zainicjowała proces. Kolejne instrukcje przełączają procesor na inny gotowy proces z kolejki i zwracają przestrzeń roboczą zakończonego procesu, dołączając ją do listy wskazywanej przez zmienną wskaźnikową **free**. Tak więc, przedstawiona implementacja uwzględnia również elementarne zarządzanie przydziałem i zwracaniem przestrzeni roboczej. Zazwyczaj, utworzenie i zakończenie procesu zdarza się znacznie rzadziej niż nadawanie sygnału, tj. przełączanie procesora. Ze względu na efektywność należy więc bardzo starannie zaprojektować procedury **Send** i **Receive**.

Blisko związane z sygnałami są także semafory proponowane przez Dijkstrę. Semafor składa się z licznika i ze skojarzonego z nim sygnału. Operacja **P** powoduje zmniejszenie zawartości licznika i — w wypadku otrzymania wartości ujemnej — oczekiwanie na odbiór sygnału. Operacja **V** powoduje zwiększenie zawartości licznika i — w przypadku dotatniego wyniku — nadanie sygnału. Ujemna zawartość licznika wskazuje, ile procesów oczekuje w kolejce na sygnał.

KANAŁY

Jeżeli system wieloprocesorowy składa się naprawdę z rozłożonych procesorów połączonych kanałami danych, a nie z procesorów mających tylko dostęp do wspólnej pamięci, to komunikacji nie można wyrazić używając zmiennych współdzielonych. Wtedy zaleca się metodę CSP zaproponowaną przez Hoare'a [4] i zrealizowaną w języku Occam [5]. Do podstawowych konstrukcji tego języka należą:

- instrukcja PAR określająca, że procesy S_0, S_1,...,S_{n-1} są wykonywane współbieżnie, np.:
  PAR S_0, S_1, ..., S_{n-1}
- deklaracja **CHAN ch** powodująca wprowadzenie kanału komunikacyjnego
- instrukcja „?" określająca odbiór wartości z kanału **ch** i przypisanie jej zmiennej **x**, np.:
  ch ? x
- instrukcja „!" określająca obliczenie wartości wyrażenia **x** i wysłanie jej do kanału **ch**, np.:
  ch ! x

Najpierw przekształcimy zwięzły zapis języka Occam na równoważną mu postać w Moduli-2, a następnie przedstawimy implementację tych konstrukcji. Oczywiście, wszystkie wymienione konstrukcje należy zdefiniować w module definicyjnym (wydruk 6).

```
IMPLEMENTATION MODULE Signals;
  FROM SYSTEM IMPORT ADDRESS, WORD, ADR, TSIZE;
  FROM Heap IMPORT Allocate;

  CONST WorkspaceSize = 2008;

  TYPE Signal = POINTER TO RingNode;
       CorPtr = POINTER TO Coroutine;

    RingNode =
      RECORD
        next, prev: Signal;  (*ring*)
        queue:      Signal;  (*queue of waiting processes*)
        cor:        CorPtr;
        ready:      BOOLEAN
      END ;

    Coroutine =
      RECORD
        G:     ADDRESS;
        L:     ADDRESS;
        PC:    ADDRESS;
        M:     BITSET;
        S:     ADDRESS;
        H:     ADDRESS;
        err:   CARDINAL;
        trapMask: BITSET;
        start: PROC;     (*start of workspace*)
        scnt:  CARDINAL;
        wsp:   ARRAY [0 .. WorkspaceSize-1] OF WORD
      END ;

  VAR cp: Signal;  (*current process*)
      aux: Signal;
      free: Signal;  (*chain of free process descriptors*)

  PROCEDURE TRANSFER(VAR from, to: CorPtr);
    CODE 256B; 0
  END TRANSFER;

  PROCEDURE StartProcess(P: PROC);

    PROCEDURE GlobalBase(): ADDRESS;
      CODE 258; 0    (*LGA 0*)
    END GlobalBase;

    PROCEDURE CALL;
      CODE 357B      (*CF*)
    END CALL;

    PROCEDURE SetPCandTransfer;
      PROCEDURE pc(): CARDINAL;
        CODE 40B; 2  (*LLW 2*)
      END pc;
    BEGIN cpt.corr.PC := pc() + 1; TRANSFER(aux.cor, cpt.cor)
    END SetPCandTransfer;

  BEGIN aux := cp;
    (*allocate a RingNode and a workspace contiguously*)
    IF free = NIL THEN
      Allocate(cp, TSIZE(RingNode)); Allocate(cpt.cor, TSIZE(Coroutine))
    ELSE cp := free; free := free.next
    END ;
    WITH cpt DO
      next := aux.next; prev := aux; queue := NIL; ready := TRUE
    END ;
    aux.next := cp; cpt.next.prev := cp;
    WITH cpt.corr DO
      G := GlobalBase(); L := 0;
      M := {};           S := ADR(wsp);
      H := ADR(wsp) + WorkspaceSize;

      err := 0;          trapMask := {};
      start := P;        scnt := 0
    END ;
    SetPCandTransfer;
    RETURN;

    CALL;      (*activate process body P*)
    aux := cp; cp := aux.next;
    cpt.prev := aux.prev; aux.prev.next := cp;
    aux.next := free; free := aux; aux := cp;
    WHILE NOT cpt.ready & (cp # aux) DO cp := cpt.next END ;
    IF cpt.ready THEN TRANSFER(free.cor, cpt.cor) END ;
    HALT (*deadlock*)
  END StartProcess;

  PROCEDURE Send(VAR s: Signal);
    VAR this: Signal;
  BEGIN this := cp;
    IF s # NIL THEN cp := s;
      s := cpt.queue; cpt.ready := TRUE
    ELSE (*release*)
      REPEAT cp := cpt.next UNTIL cpt.ready
    END ;
    IF cp # this THEN TRANSFER(this.cor, cpt.cor) END
  END Send;

  PROCEDURE Receive(VAR s: Signal);
    VAR this: Signal;
  BEGIN (*insert cp at end of queue s*)
    IF s = NIL THEN s := cp
    ELSE this := s;
      WHILE this.queue # NIL DO this := this.queue END ;
      this.queue := cp
    END ;
    this := cp; this.queue := NIL;
    REPEAT cp := cpt.next UNTIL cpt.ready;
    this.ready := FALSE;
    IF cp = this THEN (*deadlock*) HALT END ;
    TRANSFER(this.cor, cpt.cor)
  END Receive;

  PROCEDURE Expected(s: Signal): BOOLEAN;
  BEGIN RETURN s # NIL
  END Expected;

  PROCEDURE InitSignal(VAR s: Signal);
  BEGIN s := NIL
  END InitSignal;

BEGIN free := NIL; Allocate(cp, TSIZE(RingNode));
  WITH cpt DO
    next := cp; prev := cp; ready := TRUE
  END
END Signals.
```

Wydruk 5

```
DEFINITION MODULE Channels;
  TYPE Message = INTEGER;
       Process;
       Channel = RECORD prod, cons: Process END ;

  PROCEDURE Parallel(P,Q: PROC);
  PROCEDURE Send(VAR ch: Channel; msg: Message);
  PROCEDURE Receive(VAR ch: Channel; VAR msg: Message);
  PROCEDURE SenderWaiting(VAR ch: Channel): BOOLEAN;
  PROCEDURE ReceiverWaiting(VAR ch: Channel): BOOLEAN;
  PROCEDURE InitChannel(VAR ch: Channel);
END Channels.
```

Wydruk 6

W celu podkreślenia ich związku z metodą sygnałów, operację wyprowadzania „!" przekształcono na procedurę **Send**, a operację wprowadzania „?" — na procedurę **Receive**. Sygnał, który można uważać za pusty komunikat, zastąpiono komunikatem przybierającym wartość. Instrukcję Occamu:

PAR P Q

przekształcono na instrukcję:

Parallel (P', Q')

gdzie **P** i **Q** są bezparametrowymi procedurami reprezentującymi instrukcje **P** i **Q** w Occamie. Instrukcję:

PAR P Q R

wyraża się jako

Parallel (P,QR)

gdzie **QR** jest procedurą mającą ciało procedury

Parallel (Q,R).

Procedury funkcyjne **SenderWaiting** i **ReceiverWaiting** mają znaczenie analogiczne jak **Expected(s).**

Właściwości kanałów są następujące:

1. Jeżeli nadawca wysyła komunikat do kanału, to jest on opóźniany aż odbiorca po drugiej stronie kanału odbierze ten komunikat. Może to być natychmiast, jeżeli odbiorca oczekuje już w tym kanale.

2. Odbiorca oczekujący na komunikat z kanału doznaje opóźnienia aż nadawca wyśle ten komunikat do kanału. Może to być natychmiast, jeżeli nadawca już oczekiwał na odebranie komunikatu.

Wynika stąd, że kanał działa automatycznie jako element synchronizacyjny — synchronizacja i komunikacja nie różnią się. Ponadto, kanał jest zwykłym „przewodem" i nie ma możliwości buforowania. Dlatego nadawca i odbiorca biorą udział w spotkaniu (ang. rendezvous). Sam kanał nie jest skrzynką pocztową, lecz tylko miejscem spotkania.

Główna zasada Occamu polega na tym, że program reprezentuje system procesów połączonych kanałami, które są ustalone. Dlatego można założyć, że każdy kanał łączy jednego nadawcę i jednego odbiorcę.

W poniższym przykładzie przedstawiono prosty, lecz typowy system używający kanału. Proces **P** odczytuje ciąg liczb z nośnika wejściowego i kieruje je do kanału. Co czwarta liczba stanowi sumę kontrolną i nie jest nadawana. Proces **Q** odbiera liczby z kanału, oblicza sumę kontrolną po każdych siedmiu liczbach i kieruje je na nośnik wyjściowy (wydruk 7).

```
MODULE Sequences;
  FROM Channels IMPORT Channel, InitChannel, Parallel, Send, Receive;
  FROM InOut IMPORT ReadInt, WriteInt;

  VAR ch: Channel;

  PROCEDURE P;
    VAR i: CARDINAL; x, sum: INTEGER;
  BEGIN ReadInt(x);
    WHILE x # 0 DO i := 3;
      WHILE i > 0 DO
        Send(ch, x); sum := sum + x; i := i-1; ReadInt(x)
      END ;
      sum := sum + x; (*check sum = 0*) ReadInt(x)
    END ;
    Send(ch, 0)
  END P;

  PROCEDURE Q;
    VAR i: CARDINAL; x, sum: INTEGER;
  BEGIN Receive(ch, x);
    WHILE x # 0 DO
      i := 6; sum := 0;
      WHILE i > 0 DO
        WriteInt(x,6); sum := sum - x; i := i-1; Receive(ch, x)
      END ;
      WriteInt(sum, 6)
    END
  END Q;

BEGIN InitChannel(ch); Parallel(P,Q)
END Sequences.
```

Wydruk 7

IMPLEMENTACJA KANAŁÓW

Implementacja kanałów przypomina w dużym stopniu implementację sygnałów. Procesy są połączone w pierścień, którego każdy element (węzeł) zawiera wskaźnik do przestrzeni roboczej współprogramu. Na kolejnym polu przechowuje się adres przekazywanego komunikatu. Konieczne jest przechowywanie adresu, a nie samego komunikatu, aby nadawca mógł umieścić komunikat w zmiennej oznaczonej przez parametr w trybie VAR oczekującego odbiorcy.

Typ danych Channel przyjmuje rolę typu Signal. Zawiera on dwa pola rekordu, jedno dla oczekującego odbiorcy a drugie dla oczekującego nadawcy. Alternatywne rozwiązanie z jednym polem dla procesu i dyskryminatorem między nadawcą i odbiorcą odrzucono, ponieważ wymaga bardziej złożonego programu.

```
IMPLEMENTATION MODULE Channels;
  FROM SYSTEM IMPORT ADDRESS, WORD, ADR, TSIZE;
  FROM Heap IMPORT Allocate;

  CONST WorkspaceSize = 2008;

  TYPE
    Process  = POINTER TO RingNode;
    ProcessState = (ready, waiting, terminated);
    CorPtr = POINTER TO Coroutine;

    RingNode =
      RECORD
        next, prev: Process;  (*ring*)
        partner:    Process;
        cor:        CorPtr;
        state:      ProcessState;
        msgAdr:     POINTER TO Message
      END ;

    Coroutine =
      RECORD
        G:     ADDRESS;
        L:     ADDRESS;
        PC:    ADDRESS;
        W:     BITSET;
        S:     ADDRESS;
        H:     ADDRESS;

        err:   CARDINAL;
        trapMask: BITSET;
        start: PROC;     (*start of workspace*)
        scnt:  CARDINAL;
        wsp:   ARRAY [0 .. WorkspaceSize-1] OF WORD
      END ;

  VAR cp: Process;  (*current process*)
    free: Process;  (*chain of free process descriptors*)
    aux: Process;

  PROCEDURE TRANSFER(VAR from, to: CorPtr);
    CODE 266B; 0
  END TRANSFER;

  PROCEDURE Parallel(P,Q: PROC);
    VAR new: Process;  (*process to be created*)

    PROCEDURE GlobalBase(): ADDRESS;
      CODE 258; 0    (*LGA 0*)
    END GlobalBase;

    PROCEDURE CALL;
      CODE 3578      (*CF*)
    END CALL;

    PROCEDURE SetPCandCall;

      PROCEDURE pc(): CARDINAL;
        CODE 40B; 2  (*LLW 2*)
      END pc;

    BEGIN new^.cor^.PC := pc() + 1;
      P;
      WHILE new^.state # terminated DO
        (*release*) aux := cp;
        REPEAT cp := cp^.next UNTIL cp^.state = ready;
        IF cp = aux THEN HALT (*deadlock*) END ;
        aux^.state := terminated; TRANSFER(aux^.cor, cp^.cor)
      END ;
      aux:= new^.prev; new^.prev^.next := new^.next;
      new^.next^.prev := aux; new^.next := free; free := new
    END SetPCandCall;

  BEGIN
    (*allocate a RingNode and a workspace contiguously*)
    IF free = NIL THEN
      Allocate(new, TSIZE(RingNode));
      Allocate(new^.cor, TSIZE(Coroutine))
    ELSE new := free; free := free^.next
    END ;
    WITH new^ DO
      next := cp^.next; prev := cp; partner := cp; state := ready
    END ;
    cp^.next := new; new^.next^.prev := new;
    WITH new^.cor^ DO
        G := GlobalBase(); L := 0;
        W := {};           S := ADR(wsp);
        H := ADR(wsp) + WorkspaceSize;
        err := 0;          trapMask := {};
        start := Q;        scnt := 0
      END ;
      SetPCandCall;
      RETURN;

      CALL(*Q*);
      aux := cp;
      IF cp^.partner^.state = terminated THEN
        cp := cp^.partner; cp^.state := ready
      ELSE
        REPEAT cp := cp^.next UNTIL cp^.state = ready;
        IF cp = new THEN HALT (*deadlock*) END
      END ;
      aux^.state := terminated; TRANSFER(aux^.cor, cp^.cor)
    END Parallel;

    PROCEDURE Send(VAR ch: Channel; msg: Message);
      VAR this: Process;
    BEGIN this := cp;
      IF ch.cons # NIL THEN (*wake up consumer*)
        cp := ch.cons; ch.cons := NIL;
        cp^.state := ready; cp^.msgAdr := msg
      ELSE (*wait for consumer*)
        IF ch.prod # NIL THEN HALT END ;
        ch.prod := cp; cp^.msgAdr := ADR(msg);
        REPEAT cp := cp^.next UNTIL cp^.state = ready;
        this^.state := waiting;
        IF cp = this THEN HALT (*deadlock*) END
      END ;
      TRANSFER(this^.cor, cp^.cor)
    END Send;

    PROCEDURE Receive(VAR ch: Channel; VAR msg: Message);
      VAR this: Process;
    BEGIN this := cp;
      IF ch.prod # NIL THEN (*wake up consumer*)
        cp := ch.prod; ch.prod := NIL;
        cp^.state := ready; msg := cp^.msgAdr
      ELSE (*wait for producer*)
        IF ch.prod # NIL THEN HALT END ;
        ch.cons := cp; cp^.msgAdr := ADR(msg);
        REPEAT cp := cp^.next UNTIL cp^.state = ready;
        this^.state := waiting;
        IF cp = this THEN HALT (*deadlock*) END
      END ;
      TRANSFER(this^.cor, cp^.cor)
    END Receive;

    PROCEDURE SenderWaiting(VAR ch: Channel): BOOLEAN;
    BEGIN RETURN ch.prod # NIL
    END SenderWaiting;

    PROCEDURE ReceiverWaiting(VAR ch: Channel): BOOLEAN;
    BEGIN RETURN ch.cons # NIL
    END ReceiverWaiting;

    PROCEDURE InitChannel(VAR ch: Channel);
    BEGIN ch.prod := NIL; ch.cons := NIL
    END InitChannel;

  BEGIN free := NIL; Allocate(cp, TSIZE(RingNode));
    WITH cp^ DO
      next := cp; prev := cp; state := ready
    END
  END Channels.
```

Wydruk 8

Ze względu na wybraną metodę tworzenia procesów występują one zawsze parami. Każdy deskryptor procesu ma pole oznaczające partnera. O ile w wypadku sygnałów za-

kończenie procesu nie wywołało innych skutków, to w tym wypadku powoduje niejawną synchronizację. Proces macierzysty jest kontynuowany dopiero po zakończeniu obu procesów potomnych. Wymaga to poznania tożsamości partnera. W rzeczywistości, w przedstawionej implementacji instrukcja **Parallel (P,Q)** nie tworzy dwóch nowych procesów, lecz tylko jeden. Drugi jest identyczny z procesem generującym. Jest to konieczność praktyczna, ponieważ w przeciwnym wypadku połowa wymaganej przestrzeni roboczej byłaby stracona dla procesów oczekujących zakończenia obu swoich procesów potomnych (wydruk 8).

Stopień złożoności implementacji modułu Channels jest nieco większy niż modułu Signals. Wynika to głównie z synchronizacji po zakończeniu procesu. Ponadto, ta metoda jest mniej elastyczna, gdyż każdy kanał może być skojarzony w określonej chwili tylko z jednym nadawcą i jednym odbiorcą, natomiast sygnał może być w tej samej chwili oczekiwany przez wiele procesów. Jednakże w wielu wypadkach jest to odbiciem naturalnej sytuacji. Złym wyjściem byłoby z pewnością utworzenie — do elementarnych operacji komunikacyjnych — superkanału, do którego można by dołączać jednocześnie wielu nadawców i odbiorców. Niełatwo byłoby opisać charakterystykę takiego superkanału, nie mówiąc o jego implementacji. Wydaje się, że rozwiązanie oparte na sygnałach powinno być lepsze od opartego na kanałach, w przypadku systemu złożonego ze współdzielonych procesorów i wspólnej, dzielonej pamięci.

SYMULACJA

Język umożliwiający wyrażenie procesów współbieżnych i jego implementacja, o niewielkim narzucie wynikającym z przełączania procesów, są szczególnie dogodne do symulacji systemów zdarzeń dyskretnych. Przedstawione moduły wymagają jedynie niewielkich rozszerzeń, w celu uwzględnienia wymagań symulacji zdarzeń. Poniżej przedstawiono rozwiązanie oparte na koncepcji mającej swe źródła w języku Simula i propozycjach Hoare'a [6].

W rzeczywistości, jedynym koniecznym rozszerzeniem jest wprowadzenie pojęcia czasu. W symulacji zdarzeń dyskretnych, każdy obiekt aktywny należy do określonej kategorii (klasy, typu) procesów (jak na przykład: klienci, kasjerzy, obsługa domu towarowego) i dlatego jego zachowanie jest scharakteryzowane przez ustalony program sekwencyjny. Każde określone działanie, trwające w rzeczywistości przez czas **t**, jest w programie symulacyjnym wyrażone odpowiednią instrukcją i wywołaniem **Hold(t)**. Ta druga instrukcja zawiesza proces, aż czas wzrośnie o odcinek **t**.

```
DEFINITION MODULE Simulation;
  TYPE Singal;
    Process = PROCEDURE (CARDINAL);
  VAR Time: CARDINAL; (*tylko odczyt*)
  PROCEDURE StartProcess(P: Process; n: CARDINAL);
    (*rozpoczęcie procesu współbieżnego programu P(n)*)
  PROCEDURE Send(VAR s: Signal);
    (*kontynuacja procesu oczekującego na odbiór sygnału s*)
  PROCEDURE Receive(VAR s: Signal);
    (*oczekiwanie aż do odbioru sygnału s*)
PROCEDURE Hold(t: CARDINAL);
    (*wstrzymanie procesu na t sekund*)
  PROCEDURE InitSignal(VAR s: Signal);
    (*obowiązkowe inicjowanie*)
END Simulation
```

Poniżej przedstawiono przykład zastosowania tych prostych mechanizmów do opisu systemu zdarzeń dyskretnych.

Dawno temu cesarz chiński wydał rozkaz, aby zmierzono najkrótsze odległości wszystkich wiosek jego cesarstwa od stolicy. Zastosowano przy tym swoistą metodę pomiaru. Duże grupy mierniczych przemierzały kraj ze stałą prędkością i we wszystkich kierunkach, tzn. po każdej istniejącej drodze. Gdy grupa docierała do kolejnej wioski, wtedy dzieliła się na podgrupy wychodząc z wioski innymi drogami. Jeden członek grupy wracał, aby zameldować czas dojścia do wioski, a inny pozostawał, aby informować grupy przybyłe po nim.

W rozwiązaniu przedstawiono każdą wioskę za pomocą rekordu określającego jej nazwę (numer), liczbę wychodzących z niej dróg, ich kierunek (przeznaczenie) i długość. Ten sam rekord służy do zarejestrowania informacji, czy wioskę odwiedził już inny mierniczy (wydruk 9).

```
MODULE Army;
  FROM InOut IMPORT Done,
      OpenInput, ReadCard, Write, WriteLn, WriteCard, CloseInput;
  FROM Simulation IMPORT Time, StartProcess, Hold;
  CONST MaxNofVil = 32; MaxNofPaths = 8;
  TYPE Village =
    RECORD nofp: CARDINAL;
      visited: BOOLEAN;
      path: ARRAY [0..MaxNofPaths-1] OF
            RECORD destination, distance: CARDINAL END
    END ;
  VAR i: CARDINAL;
    vil: ARRAY [0..MaxNofVil-1] OF Village;
  PROCEDURE Scout(x: CARDINAL);
  (*x = currentVillage * MaxNofPath + direction*)
    VAR here, dir: CARDINAL;
  BEGIN Write("*"); here := x DIV MaxNofPaths; dir := x MOD MaxNofPaths;
    LOOP Hold(vil[here].path[dir].distance);
      here := vil[here].path[dir].destination;
      IF vil[here].visited THEN EXIT END ;
      WriteCard(here, 6); WriteCard(Time, 6); WriteLn;
      vil[here].visited := TRUE; dir := vil[here].nofp - 1;
      WHILE dir > 0 DO
        StartProcess(Scout, here * MaxNofPaths + dir); dir := dir-1
      END
    END ;
    Write("-")
  END Scout;
  PROCEDURE ReadData;
    VAR A, B, d, i: CARDINAL;
  BEGIN OpenInput("NUM"); ReadCard(A);
    FOR i := 0 TO MaxNofVil-1 DO
      vil[i].nofp := 0; vil[i].visited := FALSE
    END ;
    WHILE Done DO
      ReadCard(B); ReadCard(d);
      WITH vil[A] DO
        path[nofp].destination := B; path[nofp].distance := d;
        nofp := nofp + 1
      END ;
      WITH vil[B] DO
        path[nofp].destination := A; path[nofp].distance := d;
        nofp := nofp + 1
      END ;
      ReadCard(A)
    END ;
    CloseInput
  END ReadData;
BEGIN ReadData; vil[0].visited := TRUE;
  FOR i := 0 TO vil[0].nofp - 1 DO StartProcess(Scout, i) END ;
  Hold(9999); Write("!"); WriteLn
END Army.
```

Wydruk 9

IMPLEMENTACJA MODUŁU SYMULACYJNEGO

Bezpośrednia implementacja polega na utworzeniu kolejki opóźnionych procesów w postaci listy odpowiadającej wewnętrznemu sygnałowi TQ, nadawanemu w chwili, gdy żaden proces nie jest gotowy. W przeciwieństwie do zasady zastosowanej w module Signals w tej kolejce nie używa się metody FIFO (ang. first-in first-out), lecz porządkuje się procesy zgodnie z ich czasem przebudzenia. Po stwierdzeniu, że żaden proces nie jest gotowy, co może nastąpić po instrukcji **Receive** lub **Hold**, zamiast wykrycia zakleszczenia następuje dokończenie pierwszego procesu w kolejce **TQ** i zwiększenie czasu systemowego (zmienna Time) do wartości określonej jako czas przebudzenia. Ta metoda wymaga użycia dodatkowego pola w deskryptorze procesu, a włączenie procesu do kolejki, po jego wstrzymaniu, musi być poprzedzone poszukiwaniem właściwego miejsca w kolejce. W rzeczywistości, sygnał **TQ** ma niski priorytet, gdyż opóźnione procesy mogą być dokończone (tzn. czas systemowy może być zwiększony) tylko wtedy, gdy w chwili bieżącej nie ma innych gotowych procesów (wydruk 10).

PODSUMOWANIE

Trzy zasady współpracy procesów za pomocą współprogramów, sygnałów i przekazywania komunikatów porównano przy użyciu trzech programów testowych **CorTest**, **SigTest** i **ChanTest** przedstawionych na wydruku 11. Każdy z nich składa się z dwóch procesów, między którymi następuje powtarzalne przełączanie procesora. Dla 30 000 przełączeń zanotowano następujące czasy wykonywania:

CorTest — 1,8 s
SigTest — 5,0 s
ChanTest — 5,5 s

Przedstawione na wydruku 11 zasady wieloprogramowości oparto na różnych zbiorach operacji elementarnych do synchronizacji i komunikacji. W czasie implementacji okazało się, że pomimo różnic pojęciowych są one ściśle związane. Na podstawie przykładowych programów, można oszacować złożoność operacji elementarnych i powodowany przez nie narzut.

Wszystkie implementacje wyrażono całkowicie w Moduli-2, co świadczy o przydatności tego języka jako narzędzia programowania systemowego. Nawet szczegóły dotyczące użycia konkretnego komputera (Lilith) można wyrazić za pomocą niskopoziomowych konstrukcji języka, a ich zakres jest bardzo mały (p. moduł definicyjny Coroutines).

Wydaje się, że celowe może być odejście od włączania do języka konstrukcji dotyczących wieloprogramowości, chyba

```
IMPLEMENTATION MODULE Simulation;
  FROM SYSTEM IMPORT ADDRESS, WORD, ADR, TSIZE;
  FROM Heap IMPORT Allocate;
  CONST WorkspaceSize = 200B;
  TYPE Signal = POINTER TO RingNode;
       CorPtr = POINTER TO Coroutine;
    RingNode =
      RECORD
        next, prev: Signal;  (*ring*)
        queue:      Signal;  (*queue of waiting processes*)
        cor:        CorPtr;
        ready:      BOOLEAN;
        waketime:   CARDINAL
      END ;
    Coroutine =
      RECORD
        G:     ADDRESS;
        L:     ADDRESS;
        PC:    ADDRESS;
        M:     BITSET;
        S:     ADDRESS;
        H:     ADDRESS;
        err:   CARDINAL;
        trapMask: BITSET;
        start: Process;    (*start of workspace*)
        param: CARDINAL;
        scnt:  CARDINAL;
        wsp:   ARRAY [0 .. WorkspaceSize-1] OF WORD
      END ;
  VAR cp: Signal;  (*current process*)
      aux: Signal;
    free: Signal;  (*chain of free process descriptors*)
      TQ: Signal;  (*chain of delayed processes; time queue*)
  PROCEDURE TRANSFER(VAR from, to: CorPtr);
    CODE 256B; 0
  END TRANSFER;
  PROCEDURE StartProcess(P: Process; n: CARDINAL);
    PROCEDURE GlobalBase(): ADDRESS;
      CODE 25B; 0    (*LGA 0*)
    END GlobalBase;
    PROCEDURE CALL;
      CODE 357B      (*CF*)
    END CALL;
    PROCEDURE SetPCandTransfer;
      PROCEDURE pc(): CARDINAL;
        CODE 40B; 2  (*LLW 2*)
      END pc;
    BEGIN cpt.cort.PC := pc() + 1; TRANSFER(aux!.cor, cpt.cor)
    END SetPCandTransfer;
  BEGIN aux := cp;
    (*allocate a RingNode and a coroutine workspace*)
    IF free = NIL THEN
      Allocate(cp, TSIZE(RingNode)); Allocate(cpt.cor, TSIZE(Coroutine))
    ELSE cp := free; free := free^.next
    END ;
    WITH cpt DO
      next := aux!.next; prev := aux; queue := NIL; ready := TRUE
    END ;
    aux!.next := cp; cpt.next!.prev := cp;
    WITH cpt.cort DO
      G := GlobalBase(); L := 0;
      M := {};           S := ADR(wsp);
      H := ADR(wsp) + WorkspaceSize;
      err := 0;          trapMask := {};
      start := P;
      param := n;        scnt := 1
    END ;
    SetPCandTransfer;
    RETURN;
    CALL;      (*activate process body P*)
    aux := cp; cp := aux!.next;
    cpt.prev := aux!.prev; aux!.prev!.next := cp;
    aux!.next := free; free := aux; aux := cp;
    WHILE NOT cpt.ready & (cp # aux) DO cp := cp^.next END ;
    IF cpt.ready THEN TRANSFER(free!.cor, cpt.cor)
    ELSIF TQ # NIL THEN
      cp := TQ; TQ := TQ^.queue; cpt.ready := TRUE;
      Time := cpt.waketime; TRANSFER(free!.cor, cpt.cor)
    ELSE (*deadlock*) HALT
    END
  END StartProcess;
  PROCEDURE Send(VAR s: Signal);
    VAR this: Signal;
  BEGIN
    IF s # NIL THEN
      this := cp; cp := s;
      WITH cpt DO
        s := queue; ready := TRUE
      END ;
      TRANSFER(this!.cor, cpt.cor)
    END
  END Send;
  PROCEDURE release;
    VAR this: Signal;
  BEGIN
    this := cp;
    REPEAT cp := cpt.next UNTIL cpt.ready;
    this!.ready := FALSE;
    IF cp = this THEN (*advance time*)
      IF TQ # NIL THEN
        cp := TQ; TQ := TQ^.queue; cpt.ready := TRUE;
        Time := cpt.waketime
      ELSE (*deadlock*) HALT
      END
    END ;
    TRANSFER(this!.cor, cpt.cor)
  END release;
  PROCEDURE Receive(VAR s: Signal);
    VAR this: Signal;
  BEGIN (*insert cp at end of queue s*)
    IF s = NIL THEN s := cp
    ELSE this := s;
      WHILE this!.queue # NIL DO this := this!.queue END ;
      this!.queue := cp
    END ;
    cpt.queue := NIL; release
  END Receive;
  PROCFOURE Hold(t: CARDINAL);
    VAR T: CARDINAL; this, q0, q1: Signal;
  BEGIN T := Time + t;
    IF TQ = NIL THEN
      TQ := cp; cpt.queue := NIL
    ELSIF TQ^.waketime > T THEN
      cpt.queue := TQ; TQ := cp
    ELSE q0 := TQ;
      LOOP (*q0 # NIL*) q1 := q0^.queue;
        IF q1 = NIL THEN
          q0^.queue := cp; cpt.queue := NIL; EXIT
        ELSIF q1^.waketime > T THEN
          cpt.queue := q1; q0^.queue := cp; EXIT
        ELSE q0 := q1
        END
      END
    END ;
    cpt.waketime := T; release
  END Hold;
  PROCEDURE InitSignal(VAR s: Signal);
  BEGIN s := NIL
  END InitSignal;
BEGIN free := NIL; Time := 0; TQ := NIL; Allocate(cp, TSIZE(RingNode));
  WITH cpt DO
    next := cp; prev := cp; ready := TRUE
  END
END Simulation.
```

Wydruk 10

że współbieżność odgrywa tak dominującą rolę, iż dostatecznie ważne stanie się operowanie wygodną składnią. W większości wypadków można używać języka uniwersalnego, mającego odpowiednie konstrukcje niskiego poziomu, które wykorzystuje się stosując pojęcia programowania strukturalnego.

```
MODULE CorTest;
  FROM Coroutines IMPORT InitCoroutine;
  FROM Terminal IMPORT Read, Write;
  FROM SYSTEM IMPORT ADR, WORD, ADDRESS;
  CONST WspSize = 200;
  VAR n: CARDINAL; ch: CHAR;
      main, proc: ADDRESS;
      wsp: ARRAY [0..WspSize-1] OF WORD;
  PROCEDURE Transfer(VAR from, to: ADDRESS);
    CODE 256B; 0
  END Transfer;
  PROCEDURE P;
  BEGIN
    LOOP n := n-1; Transfer(proc, main) END
  END P;
BEGIN n := 60000; proc := ADR(wsp); InitCoroutine(P, proc, WspSize);
  Write("["); Read(ch);
  REPEAT Transfer(main, proc) UNTIL n = 0;
  Write("]")
END CorTest.

MODULE SigTest;
  FROM Signals IMPORT Signal, StartProcess, Send, Receive, InitSignal;
  FROM Terminal IMPORT Read, Write;
  VAR n: CARDINAL; ch: CHAR; tested: Signal;
  PROCEDURE P;
  BEGIN
    LOOP n := n-1; Receive(tested) END
  END P;
BEGIN n := 60000; InitSignal(tested);
  Write("["); Read(ch); StartProcess(P);
  REPEAT (* n>0 *) Send(tested) UNTIL n = 0;
  Write("]")
END SigTest.

MODULE ChanTest;
  FROM Channels IMPORT Channel, Parallel, Send, Receive, InitChannel;
  FROM Terminal IMPORT Read, Write;
  VAR n: INTEGER; ch: CHAR; chan: Channel;
  PROCEDURE P;
    VAR k: INTEGER;
  BEGIN
    REPEAT Receive(chan, k) UNTIL k = 0
  END P;
  PROCEDURE Q;
  BEGIN
    REPEAT n := n-1; Send(chan, n) UNTIL n = 0
  END Q;
BEGIN n := 30000; InitChannel(chan);
  Write("["); Read(ch); Parallel(Q,P); Write("]")
END ChanTest.
```

Wydruk 11

LITERATURA

[1] Ben-Ari M.: Principles of Concurrent Programming. Prentice-Hall, Englewood Cliffs, NJ, 1982

[2] Dijkstra E. W.: Cooperating sequential processes. Programming Languages, F. Genuys (ed.). Academic Press, 1986

[3] Hoare C. A. R.: Monitors — An operating system structuring concept. Communications of the ACM, vol. 17 (18), 549—557 (1974)

[4] Hoare C. A. R.: Communicating sequential processes. Communications of the ACM, vol. 21 (8), 666—677. (1978)

[5] Inmos Ltd.: Occam Programming Manual. Prentice-Hall Englewood Cliffs, NJ, 1984

[6] Kaubisch W. H., Perrott R. H., Hoare C. A. R.: Quasiparallel programming. Software — Practice and Experience, vol. 6, 341—356 (1976)

[7] Welsh J., Lister A.: A comparative study of task communication in Ada. Software — Practice and Experience, vol 11, 257—290 (1981)

[8] Welsh J., Lister A., Salzman E.: A comparison of two notations for process communications. Language Design and Programming Methodology, vol. 1, 225—254 (1980)

[9] Williamson R., Horowitz E.: Concurrent communication and synchronization mechanism. Software — Practice and Experience, vol. 14 (2), 135—151 (1984)

[10] Wirth N.: Modula — A programming language for modular multiprogramming. Software — Practice and Experience, vol. 7 (1), 37—52 (1977)

[11] Wirth N.: Programming in Modula-2. Springer—Verlag, Heidelberg, 1982

LITERATURA DODATKOWA

[12] Abramowicz W.: Modula-2 i Lilith — zgodność metod i narzędzi informatycznych. Informatyka, nr 4, 1984

[13] Fuglewicz P.: Modula-2 — język lat osiemdziesiątych. Informatyka, nr 1, 2, 3, 1984.

Zmiany i uzupełnienia w Moduli-2

W listopadzie 1983 r. odbyło się spotkanie przedstawicieli wielu firm, które zaimplementowały Modulę-2. Zaproponowano wówczas dodanie lub ulepszenie wielu konstrukcji języka. Poniżej przedstawiono zbiór uzgodnionych propozycji. Należy je uważać za zmiany w definicji języka. Autorom przyszłych implementacji zaleca się uwzględnienie tych zmian, a istniejące kompilatory należy poddać odpowiednim przeróbkom. Choć każda zmiana w definicji języka wywołuje sprzeciw, zakres poniższych zmian jest bardzo mały, a autor wierzy, że każda z nich stanowi znaczne ulepszenie.

1. WYJAŚNIENIA

1.1 Typy parametru formalnego **VAR** i odpowiadającego mu parametru aktualnego muszą być identyczne (a nie tylko zgodne). Reguła ta nie dotyczy parametru formalnego typu ADDRESS, który może być zgodny ze wszystkimi typami wskaźnikowymi, oraz typu **WORD**, dla którego typy zgodne są określone w każdej implementacji oddzielnie.

1.2 Typy wyrażeń określających początkową i końcową wartość zmiennej sterującej w instrukcji **for** muszą być zgodne z typem tej zmiennej (tzn. nie tylko zgodne przez przypisanie).

1.3 Proces zainicjowany w module o priorytecie n nie może wywoływać procedury zadeklarowanej w module o priorytecie **m<n**. Dopuszczalne są wywołania procedur zadeklarowanych bez priorytetu.

1.4 Typy wskaźnikowe mogą być eksportowane z modułu definicyjnego jako typy nieprzejrzyste (ang. opaque). Nieprzejrzysty eksport innych typów może podlegać ograniczeniom implementacyjnym. Do typów nieprzejrzystych stosuje się przypisanie i badanie równości.

1.5 Wszystkie moduły importowane do modułu głównego są inicjowane przed zainicjowaniem modułu importującego. Jeżeli istnieją odwołania wzajemne, to kolejność inicjowania nie jest określona.

1.6 [1]) Jeżeli importowany jest identyfikator modułu, nie oznacza to, że widoczne stają się identyfikatory obiektów z tego modułu. Jednakże, obiekty eksportowane w trybie kwalifikowanym mogą być dostępne przez poprzedzenie ich identyfikatorem modułu.

2. ZMIANY

2.1 Wszystkie obiekty zawarte w module definicyjnym są eksportowane. Usuwa się jawną listę eksportową. Moduł definicyjny można uważać za wydzieloną i rozszerzoną listę eksportową modułu implementacyjnego [2]).

ModułDefinicyjny = DEFINITION MODULE identyfikator ";" {import} {definicja} END identyfikator ";".

2.2 Składnię deklaracji typu rekordu wariantowego, bez pola znacznikowego (ang. tag field) zmienia się z obowiązującej dotychczas:

ListaPól = |CASE [identyfikator ":"]...

na następującą:

ListaPól = |CASE [identyfikator] ":" ...

Dzięki obecności dwukropka oczywiste staje się, którą część pominięto.

2.3 [3]) Parametry procedury TRANSFER są typu ADDRESS. Typ PROCESS (= ADDRESS) i procedury NEWPROCESS i TRANSFER mogą — lecz nie muszą — być zawarte w module SYSTEM.

[1]) Punkt ten zawarto w raporcie [1], natomiast usunięto z publikacji [2]

[2]) Por. opis składni w Informatyce nr 2, 1984, str. 9

3. ROZSZERZENIA

3.1 Składnia instrukcji CASE i deklaracji rekordu wariantowego zostaje zmieniona z dotychczas obowiązującej [4]):

Ewentualność = ListaEtykiet ":" CiągInstrukcji.
Wariant = ListaEtykiet":" Ciąg ListPól.

na następującą:

Ewentualność = [ListaEtykiet ":" CiągListPól].
Wariant = [ListaEtykiet ":" CiągListPól].

Włączenie pustej ewentualności i pustego wariantu umożliwia umieszczenie w programie nadmiarowej belki (ang. bar — kreska pionowa), podobnie jak instrukcja pusta umożliwia umieszczenie nadmiarowych średników.

3.2 Mówi się, że napis złożony z n znaków ma długość n. Napis o długości równej 1 jest zgodny z typem **CHAR**.

3.3 Składnia typu okrojonego zostaje zmieniona z dotychczas obowiązującej:

TypOkrojony = "["WyrażenieStałe".." WyrażenieStałe"]".

na następującą:

TypOkrojony = [identyfikator] "["WyrażenieStałe".." WyrażenieStałe"]".

Opcjonalny identyfikator umożliwia określenie typu bazowego, np. **INTEGER [0..99]**.

3.4 Zniesiono ograniczenie, aby elementy zbiorów były stałymi. Składnię zbiorów i czynników zmieniono na następującą:

CzynnikStały = ... |ZbiórStały| ...
ZbiórStały = [identkwalif] "{" [ElementStały {"," ElementStały}] "}".
ElementStały = WyrażenieStałe [".." WyrażenieStałe].
Czynnik = ... |Zbiór| ...
Zbiór = [identkwalif] "{" [element {"," element}] "}".
element = wyrażenie [".." wyrażenie].

3.5 Znak ~ jest synonimem symbolu **NOT**.

3.6 Identyfikatory **LONGCARD, LONGINT** i **LONGREAL** oznaczają typy standardowe, które mogą nie być dostępne w pewnych implementacjach.

3.7 Typ **ADDRESS** jest zgodny ze wszystkimi typami wskaźnikowymi oraz z jednym z typów: **CARDINAL** lub **LONGCARD** [5]). Interpretacja adresów jako liczb zależy od implementacji.

3.8 Funkcje standardowe MIN i MAX mogą mieć argumenty dowolnego typu skalarnego (włącznie z typem REAL). Udostępniają one najmniejszą i największą wartość typu.

Oprac. JZ

LITERATURA

[1] Wirth N.: Revisions and Amendments to Modula-2. ETH Institut fur Informatik, Zurich, 24 May 1984

[2] Wirth N.: Revisions and Amendments to Modula-2. Journal of Pascal, Ada and Modula-2, Vol. 4, No 1, pp. 25—28, 1985

[3]) W raporcie [1] punkt ten ma następujące brzmienie: typ PROCESS zostaje usunięty z modułu SYSTEM, a jego miejsce zajmuje typ ADDRESS

[4]) Termin ewentualność jest odpowiednikiem angielskiego case

[5]) W raporcie [1] wymieniono również typ LONGINT

Do artykułu Niklausa Wirtha i opracowania na temat współbieżności Moduli-2, dołączamy listę znanych implementacji tego języka i program wzorcowy, który służył do testowania translatorów. Sądzimy, że ta informacja stanowi istotne uzupełnienie naszych dotychczasowych publikacji o tym języku. **(Red.)**

Modula-2
lista implementacji i program wzorcowy

Lista znanych implementacji Moduli-2 (oprac. JZ.)

Instytucja	Kontakt	Komputer macierzysty	System operacyjny	Komputer docelowy	Koszt wersji	
					źródłowej	binarnej
Institut für Informatik ETH-Zentrum Clausiusstrasse 55, CH-8092 Zurich, Szwajcaria	—	PDP-11/40 Lilith	RT-11 Medos	PDP-11/40 M-code	350 SFR 350 SFR	— —
Rechenzentrum ETH-Zentrum CH-8092 Zurich, Szwajcaria	Mr. H. Seiler	Cyber	NOS/BE	PDP-11 MC 6809 MC 68000	350 SFR 350 SFR 350 SFR	— — —
Institut für Elektronik ETH-Zentrum Gloriastrasse 35, CH-8092 Zurich, Szwajcaria	Dr. H. Burkhart	MC 68000	specjalny	MC 68000	350 SFR	—
University of New South Wales Dept. of Computer Science P.O. Box 1, Kensington, NSW 2033, Australia	Dr. J. Tobias	PDP-11	UNIX V7	PDP-11	150 dol. austr.	—
Digital Equipment Corporation Western Research Laboratory, 4410 El Comino Real, Los Altos, CA 74022, USA	M.L. Powell	VAX	4 x BSD UNIX	VAX	—	100 dol. USA
University of Cambridge Computer Laboratory. Corn Exchangestreet Cambridge, B2 3QG, Wielka Brytania	Dr. P. Robinson	VAX 68000 FNS 16032	4.1BSD UNIX UNIX V7 UNIX	VAX 68000 NS 16032	100 funt. 100 funt. w opracowaniu	— —
Acorn Research, Suite 910 5 Palo Alto Square, Palo Alto, Ca 94306, USA	Dr M.J. Jordan	NS 16032		NS 16032	w opracowaniu	
Nottingham University, Dep. of Psychology Nottingham, NG7 2RD, Wielka Brytania	Dr R.B. Henry	PDP-11	RT-11 UNIX V7	PDP-11	—	—
Universität Karlsruhe Fakultät für Informatik Postfach 6380, D-7500 Karlsruhe 1, RFN	Dr D. Schwarz	68000	UNIX V7	68000	400 marek	—
Universität Dortmund, Informatik III Postfach 500500, D-4600 Dortmund 50, RFN	Dr W. Kuhnhauser	PDP-11	UNIX V7	PDP-11	—	— —
BBC Brown Bovery and Co. Dept. ESL CH-5401 Baden/Turgi, Szwajcaria	Dr J. Muheim	PDP-11	RSX-11M/S	PDP-11	—	1000—2500 SFR
University of Virginia Medical Center Dep. of Biomedical Engineering Box 377, Charlottesville, VA, 22908, USA	T. Breeden	PDP-11	RSX-11	PDP-11	—	—
J. Kepler Universität Institut für Informatik Altenbergerstrasse 69, A-4040 Linz Auhof, Austria,	Prof. P. Rechenberg Dr. G. Pomberger	8080	ISIS-2	8086	—	—
Logitech SA, CH-1143 Apples, Szwajcaria	Mr. W. Steiger	VAX 8086 IBM PC Z80 8080	VMS CP/M MS-DOS 2.0 CP/M, UCSD CP/M, UCSD	VAX 8086 8086 Z80 8080	— — — —	? 495 dol. 495 dol. 495 dol. 495 dol.
Volition Systems, P.O. Box 1236, Del Mar, CA 92014, USA	Ms. T. Barrett	68000 IBM PC Z80 8080 Apple II Apple III	UCSD V2 UCSD V2 UCSD V2 UCSD V2 Pascal SOS/Pascal	68000 8086 Z80 8080 6502 6502	— — — — — —	495 dol. 395 dol. 595 dol. 595 dol. 295 dol. 495 dol.
Universität Frankfurt, Fachbereich Informatik Dantestrasse 9, Frankfurt, /Main, RFN	Dr. J.W. Schmidt	VAX	VMS	VAX		200 dol.
V.U.B. Rekenzentrum Pleinlaan 2, B-1050 Brussels, Belgia	Mr. F. Maene	Cyber	NOS 1.4	PDP-11 M-code 68000 6809 Z8002 Z80 8080	— — — — — — —	50 dol. 50 dol. 50 dol. 50 dol. 50 dol. 50 dol. 50 dol.
CERN DD Division CH-1211 Geneva 23, Szwajcaria	Dr. J.D. Blake	VAX IBM 370 IBM 370 IBM 370	4.2BSD UNIX MVS MVS MVS	68000 68000 6809 TMS 9900	— — — —	— — — —
Borroughs Machines Software Dept. Castle Cary Road, Cumbernauld, Scotland, Wielka Brytania	Mr. R. Jones	B-6800	Borroughs	MC 68000	—	—

Program wzorcowy do testowania kompilatorów Moduli-2

```
MODULE BenchMark;
  (*$T—
    a: pusta pętla REPEAT
    b: pusta pętla WHILE
    c: pusta pętla FOR
    d: arytmetyka liczb typu CARDINAL
    e: arytmetyka liczb typu REAL
    f: funkcje standardowe
    g: tablica jednowymiarowa
    h: tablica jednowymiarowa z kontrolą indeksów
    i: dostęp do macierzy
    j: dostęp do macierzy z kontrolą indeksów
    k: wywołanie pustej procedury bezparametrowej
    l: wywołanie pustej procedury 4-parametrowej
    m: kopiowanie macierzy (blokowe)
    n: łączenie wskaźników
    o: odczyt pliku*)

  FROM Storage IMPORT ALLOCATE;
  FROM Terminal IMPORT Read, BusyRead,
       Write, WriteLn;
       InOut IMPORT WriteCard;
  FROM FileSystem IMPORT
       File, Lookup, ReadWord, Reset,
       Response;
  FROM MathLib0 IMPORT sin, exp, ln, sqrt;

  TYPE NodePtr = POINTER TO Node;
       Node = RECORD x, y: CARDINAL;
       next: NodePtr END;

  VAR A,B,C: ARRAY [0..255] OF
       CARDINAL;
       M: ARRAY [0..99],[0..99] OF
       CARDINAL
       m: CARDINAL;
       head: NodePtr;

PROCEDURE Test(ch: CHAR);
  VAR i, j, k: CARDINAL;
  r0, r1, r2: REAL;
  p: NodePtr;

  PROCEDURE P;
  BEGIN
  END P;

  PROCEDURE Q(x, y, z, w: CARDINAL);
  BEGIN
  END Q;

BEGIN
  CASE ch OF
    "a": k := 20000;
         REPEAT
           k := k-1
         UNTIL k = 0 |
    "b": i := 20000;
         WHILE i > 0 DO
           i := i-1
         END |
    "c": FOR i := 1 TO 20000 DO
         END |
    "d": j := 0; k := 10000;
         REPEAT
           k := k-1; j := j+1;
           i := (k*3) DIV (j*5)
         UNTIL k = 0 |
    "e": k := 5000; r1 := 7.28;
         r2 := 34.8;
         REPEAT
           k := k-1; r0 := (r1*r2) /
           (r1+r2)
         UNTIL k = 0 |
    "f": k := 500;
         REPEAT
           r0 := sin(0.7);
           r1 := exp(2.0);
           r0 := ln(10.0);
           r1 := sqrt(18.0);
           k := k-1
         UNTIL k = 0 |
    "g": k := 20000; i := 0; B[0] := 73;
         REPEAT
           A[i] := B[i]
           B[i] := A[i];
           k := k-1
         UNTIL k = 0 |
    "h": (*$T+ *) k := 20000; i := 0;
         B[0] := 73;
         REPEAT
           A[i] := B[i];
           B[i] := A[i];
           k := k-1
         UNTIL K = 0 (*$T – *) |
    "i": FOR i := 0 TO 99 DO
         FOR j := 0 TO 99 DO
           M[i,j] := M[j,i]
         END
         END |
    "j": (*$T + *)
         FOR i := 0 TO 99 DO
           FOR j := 0 TO 99 DO
             M[i,j] := M[j,i]
           END
         END (*$T + *) |
    "k": k := 20000;
         REPEAT
           P; k := k-1
         UNTIL k = 0 |
    "l": k := 20000;
         REPEAT
           Q(i,j,k,m); k := k-1
         UNTIL k = 0 |
    "m": k := 500;
         REPEAT
           k := k-1; A := B; B := C;
           C := A
         UNTIL k = 0 |
    "n": k := 500;
         REPEAT
           p := head;
           REPEAT p := p↑.next
           UNTIL p = NIL;
           k := k-1
         UNTIL k = 0|
    "o": k := 5000;
         REPEAT
           k := k-1; ReadWord(f,i)
         UNTIL k = 0;
         Reset (f)

  END (* CASE *)
END Test;

VAR ch,ch1: CHAR;
    n: CARDINAL;
    f: File;
    q: NodePtr;

BEGIN
  Lookup(f,"anyfile", FALSE);
  head := NIL; n := 100;
  REPEAT
    q := head; NEW(head); head↑.next := q;
    n := n-1
  UNTIL n = 0;
  Write(">"); Read(ch);
  WHILE ("a" <= ch) & (ch <= "p") DO
    Write(ch); WriteLn; n := 0;
    REPEAT
      n := n+1; Test(ch);
      IF (n MOD 50) = 0 THEN WriteLn END;
      Write("."); BusyRead(ch1)
    UNTIL ch1 # 0C;
    WriteCard(n,6); WriteLn; Write(">"); Read(ch)
  END;
  Write(14C)
END BenchMark.
```

Konferencje

II Międzynarodowa Szkoła MIKROKOMPUTER '86

Instytut Cybernetyki Technicznej Politechniki Wrocławskiej organizuje w dniach 23—26 września 1986 r. w Bierutowicach, II Międzynarodową Szkołę MIKROKOMPUTER '86.

CELE SZKOŁY

- Wymiana wiedzy i doświadczeń praktycznych dotyczących niezawodności i diagnostyki systemów mikroprocesorowych
- Prezentacja wyników wdrożeń
- Przegląd najnowszego dorobku badawczego w dziedzinie techniki mikroprocesorowej

TEMATYKA OBRAD I SEKCJE

- Niezawodność systemów mikroprocesorowych
- Diagnostyka systemów mikroprocesorowych
- Niezawodność oprogramowania mikrokomputerów
- Systemy mikroprocesorowe tolerujące uszkodzenia
- Postępy techniki mikroprocesorowej

ORGANIZACJA OBRAD

- Cykl wykładów wygłaszanych przez zaproszonych specjalistów
- Referaty uczestników, prezentujące ich dorobek badawczy

JĘZYKI SZKOŁY

- Angielski, rosyjski (teksty)
- Dopuszcza się wygłaszania wykładów i referatów w języku polskim

WARUNKI UCZESTNICTWA

- Nadesłanie wypełnionej deklaracji uczestnictwa w terminie do 1 czerwca 1986
- Wpłacenie, w terminie do 1 czerwca 1986, kwoty 9400 zł na konto nr 93057-3418-131 w NBP V O/M Wrocław. Na przekazie należy podać imię i nazwisko uczestnika oraz skrót ICT-MIKROKOMPUTER '86
- Ze względu na ograniczoną liczbę miejsc pierwszeństwo uczestnictwa mają Autorzy prac. W pozostałych wypadkach decydować będzie kolejność zgłoszeń.
- Potwierdzenie udziału w obradach Szkoły zostanie przesłane w terminie do 30 sierpnia 1986.
- Koszty uczestnictwa obejmują: udział w obradach, materiały Szkoły, wyżywienie i zakwaterowanie (4 doby á 1500 zł).

IMPREZY TOWARZYSZĄCE

- Ekspozycja sprzętu mikrokomputerowego i jego oprogramowania
- Giełda oprogramowania
- Udział w wystawie i giełdzie prosimy zgłaszać w terminie do 1 czerwca 1986, przedstawiając zakres oferty oraz wymagania dot. powierzchni, zasilania, plansz itp.

MIEJSCE OBRAD

- Bierutowice, DW „Szczyt", ul. Śnieżki 6
- Bierutowice, DW „Wang", ul. Na Śnieżkę 3

Dojazd z Jeleniej Góry autobusami PKS do przystanku Bierutowice—Wang

ADRES DO KORESPONDENCJI

MIKROKOMPUTER '86
Instytut Cybernetyki Technicznej
Politechniki Wrocławskiej
ul. Janiszewskiego 11—17
50-372 Wrocław
Telefony: 20-27-45, 21-26-77
Teleksy: 0712254 pwr, 0712559 pwr pl

JERZY KARCZMARCZUK
Instytut Fizyki
Uniwersytet Jagielloński
Kraków

Język programowania Icon (I)

Icon jest stosunkowo nowym językiem programowania, przeznaczonym głównie do analizy i przetwarzania tekstów i innych struktur danych o dużym stopniu złożoności strukturalnej i kontekstowej, do rozwiązywania problemów z dziedziny sztucznej inteligencji itd. O potrzebie rozwijania narzędzi programowania w tych dziedzinach nie trzeba nikogo przekonywać. Icon jest jednak również językiem uniwersalnym o „dużej mocy", w sensie liczby wbudowanych operatorów i funkcji, typów danych i wyrafinowanych struktur sterowania. Jest językiem zewnętrznie podobnym do Pascala, bardzo klasycznym, co znacznie ułatwia pierwszy z nim kontakt. Jest on jednak bezpośrednim potomkiem języka Snobol 4 [1] i ma z nim wiele cech wspólnych. Głównym twórcą Iconu, jak i Snobolu, jest prof. Ralph E. Griswold z University of Arizona w Tucson. Opracowanie Iconu, zainicjowano w drugiej połowie lat siedemdziesiątych, gdy już było oczywiste, że Snobol — mimo że nadal intensywnie eksploatowany i użyteczny — po prostu przestaje wystarczać. Wraz ze starzeniem się języka programowania (i nie tylko języka) niektóre jego cechy stają się coraz bardziej dokuczliwe, zwłaszcza gdy odbiegają od współczesnych standardów. Tak też było z prymitywnymi strukturami sterowania w Snobolu, z jego niezbyt efektywną gospodarką pamięcią i czasem maszyny, z pewnymi manieryzmami składniowymi.

Icon nie jest jednak tylko Snobolem po „generalnym remoncie". Pewne rozwiązania w nim zawarte czynią programowanie w Iconie czymś stylistycznie bardzo charakterystycznym, podobnie jak charakterystycznym dla Snobolu jest programowanie przy użyciu wzorców tekstowych. Wzorców takich w Iconie nie ma (nie są one potrzebne jako specjalne struktury danych), co powoduje, że zagadnienie przeszukiwania i rozbioru tekstów jest w nim bardziej niż w Snobolu zintegrowane z resztą języka.

Obecnie rozpowszechniane są dwie wersje języka: wersja 2 napisana w Ratforze wyłącznie po to, aby można ją praktycznie wszędzie łatwo instalować. Wersja ta powinna wkrótce zostać oddana do powszechnego użytku w Polsce. Nowsza wersja 5 jest napisana w języku C i pracuje pod nadzorem UNIXA. Sprowadzenie jej do kraju jest kwestią przyszłości, miejmy nadzieję, bliskiej.

Autor artykułu nie ma ambicji nauczania Iconu ani przedstawienia wyczerpującego opisu języka. Artykuł ma zadanie przybliżyć Icon polskiemu Czytelnikowi, zanim ukaże się jakaś książka na ten temat, np. [2]. Jego treść podzielono na trzy części: w pierwszej omówiono ogólną charakterystykę języka, jego składnię i struktury sterowania. Część druga zawierać będzie przegląd wbudowanych funkcji i operatorów oraz struktur danych — standardowych i konstruowanych w programie. Część ta będzie poświęcona także generatorom — mechanizmowi, dzięki któremu w Iconie w prosty i wygodny sposób programuje się algorytmy z nawrotami (ang. backtracking). Implementacja tych algorytmów przy użyciu tradycyjnych języków programowania bywa dość uciążliwa. W części trzeciej będą omówione podstawowe techniki stosowania generatorów. Generatory stanowią bardzo silne narzędzie, zdaniem autora artykułu, wygodniejsze niż deklaratywne konstrukcje PROLOGU, zwłaszcza dla osób przyzwyczajonych do klasycznych, proceduralnych języków programowania. W tej części zostanie dokończone omawianie elementów języka, podane kilka przykładów oraz poruszone niektóre zagadnienia związane z implementacją Iconu, które stanowią interesujący wkład do informatyki.

Opis języka będzie dość nieformalny. W celu uniknięcia nadmiernej rozwlekłości przy omawianiu konstrukcji języka, autor często będzie posługiwał się intuicją oraz analogią z innymi językami programowania, nawiązując także do Snobolu. Nie powinno to nadmiernie przeszkodzić Czytelnikom nie znającym Snobolu, natomiast znajomość tego języka może ułatwić zrozumienie artykułu.

OGÓLNA CHARAKTERYSTYKA JĘZYKA

Icon jest językiem proceduralnym, zewnętrznie dość podobnym do Algolu, Pascala czy C. Postać zmiennych (identyfikatorów) oraz stałych liczbowych całkowitych i rzeczywistych jest taka sama jak w innych językach. Podobnie jest z operatorami arytmetycznymi i relacyjnymi. Aczkolwiek Icon operuje wieloma typami danych, kontrola typów następuje dynamicznie, a zmiennych na ogół się nie deklaruje. Wartością niezdefiniowanej zmiennej jest wartość specjalnego słowa zastrzeżonego **&null**, która jest traktowana jako nielegalny argument większości operacji, ale którą można przypisywać i porównywać. Oprócz **&null**, Icon zawiera jeszcze wiele innych obiektów, których nazwy rozpoczynają się od znaku **"&"**. Pełnią one rolę wbudowanych stałych, zmiennych o specjalnym przeznaczeniu, oraz — specjalnych procedur systemowych.

Program w Iconie składa się z procedur, których składnię omówiono poniżej. Wszystkie procedury definiuje się na tym samym poziomie, nie ma procedur statycznie zanurzonych w innych. Program główny jest procedurą o wyróżnionej nazwie **main**. Poza procedurami, w programie występują jeszcze deklaracje zmiennych globalnych, które mają postać:

global <lista zmiennych>

Lista zmiennych zawiera identyfikatory oddzielone przecinkami. Nie jest to deklaracja typów, gdyż zmienne globalne, tak jak i lokalne, mogą w programie wielokrotnie przyjmować wartości różnych typów.

Symbol **#** jest traktowany jako początek komentarza i jest ignorowany przez procesor języka razem z pozostałymi znakami w wierszu.

WYRAŻENIA

Podstawową jednostką składniową w Iconie jest wyrażenie — podobnie jak w Lispie lub Algolu 68. Nie ma oddzielnego pojęcia instrukcji, np. znana z wielu języków programowania instrukcja przypisania:

ALFA := 5

jest w Iconie wyrażeniem, którego wartością jest 5, a nadanie tej wartości zmiennej ALFA jest ubocznym skutkiem ewaluacji (obliczenia) tego wyrażenia. Umożliwia to pisanie wielokrotnych instrukcji przypisania i ogólniej — pisanie bardzo zwartych, a jednocześnie skomplikowanych konstrukcji, gdyż dowolnie rozbudowana struktura składniowa może być elementem większej. Oczywiście nadużywanie tej możliwości czyni program nieczytelnym.

Wyrażenia można oddzielać średnikami lub znakami końca wiersza, podobnie jak w Snobolu. Grupowania wyrażeń w wyrażenie złożone, ewaluowane sekwencyjnie, podobnie jak instrukcje złożone w innych językach, dokonuje się przy użyciu nawiasów klamrowych {i}

Wartością wyrażenia może być liczba, ciąg znaków (ang. string), tablica lub lista jako całość, albo inna złożona struktura danych (które omówiono w dalszej części artykułu), a także — odniesienie do zmiennej, a nawet do procedury. Można więc użyć jako argumentu, nazwy wbudowanej procedury drukującej, np.

pisz := write

i odtąd zamiast:

write (...)

wywoływać ją przez:

pisz (...)

Odróżnienie odniesienia do procedury od jej wywołania jest czysto składniowe — każde wywołanie musi zawierać listę (choćby pustą) argumentów w nawiasach, np. wyrażenie **read** powoduje odczytanie jednego wiersza znaków z pliku wejściowego i dostarczenie go.

Odniesienie do zmiennej, znajdującej się po prawej stronie operacji przypisania lub w innych podobnych kontekstach, służy do pobrania wartości zmiennej. Operacja ta nosi w języku angielskim nazwę dereferencing, która nie ma dobrego odpowiednika polskiego, choć proponowano różne tłumaczenia, np. wyłuskanie [3]. W Iconie odróżnienie wartości zmiennej od samej zmiennej, tj. odniesienia do niej, jest bardzo ważne i bywa czasami dość kłopotliwe. Aczkolwiek wszystkie argumenty procedur są przekazywane przez wartość, jeśli argumenty są zmiennymi lub wyrażeniami dostarczającymi odniesień do zmiennych, to najpierw wszystkie argumenty ewaluuje się do odpowiednich odniesień, a potem dopiero wyłuskuje wartości, co w przypadku nieostrożnego programowania może być źródłem trudnych do wykrycia błędów. Typową operacją, której wynik jest odniesieniem do zmiennej, jest pożyteczna operacja wymiany wartości dwóch zmiennych:

ALFA :=: BETA

Wynikiem tego wyrażenia jest odniesienie do zmiennej po lewej stronie operatora, który podobnie jak operator przypisania łączy do prawej strony, tj.:

A :=: B :=: C

jest równoważne

A:=: (B :=: C)

Czytelnik może prześledzić skutek tej złożonej operacji.

POWODZENIE I NIEPOWODZENIE EWALUACJI WYRAŻENIA

Główną cechą języka odziedziczoną po Snobolu jest charakterystyczny mechanizm powodzenia i niepowodzenia ewaluacji wyrażenia — podstawowy mechanizm służący podejmowaniu decyzji w programie. W Iconie nie ma pojęcia wartości logicznej, prawdy lub fałszu, ani osobnego typu danych boolowskich. Nie są też one, jak np. w Basicu, symulowane przez **0** i **1** lub inną liczbę różną od zera. W Iconie ewaluacja pewnego wyrażenia, np. relacji **ALFA >= 5** może zakończyć się niepowodzeniem lub krócej — zawieść, co w tym kontekście odgrywa rolę fałszu.

Niepowodzenia nie należy jednak traktować jako specyficznej wartości, raczej jako odmowę dostarczenia wartości. Niepowodzenie przenosi się (z pewnymi zastrzeżeniami) aż do granic jednostki składniowej programu, w której zostało wygenerowane. Jeśli powyższa przykładowa relacja jest jednym z argumentów procedury, to dalsze argumenty nie są obliczane i całe wywołanie procedury też zawodzi. Jeśli to wywołanie jest fragmentem większego wyrażenia, to jego ewaluacja ulega również przerwaniu i kończy się niepowodzeniem. Niepowodzenie można oczywiście kontrolować i neutralizować. Używa się go jako warunku logicznego, np. w wyrażeniu

if ALFA >= 5 **then** BETA := 5 **else** BETA := ALFA

niepowodzenie relacji nie jest krytyczne i zgodnie z oczekiwaniami spowoduje ewaluację członu **else**. Powyższe wyrażenie można napisać podobnie jak w Algolu:

BETA := **if** ALFA >= 5 **then** 5 **else** ALFA

jednak w odróżnieniu od Algolu, Icon dopuszcza zapisanie wyrażenia warunkowego bez członu **else**, gdyż wiadomo jak postąpić, gdy warunek logiczny nie jest spełniony — należy przenieść niepowodzenie i nie wykonać operacji przypisania.

Niepowodzenie można wygenerować explicite nakazując w programie ewaluację słowa zastrzeżonego **&fail**. Niepowodzenie służy w Iconie do sterowania wykonaniem pętli. Jest generowane w takich kontekstach jak: próba czytania danych z pliku poza jego końcem lub — próba odwołania do elementu tablicy poza jej granicami. Umożliwia to także pisanie bardzo zwartych konstrukcji iteracyjnych, w których następuje przejście całego pliku lub całej tablicy, a programista nie musi martwić się jawnym ubezpieczeniem pętli na wypadek wystąpienia końca.

Główną wadą mechanizmu powodzenia i niepowodzenia jest zatarcie granicy między sytuacją normalną w programie a wyjątkową, traktowaną przez inne języki programowania jako błąd.

PROCEDURY

Deklaracja procedury składa się z nagłówka zawierającego słowo **procedure**, po którym następuje nazwa procedury i lista parametrów w nawiasach okrągłych. Po nagłówku występują opcjonalne deklaracje zmiennych lokalnych, następnie (również opcjonalna) klauzula inicjująca i ciało procedury, które jest wyrażeniem lub ciągiem wyrażeń. Procedura kończy się słowem **end**. Poniżej podano przykład prostej procedury obliczającej i-tą liczbę ciągu Fibonacciego:

```
procedure Fib(i)
  if i=1 then return 1
  if i=2 then return 1
  return Fib(i—1) + Fib(i—2)
end
```

Dozwolona jest pełna rekursja, również pośrednia, bez konieczności deklaracji z wyprzedzeniem (ang. forward), znanej z Pascala. Wywołanie procedury dostarcza wartości wyrażenia będącego argumentem słowa zastrzeżonego **return**, które można traktować jako jednoargumentowy (prefiksowy) operator o niskim poziomie priorytetu. Jeśli to wyrażenie zawiedzie, to zawiedzie również wywołanie procedury. Niepowodzenie wywołania można też wymóc jawnie pisząc:

return &fail

albo prościej — słowo zastrzeżone **fail**.

Można też dopuścić, aby ewaluacja ciała procedury nigdy nie napotkała **return**. Bezargumentowe wyrażenie **return** dostarcza wartości **&null**.

Wszystkie argumenty są przekazywane przez wartość. Nie ma innych sposobów przekazywania argumentów, jednak należy pamiętać, że jeśli argumentem jest struktura, np. tablica, to procedurze przekaże odniesienie do tej struktury. Skutkiem ubocznym działania procedury może być więc modyfikacja argumentu. Ponadto, o czym wspomniano powyżej, przekazywanie procedurze wartości argumentów odbywa się w specyficznej kolejności. Przykładowo, procedura o nagłówku

proc(x,y)

wywołana następująco:

proc(ALFA,ALFA:=5)

przypisze obu swoim parametrom x i y wartość 5 niezależnie od wartości ALFA przed wywołaniem.

Lokalne zmienne w procedurze można — np. dla celów dokumentacyjnych — zadeklarować jawnie:

local <lista identyfikatorów>

lub równoważnie

dynamic <lista identyfikatorów>

Te deklaracje są opcjonalne, niezadeklarowane zmienne są automatycznie traktowane jako lokalne i dynamiczne. Ich wartości podczas rekursywnych wywołań są pamiętane na stosie. Parametry procedury są również traktowane jako zmienne lokalne. Drugi sposób deklaracji jest następujący:

static <lista identyfikatorów>

W ten sposób deklaruje się zmienne statyczne, o wartościach zachowywanych między wywołaniami procedury, podobnie jak zmienne **own** w Algolu 60 lub zmienne lokalne w większości implementacji Fortranu.

Klauzula inicjująca ma postać:

initial <wyrażenie>

i pełni rolę wyrażenia, które jest ewaluowane tylko raz, podczas pierwszego wywołania procedury. Jako przykład wykorzystania zmiennych statycznych i klauzuli inicjującej przedstawiono modyfikację procedury **Fib**, która w pierwszej wersji jest praktycznie bezużyteczna, gdyż charakteryzuje się wykładniczą złożonością ze wzrostem **i**. Można ją zoptymalizować przez przechowanie raz obliczonych wartości w specjalnej tablicy. Tablica ta, o nazwie **fibtab**, będzie — co jest charakterystyczne dla Iconu — rozszerzana w miarę potrzeby przez dołączanie na jej końcu nowych elementów.

```
procedure Fib(i)
  static fibtab
  local j
  initial fibtab := [1,1] #dwuelementowy wektor
  if j := fibtab[i] then return j
     fibtab := fibtab ||| [j := Fib(i—2) + Fib(i—1)]
     return j
end
```

Operator ||| jest operatorem spinania (konkatenacji). Dołącza on do tablicy jednoelementową tablicę zawierającą obliczoną liczbę. Obliczenie to następuje, jeśli wskutek próby pobrania wartości nieistniejącego elementu tablicy wyrażenie **if** zawiodło. Czytelnik może sprawdzić, czy zamiana kolejności rekursywnych wywołań **Fib** wywoła skutki negatywne czy nie.

Wartością typowo dostarczoną przez wywołanie procedury jest liczba, ciąg znaków, odniesienie do tablicy itp. Może być nią jednak odniesienie do zmiennej, tak że wywołanie procedury może znaleźć się po lewej stronie operacji przypisania! Jeżeli, na przykład, procedura **maxel(x)** oblicza wartość największego elementu tablicy **x**:

```
procedure maxel(x)
   ...
   return x[i] #gdzie i — indeks największego elementu
end
```

to dopuszczalna jest konstrukcja:
maxel(a) := 0

której wynikiem jest wyzerowanie odpowiedniego elementu tablicy **a**. Ogólnie — wywołanie procedury dostarcza odniesienia do zmiennej, jeśli argumentem **return** jest zmienna globalna lub element tablicy czy innej struktury złożonej.

PODSTAWOWE STRUKTURY STEROWANIA

O ile w Snobolu jedyną strukturą sterowania są skoki (pomijające automatyczne nawroty podczas operacji porównania z wzorem oraz — wywołania procedur), to w Iconie skoków nie ma w ogóle!

Wyrażenie warunkowe **if ... then ... else** przedstawiono powyżej. Ze względu na dynamiczny charakter powodzenia i niepowodzenia operacji w Iconie, konstrukcja:

not E

gdzie **E** jest wyrażeniem, zalicza się raczej do struktur sterowania niż do operacji logicznych. Jej sens jest dość oczywisty — zawodzi, gdy ewaluacja **E** zakończy się sukcesem. W przeciwnym wypadku dostarcza wartości **&null**.

Dynamiczny charakter ma również konstrukcja koniunkcji logicznej

E1 & E2 & E3 ... & En

Wyrażenie to jest ewaluowane podobnie jak **AND** w Lispie — oblicza się kolejno **E1, E2** itd., aż do momentu pierwszego niepowodzenia, które kończy ewaluację całości, lub — do zakończenia ewaluacji **En**, którego wartość jest dostarczana.

Nieco wygodniejszym, równoważnym zapisem powyższego wyrażenia jest tzw. wyrażenie łączne

(E1,E2, ... , En)

Jeśli konieczna jest łączna ewaluacja kilku wyrażeń, ale interesująca wartość jest dostarczana przez jedno z wyrażeń środkowych, to można użyć konstrukcji postaci:

E(E1, E2, ... , En)

gdzie **E** jest wyrażeniem dostarczającym liczbę całkowitą. Jeśli ewaluacja wszystkich **Ei** zakończyła się powodzeniem, a wartością **E** jest **i**, to wynikiem całości będzie wartość **i-tego** wyrażenia w nawiasach. Liczba **i** może być ujemna, wtedy obowiązuje konwencja odliczania od końca, np. dla **i** równego - 1 dostarczana jest wartość przedostatniego wyrażenia.

W Iconie występuje duża różnorodność pętli, np.:

while E1 do E2

oraz komplementarna:

until E1 do E2

W obu wypadkach warunek **E1** jest ewaluowany zanim nastąpi ewentualna ewaluacja **E2**. W pierwszym wypadku warunkiem przerwania pętli jest niepowodzenie ewaluacji **E1**, w drugim — powodzenie. Zwrot do **E2** jest opcjonalny, gdyż można również napisać:

while write (read())

co powoduje skopiowanie pliku wejściowego. Pętla ulegnie przerwaniu, gdy zawiedzie read(), co nastąpi, gdy skończą się dane. Niepowodzenie przenosi się na **write**, a następnie przerywa **while.**

Jeśli umieszczenie sprawdzanego warunku na początku pętli nie jest odpowiednie, w Iconie można użyć pętli:

repeat E

która ewaluuje wyrażenie **E**, dopóki nie zostanie jawnie przerwana z wnętrza. Mechanizmem przerywającym pętlę **repeat**, a także i inne pętle niezależnie od automatycznie sprawdzanych warunków jest słowo **break.**

Wyrażenie:

break E

przerywa jedną, najbardziej wewnętrzną pętlę, w której się znajduje i dostarcza wartości **E** na wyższy poziom, do pętli obejmującej. Argumentem **break** może być następne wyrażenie **break**, co umożliwia opuszczenie od razu dwóch pętli.

Ostatnią omówioną w tej części strukturą jest wyrażenie wyboru:

```
case E of {
     <lista wyboru>
     ....
     } .
```

Element listy wyboru jest parą wyrażeń oddzielonych dwukropkiem **E1 : E2.**

Elementy są oddzielane średnikami lub znakami zmiany wiersza. Wartością dostarczaną jest wartość jednego z wyrażeń **E2.** Ewaluowane jest to wyrażenie **E2**, które odpowiada **E1** równemu wartością aktualnemu **E**. Jeśli wartość żadnego **E1** nie jest równa wartości **E**, to wyrażenie **case** zawodzi. Możliwy jest jednak jeszcze opcjonalny zwrot:

default : E2

który jest ewaluowany, jeśli wszystkie inne możliwości zawiodły.

Należy zauważyć, że nie omówiono dotąd odpowiednika pętli **for.** Taka pętla istnieje w Iconie, ale jej semantyka jest na tyle różna od **for** w Algolu lub Pascalu, że zostanie ona omówiona w kolejnej części artykułu, razem z generatorami. Okaże się wówczas, że konstrukcje omówione powyżej nie są tak proste i kryją w sobie sporo niespodzianek.

LITERATURA

[1] Gizbert-Studnicki P., Karczmarczuk J.: Snobol 4. WNT, Warszawa, 1984

[2] Griswold R. E., Griswold M. T.: The Icon Programming Language. Prentice-Hall, Englewood Cliffs (NJ), 1983

[3] Małuszyński J., Piasecki K.: Wprowadzenie do języka ALGOL 68; ALGOL 68. WNT, Warszawa, 1980.

JAN BIELECKI
Instytut Informatyki
Politechnika Warszawska

Zaawansowane konstrukcje języka C

Doświadczenia kilku ostatnich lat dobitnie wykazują, iż język C staje się głównym językiem programowania w zastosowaniach mikrokomputerowych. Wynika to w głównej mierze z łatwości programowania w nim, przenośności oprogramowania i wysokiej jakości kodu generowanego przez najlepsze kompilatory tego języka.

Chociaż pisanie prostych programów w języku C nie sprawia specjalnych trudności, posługiwanie się jego pełnymi możliwościami wymaga dogłębnej znajomości mechanizmów języka.

WYRAŻENIA WSKAZUJĄCE

Podobnie jak w wielu językach programowania, podstawowym pojęciem języka C jest pojęcie danej — obiektu, który jest przedmiotem przetwarzania. Wśród danych przetwarzanych przez program, szczególną rolę odgrywają dane wskazujące. Są to dane, których wartości reprezentują wskazania na inne dane. Znaczenie danych wskazujących (niekiedy nazywanych wskaźnikowymi) jest oczywiste dla wszystkich, którzy zetknęli się z programowaniem systemowym.

Dane są w programach reprezentowane przez wyrażenia, obejmujące także nazwy zmiennych i literały. Dane wskazujące są reprezentowane przez wyrażenia wskazujące. Szczególnym rodzajem wyrażeń wskazujących są tzw. l-wyrażenia, tj. takie wyrażenia reprezentujące zmienne, których składnia wynika z definicji:

```
l-wyrazenie:
      identyfikator
      wyrazenie-pierwotne [ wyrazenie ]
      l-wyrazenie-pierwotne . identyfikator
      wyrazenie-pierwotne -> identyfikator
      * wyrazenie
      ( l-wyrazenie )

wyrazenie-pierwotne:
      identyfikator
      literal
      ( wyrazenie )
      wyrazenie-pierwotne ( lista-wyrazen )
      wyrazenie-pierwotne [ wyrazenie ]
      l-wyrazenie-pierwotne . identyfikator
      wyrazenie-pierwotne -> identyfikator

wyrazenie:
      wyrazenie-pierwotne
      * wyrazenie
      & l-wyrazenie
      - wyrazenie
      ! wyrazenie
      ~ wyrazenie
      ++ l-wyrazenie
      -- l-wyrazenie
      l-wyrazenie ++
      l-wyrazenie --
      sizeof wyrazenie
      ( nazwa typu ) wyrazenie
      wyrazenie operator-dwuargumentowy wyrazenie
      wyrazenie ? wyrazenie : wyrazenie
      l-wyrazenie operator-przypisania wyrazenie
      wyrazenie , wyrazenie

operator-dwuargumentowy:
      *  /  %
      +  -
      >>  <<
      <  >  <=  >=
      ==  !=
      &
      ^
      |
      &&
      ||

operator-przypisania:
      =  +=  -=  /=  %=  >>=  <<=  &=  ^=  |=

char arr[2][2] = { "j","b" },
     (*ref)[2] = arr + 1;
```

Nazwy tablic występujące w wyrażeniach są traktowane specjalnie. Jeśli arr jest wyrażeniem reprezentującym tablicę, to jest ono niejawnie przekształcane w wyrażenie **&arr[0]**, a więc staje się wyrażeniem wskazującym, ale nie jest l-wyrażeniem. Przede wszystkim z tego względu wyrażenia reprezentujące tablice nie mogą wystąpić w znaczeniu lewej strony operatora przypisania. Fakt ten musi być uwzględniony podczas interpretowania przytoczonych reguł składniowych, jako że

l-wyrażenie-pierwotne . identyfikator

oraz

wyrażenie-pierwotne —> identyfikator

są l-wyrażeniami tylko wtedy, gdy nie reprezentują tablic ani pól rekordu. Na uwagę zasługuje także fakt, że argumentem operatora "&" musi być l-wyrażenie. Na skutek pomyłki, wymagania tego nie uwzględniono w [3]. Z tych samych powodów nie uwzględniono też wymagania, aby przed operatorem "." występowało l-wyrażenie-pierwotne. Oba te uchybienia naprawiono w przytoczonym tu opisie. Nie występują one także w lepszych kompilatorach języka (Lattice/Microsoft, Mark Williams, Aztec).

KONWERSJE

Odrębny, ważny element języka C stanowią konwersje. Występują one w dwóch postaciach: jako konwersje jawne, zadawane za pomocą operatora konwersji:

(nazwa-typu)

oraz jako konwersje niejawne, występujące np. podczas wykonywania instrukcji **return** zawierającej wyrażenie. Ponieważ konwersje niejawne stanowią jedynie pewne uproszczenie zapisu programu i mogą być zawsze zastąpione konwersjami jawnymi, wystarczy ograniczyć się do omówienia tych ostatnich. Aby to zadanie ułatwić, przytoczymy składnię operatora konwersji:

```
operator-konwersji:
   (nazwa-typu)
nazwa-typu:
   oznaczenie-typu pseudodeklarator
oznaczenie typu:
   char
   int
   short int
   long int
   unsigned int
   float
   double
   long float
   opis-struktury-lub-unii
   identyfikator-typu
opis-struktury-lub-unii:
   struct-lub-union { wykaz-komponentów }
   struct-lub-union nazewnik { wykaz-komponentów }
   struct-lub-union nazewnik
struct-lub-union:
   struct
   union
nazewnik:
   identyfikator
wykaz-komponentów:
   oznaczenie-typu lista-deklaratorów-komponentów;
deklarator-komponentu:
   deklarator
   deklarator : wyrażenie-stałe
   : wyrażenie-stałe
deklarator:
   identyfikator
```

(deklarator)
deklarator [wyrażenie-stałe]
* deklarator
deklarator ( )

WYZNACZANIE WARTOŚCI WYRAŻEŃ

Omówiwszy zasady konstruowania wyrażeń w języku C, można przystąpić do przedstawienia zasad wyznaczania ich wartości. Zasady te w głównej mierze wynikają z priorytetów przypisywanych poszczególnym operatorom języka. Na podkreślenie zasługuje fakt, że priorytety i wiązania operatorów, wymienione w tabeli, służą jedynie do określenia zasad interpretowania wyrażeń języka, ale nie określają porządku, w jakim są wyznaczane wartości elementów wyrażenia.

Priorytet	Wiązanie	Operator
15	lewe	() [] -> .
14	prawe	! ~ ++ -- - (typ) * & sizeof
13	lewe	* / %
12	lewe	+ -
11	lewe	<< >>
10	lewe	< <= > >=
9	lewe	== !=
8	lewe	&
7	lewe	^
6	lewe	|
5	lewe	&&
4	lewe	||
3	prawe	?:
2	prawe	= += -= *= /= %= >>= <<= &= ^= |=
1	lewe	,

Dla skonkretyzowania tego problemu rozpatrzymy wyrażenie arytmetyczne

a — b + c * d

Ponieważ priorytet mnożenia jest wyższy od priorytetów dodawania i odejmowania, wyrażenie to jest interpretowane tak jak wyrażenie

a — b + (c * d)

a wobec tego, iż operatory "+" i "—" wiążą argumenty od lewej do prawej, jest ono interpretowane tak jak wyrażenie

(a — b) + (c * d)

a nie jak wyrażenie

a — (b + (c * d)

co nastąpiłoby, gdyby "+" i "—" wiązały argumenty od prawej do lewej.

Mimo iż rozpatrywane wyrażenie jest interpretowane tak jak wyrażenie

(a — b) + (c * d)

nie można wnioskować o kolejności wyznaczania wartości argumentów dodawania, gdyż jest ona pozostawiona do wyboru w implementacji. Ma to miejsce nawet wtedy, gdy pociąga za sobą skutki uboczne.

W języku C zezwolono również na matematycznie równoważne przekształcenia argumentów operacji łącznych i przemiennych, nawet jeśli wymagałoby to innego rozmieszczenia lub eliminacji nawiasów. Fakt ten należy mieć na uwadze podczas opracowywania programów przenośnych.

PRZYKŁADY PROGRAMÓW

Po tych wyjaśnieniach można przystąpić do omówienia przykładowych programów.

Program i01

```
/*** i01 ***/

int var;
main(){
  printf("%s", (--var, var += ++var) + "jb" +1);
}
```

Zmienna **var**, jako zmienna zewnętrzna, otrzymuje przez domniemanie wartość początkową **0**. Wynikiem operacji — — **var** jest —1, a wynikiem przypisanie —1 lub **0**. Ten ostatni rezultat wynika z faktu, iż przypisanie

var += ++var

traktowane jak

var = var + (++var)

może być w języku C interpretowane na jeden z dwóch sposobów, a mianowicie

temp = var, var = temp + (++var)

albo

temp = ++var, var = var + temp

Niejednoznaczność przytoczonego przypisania powoduje, że jest ono konstrukcją nieprzenośną, a więc należy go unikać.

Jeśli przyjąć, że w pewnej implementacji (np. Lattice/Microsoft, Aztec) rezultatem wyrażenia w nawiasach jest **0**, to drugim argumentem funkcji **printf** jest **"jb" + 1**. Ponieważ literał **"jb"** reprezentuje daną typu **(char [3])** i zostaje niejawnie przekształcony w **"jb"** typu **(char *)**, wspomniany argument reprezentuje daną wskazującą literę **b**, a wynikiem wykonania programu jest napis składający się z tej litery. Jak wynika z uprzednich wyjaśnień, poprawna jest także implementacja, w której wynikiem wykonania programu jest napis **jb**.

Program i02

```
/*** i02 ***/

char fun();

main(){
  printf("%d", fun());
}

char fun()
{return 259;}
```

W implementacji zgodnej z [3] użyta tu instrukcja **return** powinna być traktowana jak instrukcja

return (char)259;

a wynikiem wykonania programu powinna być liczba 3. W szeregu implementacji (m.in. Lattice/Microsoft, ale nie Aztec), deklarator funkcji **fun** jest traktowany tak, jakby miał postać

int fun( );

a wynikiem wykonania programu jest liczba 259.

Program i03

```
/*** i03 ***/

main(){
  printf("%c", 'jb');
}
```

W standardzie języka C wypowiedziano się niejednoznacznie na temat poprawności literału znakowego, składającego się z więcej niż jednego znaku. W punkcie 2, 4, 3 zawarto zakaz wystąpienia więcej niż jednego znaku, natomiast w punkcie 16 decyzje o poprawności i sposobie interpretowania takiej konstrukcji pozostawiono do wyboru implementacji.

W implementacji Lattice/Microsoft literał **'jb'** jest poprawny i traktowany tak, jakby był wyrażeniem **'j'<<8+'b'** typu **(int)**, co powoduje, że wynikiem wykonania rozważanego programu jest napis składający się z litery **b**. W implementacji Aztec wynikiem jest napis składający się z litery **j**.

Program i04

Ponieważ w języku C każdy wektor wektorów jest uznawany za tablicę, zmienna **ref** jest tego samego typu co **ptr**, a każda z tych zmiennych wskazuje tablicę dwuwymiaro-

```
/***  i04  ***/

char (*ptr)[3][4],
     ((*ref)[3])[4];

main(){
  ptr = 0;
  ref = ptr;
  printf("%d,%d", (int)++ptr,(int)++ref);
}
```

wą typu **(char[3][4]**. Z tego powodu wynikiem wykonania programu jest wyprowadzenie pary liczb 12, 12. Rozwiązanie to, przyjęte w większości implementacji, m.in. przez Lattice/ /Microsoft oraz Aztec, budzi jednak pewne zastrzeżenia, ponieważ sensowniej byłoby przyjąć, że to właśnie każda tablica dwuwymiarowa jest wektorem wektorów, a wtedy wynikiem wykonania programu byłoby wyprowadzenie pary liczb 4, 4.

Program i05

```
/***  i05  ***/

#define s 3

char arr[s][s][s] = { "jb" },
     *fun();

main(){
  printf("%s", fun(arr));
}

char *
fun(par)
 char (*par)[s][s];
{ return *par; }
```

Argument funkcji **fun** zostaje niejawnie przekształcony na **(&arr[0])**, a więc reprezentuje dwuwymiarową tablicę **arr[0]**. Zgodnie z zasadami kojarzenia parametrów funkcji z jej argumentami, również **par** reprezentuje wskazanie na tę tablicę, a ***par** reprezentuje samą tablicę.

Ponieważ ***par** reprezentuje tablicę **arr[0]**, zostaje niejawnie przekształcone na **&arr[0][0]**, reprezentujące wskazanie na wektor **arr[0][0] typu (char [3])**. W ramach wykonania instrukcji **return** traktowanej tak, jakby miała postać

return (char *)*par;

wskazanie to zostaje poddane konwersji **(char *)** i tym samym reprezentuje wskazanie na pierwszą literę ciągu **jb**. Rezultatem wykonania programu jest napis **jb**.

Należy nadmienić, iż w pewnych implementacjach (m.in. Aztec, ale nie Lattice/Microsoft) wymaga się, aby w przytoczonym programie domniemana konwersja **(char *)** została użyta jawnie.

Program i06

```
/***  i06  ***/

char var = 'b';

main(){
  char ins = var,
       var = 'j';
  printf("%c%c", var,ins);
}
```

Ponieważ zasięgiem deklaracji zmiennej zewnętrznej **var** są trzy pierwsze wiersze programu, zmienna **ins** otrzymuje wartość początkową **'b'**. Wynikiem wykonania programu jest napis **jb.**

Program i07

```
/***  i07  ***/

char arr[3][4];

main(){
  printf("%d", (arr[2] - (char *)arr) / sizeof(char));
}
```

Odwołanie **arr[2]** reprezentuje ostatni „wiersz" tablicy **arr** i zostaje niejawnie przekształcone na wskazanie na element **arr[2][0]**. Odwołanie **arr** reprezentuje tablicę **arr** i zostaje niejawnie przekształcone na wskazanie na wektor **arr[0]**, a ono z kolei — poddane konwersji na wskazanie elementu **arr[0][0]**. Wynikiem wykonania programu jest wyprowadzenie liczby 8.

Program i08

```
/***  i08  ***/

char *
sub(p){
  return "jb";
}

int (*fun(p1,p2))(){
  return sub;
}

int (*(*prg())())(){
  return fun;
}

main(){
    printf("%s", (*(*prg())(0,0))(0));
}
```

W ślad za poważnym błędem zawartym w punkcie 10.1, uchodzącego za standard języka opracowania [3], wielu autorów twierdzi, że ciało definicji funkcji następuje bezpośrednio po nawiasie okrągłym zamykającym listę parametrów. Stwierdzenie to jest nieprawdziwe, a przytoczony program poprawny. Wynikiem jego wykonania jest napis **jb** (w implementacji Aztec wymaga się, aby w instrukcji **return** jawnie użyto operatora konwersji).

Program i09

```
/***  i09  ***/

char *(*ref)[3],
     *ptr[4] = { 0, "jan", "bielecki" };

main(){
  ref = (char *(*)[3])ptr;
  printf("%c%c", *(*ref++)[1],
                 *(*--ref)[2]);
}
```

Zmienna **ref** wskazuje trzyelementową tablicę wskazań na dane typu **(char)**. Po wykonaniu przypisania, wskazuje ona tablicę częściową składającą się z pierwszych trzech elementów tablicy **ptr**. W tym momencie ***ref++** reprezentuje tablicę częściową i zostaje niejawnie przekształcone na wskazanie na pierwszy element tej tablicy, tj. na wskazanie na **ptr[0]**. Zaindeksowanie tego wskazania indeksem **[1]** powoduje, że wyrażenie **(*ref++)[1]** reprezentuje element **ptr[1]**, zaś ***(*ref++)[1]** reprezentuje literę **j**. Po tych operacjach, zmienna **ref** ma wartość równą wskazaniu na tablicę, której pierwszym elementem jest **ptr[3]**. Program jest wykonywany poprawnie tylko w tych implementacjach (np. Lattice/Microsoft, ale nie Aztec), w których wyznaczenie wartości drugiego argumentu funkcji **printf** następuje przed wyznaczeniem wartości trzeciego argumentu. Wynikiem wykonania programu jest wówczas napis **jb.**

Program i10

```
/***  i10  ***/

#define put(par) printf("%c",par)

char *(*ref)[2],
     *ptr[3] = { "satan", 0, "dracula" + 2 };

main(){
  ref = (char *(*)[2])ptr;
  put(* (*++ref)[0] --);
  put(***ref);
  put(* (*ref--)[0] ++);
  put(**ref[1]);
  put(***ref);
}
```

Zmienna **ref** wskazuje dwuelementową tablicę wskazań na dane typu **(char)**. Po wykonaniu przypisania wskazuje ona tablicę składającą się z dwóch pierwszych elementów tablicy **ptr**, a po wykonaniu preinkrementacji wskazuje tablicę składającą się z **ptr[2]** i **ptr[3]**. Z tego powodu **(*++ref)[0]** reprezentuje element **ptr[2]**, a argument pierwszego wywołania makrodefinicji **put** reprezentuje literę **a** ciągu **dracula.** W tym momencie, a więc po postdekrementacji, element **ptr[2]** wskazuje literę **r** tego samego ciągu. Operację ***ref** można przedstawić w postaci równoważnej ***(*ref)[0]**. Wynika stąd, że argumentem drugiego wywołania makrodefinicji **put** jest litera **r.** Wynikiem wykonania programu jest napis **arras.**

Program i11

```
/***   i11   ***/

typedef char Vec[2];
Vec arr[2] = { "j","b" },
    *ref = arr + 1;

main(){
  printf("%s%s", *arr,ref);
}
```

Zgodnie z definicją, **Vec** identyfikuje tablicę typu (**char [2]**), a zatem **arr** jest dwuelementową tablicą, której elementami są dane typu (**char [2]**). Analogicznie, **ref** jest zmienną wskazującą dane typu (**char [2]**). Z tego powodu, jawne deklaracje **arr** i **ref** można przedstawić w sposób równoważny jako

```
char arr[2][2] = { "j","b" },
     (*ref)[2] = arr + 1;
```

Wynikiem wykonania programu jest napis **jb**.

Program i12

```
/***   i12   ***/

main(){
  char (*ptr)[3] = (char (*)[3])"eb";
  printf("%s", (fun(), ptr));
}

fun(arr)
 char arr[2][3];
{ **arr = 'j'; }
```

Parametr funkcji **fun** jest traktowany tak, jakby miał deklarację

```
char arr[ ] [[3];
```

równoważną deklaracji

```
char (*arr)[3];
```

Ponieważ **arr można przedstawić jako **(*arr)[0]**, łatwo wywnioskować, że wykonanie instrukcji przypisania powoduje zmianę wartości pierwszego znaku danej, reprezentowanej przez literał "eb". Wynikiem wykonania programu jest napis **jb**.

ZBIGNIEW SZKARADNIK
Instytut Informatyki
Politechnika Śląska
Gliwice

LITERATURA

[1] Bielecki J. A.: Język C. WNT, Warszawa, 1986 (w druku)
[2] Harbison S. P., Steele G. L.: C. Reference Manual. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1984
[3] Kernigham B. W., Ritchie D. M.: The C programming language. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1978.

Nie mogę wyobrazić sobie lepszego języka do pisania programów, wyrażania algorytmów i zrozumienia komputerów

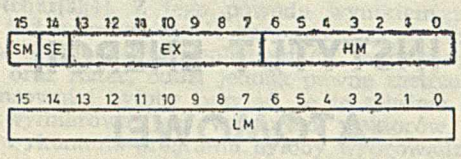
Charles H. Moore

Operacje zmiennoprzecinkowe w języku FORTH

Ideą twórców języka Forth było stworzenie prostego narzędzia, które w razie potrzeby można by łatwo rozbudować. Uznali oni operacje zmiennoprzecinkowe za niepotrzebne, czego wyrazem był ich konsekwentny brak w kolejnych standardach języka. Istotnie, w bardzo wielu zastosowaniach operacje zmiennoprzecinkowe są niepotrzebne, zwykle też mogą zostać zastąpione przez operacje na odpowiednio długich liczbach stałoprzecinkowych. Standard języka Forth przewiduje liczby całkowite pojedynczej i podwójnej długości (32 bity). Jeżeli ich zakres okaże się niewystarczający, wtedy można, korzystając z rozszerzalności języka, wzbogacić go o operacje zmiennoprzecinkowe lub operacje na liczbach stałoprzecinkowych odpowiedniej długości. W praktyce, większość firmowych implementacji języka jest lub może być wyposażona w pakiet operacji zmiennoprzecinkowych.

W niniejszym artykule postaram się udzielić odpowiedzi na pytanie, jak wyposażyć własny translator Fortha w taki pakiet. Posłużę się przykładem implementacji dla mikrokomputera Mera 60 (LSI-11), którego postać słowa zmiennoprzecinkowego przedstawiono na rysunku.

Istnieją dwie zasadnicze metody rozwiązania tego zagadnienia. Pierwsza polega na napisaniu pakietu operacji zmiennoprzecinkowych w języku asemblera, druga zaś na napisaniu całości w języku Forth [2]. Obydwie metody mają swoje wady i zalety. Poniżej wybrano metodę pośrednią. Część programów napisano w języku asemblera, a część w języku Forth, starając się maksymalnie wykorzystać już istniejące oprogramowanie.



Format liczby zmiennoprzecinkowej

SM — znak mantysy (1 bit, 0 znaczy plus, 1 minus)
SE — znak wykładnika (1 bit, 1 znaczy plus, 0 minus)
EX — wykładnik (7 bitów)
HM — bardziej znacząca część mantysy (7 bitów + 1 domyślny bit równy 1)
LM — mniej znacząca część mantysy (16 bitów)
liczba zmiennoprzecinkowa = mantysa*podstawa ↑ wykładnik
*** mnożenie**
↑ potęgowanie

SŁOWNIK

Przed przystąpieniem do pracy należy zdefiniować słownik (zestaw słów Fortha realizujących operacje zmiennoprzecinkowe). Ponieważ nie precyzuje go standard języka, wybór słów należy do programisty. Zaproponowany zestaw słów zmiennoprzecinkowych zamieszczono w tabeli 1. Zebrane w niej słowa można podzielić na grupy słów:
— realizujących operacje arytmetyczne

F+ , F— , F* , F/ i FMINUS

— realizujących operacje stosowe

FDROP , FDUP , FSWAP , FOVER i FROT

— realizujących operacje testowania

FO= , FO< , F= , F< i F>

— deklarujących stałe i zmienne zmiennoprzecinkowe

FCONSTANT i FVARIABLE

— realizujących przesłania między stosem i pamięcią

Fu i F!

— realizujących konwersje między liczbami stało- i zmiennoprzecinkowymi

S—>F i F—>S

— realizujących wprowadzanie liczb zmiennoprzecinkowych

F. , F.R i .F

— realizujących konwersję ciągu znaków na liczbę zmiennoprzecinkową

FLOAT

— pozostałe

FLITERAL , FBASE , FERROR i EXP-CHAR .

Tabela 1. Wykaz słów pakietu zmiennoprzecinkowego

Oznaczenie	Opis działania
Fu	n→fp
F!	fp n→
FDUP	fp → fp fp
FDROP	fp →
FOVER	fp1 fp2 → fp1 fp2 fp1
FSWAP	fp1 fp2 → fp2 fp1
FROT	fp1 fp2 fp3 → fp2 fp3 fp1
F+	fp1 fp2 → fp1+fp2
F—	fp1 fp2 → fp1—fp2
F*	fp1 fp2 → fp1*fp2
F/	fp1 fp2 → fp1/fp2
FMINUS	fp → —fp
S→F	n → fp
F→S	fp → n
FO=	fp → tf, jeśli fp = 0 fp → ff, jeśli fp<> 0
FO<	fp → tf, jeśli fp< 0 fp → ff, jeśli fp> = 0
F=	fp1 fp2 → tf, jeśli fp1 = fp2 fp1 fp2 → ff, jeśli fp1<> fp2
F<	fp1 fp2 → tf, jeśli fp1 < fp2 fp1 fp2 → ff, jeśli fp1 > = fp2
F>	fp1 fp2 → tf, jeśli fp1 > fp2 fp1 fp2 → ff, jeśli fp1 < = fp2
FCONSTANT	fp → , definicja: FCONSTANT fname → fp, wykonanie: fname Deklaracja stałej zmiennoprzecinkowej
FVARIABLE	fp → , definicja: FVARIABLE fname → n, wykonanie: fname Deklaracja zmiennej zmiennoprzecinkowej
FLITERAL	fp → Kompilacja liczby fp; typowe użycie: : cccc ... [...] FLITERAL ... ;
F.	fp → Wyprowadzenie liczby fp w dziesiętnym zapisie wykładniczym. Format wyjściowy jest następujący: spacja, [znak], cyfra, kropka, 6 cyfr, litera 'E', znak, 2 cyfry. Przykładowo, 1.234 ↑ 5 będzie wyprowadzone jako: FLOAT 1.234 ↑ 5 F. <cr> 1.234000E+05 OK
F.R	fp n1 n2 → Wyprowadzenie liczby fp w zapisie dziesiętnym. Format wyjściowy jest zależny od n1, oznaczającego liczbę pozycji po kropce i od n2 określającego szerokość pola wydruku. Format wyjściowy jest następujący: spacja, [znak], cyfry, kropka, n1 cyfr. Całkowita liczba znaków jest równa n2. Jeśli n2 jest ujemne, F.R wyprowadza liczbę w zapisie wykładniczym (jak F.). Przykładowo, 1.234 ↑ 5 będzie wyprowadzone jako: FLOAT 1.234 ↑ 5 2 10 F.R <cr> 123400.00 OK
.F	fp → Wyprowadzenie liczby fp w dowolnym zapisie. Zapis ten jest zdefiniowany zawartością zmiennych BASE i FBASE. Format wyjściowy jest następujący: [znak], cyfra, kropka, 3 cyfry, ogranicznik, [znak], cyfry. Ogranicznik może być zdefiniowany przez użytkownika (patrz:EXP-CHAR). Przykładowo, 1.234 ↑ 5 będzie wyprowadzone jako: 1.234 ↑ 5, w zapisie dziesiętnym 3.610 ↑ 5, w zapisie ósemkowym.
FLOAT	→ fp Pobranie następnego słowa, konwersja na liczbę zmiennoprzecinkową fp oraz umieszczenie jej na stosie. W czasie kompilacji liczba ta jest dodatkowo kompilowana. Słowo pobierane przez FLOAT musi być liczbą zmiennoprzecinkową zapisaną w formacie: [znak], cyfry, kropka, cyfry, ogranicznik, [znak], cyfry. Ogranicznik może być zdefiniowany przez użytkownika (patrz:EXP-CHAR). Słowo to umożliwia konwersje liczb zapisanych w dowolnej notacji. Zdefiniowana jest ona przez zawartość zmiennych BASE i FBASE (analogicznie jak dla .F). Przykładowo: FLOAT 123.4567 ↑ —19, w zapisie dziesiętnym FLOAT —AB.CD ↑ E, w zapisie szesnastkowym.
EXP—CHAR	→ n Zmienna zawierająca ogranicznik rozdzielający mantysę i wykładnik (zawiera znak ,, ↑ ''). Może on zostać zmieniony przez użytkownika.
FBASE	→ n Zmienna przechowująca zmiennoprzecinkową podstawę liczbową. Normalnie (po ABORT) jest ustawiona na 10. Użytkownik może ją zmienić, jeżeli chce prowadzić obliczenia w innych systemach liczbowych.
FERROR	→ n Przesłanie na stos numeru błędu zmiennoprzecinkowego. Jest używane jedynie przez procedurę restartu po błędzie powodującym przerwanie.

fp — liczba zmiennoprzecinkowa, n — słowo 16-bitowe (liczba stałoprzecinkowa lub adres), tf — wartość logiczna TRUE (n=1), ff — wartość logiczna FALSE (n=0)

ZAPIS LICZBOWY

Szczególnej uwagi wymagają operacje we-wy. Na wstępie należało zdecydować, jaki przyjąć rodzaj zapisu liczby. Założono, że liczba zmiennoprzecinkowa będzie występować w zapisie wykładniczym, poprzedzona słowem **FLOAT** dokonującym jej konwersji. Język Forth umożliwia wprowadzenie i wyprowadzenie liczb stałoprzecinkowych przy zastosowaniu dowolnego zapisu liczbowego (dwójkowy, ósemkowy, dziesiętny itd.). Ze względu na spójność języka należało zachować te cechy również w stosunku do liczb zmiennoprzecinkowych.

Zarówno program słowa **FLOAT** przekształcający liczbę w zapisie wykładniczym na postać maszynową, jak i program słowa **.F**, wyprowadzający tę liczbę na terminal, czynią to wykorzystując aktualnie obowiązujący zapis liczbowy, zdefiniowany przez wartość zmiennej **BASE** (przechowującej podstawę stałoprzecinkową) i zmiennej **FBASE** (przechowującej podstawę zmiennoprzecinkową). Można również zdefiniować nowe słowa **FBIN, FDECIMAL** itd., zmieniające aktualną podstawę (wydruk 1).

```
: FBIN 2 S->F FBASE F! 2 BASE ! ;
: FOCTAL 8 S->F FBASE F! OCTAL ;
: FDECIMAL 10 S->F FBASE F! DECIMAL ;
: FHEX 16 S->F FBASE F! HEX ;  ( etc )
```

Wydruk 1. Definicje słów zmieniających podstawę

W związku z dopuszczeniem do stosowania dowolnej podstawy pojawia się problem wynikający z użycia litery **E** w systemie dziesiętnym, jako znaku rozdzielającego mantysę i cechę (ponieważ oznacza ona jednocześnie liczbę 14). Należało więc wybrać jakiś inny symbol. Użyto w tym celu znaku '↑' (strzałka w górę), który jednak może zostać przez użytkownika zmieniony, ponieważ jest przechowywany przez zmienną **EXP-CHAR**.

Wykorzystanie słów **FLOAT** i **.F** zilustrowano na wydruku 2.

```
FLOAT 1.2345678↑8  (cr)
 OK
FDUP FBIN .F  (cr)
1.110↑11010 OK
FDUP FOCTAL .F  (cr)
7.267↑10 OK
FDUP FDECIMAL .F  (cr)
1.234↑8 OK
FDUP FHEX .F  (cr)
7.5BC↑6 OK
```

Wydruk 2. Wykorzystanie słów FLOAT i .F

IMPLEMENTACJA

Napisanie biblioteki operacji zmiennoprzecinkowych w języku asemblera jest czynnością pracochłonną i dość żmudną. Znacznie wygodniejsza jest adaptacja istniejącego oprogramowania, co też zostało zrobione. W celu realizacji czterech podstawowych operacji arytmetycznych wykorzystano fortranowe procedury **$ADDF, $SUBF, $MULF** i **$DIVF**. Ponieważ są to procedury używane wyłącznie przez kompilator, sposób przekazywania parametrów stosowany przy ich wywołaniu różni się od obowiązującego w Fortranie dla instrukcji **CALL** (gdzie rejestr **R5** wskazuje obszar parametrów). W takim wypadku, dla programisty próbującego rozszyfrować sposób przekazywania parametrów, pomocne może być podglądnięcie kodu generowanego przez kompilator dla określonego wyrażenia arytmetycznego (wydruk 3).

```
;  z = x + y    ! EIS option

MOV     #2,@#AOTS
MOV     X+2,-(SP)
MOV     X,-(SP)
MOV     Y+2,-(SP)
MOV     Y,-(SP)
JSR     PC,#ADDF
MOV     (SP)+,Z
MOV     (SP)+,Z+2

;  z = x + y    ! FIS option

MOV     #2,@#AOTS
MOV     X+2,-(SP)
MOV     X,-(SP)
MOV     Y+2,-(SP)
MOV     Y,-(SP)
FADD    SP
MOV     (SP)+,Z
MOV     (SP)+,Z+2
```

Wydruk 3. Kod generowany przez kompilator Fortranu dla wyrażenia z=x+y w obu realizacjach

Wydruk 4. Program słów zrealizowanych w języku asemblera

```
;
W=%2            ; temporary used by 'NEXT' macro ( the inner interpreter )
U=%3            ; pointer to the user area
IP=%4           ; Forth instructions counter
S=%5            ; Forth stack pointer
RP=SP           ; Forth return-stack pointer
;
;
LINK=0          ; last link field is 0, indicating end of the dictionary.
;
        .MACRO HEAD,LENGTH,NAME,LCHAR,LABEL,CODE
LINK2=.
        .BYTE   LENGTH
        .ASCII  ^NAME^
        .EVEN
.=.-1
        .BYTE   LCHAR   ; last character of name ( or blank fill ),
                        ; passed in octal with high bit set.
        .WORD   LINK
LINK=LINK2
LABEL:  .IF NB  CODE
        .WORD   CODE
        .IFF
        .WORD   .+2
        .ENDC
        .ENDM
;
;
;
        .MACRO  NEXT
        MOV     (IP)+,W
        JMP     @(W)+
        .ENDM
;
;       .....
;
;*****************************************************************************
;
;       Floating point package implemented and developed by Zbigniew
;       Szkaradnik.
;
;*****************************************************************************
;
        .IFDF   FLT
;
        .IFDF   F79
        HEAD    202,F0,240,FAT,DOCOL                    ; ***** F0
        .WORD   TWAT,SEMIS
        .IFF
        HEAD    202,F0,240,FAT                          ; ***** F0
        MOV     (S),R0
        MOV     2(R0),(S)
        MOV     (R0),-(S)
        NEXT
        .ENDC
;
        .IFDF   F79
        HEAD    202,F!,240,FSTOR,DOCOL                  ; ***** F!
        .WORD   TWSTO,SEMIS
        .IFF
        HEAD    202,F!,240,FSTOR                        ; ***** F!
        MOV     2(S),0(S)
        ADD     #2,(S)
        MOV     4(S),0(S)
        ADD     #6,S
        NEXT
        .ENDC
;
        .IFDF   F79
        HEAD    204,FDUP,240,FDUP,DOCOL                 ; ***** FDUP
        .WORD   TWOD,SEMIS
        .IFF
        HEAD    204,FDUP,240,FDUP                       ; ***** FDUP
        MOV     2(S),-(S)
        MOV     2(S),-(S)
        NEXT
        .ENDC
;
        HEAD    205,FDROP,320,FDROP                     ; ***** FDROP
        ADD     #4,S
        NEXT
;
        HEAD    205,FOVER,322,FOVER                     ; ***** FOVER
        MOV     6(S),-(S)
        MOV     6(S),-(S)
        NEXT
;
        HEAD    205,FSWAP,320,FSWAP                     ; ***** FSWAP
        JSR     PC,SWAPF
        NEXT
;
SWAPF:  MOV     (S),R0
        MOV     4(S),(S)
        MOV     R0,4(S)
        MOV     2(S),R0
        MOV     6(S),2(S)
        MOV     R0,6(S)
        RTS     PC
;
        HEAD    204,FROT,240,FROT                       ; ***** FROT
        MOV     (S),R0
        MOV     2(S),R1
        MOV     10(S),(S)
        MOV     12(S),2(S)
        MOV     4(S),10(S)
        MOV     6(S),12(S)
        MOV     R0,4(S)
        MOV     R1,6(S)
        NEXT
;
;  Subroutines : $ADDF, $SUBF, $MULF, $DIVF, $CVTIF, $CVTFI
;                are taken from FORTRAN library.
;
        .MACRO  PUSHR
        MOV     R3,-(SP)
        MOV     R4,-(SP)
        .ENDM
;
        .MACRO  POPR
        MOV     (SP)+,R4
        MOV     (SP)+,R3
        .ENDM
;
        .MACRO  FPOP
        MOV     2(S),-(SP)
        MOV     (S)+,-(SP)
        ADD     #2,S
        .ENDM
;
        .MACRO FPUSH
        MOV     2(SP),-(S)
        MOV     (SP)+,-(S)
        ADD     #2,SP
```

```
.ENDM
;
        .IFDF   FIS                                     ; ***** F+
        HEAD    202,F+,240,FPLUS
        FADD    S
        NEXT
        .IFF                                            ; ***** F+
        HEAD    202,F+,240,FPLUS
        PUSHR
        SUB     #2,SP
        FPOP
        FPOP
        MOV     S,10(SP)
        JSR     PC,$ADDF
        .GLOBL  $ADDF
        MOV     4(SP),S
        FPUSH
        ADD     #2,SP
        POPR
        NEXT
        .ENDC

        .IFDF   FIS                                     ; ***** F-
        HEAD    202,F-,240,FMIN
        FSUB    S
        NEXT
        .IFF                                            ; ***** F-
        HEAD    202,F-,240,FMIN
        JSR     PC,SWAPF
        PUSHR
        SUB     #2,SP
        FPOP
        FPOP
        MOV     S,10(SP)
        JSR     PC,$SUBF
        .GLOBL  $SUBF
        MOV     4(SP),S
        FPUSH
        ADD     #2,SP
        POPR
        NEXT
        .ENDC

        .IFDF   FIS                                     ; ***** F*
        HEAD    202,F*,240,FSTAR
        FMUL    S
        NEXT
        .IFF                                            ; ***** F*
        HEAD    202,F*,240,FSTAR
        PUSHR
        SUB     #2,SP
        FPOP
        FPOP
        MOV     S,10(SP)
        JSR     PC,$MULF
        .GLOBL  $MULF
        MOV     4(SP),S
        FPUSH
        ADD     #2,SP
        POPR
        NEXT
        .ENDC

        .IFDF   FIS                                     ; ***** F/
        HEAD    202,F/,240,FSLAS
        FDIV    S
        NEXT
        .IFF
        HEAD    202,F/,240,FSLAS                        ; ***** F/
        JSR     PC,SWAPF
        PUSHR
        SUB     #2,SP
        FPOP
        FPOP
        MOV     S,10(SP)
        JSR     PC,$DIVF
        .GLOBL  $DIVF
        MOV     4(SP),S
        FPUSH
        ADD     #2,SP
        POPR
        NEXT
        .ENDC

        HEAD    204,S->F,240,STOF                       ; ***** S->F
        PUSHR
        SUB     #2,SP
        MOV     (SP)+,-(SP)
        MOV     S,2(SP)
        JSR     PC,$CVTIF
        .GLOBL  $CVTIF
        MOV     4(SP),S
        FPUSH
        ADD     #2,SP
        POPR
        NEXT

        HEAD    204,F->S,240,FTOS                       ; ***** F->S
        PUSHR
        SUB     #2,SP
        MOV     (SP)+,-(SP)
        MOV     S,2(SP)
        JSR     PC,$CVTFI
        .GLOBL  $CVTFI
        MOV     4(SP),S
        FPUSH
        ADD     #2,SP
        POPR
        NEXT

        HEAD    203,F0=,275,FZEQU                       ; ***** F0=
        BIC     #77600,(S)
        TST     2(S)
        BNE     1$
        MOV     (S)+,-(SP)
        ADD     #2,S
        MOV     #1,(S)
        NEXT
1$:     ADD     #2,S
        CLR     (S)
        NEXT

        HEAD    203,F0(,274,FZLES                       ; ***** F0(
        MOV     (S)
        BNE     1$
        ADD     #2,S
        CLR     (S)
        NEXT

        HEAD    202,F.,240,FLOT                         ; ***** F.
        PUSHR
        SUB     #2,SP
        FPOP
        MOV     S,4(SP)
        CLR     -(SP)
        CLR     -(SP)
        DEC     (SP)
        JSR     PC,$B28
        .GLOBL  $B28
        MOV     (SP)+,S
        POPR
        NEXT
;
;       Subroutine $B28 is taken from PASCAL library.
;
        HEAD    203,F.R,322,FDOTR                       ; ***** F.R
; ( fp prec  field  -- )
        JSR     PC,SWAPF
        PUSHR
        SUB     #2,SP
        FPOP
        MOV     (S)+,-(SP)
        MOV     (S)+,-(SP)
        MOV     S,10(SP)
        JSR     PC,$B28
        .GLOBL  $B28
        MOV     (SP)+,S
        POPR
        NEXT
;
        HEAD    206,FERROR,240,FPERR                    ; ***** FERROR
        MOV     FPE,-(S)
        CLR     FPE
        NEXT
FPE:    .WORD   0                       ; Number of floating point error
FPER:   MOV     #10.,FPE
        MOV     ORIGIN+20,U
        MOV     ORIGIN+22,S
        MOV     ORIGIN+24,RP
        MOV     #RES,IP
        NEXT
;
RES:    .WORD   FPERR,DUP,QERR,QUIT     ; Start after floating point error
;
        .ENDC
;
```

Podobnie, tzn. przez wykorzystanie fortranowych procedur **$CVTIF** i **$CVTFI,** zrealizowano bardzo przydatne konwersje między liczbami stało- i zmiennoprzecinkowymi. Jeśli procesor wyposażony jest w zestaw rozkazów zmiennoprzecinkowych (procesor M2 dla Mery 60 lub LSI-11/FIS), to operacje arytmetyczne należy zaimplementować przy użyciu rozkazów **FADD, FSUB, FMUL** i **FDIV.** Programy słów **F.** i **F.R** wyprowadzające liczbę zmiennoprzecinkową w systemie dziesiętnym, przy użyciu formatów wyjściowych odpowiadających fortranowym formatom **E** i **F,** zrealizowano przez dołączenie pascalowej procedury **$B28** (zajmuje ona mniej pamięci niż podobne procedury fortranowskie). Operacje stosowe i niektóre testy wygodnie jest napisać w języku asemblera. Wpływa to korzystnie na szybkość ich wykonania i nie powinno powodować większych kłopotów realiza-

```
Screen # 1

 0 ( Floating point package
 1   implemented and developed by Zbigniew Szkaradnik )
 2 DECIMAL
 3 : FCONSTANT (BUILDS HERE 4 ALLOT F! DOES> F@ ;
 4 : FVARIABLE (BUILDS HERE 4 ALLOT F! DOES> ;
 5 : FLITERAL DLITERAL ;
 6 : F= F- F0= ;
 7 : F< F- F0< ;
 8 : F) FSWAP F< ;
 9 1 S->F FCONSTANT F1.
10 10 S->F FCONSTANT F10.
11 F10. FVARIABLE FBASE
12 94  ( '^' character ) VARIABLE EXP-CHAR
13 : FNUMBER EXP-CHAR @ WORD HERE NUMBER DPL-@ )R
14 BL WORD HERE NUMBER R) DPL ! ;
15 -->

Screen # 2

 0 ( Floating point package
 1   implemented and developed by Zbigniew Szkaradnik )
 2 : EXP* 0 DO FBASE F@ F* LOOP ;
 3 : EXP/ 0 DO FBASE F@ F/ LOOP ;
 4 : FLOAT FNUMBER DROP )R FDUP DABS
 5 0= IF DROP DROP S->F ELSE
 6 DROP 1000 M/  S->F 3 EXP* ROT S->F F+ ENDIF
 7 DPL @ DUP 0 > IF EXP/ ELSE DROP ENDIF
 8 R DUP 0 > IF EXP* ELSE DROP ENDIF
 9 R> DUP 0 < IF ABS EXP/ ELSE DROP ENDIF
10 STATE @ IF [COMPILE] FLITERAL ENDIF ; IMMEDIATE
11 -->
12
13
14
15

Screen # 3

 0 ( Floating point package
 1   implemented and developed by Zbigniew Szkaradnik )
 2 : .F DPL @ )R 0 DPL ! FDUP FBASE F@ F< IF
 3 FDUP BEGIN 1 DPL +!  1 EXP/ FDUP  FBASE F@ F< UNTIL
 4 F->S DUP )R S->F DPL @ EXP* F-
 5 BEGIN 1 EXP/ FDUP F1. F< UNTIL
 6 3 EXP* F->S R> ELSE FDUP F1. F< IF
 7 FDUP BEGIN -1 DPL +! 1 EXP* FDUP F1. F> UNTIL
 8 F->S DUP )R S->F DPL @ ABS EXP/ F-
 9 BEGIN 1 EXP* FDUP F1. F> UNTIL
10 2 EXP* F->S R> ELSE
11 FDUP F->S DUP )R S->F F- 3 EXP* F->S R> ENDIF ENDIF
12 S->D 0 D.R 46 EMIT S->D 0 D.R EXP-CHAR @ EMIT DPL @ .
13 R> DPL ! ;
14
15
```

Wydruk 5. Program słów zrealizowanych w języku Forth

cyjnych. Jeżeli dysponujemy translatorem języka Forth-79, to można wykorzystać odpowiednie operacje stosowe dla słów o podwójnej długości, jak **2SWAP, 2DUP** itd.

Na wydruku 4 zestawiono słowa zrealizowane w języku asemblera.

Programy słów **FLOAT** i **F.** dotyczące liczb zmiennoprzecinkowych w dowolnej notacji, napisano w języku Forth (wydruk 5). Są one dość eleganckie i niezwykle zwięzłe (bardzo ważna zaleta języka Fortha), choć na pierwszy rzut oka mało czytelne.

Osobnego omówienia wymaga obsługa sytuacji błędnych. Błędy mogą powstać na skutek nadmiaru lub niedomiaru, próby dzielenia przez zero oraz próby konwersji zbyt dużej liczby zmiennoprzecinkowej na stałoprzecinkową. Błąd wykryty przez fortranowe procedury, realizujące operacje zmiennoprzecinkowe, powoduje przerwanie programowe (TRAP) do komórki 34 (ósemkowo). Jeżeli operacje arytmetyczne są wykonywane sprzętowo, to błąd powoduje przerwanie do komórki 244 (ósemkowo). W związku z tym należy odpowiednio oprogramować układ przerwań oraz napisać procedurę wznowienia po błędzie zmiennoprzecinkowym (patrz: wydruk 2).

```
        HEAD    204,COLD,240,COLD                   ; ***** COLD
CENT:                                   ; Cold start entry point
        .IFDF   FLT                     ; Initialization of
FPTRAP=34                               ; floating point traps
        MOV     #FPER,FPTRAP
        .IFDF   FIS
FPTRAP=244
        MOV     #FPER,FPTRAP
        .ENDC
        .ENDC
        MOV     ORIGIN+14,FORTH+6       ; Set 'FORTH' vocabulary from
                                        ; start up table
        MOV     ORIGIN+20,U             ; Initialize user pointer
;       etc.
```

Wydruk 6. Zmiany w procedurze COLD

Ponadto, należy zmodyfikować programy słów **ABORT** i **COLD** tak, aby po błędzie zmiennoprzecinkowym i wznowieniu działania podstawa była równa **10**, a adresy w wektorze przerwań ustawione (wydruk 6). Poza tym, w obszarze komunikatów o błędach należy umieścić odpowiednią informację.

• • •

W tabeli 2 przedstawiono wyniki porównania tak zmodyfikowanego języka Forth z innymi językami programowania: Fortranem, Pascalem i Basicem. Prównanie polegało na realizacji wybranego programu testowego w wymienionych językach (wydruki 7—10) oraz pomiarze czasu jego realizacji.

Tabela 2. Porównanie szybkości realizacji operacji zmiennoprzecinkowych

Język	Fortran	Fortran	Pascal	Basic	Forth
Kod	thr	nat	nat	int	thr
Czas (s)	68	63	92	580	105
Czas względny	1,08	1,00	1,46	9,21	1,67

procesor FIS, nat — kod naturalny, thr — kod kaskadowy, int — interpreter

Tabela 3.
Porównanie obydwu realizacji języka Forth

Język	Forth	Forth
Procesor	fis	eis
Czas [s]	105	202

```
100 REM fptest
110 Y = 1 \ X = 1
120 FOR JX = 1 TO 5000
130 FOR IX = 1 TO 10
140 Y = Y * X / X + X - X
150 X = IX
160 NEXT IX
170 NEXT JX
180 END
```

Wydruk 7. Program testowy w Basicu

Prównano także wersję translatora, wykorzystującą rozkazy zmiennoprzecinkowe (FIS), z wersją wykorzystującą bibliotekę podprogramów realizujących operacje zmiennoprzecinkowe (FIS). Wyniki tego porównania przedstawiono w tabeli 3.

```
c       fortran
        program fptest
        real x, y
        integer i, j
        y = 1
        x = 1
        do 20 j = 1, 5000, 1
          do 10 i = 1, 10, 1
            y = y * x / x + x - x
            x = i
10        continue
20      continue
        call exit
        end ! fptest
```

Wydruk 8. Program testowy w Fortranie

```
(pascal)
program fptest ;
var
  x, y : real ;
  i, j : integer ;
begin
  y := 1 ;
  x := 1 ;
  for j := 1 to 5000 do
    for i := 1 to 10 do
      begin
        y := y * x / x + x - x ;
        x := i
      end
end (fptest) .
```

Wydruk 9. Program testowy w Pascalu

```
( forth )
( fptest )
1 s->f fvariable x
1 s->f fvariable y
: fptest
  5001 1 do
    11 1 do
      y f@ x f@ f* x f@ f/ x f@ f+ x f@ f- y f!
      i s->f x f!
    loop
  loop ; ( fptest )
```

Wydruk 10. Program testowy w języku Forth

Myślę, że w podobny sposób można łatwo i szybko zmodyfikować każdą odmianę języka Forth, pod warunkiem posiadania źródłowej wersji translatora oraz pewnej znajomości procesora, systemu operacyjnego i istniejącego oprogramowania (makroasembler, kompilatory, biblioteki).

LITERATURA

[1] Brodie I.: Starting Forth. Prentice Hall, Englewood Cliffs (NJ), 1981

[2] Monroe A. J.: Forth Floating-Point. Dr Dobb's Journal, No. 71, September 1982

[3] Trojniar W.: FORTH — język i system programowania. INFORMATYKA, nr 5, 7, 8, 1984.

JERZY ZAKRĘCKI
Instytut Podstaw Informatyki
Polska Akademia Nauk
Warszawa

Abstrakcje w programowaniu (2)

W drugiej części artykułu przedstawiono przykład pakietu arytmetyki liczb wymiernych w Adzie oraz związki między stosowaniem abstrakcji i modularyzację programu.

JĘZYK ADA

Podstawowymi mechanizmami umożliwiającymi modularyzację programów w języku Ada są procedury, funkcje i pakiety. Definiowanie typów i obiektów abstrakcyjnych w Adzie jest możliwe dzięki pakietom. Definicja pakietu składa się z dwóch części:

- specyfikacji pakietu
- ciała pakietu

Specyfikacja pakietu zawiera opis zasobów dostarczanych przez pakiet. Ciało pakietu zawiera implementację tych zasobów. Ciało pakietu może być kompilowane oddzielnie od specyfikacji, dlatego przy modyfikacjach pakietu nie wymagających zmian w specyfikacji wystarczy kompilować powtórnie tylko jego ciało. Konstrukcję pakietu omówimy na przykładzie pakietu RATIONAL_ARITHMETIC (wydruk).

W programie tym korzystano z rozwiniętego mechanizmu typów i struktury blokowej w języku Ada. Mechanizm typów pozwala także na bardziej precyzyjny opis procedur. Przykładowo, nagłówek procedury CREATE zawiera informację o tym, że wartość parametru aktualnego odpowiadającego parametrowi formalnemu D powinna być nie mniejsza od 1. Analogiczną informację w Fortranie można uzyskać dopiero po zapoznaniu się z treścią procedury CREATE.

Wadą mechanizmów modularyzacji języka Ada jest opcjonalność ich stosowania. Znaczy to, że można w nim pisać bardzo czytelne i bezpieczne moduły, nie jest to jednak wymuszone przez język, tzn. możliwe jest tworzenie modułów o bardzo złej jakości. Nośnikiem tej wady jest struktura blokowa, wielofunkcyjność konstrukcji **package** i opcjonalność stosowania typów z atrybutem **private.** Struktura blokowa powoduje, że zasoby pakietu są dostępne w każdym miejscu bloku, w którym pakiet ten został zadeklarowany. Dodatkową wadą struktury blokowej jest fakt, że procedury i funkcje deklarowane w pakiecie mogą odwoływać się do zmiennych zewnętrznych. Konsekwencją tego faktu może być zależność wyniku działania tych procedur i funkcji od historii działania programu, co może powodować duże trudności przy testowaniu i modyfikacjach programu. Opcjonalność deklarowania typów prywatnych powoduje, że możliwe jest działanie na wartościach abstrakcyjnego typu danych (w przypadku gdy nie został on zadeklarowany jako prywatny) przy użyciu operacji różnych od dostarczanych przez pakiet, co może doprowadzić do zaburzenia ich funkcjonowania. Oprócz tego pewne zastrzeżenia może budzić konieczność przytaczania definicji typów prywatnych w specyfikacji pakietu.

ABSTRAKCJE A MODULARYZACJA PROGRAMU

Pojęcia modułu i modularyzacji programów nie doczekały się jeszcze ogólnie przyjętych definicji. Obecnie definicje obejmują praktycznie wszystko, co mieści się między modułami definiowanymi jako przypadkowo powiązane części programu a wyrafinowanymi realizacjami abstrakcyjnych typów danych. Ostatnio obserwuje się jednak pewną stabilizację definicji tych pojęć.

W bardzo często cytowanej pracy Parnasa [3] modularyzacja rozpatrywana jest jako środek do ułatwiania zrozumienia i pielęgnacji programów. Wyszczególnione są korzyści z jej zastosowania korespondujące z wymienionymi celami:

- stworzenie techniki programowania, która umożliwia programowanie modułów przy małej wiedzy o kodzie innych modułów oraz ponowną kompilację i wymianę modułów bez ponownej kompilacji całego programu (wymaganie to czasami nie jest uważane za konieczne — jako alternatywę proponuje się szybką kompilację)
- udoskonalenie zarządzania procesem programowania — czas tworzenia programów ulega skróceniu, ponieważ oddzielne grupy programistów mogą pisać różne moduły przy ściśle zdefiniowanej komunikacji między nimi
- ułatwienie pielęgnacji programów, co wpływa na możliwość dokonywania dużych zmian w jednych modułach, bez potrzeby modyfikacji innych modułów

```
package RATIONAL_ARITHMETIC is
  type RATIONAL is private; -- definicja typu
  function CREATE(N: INTEGER; D: POSITIVE) return RATIONAL;
  function SUM(I, J: RATIONAL) return RATIONAL;
  function MULT(I, J: RATIONAL) return RATIONAL;

  -- funkcja EQ - "=" i instrukcja przypisania ASSIGN - ":="
  -- dostarczana jest przez język dla każdego typu danych,
  -- definicja którego nie zawiera słowa kluczowego limited

  private
    type RATIONAL is record
                       N: INTEGER;
                       D: POSITIVE
                     end record;

end RATIONAL_ARITHMETIC;


package body RATIONAL_ARITHMETIC is

  package body RATIONAL-ARITHMETIC is

    function CREATE(N: INTEGER; D: POSITIVE) return RATIONAL is
      function GCD(N, D: POSITIVE) return POSITIVE is
        begin
          if N = D then return N
          elsif N > D then return GCD(N - D, D);
          else return GCD(N, D - N);
          end if;
        end GCD;

      begin;
        if N = 0 then return (0, 1);
        else
          declare A: POSITIVE;
          begin A := GCD(ABS(N), D); return (N/A, D/A);
        end if;
      end CREATE;

      function SUM(I, J: RATIONAL) return RATIONAL is
      begin
          return CREATE(I.N*J.D + J.N*I.D, I.D*J.D);
      end SUM;

      function MULT(I, J: RATIONAL) return RATIONAL is
      begin
          return CREATE(I.N*J.N, I.D*J.D);
      end MULT;

end RATIONAL_ARITHMETIC;


declare
  use RATIONAL_ARITHMETIC;
  A, B, C: RATIONAL;

begin
  A := CREATE(5, 6);
  B := CREATE(1, 2);
  C := SUM(A, B);
  if A = B then B := C end if;
end


/***   iii   ***/


typedef char Vec[2];
Vec arr[2] = { "j","b" },
    *ref = arr + 1;

main(){
  printf("%s%s", *arr,ref);
}
```

● zwiększenie zrozumiałości programów, dzięki możliwości studiowania programów częściami.

W artykule [4] lista ta została uzupełniona o dwa dodatkowe punkty:

● zwiększenie przenośności programów przez wyabstrahowanie cech zależnych od komputera we względnie małej liczbie modułów (wynik zastosowania tej techniki został opisany w [1])

● stworzenie ułatwień dla dowodzenia twierdzeń o programach, przez ścisłą i sformalizowaną dekompozycję problemu (i dowodu) na podproblemy.

Wyliczone zalety modularyzacji sprawiają, że potrzeba jej stosowania nie wymaga dodatkowych uzasadnień. Istota problemu sprowadza się do sformułowania zasady dekompozycji programów na moduły i na dostarczeniu narzędzi do ich programowania.

Zasada dekompozycji programów na moduły spełniające powyższe wymagania została sformułowana w pracy Parnasa [3]. Sprowadza się ona do ukrycia (wyabstrahowania) w oddzielnych modułach pewnej informacji o programie (tzn. informacji o strukturach danych, algorytmach, zasobach lub urządzeniach wykorzystywanych w programie) w taki sposób, by dostęp do tej informacji był realizowany tylko w ukrywających ją modułach. Z zasady tej wynikają następujące właściwości modułów:

1) moduły powinny komunikować się ze sobą w sposób maksymalnie uproszczony, ideałem jest jawne przekazywanie wszystkich danych i wyników przy użyciu prostych (niestrukturalnych) parametrów (odpowiada to minimalizacji więzi modułowej [2]);

2) moduł powinien realizować jedną funkcję lub kilka funkcji określonych na tej samej strukturze danych (odpowiada to maksymalizacji mocy modułu [2]); oznacza to ukrycie szczegółów implementacyjnych funkcji realizowanych przez moduł lub obsługiwanego przez niego zasobu programu.

Programy spełniające wymagania sformułowanej powyżej zasady są ideałem, do którego należy dążyć, aczkolwiek pewne świadome i uzasadnione odstępstwa od niej nie oznaczają dyskwalifikacji całego programu. W zależności od stopnia spełnienia zasady dekompozycji można mówić o większej lub mniejszej modularności programu.

Nie trudno zauważyć, że moduły spełniające powyższe wymagania są doskonałym mechanizmem do definiowania abstrakcji (procedur, abstrakcyjnych typów danych i obiektów abstrakcyjnych). Wynika to z prostoty i jawności komunikowania się modułów między sobą (programista ma pełną kontrolę nad przepływem informacji do modułów i z modułów) oraz z faktu, że każdy moduł realizuje albo jedną funkcję (w tym przypadku odpowiada on proceduralnej abstrakcji), albo kilka funkcji określonych na tej samej strukturze danych (w tym przypadku jest on implementacją abstrakcyjnego typu danych lub obiektu abstrakcyjnego). Wydaje się więc, że lista korzyści płynących z modularyzacji powinna być rozszerzona o jeszcze jeden, bardzo istotny element:

● **umożliwienie programowania w abstrakcjach.**

Nie oznacza to, że nie można programować w abstrakcjach bez stosowania modułów, jednak programowanie w abstrakcjach bez użycia środków umożliwiających modularyzację programów ma zbyt wiele wad. W związku z tym można mówić o większej lub mniejszej modularyzacji języka programowania w zależności od obecności w nim konstrukcji ułatwiających definiowanie abstrakcji.

Warto zaznaczyć, że modularne właściwości języków programowania są w pewnym stopniu sprzeczne z możliwością ich efektywnej implementacji, np. włączenie definicji prywatnego typu danych do specyfikacji pakietu w języku Ada (co pogarsza modularyzacyjne właściwości tego języka) podyktowane było wyłącznie względami implementacyjnymi. Dzięki takiej konstrukcji możliwa jest efektywniejsza implementacja operacji dostarczanych przez pakiet.

Z przedstawionej analizy można wyciągnąć następujące wnioski (postulaty) dotyczące pożądanych właściwości nowoczesnych języków programowania:

● język powinien zapewniać tylko jeden rodzaj komunikacji między modułami — komunikację przy użyciu prostych (niestrukturalnych) parametrów:

● język powinien zachęcać do pisania modułów realizujących jedną funkcję lub kilka funkcji określonych na tej samej strukturze danych, tzn. powinien udostępniać mechanizmy do definiowania abstrakcji

● korzystanie z modułu powinno być oddzielone od implementacji dostarczanych przez niego zasobów

● powinna istnieć możliwość efektywnej implementacji oddzielnej kompilacji modułów.

Nie wydaje się, aby naturalna, prosta i efektywna realizacja wszystkich tych postulatów w jednym języku programowania była zadaniem łatwym. W każdym konkretnym projekcie językowym, w zależności od jego celów i założeń, należy uwzględniać część tych postulatów, a pozostałe traktować jako pożądane, aczkolwiek niekonieczne.

Autor pragnie gorąco podziękować prof. dr. hab. Leonowi Łukaszewiczowi za pomoc w opracowaniu przykładów i wiele cennych uwag podczas przygotowywania kolejnych wersji artykułu.

LITERATURA

[1] Henderson P., Simson R. B.: Modularization of large programs. Software Practice and Experience, 1981, Vol. 11, pp. 497—520

[2] Myers G. J.: Projektowanie niezawodnego oprogramowania. WNT, Warszawa, 1980

[3] Parnas D. L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM, 1972, Vol. 15, pp. 1053—1058

[4] Schwabe D., Lucena C. J.: Design and implementation of abstraction definition facility, Software Practice and Experience, 1978, Vol. 8, pp. 707—719.

LESZEK KOTULSKI
Katedra Informatyki
Uniwersytet Jagielloński
Kraków

Monitor jako narzędzie strukturalizacji programów współbieżnych (I)

U podstaw rozwoju programowania współbieżnego legło oczywiste spostrzeżenie: jeżeli pewne czynności mogą być wykonane równocześnie, to cała praca zostanie ukończona szybciej. Otwarta pozostała jedynie metoda opisu wykonania programu współbieżnego. W przeprowadzonych badaniach uwidoczniły się dwa podejścia:
— implementacja „inteligentnego" kompilatora, który sekwencyjny program, napisany przez programistę, przekształci w trakcie kompilacji na współbieżny program wynikowy
— utworzenie programu współbieżnego jako systemu programów sekwencyjnych, które mogą być wykonywane równocześnie jako procesy współbieżne.

Wstępna analiza wykazuje, że pierwsze podejście wymaga dużej regularności struktur danych (np. takich jak tablice). W praktyce więc, rozwiązanie to zdobyło sobie większą popularność jedynie przy realizowaniu operacji macierzowych, szczególnie przy wykorzystaniu sprzętu w rodzaju procesorów tablicowych. Podejście drugie pozostawia programiście wyłączną decyzję, które procesy mogą być wykonywane współbieżnie. Intensywne badania dotyczą jedynie dostarczenia efektywnych oraz elastycznych narzędzi, służących do wydzielenia procesów sekwencyjnych oraz pozwalających zorganizować ich współpracę.

Podstawowym elementem środowiska programu współbieżnego jest proces sekwencyjny, rozumiany jako wykonanie pewnej procedury. W większości języków programowania, zapis fragmentu programu określającego proces jest identyczny z zapisem procedury, z dokładnością do zamiany słowa kluczowego **procedure** przez słowo **process** (w językach Concurrent Pascal, Modula, Loglan, Chill) lub **task** (w języku Ada).

O dwóch procesach powiemy, że są **współbieżne**, jeżeli jeden z nich rozpocznie swoją pracę w przedziale czasu pomiędzy rozpoczęciem a zakończeniem pracy procesu drugiego. Wśród procesów wyróżniamy **procesy interakcyjne** oraz **nieinterakcyjne**. Procesami interakcyjnymi są takie pary procesów, w których działanie każdego z nich zależy również od działania procesu z nim współpracującego. Wykorzystywanie procesów nieinterakcyjnych nie stwarza żadnych dodatkowych problemów w porównaniu z programowaniem sekwencyjnym.

KONCEPCJA MONITORA

W wypadku procesów interakcyjnych istnieją dwie równoważne formy [15] wymiany informacji:
— przesyłanie komunikatów
— przekazywanie informacji przez zmienne wspólne.

Zmienne wspólne mogą być dostępne dla kilku procesów, które wykorzystując operacje podstawienia wpisują informacje do tej zmiennej oraz na bieżąco testują jej wartość w programie. Najpowszechniej stosowaną metodą synchronizacji procesów z wykorzystaniem obszarów pamięci wspólnej jest mechanizm zawarty w zapisie monitora. Monitor został wykorzystany w kilku językach programowania współbieżnego, takich jak Concurrent Pascal [2], Modula [18], Simone [8], Pascal-Plus [17], Mesa [14], Chill [16] i in. Stworzenie koncepcji monitora było poprzedzone kilkoma wcześniejszymi opracowaniami, formułującymi zasady wykorzystywania zmiennych wspólnych.

Niekontrolowane wykorzystanie zmiennych wspólnych niesie w sobie niebezpieczeństwo wystąpienia błędów uwarunkowanych czasowo. Wykrywanie przyczyn powstania tego rodzaju błędów jest na etapie wykonania bardzo kosztowne lub nawet niemożliwe, z uwagi na brak w programowaniu współbieżnym możliwości powtórzenia przebiegu w identycznych warunkach (czyli brak znanej z programowania sekwencyjnego cechy zwanej reprodukcyjnością), gdyż wymagałoby to ciągłego zbierania informacji przez programy śledzące. Z drugiej strony nie jest ono konieczne, gdyż warunkiem wystarczającym do wyeliminowania błędów uwarunkowanych czasowo jest spełnienie wymagania, aby każdy proces miał zagwarantowany wyłączny dostęp do zmiennej wspólnej przez taki czas, jaki uzna on za stosowne. Koncepcja ta przyjęła nazwę **rejonów krytycznych** i została lingwistycznie wprowadzona przez Brinch Hansena [1] w postaci:

region V do ... od

Podejście to ma dwie wady:

1. Klasa problemów, które można rozwiązać w ten sposób, jest zbyt wąska. Przykładowo, dla problemu wymiany komunikatów przez bufor wymagane jest, aby proces odbierający komunikat po wejściu do rejonu mógł sprawdzić, czy komunikat został umieszczony i ewentualnie mógł poczekać na jego umieszczenie. Takie oczekiwanie musi odbywać się na zewnątrz rejonu krytycznego, aby proces nadawczy mógł wstawić komunikat do bufora. Jednocześnie, pozostawienie procesu w aktywnej pętli oczekiwania (ang. busy from waiting) na spełnienie warunku jest rozwiązaniem efektywnym.

2. Drugą wadą jest pracochłonność sprawdzania, czy procesy nie nadużywają zaufania, przebywając w rejonie krytycznym dłużej niż jest to konieczne, a nawet czy go w ogóle opuszczają. Sprawdzanie to wymaga bowiem z konieczności zbadania całego tekstu programu współbieżnego (w najlepszym wypadku, gdy zmienne wspólne są jawnie deklarowane, jedynie w tych procesach, w których występuje interesująca nas zmienna). Co więcej, zrealizowanie tych samych funkcji za pomocą innego algorytmu wymaga powtórnej weryfikacji poprawności rozwiązania.

Rozwiązanie pierwszego problemu wymaga zdefiniowania dodatkowych operacji zwanych **operacjami synchronizacyjnymi**. Dla rozwiązania drugiego problemu, zdecydowano się przenieść zmienną wspólną (ewentualnie kilka zmiennych wspólnych) do odrębnego modułu. Jednocześnie przeniesiono do tego modułu wszystkie operacje wykonywane dotychczas

Dr LESZEK KOTULSKI ukończył w 1979 r. studia informatyczne na Uniwersytecie Jagiellońskim. W 1984 r. obronił pracę doktorską na Akademii Górniczo-Hutniczej w Krakowie. Pracuje w Instytucie Informatyki Uniwersytetu Jagiellońskiego jako pracownik dydaktyczno-naukowy. Zajmuje się teorią programowania współbieżnego, ze szczególnym uwzględnieniem systemów operacyjnych.

w rejonach krytycznych, związanych z tą zmienną (zmiennymi), zastępując treść instrukcji procedurą zapisaną w tym module, a samą instrukcję **region** — wywołaniem zdefiniowanej procedury. Koncepcja zgromadzenia zmiennych w jednym module i udostępniania pozostałym modułom programowym możliwości użytkowania tych zmiennych jedynie w postaci dobrze zdefiniowanych operacji, została wprowadzona w programowaniu sekwencyjnym już w roku 1967 w języku Simula [4] i nosi nazwę **klasy.**

Koncepcja klasy nie zapobiega jednak możliwości równoczesnego wykonania dwóch procedur klasy, jeżeli zostaną one wywołane przez procesy współbieżne. W celu uzyskania wzajemnego wykluczenia, wprowadzono pojęcie monitora jako modyfikację pojęcia klasy. W monitorze zawsze jest spełniony warunek, że tylko jedno z wielu zgłoszeń wykonania procedur modułu jest w danej chwili możliwe. Dodatkowo, niemożliwe jest zewnętrzne wywłaszczenie wykonywania tej procedury na rzecz danego procesu. Zadania z pozostałych procesów są zapamiętywane w kolejce wejściowej i realizowane po zwolnieniu monitora. Korzyści z wprowadzenia monitora sformułowano w raporcie Concurrent Pascala:

1. Ułatwia on strukturalne wieloprogramowanie, umożliwiając programiście skupienie szczegółów implementacyjnych na współdzielonym typie danych w jednym miejscu. W rezultacie, procesy komunikacyjne muszą wiedzieć, co robi monitor, a nie muszą i nawet nie powinny wiedzieć, jak on to robi.

2. Ułatwione jest testowanie programu, ponieważ kompilator ma prawo uznać za nielegalne każde odwołanie do struktur danych spoza monitora. Oznacza to, że monitor, którego poprawność została sprawdzona, będzie zachowywał się poprawnie w sensie lokalnym, gdy zostanie włączony do większego systemu.

3. Upraszcza się dowodzenie poprawności programów współbieżnych.

Reasumując — **pod pojęciem monitora rozumie się wspólną strukturę danych i zbiór wzajemnie wykluczających się w czasie operacji, służących do wymiany informacji oraz synchronizacji pracy procesów współbieżnych.** Naszkicowana tu składnia oraz semantyka obiektu monitora została zaakceptowana we wszystkich znanych implementacjach monitorów różniących się jedynie szczegółami notacji.

Konieczność synchronizacji pracy procesów, przez wyznaczenie względnej kolejności ich wykonania, niesie ze sobą konieczność uzupełnienia wymienionych elementów monitora o specjalne operacje synchronizacyjne, które pozwolą zawiesić proces w oczekiwaniu na spełnienie pewnego warunku, określonego przez wspólną strukturę danych oraz odpowiednio odwiesić zawieszony proces. Postać tych operacji, ich efektywność, ekspresyjność notacji oraz reguły szeregowania, określające wzajemne powiązania pomiędzy procesami, umożliwiają realizację dużej liczby różnych implementacji monitora. Czytelników zainteresowanych formalnymi właściwościami monitora warto odesłać do pracy Buhra i Bowena [3], którzy wykorzystując diagramy przepływowe opisują możliwe do zrealizowania modele monitorów.

PRZEGLĄD OPERACJI SYNCHRONIZACYJNYCH

Zastosowanie koncepcji programowania współbieżnego wymaga zapewnienia możliwości zsynchronizowania pracy kilku procesów. Tak więc, monitor w ramach obsługi procesu może świadomie opóźnić pracę tego procesu, do momentu aż inny proces przekaże do niego informację o możliwości spełnienia żądań zawieszonego procesu. W takim wypadku konieczne jest, aby proces mógł opuścić monitor bez podejmowania dalszej pracy oraz mógł „automatycznie" wrócić do przerwanej obsługi. Najbardziej eleganckim rozwiązaniem, pozwalającym wpisywać warunek dalszej obsługi wprost w operacji synchronizacyjnej jest wywołanie:

wait (B)

gdzie **B** oznacza wyrażenie logiczne, którego obliczona wartość ma wpływ na zawieszenie procesu. Kolejne obliczanie wartości wyrażenia **B** oraz pozostałych wyrażeń związanych z operacją **wait** odbywa się po każdym zakończeniu obsługi procesu przez monitor lub po przerwaniu tej obsługi. Jeżeli istnieją procesy, których warunki zawieszenia są spełnione, to jeden z nich wróci do obsługi — w przeciwnym razie monitor będzie obsługiwał jeden z procesów czekających na wejściu.

Rozwiązanie to ma dwie zasadnicze wady:

1. Mała efektywność (nawet przy proponowanych przez Kessel'a [10] próbach jej zwiększenia) spowodowana ciągłą koniecznością obliczania wyrażeń logicznych.
2. Istnienie możliwości tzw. zagłodzenia procesu. Przy dużym obciążeniu monitora można wyobrazić sobie sytuację, że zawsze, gdy dany proces **P** mógłby być obsługiwany, będą istniały inne procesy, dla których warunek zawieszenia będzie obliczony wcześniej i które zmienią stan zmiennych tak, że proces **P** znów będzie musiał czekać. Przy jednym procesorze warunki będą obliczane w zadanej kolejności. W wypadku systemów wieloprocesorowych może decydować o tym względna szybkość procesorów.

Hoare w swym fundamentalnym artykule [5], zwracając uwagę na ciągłą konieczność stosowania kompromisu pomiędzy elegancją notacji a efektywnością implementacji, zaproponował kolejne rozwiązanie — noszące odtąd nazwę monitora Hoarea. W propozycji tej z każdym potencjalnym zdarzeniem wiąże się odpowiednią zmienną warunkową typu **condition.** Monitor wykorzystując tę zmienną może zawiesić aktualnie wykonywany proces, wykonując operację:

wait (cond)

oraz odwiesić jeden z procesów zawieszonych przy tej zmiennej wykonując operację:

signal (cond)

Po wykonaniu operacji, powodującej odwieszenie procesu, powstaje sytuacja, w której dwa procesy znajdą się jednocześnie aktywne w monitorze — proces odwieszający i odwieszany. Trzeba więc podjąć decyzję, który z nich będzie obsługiwany wcześniej. W wypadku operacji **signal** zdecydowano, że wcześniej jest obsługiwany proces odwieszany, o ile taki istnieje. Po zakończeniu tej obsługi, czyli po wykonaniu całej procedury monitora, lub przerwaniu obsługi, czyli wykonaniu kolejnej operacji **wait,** monitor powraca do obsługi procesu odwieszającego — wykonując operację następną po **signal.**

Zmienne warunkowe nie mają związanej z nimi wartości logicznej i mimo że są jawnie deklarowane, mogą być użyte jedynie jako argumenty powyżej opisanych operacji **signal** i **wait.** W praktyce, zmienne warunkowe są więc raczej zmiennymi typu kolejkowego, w których procesy oczekują na spełnienie warunków. Taką też interpretację przyjął Brinch Hansen w swojej definicji monitora podanej w [2], a zaimplementowanej w języku Concurrent Pascal. Istnieje więc następująca odpowiedniość między typami zmiennych:

condition — queue
wait — delay
signal — continue
queue — empty

Ze względu na trudności implementacyjne Brinch Hansen poczynił pewne ograniczenia:

- w danej kolejce może być zawieszony co najwyżej jeden proces
- operacja **continue** musi być ostatnią wykonywaną operacją monitora.

Drugie z tych założeń usuwa ponadto zagrożenie wystąpienia błędów, uwarunkowanych czasowo, spowodowane przez odwieszony proces modyfikacją struktury danych przechowującej informację, niezbędną do prawidłowej obsługi procesu odwieszającego. Założenie to jest jednak zbyt silne. Co prawda, Hoare w analizowanych przez siebie przykładach nie wykazuje potrzeby rezygnacji z niego, to jednak do potwierdzenia naszej tezy wystarczy rozważyć problem oczyszczenia kolejki po spełnieniu warunku, na który oczekuje kilka procesów. Nie da się tego zrealizować za pomocą konstrukcji:

while not empty(Q) **do continue**(Q);

gdyż pierwsze wykonanie operacji **continue** spowoduje opuszczenie monitora. Konieczne jest więc, aby parametr wyjściowy informował o stanie kolejki i proces mógł powtórnie wywołać odpowiednią procedurę monitora, by odwiesić kolejne procesy z kolejki. Nawet jeżeli istnieje możliwość zdefiniowania takiej akcji w odrębnym module, przy użyciu koncepcji klasy, to rozwiązanie takie zmusza do badania powiązań pomiędzy kilkoma modułami. W Concurrent Pascalu

klasa może być wywoływana jedynie z monitora, a realizacja powyższego problemu za pomocą dwóch monitorów powoduje zakleszczenie.

Przedstawione założenie nie stanowi warunku koniecznego dla uniknięcia występowania błędów uwarunkowanych czasowo, wystarczy bowiem zadbać o zagwarantowanie niepodzielności operacji elementarnych, definiowanych w monitorze i zezwalać na wykonanie operacji **continue** jedynie jako ostatniej w tych operacjach lub pomiędzy nimi. Można to zapewnić stosując metodę podaną w [11, 12].

Podjęcie decyzji o nadaniu priorytetu w obsłudze przez monitor procesu odwieszanego — w stosunku do procesu odwieszającego — nie jest jedynym praktycznie stosowanym rozwiązaniem. Przykładowo, w języku Mesa [14] operacja **notify** powoduje odwieszenie jednego z procesów zawieszonych w danej kolejce, wskutek wykonania przez obsługującą je procedurę monitora operacji **wait**. Monitor będzie kontynuował obsługę odwieszanego procesu po pewnym czasie, nie wcześniej jednak niż po zakończeniu obsługi lub zawieszeniu procesu odwieszającego. Czas podjęcia tej obsługi nie jest precyzyjnie określony, tak więc dopuszczalne jest nawet, że inne procesy oczekujące na obsługę przez monitor będą obsłużone wcześniej. Oczywiście w takim wypadku nie można zapewnić, że warunek, na którego spełnienie oczekiwał proces, będzie spełniony w momencie podjęcia obsługi tego procesu. W rezultacie, w miejsce tradycyjnie stosowanej w monitorach Brinch Hansena i Hoarea konstrukcji:

if not <warunek kontynuacji> **then wait (Q)**

w języku Mesa jako wzorzec proponowana jest konstrukcja:

while not <warunek kontynuacji> **do wait (Q)**

Istotnym ograniczeniem przedstawionych propozycji operacji synchronizacyjnych jest fakt, że ani **signal**, ani **continue** nie pozostawiają śladu swojego wykonania — jeżeli kolejka procesów oczekujących na spełnienie zdarzenia jest pusta. Tak więc, informacje o tym wydarzeniu muszą być pamiętane w globalnej strukturze danych monitora, co znacznie utrudnia weryfikację programu.

Kolejny zarzut stawiany powyższym operacjom synchronizacyjnym polega na tym, że nie pozwalają one oczekiwać na spełnienie alternatywy lub koniunkcji warunków [9]. Przykładowo załóżmy, że należy zawiesić proces polecający realizację transmisji i odwiesić go po jej zrealizowaniu, po upłynięciu czasu oczekiwania lub po otrzymaniu sygnału o niesprawności urządzenia. Na zrealizowanie tych wymagań pozwala implementacja rozszerzonej operacji **delay** [11, 13], która wprowadza wyrażenie kolejkowe jako parametr operacji. Wyrażenie to powstaje wskutek połączenia nazw kolejek za pomocą operatorów **or** i **and** w taki sposób, że podstawiając w miejsce nazw kolejek warunki z nimi związane, otrzymuje się złożony warunek, na którego spełnienie oczekuje proces. Przykładowo, jeżeli odpowiednie zmienne kolejkowe oznaczają:

KONIEC_TRANSM — oczekiwanie na zakończenie transmisji
ZBYT_DŁUGO — oczekiwanie na sygnał przekroczenia czasu realizacji
AWARIA — wystąpienie sygnału o awarii urządzenia

to przedstawiony problem można rozwiązać za pomocą rozszerzonej instrukcji **delay** w następujący sposób:

delay (KONIEC_TRANSM or ZBYT_DŁUGO or AWARIA)

gdzie operacje **continue** będą sygnalizowały spełnienie odpowiednich warunków.

PRZYKŁAD UŻYCIA MONITORA

Sposób wykorzystania monitora zilustrowano poniżej, rozwiązując postawiony przez Dijkstrę problem „czytelnicy-pisarze" — pierwszy raz rozwiązany przy użyciu monitora przez Hoarea [5]. W problemie tym zakłada się istnienie dwóch klas procesów zwanych „czytelnikami" i „pisarzami", współzawodniczących o zasób (zwany dalej stroną). Każdy z procesów „pisarzy" wpisując informację musi mieć zagwarantowany wyłączny dostęp do strony. Procesy zwane „czytelnikami" mogą jednocześnie korzystać z informacji zapisanej na stronie, o ile żaden proces „pisarz" nie wpisuje nowej informacji. W celu zsynchronizowania pracy tych procesów zdefiniowano monitor MON1.

```
monitor MON1;
const MAXPROC = 10   {MAKSYMALNA LICZBA PROCESOW CZYTELNIKOW
                          I PISARZY}
var LC,LP:INTEGER      {LICZBA PROCESOW CZYTAJACYCH I PISZACYCH}
    CZYT,PISZ:CONDITIONAL;
    PIS:BOOLEAN;       {CZY ODBYWA SIE PISANIE}

procedure entry POCZATEK_CZYTANIA;
begin
if LP <> 0 or PIS then wait(CZYT);
LC:=LC+1
end;

procedure entry POCZATEK_PISANIA;
begin
LP:=LP+1;
if  LC > 0 then wait(PISZ);
PIS:=TRUE
end;

procedure entry KONIEC_CZYTANIA;
begin
LC:=LC-1;
if LC = 0 then signal(PISZ)
end;

procedure entry KONIEC_PISANIA;
begin
PIS:=FALSE;
LP:=LP-1;
if LP > 0 then signal(PISZ)
   else while not queue(czyt) do signal(CZYT);
end;
begin
LP:=0;
LC:=0;
PIS:=FALSE;
end.
```

Monitor MON1 do synchronizacji procesów

Każdy proces „pisarza", przed wpisaniem informacji, powiadamia o tym monitor — wywołując procedurę POCZĄTEK_PISANIA (tzn. wykonując operację MON1.POCZĄTEK_PISANIA), a po wpisaniu wywołuje procedurę KONIEC_PISANIA. Podobnie, procesy „czytelników" zgłaszają chęć czytania, wywołaniem procedury POCZĄTEK_CZYTANIA, a koniec — wywołaniem procedury KONIEC_CZYTANIA.

LITERATURA

[1] Brinch Hansen P.: Structured multiprogramming. Communications of the ACM, Vol. 15, No. 7, pp. 574—578, 1972

[2] Brinch Hansen P.: The programming lanquage Concurrent Pascal. IEEE Trans. on Software Engineering, Vol. SE-1, No.2, pp. 199—207, 1975

[3] Bowen B. A., Buhr R. J. A.: The logical design of multiple-microprocesor systems. Prentice Hall Inc., Englewood Cliffs, New Jersey ,1980

[4] Dahl O. J., Myhrhaug B., Nygaard K.: Simula 67 Common Base Lanquage. Norwegian Computer Center, 1970

[5] Hoare C. A. R.: Monitor — an operating system structuring concept. Communications of the ACM, Vol. 13, No. 10, pp. 549—557, 1974

[6] Iszkowski W., Maniecki M.: Programowanie współbieżne. WNT, Warszawa, 1982

[7] Iszkowski W.: Przegląd mechanizmów komunikacji. Informatyka nr 3, 1985

[8] Kaubisch W. H., Perrot R. H., Hoare C. A. R.: Quasiparallel programming. Software-Practice and Experience, Vol. 6, pp.341—384, 1976

[9] Keedy J. L.: On structuring Operating Systems for Monitors. The Australian Computer Journal, Vol. 10, No 1, pp. 23—27, 1978

[10] Kessels J. L. W.: An alternative to event queues for synchronization in monitors. Communications of the ACM, Vol. 19, No 7, pp. 371—384, 1976

[11] Kotulski L.: Metoda eliminacji błędów uwarunkowanych czasowo w systemach procesów współbieżnych. Rozprawa doktorska, Uniwersytet Jagielloński, Kraków, 1984

[12] Kotulski L.: Time depending errors exclusion method in monitor structures. Zeszyty Naukowe Uniwersytetu Jagiellońskiego, seria Informatyka, nr 3, 1985

[13] Kotulski L.: On the use of monitors operating systems. Zeszyty Naukowe Uniwersytetu Jagiellońskiego, seria Informatyka, nr 1, 1983

[14] Lampson B. W., Redell D. D.: Experience with Processes and Monitors in Mesa. Communications of the ACM, Vol. 23, No. 2, pp. 105—177, 1980

[15] Lauer H. C., Needham R. M.: On the duality of Operating Systems Structures. Operating Systems Review, Vol. 13, No. 2, pp. 3—19, April 1979

[16] Udrycki Wł., Wilczyński W.: Chill — język programowania systemów komunikacyjnych. Informatyka, nr 9—11, 1984

[17] Welsh J., Bustard D. W.: Pascal-Plus — Another Language for Modular Moltiprogramming. Software — Practice and Experience, Vol. 9, pp. 947—957, 1979

[18] Wirth N.: Modula — a programming language, for modular multiprogramming. Software — Practice and Experience, Vol. 7, No. 1, pp. 3—81, 1977.

MICHAŁ KLEIBER
MACIEJ LEŚNY
ROMUALD SZUNIEWICZ

Komputery osobiste w zastosowaniach profesjonalnych (I)

Gwałtowne rozpowszechnienie mikrokomputerów wśród użytkowników indywidualnych w krajach wysoko rozwiniętych można przypisać dwóm czynnikom. Pierwszym z nich jest spadek cen do poziomu umożliwiającego zakup zestawu o dużych możliwościach obliczeniowych przez osoby średniozamożne, a prostego mikrokomputera — także przez młodzież. Drugim, ważniejszym czynnikiem jest rozwój oprogramowania do zastosowań profesjonalnych, nie wymagającego od użytkownika żadnej wiedzy informatycznej. Obok tych dwóch czynników można wymienić także szereg innych, jak choćby dostępność atrakcyjnych gier dla dzieci i dla dorosłych w wielkim wyborze, zwłaszcza na tańsze komputery typu domowego — i coraz powszechniejsze przekonanie rodziców (poparte intensywną reklamą producentów), że przyszłość ich dzieci zależy od poznania i opanowania mikrokomputerów.

Użytkownik profesjonalny traktuje komputer jako inwestycję, której koszt powinien się zwrócić, choć niekoniecznie w wymiarze finansowym. Zysk z używania komputera może przejawiać się w oszczędności czasu, w większym komforcie pracy, uzyskanym dzięki zautomatyzowaniu nużących czynności rutynowych, a przede wszystkim w otwarciu nowych możliwości, także twórczych. Ten artykuł jest poświęcony zastosowaniom profesjonalnym, niespecjalistycznym. Chodzi więc o podstawowe zastosowania w pracy naukowca, ekonomisty, inżyniera, a częściowo także humanisty lub każdego innego użytkownika, który pracuje przy biurku. Zastosowania mikrokomputerów do innych profesjonalnych celów, takich jak sterowanie procesami, projektowanie elementów mechanicznych, komponowanie muzyki itp., są tak szerokie i różnorodne, że samo ich wyliczenie przekroczyłoby ramy jednego artykułu.

Dziś uznaje się już powszechnie, że o pożytku z komputera decyduje oprogramowanie. Dlatego przed kupnem systemu komputerowego trzeba najpierw określić cele, następnie wyszukać oprogramowanie, które te cele najlepiej zrealizuje, a dopiero na końcu wybrać konfigurację systemu komputerowego niezbędną do pełnego wykorzystania programów. Jednakże, wielka uniwersalność komputerów osobistych sprawia, że można pokusić się o zdefiniowanie takiego zestawu mikrokomputerowego, który przy rozsądnej cenie spełni większość niespecjalistycznych wymagań użytkownika profesjonalnego.

SPRZĘT MIKROKOMPUTEROWY

Pierwszą sprawą jest wybór podstawowej odmiany mikrokomputera. Można przy tym rozważać typ mikroprocesora (np. 8-bitowe — Z80, 6502, 8080, 16-bitowe — 8086, 8088, 32-bitowe — 68000), szybkość działania na liczbach stało- i zmiennoprzecinkowych, rozdzielczość monitora i inne parametry techniczne, ale jak wskazuje doświadczenie wielu producentów, doskonałość techniczna i innowacyjność nie gwarantuje zdobycia zaufania użytkowników. Bezpiecznym kryterium, według którego użytkownik może oceniać wartość mikrokomputera, jest oferta programowa na ten komputer — stanowiąca najbardziej kompetentną zbiorową ocenę wystawioną przez firmy produkujące oprogramowanie, których egzystencja zależy od trafności wyboru. Kolejnym bezpiecznym kryterium jest popularność i rozpowszechnienie mikrokomputera wśród innych użytkowników o podobnym profilu zastosowań. Pomimo wielkich zasług firmy Apple, w popularyzacji komputerów osobistych i niewątpliwych zalet komputerów Apple i Macintosh, trzeba zdawać sobie sprawę, że dziś na rynku komputerów osobistych typu profesjonalnego dominuje zdecydowanie IBM.

Według czasopisma The Economist (1985, nr 7391, s. 78) w 1984 r. na rynku amerykańskim sprzedano mikrokomputery profesjonalne o wartości 6,6 mld dol., z czego 41,5% stanowiły mikrokomputery IBM, 4,5% naśladujące IBM komputery Compaq, 11% komputery firmy Apple, 6,3% komputery firmy Tandy, 3,9% Hewlett-Packard, 3,7% DEC, 3,7% Wang i 25,4% pozostałe, wśród których było wiele kompatybilnych z IMB. Według innego źródła (International Herald Tribune z 15 kwietnia 1985 r.) udział mikrokomputerów Apple jest oceniany wyżej — na 21%, ale IBM zdecydowanie przewodzi z 42% udziałem, przy czym prognozy na 1990 r.

Prof. dr hab. MICHAŁ KLEIBER pracuje w Instytucie Podstawowych Problemów Techniki PAN, jest specjalistą w zakresie mechaniki ciał odkształcalnych i zastosowań metod numerycznych w mechanice. W roku 1968 ukończył Wydział Inżynierii Lądowej Politechniki Warszawskiej, zaś w roku 1971 Wydział Matematyki i Mechaniki Uniwersytetu Warszawskiego. Doktorat obronił w 1972 roku, zaś habilitację w 1978. W latach 1975—1977 pracował na Uniwersytecie w Stuttgarcie (RFN), zaś w roku akademickim 1983, 1984 wykładał na Uniwersytecie Kalifornijskim w Berkeley (USA). Był zapraszany do wygłaszania generalnych referatów z zakresu zastosowań metod numerycznych w mechanice na międzynarodowych kongresach w Stuttgarcie w 1981 r. i Chicago w 1983 r. Jest autorem wielu programów z zakresu nieliniowej analizy złożonych układów konstrukcyjnych.

Mgr MACIEJ LEŚNY ukończył studia na Wydziale Ekonomii Uniwersytetu Warszawskiego (1969). W latach 1969—1975 pracuje w Zakładzie Informatyki Przemysłu Okrętowego w Gdańsku. W okresie 1975—1983 związany z siecią ZETO (ZETO Gdynia, OBRI Warszawa, ostatnio CPiZI-ZETO ZOWAR). W czasie pracy w CPiZI uczestniczy m.in. w pracach Komisji ds. Automatyzacji Systemu Rozliczeń Finansowych MKETO. Od 1983 roku w Komisji Planowania, w grupie ds. Modeli Ekonomicznych.

przewidują zwiększenie tego udziału do 60—70%. Mikrokomputery IBM serii PC, XT, AT stały się faktycznym standardem, który starają się kopiować lub przynajmniej naśladować setki firm na całym świecie. Pomimo że wielu producentów oferuje po konkurencyjnych cenach mikrokomputery przewyższające parametrami technicznymi komputery IBM, to jednak pozycja tej firmy na rynku ulega umocnieniu. Warto zacytować stwierdzenie prezesa firmy Tandy/ /Radio Shack, której mikrokomputery do niedawna zajmowały wysoką pozycję na rynku Stanów Zjednoczonych: *„Działamy w przemyśle technicznym, gdzie promocja i dystrybucja są ważniejsze niż technika"* (Computers and Electronics, 1984, Vol. 22, nr 9, s. 23). Występuje tu sytuacja podobna jak na rynku magnetowidów, gdzie doskonałe pod względem technicznym magnetowidy systemu Beta firmy Sony przegrywają zdecydowanie z magnetowidami systemu VHS, głównie dlatego, że kupujący myśląc o wymianie kaset wolą mieć ten sam system, co ich znajomi.

W przypadku mikrokomputerów dochodzi do tego pewność, że wszystkie nowe programy tworzone w niezależnych firmach są pisane głównie z myślą o IBM lub natychmiast adaptowane do mikrokomputerów IBM. Decydują o tym oczywiście zyski, tym większe im szerszy jest rynek użytkowników. Warto tu przytoczyć przykład innowacyjnego komputera Macintosh firmy Apple, który uznano za prawdziwą rewelację, głównie ze względu na oryginalny sposób przekazywania poleceń użytkownika przez urządzenie zwane myszką, które umożliwia szybki wybór na ekranie symboli graficznych (tzw. ikon) odpowiadających określonym poleceniom. Otóż w momencie wprowadzenia na rynek, oprogramowanie Macintosha składało się zaledwie z kilku programów, co powstrzymało ewentualnych użytkowników przed zakupem tego komputera. Dziś Macintosh ma dość bogate oprogramowanie, ale innowacyjne rozwiązania sprzętowe (myszka) i programowe (ikony) stały się dostępne także na komputerach IBM.

Przedstawione uwagi nie powinny jednak pozostawiać wrażenia, że na IBM zaczyna i kończy się świat mikrokomputerów profesjonalnych. Użytkownicy komputerów Apple mogą dostarczyć przekonywających dowodów, że przy znacznie niższej cenie mogą osiągać te same lub lepsze wyniki w różnych zastosowaniach, a użytkownicy mikrokomputerów domowych Commodore czy Sinclair ZX Spectrum lub QL będą mieli rację, gdy stwierdzą, że niewielkim kosztem mogą rozbudować te mikrokomputery tak, aby wykorzystać je w wielu zastosowaniach profesjonalnych. Przykład IBM ma posłużyć w tym artykule tylko jako pewien standard, do którego warto odnosić rozważania dotyczące wyboru sprzętu.

Takimi standardami są z pewnością klawiatura i monitor ekranowy IBM. Użytkownik profesjonalny powinien zdecydować się na wybór klawiatury o normalnych wciskanych, sprężynujących klawiszach, w układzie takim jak w maszynie do pisania (amerykański system QWERTY), z dodatkowymi klawiszami numerycznymi, kierunkowymi i funkcyjnymi. W przypadku IBM PC klawisze numeryczne pokrywają się z kierunkowymi, co nie sprawia kłopotów w większości zastosowań. Prymitywna płaska klawiatura, taka jak w ZX81, lub nieco lepsza, podobna do stosowanej w kalkulatorach klawiatura Spectrum, nie nadaje się do zastosowań związanych z pisaniem tekstów. Jedną z przyczyn początkowego niepowodzenia sprzedaży domowego komputera IBM PCjr była właśnie „oszczędna" klawiatura. Klawiatura IBM PC zebrała także nieco uwag krytycznych ze względu na inne położenie kilku klawiszy niż w maszynach do pisania tej samej firmy, oraz ze względu na brak możliwości rozpoznania klawiszy, zmieniających litery małe na duże i klawisze kierunkowe na numeryczne. Klawiatury dostarczane z mikrokomputerami naśladującymi IBM posiadają z reguły wbudowane diody świetlne informujące o stanie zmieniaczy.

Monitor ekranowy do zastosowań profesjonalnych powinien umożliwiać wyświetlanie przynajmniej 25 wierszy po 80 znaków w wierszu. Telewizory, które doskonale mogą zastępować monitor przy pracy z komputerami domowymi, pozwalają na rozróżnienie 40 znaków w wierszu (w specjalnych rozwiązaniach 64 znaki), a niektóre telewizory ze specjalnym wejściem monitorowym (oddzielne wejście video) umożliwiają odczyt 80 znaków, lecz tylko czarno-białych. Do pracy profesjonalnej w kolorze nadają się praktycznie tylko monitory RGB. W większości zastosowań niespecjalistycznych kolor nie jest jednak niezbędny. Przy pracy w trybie graficznym wystarczająca jest rozdzielczość monitora taka jak w komputerach IBM PC (640×200 elementów obrazu) czy Macintosh (512×342), choć z pewnością do zastosowań specjalistycznych niezbędna jest większa rozdzielczość, np. 1024×1024. Tryb pracy i jakość obrazu zależą też od układów elektronicznych komputera, które umieszczane są zwykle na wymiennych płytkach (kartach). Przykładowo, w IBM PC można stosować kartę monochromatyczną, która współpracuje ze specjalnym monitorem TTL tylko w trybie tekstowym, wyświetlając znaki alfanumeryczne na polach składających się z 14×9 punktów. Natomiast przy użyciu karty graficznej możliwe jest zastosowanie różnorodnych monitorów, ale jakość wyświetlanych znaków w trybie tekstowym jest niższa (8×8 punktów). Wielu niezależnych producentów oferuje do IBM PC karty umożliwiające pracę z monitorem TTL w trybie tekstowym i graficznym. W tym miejscu warto podkreślić, że jedną z tajemnic sukcesu IBM jest opublikowanie przez tę firmę pełnej dokumentacji technicznej IBM PC, co niezależnym producentom wyposażenia pozwoliło natychmiast opracować setki elementów rozszerzających system bazowy, a użytkownikom otworzyło możliwości skompletowania systemu dostosowanego do ich wymagań.

Znaki alfanumeryczne generowane są zwykle układowo, trudno więc oczekiwać, żeby produkowany za granicą sprzęt miał możliwość wyświetlania znaków charakterystycznych dla języka polskiego. Wiele mikrokomputerów, np. IBM PC, umożliwia jednak generowanie dowolnych znaków w trybie graficznym. Problem polega tylko na zakupie lub opracowaniu odpowiedniego oprogramowania wykorzystującego ten tryb pracy do redagowania tekstów. Można też wykorzystać karty dostarczane przez producentów niezależnych, np. Hercules, gdzie generator znaków jest oparty na układzie EPROM, co pozwala zmodyfikować zestaw znaków wyświetlanych na ekranie w trybie tekstowym.

Pamięć wewnętrzna komputera do zastosowań profesjonalnych nie powinna być mniejsza od 256 KB, ze względu na wymagania wielu rozpowszechnionych programów. Ostatnio pojawiły się pożyteczne programy, które wymagają 512 KB. Ta ostatnia wielkość pamięci (lub 640 K) jest optymalna ze względu na możliwość symulowania dysku w pamięci, co wielokrotnie przyspiesza operacje na plikach.

Pamięć zawnętrzną powinny stanowić przynajmniej dwie stacje dysków elastycznych. Kasety magnetofonowe nie zapewniają odpowiednio szybkiego dostępu do informacji. W zasadzie, jedna stacja dyskietkowa jest w większości zastosowań wystarczająca, ale manipulacja dyskietkami zajmuje wówczas zbyt wiele czasu użytkownika. Przy operacji na wielkich zbiorach danych niezbędny jest dysk twardy, a do niego odpowiednie urządzenie kopiujące (ang. streamer). Powiększa to jednak znacznie koszt zestawu. Na ogół, dyskietki dwustronne o podwójnej gęstości zapisu, takie jak stosuje się w IBM PC (po 360 KB pod nadzorem systemu operacyjnego DOS 2.0), wystarczają do niespecjalistycznych zastosowań, choć doświadczenie uczy, że pamięci i dyskietek nigdy nie jest zbyt wiele.

W zastosowaniach uniwersalnych nieodzowna jest drukarka graficzna, która umożliwia przeniesienie dowolnego obrazu i tekstu z ekranu na papier. Drukarka powinna pracować na zwykłym papierze maszynowym w pojedynczych

Dr inż. ROMUALD SZUNIEWICZ ukończył w 1970 r. Wydział Chemiczny Politechniki Warszawskiej, specjalność projektowanie technologiczne, a następnie studia podyplomowe w zakresie automatyki na Wydziale Elektroniki PW (w 1972 r.) oraz ekonomiki przemysłu i informatyki na Wydziale Ekonomiki Produkcji SGPiS (w 1982 r.). Podczas pracy w Resortowym OBR Automatyzacji Procesów Chemicznych „Chemoautomatyka" zajmował się komputerowym sterowaniem procesów chemicznych. Od 1979 r. w Komisji Planowania przy Radzie Ministrów zajmuje się planowaniem długookresowym rozwoju gospodarczego kraju. W 1979 r. obronił doktorat w Instytucie Inżynierii Chemicznej PW. W roku akademickim 1983, 84 uzyskał stypendium Fulbrighta w Stanach Zjednoczonych, na Uniwersytecie Delaware.

arkuszach formatu A3 lub przynajmniej A4. Większość nowoczesnych drukarek graficznych ma możliwość uzyskania druku jakości listowej. Ze względu na znaki charakterystyczne dla języka polskiego, warto jest przy zakupie drukarki zwrócić uwagę na możliwość definiowania części znaków przez użytkownika (tzw. download characters). Ze względów eksploatacyjnych trzeba też pamiętać o koszcie kaset z taśmą lub pojemników z tuszem (do drukarek strumieniowych). Do niedrogich drukarek graficznych, w których nie używa się kaset lecz szpul taśmy 13 mm, takiej jak w polskich maszynach do pisania, należą drukarki produkowane przez Star Micronics, np. seria Gemini. Do pracy polegającej wyłącznie na pisaniu i redagowaniu tekstów najlepsze są drukarki rozetkowe, w których czcionki są rozmieszczone na płatkach wymiennych rozetek.

Połączenia komputera z drukarką dokonuje się najczęściej przez sprzęg równoległy typu Centronics. Drukarki ze sprzęgiem szeregowym RS-232 C są nieco droższe. Niezależnie od drukarki, wyjście szeregowe jest pożyteczne ze względu na możliwość bezpośredniego łączenia komputerów lub dołączania przez modem do linii telefonicznej.

Stosowanie innych urządzeń, takich jak pisaki xy, pióra świetlne, myszki, manetki itp., zależy od indywidualnych potrzeb lub przyzwyczajeń użytkowników. Z rzeczy, które nie są niezbędne, lecz są pożyteczne, należy wymienić zegar zasilany z baterii, który uwalnia użytkownika od nużącego wczytywania aktualnej daty i godziny przy każdym ładowaniu systemu operacyjnego.

OPROGRAMOWANIE UŻYTKOWE

Przy omawianiu oprogramowania użytkowego trzeba podkreślić, że przełom w oprogramowaniu został spowodowany przede wszystkim szerokim rozpowszechnieniem sprzętu. Olbrzymi, choć chłonny i zyskowny rynek użytkowników indywidualnych, który otworzył się przed programistami, wymaga dostarczenia oprogramowania „przyjaznego", a więc takiego, które może być stosowane bez znajomości informatyki, a także — o ile to możliwe — bez potrzeby odwoływania się do drukowanych instrukcji. Osiągnięto to, dzięki wprowadzeniu techniki menu, polegającej na przedstawieniu użytkownikowi do wyboru na ekranie monitora różnych opcji odpowiadających określonym poleceniom lub instrukcjom. Taka przygotowana lista poleceń może występować w strukturze hierarchicznej, gdzie wybór polecenia otwiera kolejne, bardziej szczegółowe opcje. Ze względu na skrótowość haseł reprezentujących różne opcje, są one z reguły uzupełniane wyświetlanymi na życzenie instrukcjami. Dostępne są też zapisane na dyskietkach samouczki, wprowadzające w praktyczne użytkowanie oprogramowania.

Można bez przesady stwierdzić, że dziś komputerem osobistym może posługiwać się każdy, kto nie obawia się włączyć go do sieci, potrafi pisać na maszynie (przy czym sprawność manualna nie ma tu większego znaczenia) i opanuje podstawowy zasób słownictwa angielskiego w zakresie oprogramowania użytkowego. Dzięki łatwości posługiwania się komputerem osobistym, znika rezerwa, z jaką menadżerowie traktowali dotąd komputery.

PROGRAMY REDAGOWANIA TEKSTÓW

Najbardziej uniwersalnym zastosowaniem profesjonalnym komputerów osobistych jest ich użycie do pisania i redagowania tekstów. Programy pisania i redagowania tekstów noszą ogólną nazwę edytorów tekstowych (ang. text editors) lub programów przetwarzania tekstów (word processors). Cechą charakterystyczną nowoczesnych programów redagowania tekstów jest praca ekranowa, tzn. kontrola redagowanego tekstu na ekranie w sposób interakcyjny. Modnym hasłem reklamowym wielu programów redagowania tekstów jest *„masz, to co widzisz"*, a więc obietnica, że tekst na ekranie odpowiada dokładnie tekstowi drukowanemu. Jest to z reguły obietnica przesadzona, albowiem możliwości drukarki różne są od możliwości monitora i przykładowo zmiana kroju czcionki jest na ekranie tylko sygnalizowana odpowiednim znakiem kontrolnym.

Ze względu na rozmiary i zakres tematyczny tego artykułu, trudno jest omówić w pełni systematykę i różne odmiany programów pisania i redagowania tekstów. Jak IBM PC wśród mikrokomputerów, tak program Wordstar firmy MicroPro wyznacza pewien standard wśród programów redagujących i warto go przyjąć jako punkt odniesienia przy omawianiu techniki przetwarzania tekstów.

Operacje przeprowadzane na tekstach można podzielić na cztery zasadnicze grupy, odpowiadające czterem różnym zbiorom instrukcji programów Wordstar.

Do pierwszej grupy należą operacje na zbiorach (dokumentach tekstowych), takie jak: otwarcie dokumentu (przez nadanie nazwy), zmiana nazwy, łączenie zbiorów przez skopiowanie całości lub części jednego zbioru do innego zbioru, skasowanie, wydrukowanie, wyświetlenie spisu dokumentów itp.

Do drugiej grupy należą operacje pisania tekstów i wprowadzania zmian, które polegają między innymi na przemieszczaniu lub usuwaniu wybranych fragmentów tekstu, wstawianiu w dowolnym miejscu nowych liter, słów lub całych fragmentów, przeglądaniu całego dokumentu wiersz po wierszu lub strona (ekranu) po stronie. Tekst wprowadzany jest z klawiatury, tak jak na maszynie do pisania i natychmiast wyświetlany na ekranie. Objętość dokumentu ograniczona jest wielkością dostępnej pamięci (pamięć 360 KB mieści ok. 200 stron maszynopisu formatu A4), ale praktycznie ze względu na czasochłonność operacji lepiej jest dzielić dłuższe dokumenty na odrębne opracowywane fragmenty, np. rozdziały książki. Szerokość szpalty może wynosić do 256 znaków. Po przekroczeniu wybranej szerokości szpalty tekst jest automatycznie przenoszony do następnego wiersza. Na ekranie wyświetlany jest wybrany przez użytkownika fragment o objętości do 22 wierszy po 79 znaków. Do wskazywania wybranego miejsca tekstu służy plamka świetlna nazywana kursorem, która sterowana jest przy użyciu klawiszy kierunkowych. Program umożliwia automatyczne wyszukiwanie w tekście podanych przez użytkownika słów (dowolnych ciągów znaków) i zastąpienie ich innymi.

Do trzeciej grupy należą operacje formatowania ekranowego, polegające na zmianie lewego i prawego marginesu, wyrównaniu tekstu do prawego marginesu z dzieleniem przenoszonych wyrazów lub bez ich dzielenia, środkowaniu tekstu (automatycznym umieszczaniu na środku strony). Połączenie operacji redagowania i formatowania pozwala uzyskać tekst kilkuszpaltowy z wyrównanymi brzegami (rys. 1).

Do czwartej grupy należą operacje formatowania wydruku, polegające na umieszczeniu w tekście znaków sterujących drukarką dla uzyskania indeksów dolnych i górnych, podkreśleń, różnego kroju czcionek, odstępów linii i liczby znaków w wierszu.

Do trzeciej grupy należą operacje formatowania ekranowego polegające na zmianie lewego i prawego marginesu, wyrównaniu tekstu do prawego marginesu z dzieleniem przenoszonych wyrazów lub bez ich dzielenia, centrowaniu tekstu (automatycznym umieszczeniu na środku strony). Połączenie operacji redagowania i formatowania pozwala uzyskać tekst kilkuszpaltowy z wyrównanymi brzegami (rys.1).

Do czwartej grupy należą operacje formatowania wydruku polegające na umieszczeniu w tekście znaków sterujących drukarką dla uzyskania takich efektow jak subskrypty i superskrypty, podkreślenia, różne kroje czcionek, odstęp linii i liczba znaków w wierszu.

W oryginalnym programie Wordstar nie ma możliwości pracy w trybie graficznym, dlatego też zmiana postaci wydruku sygnalizowana jest na ekranie symbolami odpowiadającymi znakom sterującym drukarką. Istnieją jednak programy, np. StarPolish, które uzupełniają Wordstar pozwalając uzyskać na ekranie efekty takie jak na drukarce.

Rys. 1. Przykład wykorzystania programu redagującego do rozmieszczenia tekstu w układzie dwuszpaltowym

W oryginalnym programie Wordstar nie ma możliwości pracy w trybie graficznym, dlatego też zmiana postaci wydruku jest sygnalizowana na ekranie symbolami odpowiadającymi znakom sterującym drukarką. Istnieją jednak programy, np. StarPolish, które uzupełniają Wordstar, pozwalając uzyskać na ekranie efekty takie, jak na drukarce.

Wymienione grupy operacji są sterowane przy użyciu klawiszy kierunkowych, literowych i funkcyjnych. Istnieje możliwość łączenia poszczególnych instrukcji w ciągi wywoływane za pomocą poszczególnych klawiszy funkcyjnych. Przy wykorzystaniu odrębnego programu definiującego funkcje klawiszy, takiego jak Prokey firmy Rosesoft, można znacznie rozszerzyć możliwości pracy, przez przypisanie klawiszom lub ich kombinacjom kilkudziesięciu różnych ciągów często wykorzystywanych instrukcji, np. w celu przywołania i wstawienia do dokumentu dowolnych, często powtarzających się tekstów lub tablic. Możliwe jest też zdefiniowanie klawiszy tak, aby w połączeniu z innymi powodowały wysłanie na drukarkę znaków uruchamiających wydruk polską czcionką. Przykładowo, klawisz a użyty łącznie ze zmieniaczem (klawiszem o nazwie **Alt**) będzie interpretowany przez drukarkę jako polska litera ą, na ekranie jednak wyświetlony będzie symbol wybrany przez użytkownika do reprezentowania tej polskiej litery.

Program Wordstar może być rozszerzony o programy: łączenia tekstów z kolejnymi elementami listy, np. przy drukowaniu korespondencji do różnych adresatów z listy na podstawie jednego standardowego tekstu (program MailMerge), sprawdzania poprawności ortograficznej tekstów angielskich (programy Spelstar, Corrstar), przygotowania indeksów (program StarIndex) i inne.

PROGRAMY OBLICZEŃ TABLICOWYCH

Drugą obok programów pisania i redagowania tekstów najszerzej stosowaną grupą programów mikrokomputerowych są programy obliczeń tablicowych (w języku angielskim nazywane arkuszami elektronicznymi lub programami planowania finansowego). Pierwszy z tych programów, o nazwie Visicalc opracowany w 1979 r. na komputer Apple, przyczynił się do rozwoju masowej sprzedaży komputerów osobistych. Obecnie, najbardziej popularny stał się 1-2-3 firmy Lotus, który wyznacza poziom odniesienia dla innych programów z tej grupy, a jego wersja na IBM PC jest przez wielu użytkowników wykorzystywana do testowania programowej kompatybilności innych komputerów z IBM.

Program 1-2-3 pozwala na wprowadzenie danych liczbowych i tekstu z klawiatury na ekran, reprezentujący fragment tablicy. Tablica ma wymiary 256 kolumn na 2048 wierszy. Na ekranie wyświetlane jest 25 wierszy, a liczba kolumn zależy od ustalonej przez użytkownika szerokości kolumny. Każdy element tablicy może pomieścić do 240 znaków alfanumerycznych. Wyboru elementu tablicy, do którego mają być wpisane dane, dokonuje się przez wskazanie tego elementu kursorem, przy użyciu klawiszy kierunkowych. Przeglądanie tablicy odbywa się przez przesuwanie kursora do odpowiedniego elementu tablicy.

Elementy tablicy mogą być wyliczane z innych elementów (wielkości), przez wskazanie tych elementów na ekranie przy użyciu kursora i wpisanie z klawiatury symbolu działania (operatora) lub zależności funkcyjnej. Do zaprogramowania złożonego modelu matematycznego nie jest wymagana znajomość żadnego języka programowania. Zapisywanie wielkości w postaci tablicowej czyni szczególnie prostym formułowanie zależności rekurencyjnych. Możliwe jest prowadzenie obliczeń iteracyjnych. W programie zdefiniowanych jest wiele funkcji matematycznych, logicznych i niektóre funkcje statystyczne (np. średnia, wariancja), które użytkownik wywołuje, wpisując ich nazwę i wskazując kursorem zakres argumentów, do których się odnoszą.

Dowolne szeregi liczb (elementów tablicy) mogą być wykorzystane do zbudowania różnego rodzaju wykresów, które wyświetlane są na ekranie na życzenie użytkownika. Każda zmiana w danych liczbowych, wprowadzona z klawiatury na ekran, powoduje natychmiastowe przeliczenie wielkości zależności i zmianę przebiegu wykresu. W sposób interakcyjny użytkownik może więc badać odpowiedź modelu na zmianę danych i korygować też postać modelu. Istnieje możliwość automatycznego wielokrotnego przeliczania modelu dla różnych danych wejściowych i zestawienia wyników w tablicę.

Wszelkie operacje wykonywane przez użytkownika wywołane są przez wybór z menu przedstawionego na ekranie. W wypadku wątpliwości, użytkownik może wywołać na ekranie tekst instrukcji, która odpowiada wybranej pozycji menu lub może odczytać dowolne inne instrukcje. Dowolny fragment tablicy oraz dowolne wykresy mogą być wydrukowane na żądanie użytkownika. Jeśli wielkość tablicy przekracza rozmiary arkusza, na którym dokonywany jest wydruk, wówczas można wybrać mniejszy krój czcionki, który umożliwia wydrukowanie do 132 znaków w wierszu formatu A4. Jeszcze większe tablice są drukowane w ten sposób, że poszczególne kartki wydruku mają margines umożliwiający sklejenie z nich dużego arkusza, o ile zachodzi taka potrzeba. Tekst na wykresach może być wydrukowany z kilku krojów czcionki.

ANALIZA KSZTAŁTOWANIA SIĘ ZADŁUŻENIA

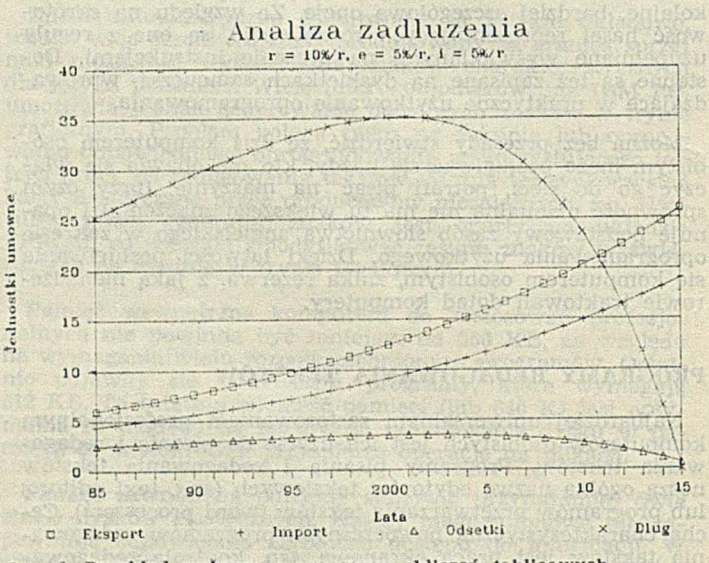
Założona stopa wzrostu %/r
Oprocentowanie 10.00%
Eksport 5.00%
Import 5.00%

Lata	Eksport	Import	Saldo	Odsetki	Dług na koniec roku	
1985	6.0	4.4	1.6	2.5	25.3	85
1986	6.3	4.6	1.7	2.5	26.1	
1987	6.6	4.9	1.8	2.6	27.0	
1988	6.9	5.1	1.9	2.7	27.8	
1989	7.3	5.3	1.9	2.8	28.7	
1990	7.7	5.6	2.0	2.9	29.5	90
1991	8.0	5.9	2.1	2.9	30.3	
1992	8.4	6.2	2.3	3.0	31.1	
1993	8.9	6.5	2.4	3.1	31.8	
1994	9.3	6.8	2.5	3.2	32.5	
1995	9.8	7.2	2.6	3.3	33.2	95
1996	10.3	7.5	2.7	3.3	33.7	
1997	10.8	7.9	2.9	3.4	34.2	
1998	11.3	8.3	3.0	3.4	34.6	
1999	11.9	8.7	3.2	3.5	34.9	
2000	12.5	9.1	3.3	3.5	35.1	2000
2001	13.1	9.6	3.5	3.5	35.1	
2002	13.8	10.1	3.7	3.5	35.0	
2003	14.4	10.6	3.9	3.5	34.6	
2004	15.2	11.1	4.0	3.5	34.0	
2005	15.9	11.7	4.2	3.4	33.2	5
2006	16.7	12.3	4.5	3.3	32.0	
2007	17.6	12.9	4.7	3.2	30.6	
2008	18.4	13.5	4.9	3.1	28.7	
2009	19.4	14.2	5.2	2.9	26.4	
2010	20.3	14.9	5.4	2.6	23.6	10
2011	21.3	15.6	5.7	2.4	20.3	
2012	22.4	16.4	6.0	2.0	16.4	
2013	23.5	17.2	6.3	1.6	11.7	
2014	24.7	18.1	6.6	1.2	6.3	
2015	25.9	19.0	6.9	0.6	.0	15
Suma	424.6	311.3	113.2	88.9		

Analiza wrażliwości długu w 2015 r. w zależności od tempa wzrostu eksportu - e i importu -i , w %/r

	+DŁUG	import 4.80%	import 4.90%	import 5.00%	import 5.10%	import 5.20%
eksport	4.80%	10.1	23.7	37.6	51.7	66.1
eksport	4.90%	-8.6	5.1	19.0	33.1	47.4
eksport	5.00%	-27.5	-13.9	.0	14.2	28.5
eksport	5.10%	-46.8	-33.1	-19.2	-5.1	9.2
eksport	5.20%	-66.3	-52.7	-38.8	-24.7	-10.3

Rys. 2. Przykład wydruku z programu obliczeń tablicowych *)
***) Użyte tu wielkości liczbowe i proporcje między nimi są wybrane jedynie dla ilustracji obliczeń tablicowych i nie mają odniesienia do rzeczywistych wielkości ekonomicznych**



Rys. 3. Przykład wykresu z programu obliczeń tablicowych

Przykład obliczeń tablicowych przedstawiono na rysunkach 2 i 3. Zaprogramowanie rekurencyjnego obliczenia zmian zadłużenia do 2015 r. zajęło autorom niewiele więcej czasu niż wpisanie wielkości wejściowych i podanie jak wielkości w następnym roku zależą od wielkości wejściowych. Można stwierdzić, że zaprogramowanie tego przykładu

nie było trudniejsze niż ręczne obliczenie zmiany zadłużenia dla jednego roku. Przy ręcznych obliczeniach, zmiana założeń wymaga powtórzenia całego procesu obliczeń. Natomiast przy użyciu programu przeliczenie dowolnego wariantu dla różnych wielkości wejściowych zajmuje praktycznie tyle czasu, ile trwa wpisanie nowej wartości z klawiatury, przy czym można też uzyskać wyniki dla wielu wariantów jednocześnie, co pokazuje tablica wrażliwości na rys. 2. Przykład ten stanowi doskonałe potwierdzenie niedawno przeprowadzonego porównania: dawniej trzeba było poświęcić tydzień pracy, aby przygotować pięciostronicowy raport czytany następnie w ciągu pięciu minut. Teraz wystarczy pięć minut, aby otrzymać 50-stronicowy raport, który będzie następnie czytany przez tydzień.

JANUSZ STOKŁOSA
Politechnika Poznańska

Abraham Stern
pierwszy polski konstruktor maszyn liczących

Schickard, Pascal, Leibniz i Babbage znani są jako pierwsi wynalazcy maszyn do liczenia [1, 2]. Mniej znani są polscy prekursorzy informatyki; za pierwszego jest uważany Abraham Stern.

Abraham Stern urodził się w Hrubieszowie w roku 1769. Oddany na naukę do zegarmistrza, zwrócił na siebie uwagę Stanisława Staszica, w którym zyskał protektora. Z jego inspiracji, jako samouk zgłębiał matematykę. W latach 1808—1826 Staszic pełnił funkcję prezesa Towarzystwa Królewskiego Warszawskiego Przyjaciół Nauk, które w warunkach rozbiorów zapoczątkowało realizację idei Polskiej Akademii Nauk. Jego członkowie pochodzili z całego obszaru dawnej Rzeczypospolitej i choć główną działalność rozwijano w Warszawie, utrzymywano również kontakty ze znanymi ośrodkami naukowymi w Wilnie i Krzemieńcu.

W grudniu 1812 roku Stern zwrócił się do Towarzystwa z prośbą o ocenę jego czterodziałaniowej „machiny arytmetycznej". W styczniu 1813 roku recenzenci przedstawili opinię, w której czytamy [3]: *„Machina ta na rozmaite zagadnienia deputacyi, co do dodawania, odciągania, mnożenia i dzielenia, odpowiedziała z wszelką dokładnością, tak, że i ułamki, jakie pozostają z niepodzielnej liczby w dzieleniu, wskazała (...). Po rozebraniu tej machiny przekonała się deputacya, o niezawodności onej, a tak i do rzeczy, jako i mechanizmu samego wynalazca onej na wielkie pochwały zasługuje".* Dalej następuje opis arytmometru: *„Machina ta ma kształt skrzyneczki, czyli równoległościanu. Znajdują się w niej na wierzchu trzy rzędy z cyferblatami złożone. Każdy cyferblat podzielony jest na dziesięć części, dla umieszczenia naokoło brzegu onego wszystkich jedności i zera. Pierwszy cyferblat po prawej ręce stanowi jedności, drugi dziesiątki, trzeci — sta itd. Każdy cyferblat będąc poziemnie osadzony, obraca się naokoło swej osi. Cyferblaty te pokryte są blaszkami z okienkami, w pewnych odstępach nad cyferblatami, na które to okienka żądane cyfry nakręcają się, we wszystkich innych zera zostawując. Dwa rzędy takich cyferblatów stanowią zagadnienie, a trzeci — wypadek wskazuje. Średni rząd cyferblatów, na którym najwięcej zależy i przy którym korba do obracania jest umieszczona, w półokręgu tylko ma jedności umieszczone, pod którymi sztyfty ruchome, na dół i do góry iść mogące, danemi są. Te sztyfty początkiem są całej sztuki, albowiem tyle onych wypadnie na dół, jaką cyfrę pod okienko podsunie się (...)".* W konkluzji, recenzenci zalecają uproszczenie maszyny.

Stern kontynuuje prace. Uzyskuje zasiłek Towarzystwa i pensję rządową, którą mu Towarzystwo wyjednało. W roku 1817 ma już udoskonaloną wersję maszyny, która oprócz dodawania, odejmowania, mnożenia i dzielenia wykonywała także pierwiastkowanie i umożliwiała sprawdzanie wyników. Na posiedzeniu Towarzystwa 30 kwietnia tegoż roku przedstawia „Rozprawę o machinie arytmetycznéy połączonéj z machiną do wyciągania piérwiastków z ułomkami" mówiąc [5]: *„(...) ułożyłem sobie, powtorną Machinę z Metalu sposobem mocnym i trwałym, z wszelką dokładnością zrobić. A chociaż takowe przedsięwzięcie, osobliwie w pierwiastkowym swym stanie, czasu i znacznego funduszu na opędzenie kosztów wymagało, czego ieszcze owczesne krytyczne woienne położenie kraju polskiego, którego iestem rodakiem, trudniejszym dla mnie uczyniło, przecież nieoszczędzając z méy strony usiłowań, to moie oświadczenie uskuteczniłem (...)".*

Był już wtedy (od 9 lutego 1817 roku) członkiem korespondentem Towarzystwa. Pracował nad innymi wynalazkami. W listopadzie 1818 roku przedkłada rozprawę o trzech nowych maszynach: młockarni, tartaku i żniwiarce [6], a w maju 1820 roku, wespół z J. K. Skrodzkim, opinię o projekcie anonimowego autora, w której zamieszczono wyniki eksperymentów z żelaznym łańcuchem. Łańcuchy takie miały być stosowane przy budowie projektowanego mostu na Wiśle.

Potem, w 1821 roku przedstawił Towarzystwu model nowego urządzenia, które nazwał „wózkiem topograficznym". Maszyna była przeznaczona „do mierzenia gruntów i razem rysowania ich figur" [4]. Testy wykonywano na podwórku uniwersyteckim. W tymże roku, 4 lutego został członkiem przybranym Towarzystwa [9]. W roku następnym zaprezentował „narzędzie swego wynalazku służące do dochodzenia odległości punktów niedostępnych i zdejmowania planów na ziemi z jednego punktu bez rachunku trygonometrycznego" [4], a w roku 1827 mówił o udoskonalonej przez siebie maszynie do żęcia.

W dniu 3 stycznia 1830 roku Abraham Stern awansował na członka czynnego Towarzystwa Przyjaciół Nauk. *„Oświadcz wdzięczność moją Towarzystwu naszemu za ten wymiar sprawiedliwości (...)"* — pisał generał, a jednocześnie dramatopisarz i chemik, Aleksander hr. Chodkiewicz, do

Antoni Blank: Portret Abrahama Sterna (1823 r.) Muzeum Narodowe w Poznaniu

ks. Edwarda Czarneckiego [3]. Po styczniowych wyborach Towarzystwo liczyło 57 członków czynnych, 35 przybranych, 48 honorowych i 88 korespondentów. Wskutek represji po powstaniu listopadowym, w roku 1832 zostało ono rozwiązane dzieląc los działającego od 1816 roku Uniwersytetu Warszawskiego. Stern nie zaprzestał jednak działalności konstruktorskiej. W roku 1836 wynalazł jeszcze *„pełen prostoty mechanizm, ochraniający w rozbieganiu się koni tak sam powóz, jakoteż i osoby w nim siedzące"* [9].

Oprócz rozpraw naukowych, pisanych w języku polskim, Stern uprawiał również działalność literacką, w tym również poetycką w języku hebrajskim.

Był również pierwszym rektorem (w latach 1826—1835) Warszawskiej Szkoły Rabinów, jedynej żydowskiej szkoły średniej w Królestwie Polskim (1826—1863). Przedmioty ogólne (wśród nich matematyka, historia, geografia) były w niej wykładane w języku polskim. Stern pełnił jednak tę funkcję wyłącznie nominalnie, nie chcąc odrywać się od swojej pracy naukowej i literackiej [7]. Zmarł w Warszawie 2 lutego 1842 roku.

Wynalazki Sterna, wśród nich maszyna arytmetyczna, nie znalazły praktycznego zastosowania. Nie znalazł się nikt, kto by podjął się ich produkcji.

Kontynuatorem myśli konstruktorskiej Sterna był tylko jego zięć Chaim Zelig Słonimski (1810—1904), który za pracę naukową dotyczącą ulepszonej wersji maszyny arytmetycznej Sterna, uzyskał w roku 1844 nagrodę Akademii Nauk w Petersburgu. Wynalazł także „sposób przekazywania 4 telegramów na jednym przewodzie" [10].

Na zakończenie warto przytoczyć prorocze fragmenty przemówienia, jakie Stern wygłosił w 1818 r. na posiedzeniu Towarzystwa [6]:

„Słabość sił fizycznych człowieka dowodzi, że przyrodzenie rozkazało mu siłami umysłu więcej niż siłami ciała pracować. Dążyć on więc powinien do rozszerzenia granic mechaniki: za nią w krok postępują bogactwa i potęga państw, w których ona jest uprawiana. (...) Człowiek powinien tworzyć machiny i nimi kierować, a one wyręczać go w uciążliwej pracy. Narody, które wydoskonalił przemysł, panują nad światem, te zaś, które go zaniedbały, popadły w słabość, ciemnotę, ubóstwo i niewolę".

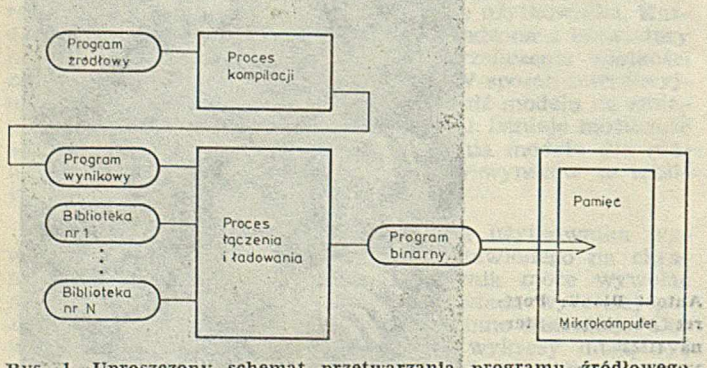
LITERATURA

[1] Czyżo E., Matusek T.: Prekursorzy współczesnej inofrmatyki. Informatyka, nr 11, 1984, str. 1—4
[2] Kaufmann H.: Dzieje komputerów. PWN, Warszawa, 1980
[3] Kraushar A.: Towarzystwo Warszawskie Przyjaciół Nauk, ks. II, 1807—1815. Gebethner, Kraków, 1902
[4] Michalski J.: Z dziejów Towarzystwa Przyjaciół Nauk. TNW, Warszawa, 1953
[5] Stern A.: Rozprawa o machinie arytmetycznéy. Roczniki Towarzystwa Królewskiego Warszawskeigo Przyjaciół Nauk, t. 12, Warszawa, 1818, str. 106—127
[6] Stern A.: Rozprawa o trzech nowych machinach: to iest młockarni, tartaku i do żęcia zboża. Roczniki Towarzystwa Królewskiego Warszawskiego Przyjaciół Nauk, t. 13, Warszawa, 1820, str. 42—55
[7] Strzemski M., Warszawska Szkoła Rabinów (1826—1863) najdziwniejsza w świecie. Znak, nr 339—340, 1983, str. 361—364
[8] Świderska E., Tendencje społeczno-kulturowe wśród Żydów polskich w XIX wieku. Znak, nr 339—340, 1983, str. 344—353
[9] Wójcicki K. W., Abraham Stern. Tygodnik Ilustrowany, nr 248, Warszawa, 25 czerwca 1864, str. 233—234
[10] Żydzi polscy. Dzieje i kultura. Interpress, Warszawa, 1982.

MARIUSZ POSTÓŁ
Zakład Badań Podstawowych Elektrotechniki
Ministerstwa Hutnictwa i Przemysłu Maszynowego
i Polskiej Akademii Nauk

Modyfikacja procesu kompilacji PASCALA/MT+

Program wynikowy otrzymany w wyniku kompilacji programu źródłowego przez kompilator PASCAL/MT+ nie stanowi jeszcze samodzielnej całości, którą można umieścić w pamięci i zainicjować. Wymagane jest uprzednie dołączenie niezbędnych modułów, znajdujących się w bibliotekach systemowych i użytkowych, oraz przekształcenie wyniku kompilacji do postaci binarnej o adresach bezwzględnych. Proces ten jest wykonywany przez specjalny program łączący (ang. linker). Pełny przebieg procesu przetwarzania programu źródłowego, z uwzględnieniem etapu kompilacji i łączenia, przedstawiono na rys. 1. W trakcie procesu łączenia możliwe jest takie rozmieszczenie programu i jego danych, aby można następnie wpisać go do pamięci stałej ROM lub do pamięci o dostępie swobodnym RAM. Wykorzystanie pamięci ROM do przechowywania programu binarnego jest uzasadnione w przypadku mikrokomputerów specjalizowanych, gdy nie zachodzi potrzeba zbyt częstych zmian jego funkcji (np. sterowniki). Powyższy warunek jest spełniony tylko w trakcie eksploatacji urządzenia, po uruchomieniu i przetestowaniu programu. w tego rodzaju urządzeniach, pamięci stałe ROM pozwalają wyeliminować kosztowne pamięci masowe na dyskach elastycznych.



Rys. 1. Uproszczony schemat przetwarzania programu źródłowego z wykorzystaniem kompilatora PASCALA/MT+

MODYFIKACJA METODY KOMPILACJI, ŁĄCZENIA I ŁADOWANIA PROGRAMU

Wykorzystanie pamięci stałych jako nośnika informacji (programu) w trakcie testowania jest bardzo niepraktyczne. Charakterystyczna dla tego procesu jest konieczność wprowadzania częstych zmian. Zmiany mogą dotyczyć programu źródłowego lub programu binarnego. W pierwszym wypadku, program musi być poddany ponownej kompilacji, co z reguły wymaga zmiany całej zawartości pamięci. Dru-

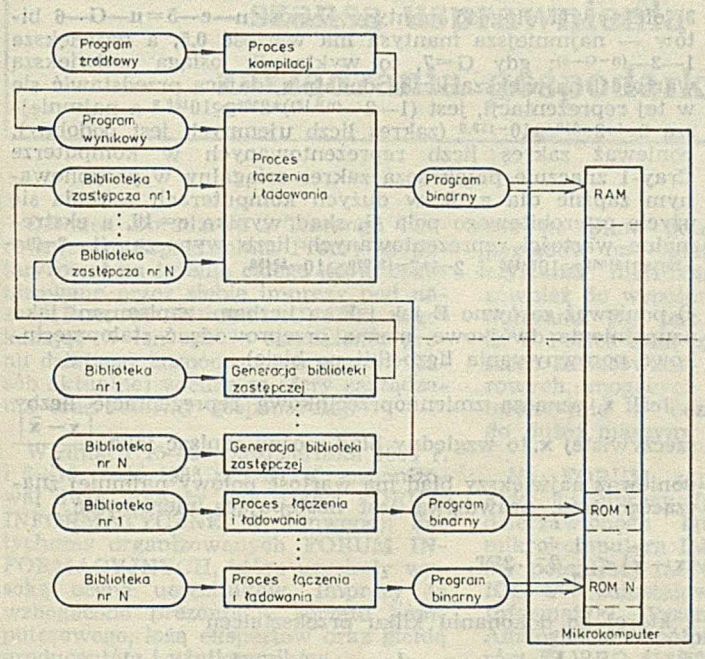
gi rodzaj zmian występuje najczęściej, gdy korzysta się z programów diagnostycznych. Zastosowanie w tym wypadku pamięci stałych w znacznym stopniu ogranicza możliwości tych programów. Dodatkowym ograniczeniem jest długi czas programowania i kasowania pamięci stałych — reprogramowanych oraz zmniejszenie ich niezawodności i trwałości przy częstych zmianach zawartości.

Dla kompilatorów języków wysokiego poziomu regułą jest, że tylko część (często nieznaczna) programu binarnego jest produktem kompilatora, a zatem — programisty. Pozostałą część stanowią moduły dołączane z bibliotek systemowych. Ponieważ nie wymagają one testowania, nie ma potrzeby, aby przechowywać tę część programu w pamięci RAM. Prowadzi to do wniosku, że program binarny można podzielić na trzy części:

- część reprezentującą program źródłowy
- część obejmującą procedury biblioteczne
- dane programu źródłowego i dane bibliotek

Części te można tak rozmieścić w pamięci, aby część pierwsza i trzecia znalazły się w pamięci RAM, natomiast część druga w pamięci ROM. Treść procedur bibliotecznych jest całkowicie niezależna od programu użytkownika, a więc program może być modyfikowany bez potrzeby zmiany zawartości pamięci stałej. Podobnie można postąpić z bibliotekami utworzonymi przez użytkownika. Przedstawiony sposób rozmieszczenia pozwala radykalnie zmniejszyć długość programu, którym programista musi operować w trakcie testowania.

Realizacja powyższych postulatów w procesie łączenia, zgodnie ze schematem przedstawionym na rys. 1, za pomocą standardowych programów łącząco-ładujących jest niemożliwa. Dlatego należy zmienić schemat postępowania przy tworzeniu programu binarnego i poddać program wynikowy procesowi łączenia niezależnie od łączenia bibliotek. Wymaga to określania adresów obiektów, znajdujących się w bibliotece, bez udziału tej biblioteki w procesie łączenia (rys. 1). W celu rozwiązania tego problemu, proponuje się zastąpienie bibliotek w procesie łączenia programu wynikowego przez ich namiastki, które zawierałyby jedynie definicje symboli globalnych.



Rys. 2. Proces przetwarzania programu źródłowego z wykorzystaniem bibliotek wpisanych do pamięci stałej

Schemat nowego postępowania przy przetwarzaniu programu źródłowego zilustrowano na rys. 2. Część dolna rysunku obejmuje wszystkie operacje, które mogą być wykonane tylko raz dla danej konfiguracji sprzętu, ponieważ nie bierze w nich udziału program wynikowy generowany przez kompilator. Część górna rysunku przedstawia natomiast operacje, które muszą być wykonane po każdej modyfikacji programu źródłowego.

Do generacji bibliotek zastępczych wykorzystuje się specjalny program, którego zasada działania jest podobna do zasady działania programu łącząco-ładującego. Oba programy odczytują treść biblioteki lub programu wynikowego i budują wewnętrznie tablicę symboli globalnych. Różnica w ich działaniu polega na tym, że pierwszy generuje tę tablicę w postaci biblioteki zastępczej, natomiast drugi używa jej tylko do generowania bezwzględnie adresowanego programu binarnego [1]).

W procesie łączenia, zgodnie ze schematem przedstawionym na rys. 1, program łącząco-ładujący dołącza do generowanego programu binarnego tylko niezbędne moduły z biblioteki. Użycie schematu z rys. 2 wymaga wpisania do pamięci całej biblioteki, ponieważ z góry nie wiadomo, które jej moduły będą niezbędne do realizacji programu. Kolejnym ograniczeniem stosowania opisanej metody rozmieszczania części programu w różnych obszarach pamięci jest przypadek, gdy biblioteka wymaga zdefiniowania przez program symbolu globalnego. Jest to sytuacja rzadko występująca w praktyce. Wymaga ona wyłączenia z biblioteki wszystkich modułów o wspomnianej właściwości i utworzenia z nich osobnej biblioteki, którą trzeba łączyć bezpośrednio z programem wynikowym.

WSPÓŁPRACA PROGRAMU Z OTOCZENIEM

Budowa bibliotek i programu wynikowego generowanego przez kompilator PASCAL/MT+ jest taka, aby program mógł być realizowany przez komputer wyposażony w system operacyjny CP/M. Jednakże system CP/M jest przeznaczony głównie do wytwarzania oprogramowania i rzadko implementowany na sprzęcie przeznaczonym do pracy w charakterze sterowników — ze względu na niefunkcjonalność w tego rodzaju zastosowaniach. Dlatego należy wprowadzić istotne zmiany w bibliotece PASLIB, odpowiedzialnej za współpracę z otoczeniem programowym i sprzętowym, tak aby program mógł być realizowany bez konieczności wykorzystywania systemu CP/M. Zasadnicze zmiany muszą objąć z tej biblioteki procedury:

▫INI, ▫BDOS, ▫HLT.

Procedura **▫INI** jest odpowiedzialna za inicjowanie zmiennych systemowych, a w szczególności zmiennych standardowych związanych z systemem we-wy. Procedura **▫HLT** jest wywoływana zawsze na zakończenie programu i służy do przekazania sterowania do systemu operacyjnego, po zakończeniu obliczeń. Procedura **▫BDOS** realizuje wszystkie funkcje związane ze współpracą z urządzeniami we-wy. Wymienione procedury muszą być opracowane dla każdego mikrokomputera oddzielnie. Obecnie istnieją ich wersje dla mikrokomputerów MSK i MM-80.

LITERATURA

[1] Pascal/MT+ Language Reference Manual. Digital Research, Pacific Grove (CA)
[2] Utility Software Package, Reference Manual Microsoft. Inc., Bellevue (WA).

[1]) Program taki opracowano w Zakładzie Badań Podstawowych Elektrotechniki

Dwie uogólnione zmiennoprzecinkowe reprezentacje liczb

Zakres i dokładność zmiennoprzecinkowej reprezentacji liczb są ważnym elementem oceny mocy obliczeniowej komputera. W związku z dynamicznym rozwojem technologii, a także mikroprogramowania, stosowanego w coraz większych i szybszych komputerach, możliwe stało się operowanie liczbami zmiennoprzecinkowymi o formacie innym niż tradycyjny (znak, wykładnik, mantysa).

W poniższym artykule opisano dwie modyfikacje reprezentacji zmiennoprzecinkowej. Pierwsza z nich jest tzw. reprezentacją podzieloną, w której zakres uzależniony jest od dokładności i na odwrót. Druga jest reprezentacją o zmiennej podstawie potęgowania.

Reprezentacja podzielona

Podzielona reprezentacja zmiennoprzecinkowa jest rozwinięciem [1]) trójpolowego formatu GEF, w którym **G**, **E** i **F** oznaczają wartości tych pól, a **g=3**, **e**, **f** są odpowiednimi długościami pól. Przy założeniu definicyjnym **e = G+1**, oznaczając przez **n** całkowitą długość słowa otrzymamy

f = n-3-e = n-G-4.

Pole o ustalonej długości **G** dzieli słowa na e-bitowy wykładnik ze znakiem i f-bitową mantysę ze znakiem. Taka postać słowa, oprócz niewątpliwej zalety, jaką jest wzajemna zależność dokładności i zakresu, ma kilka wad:

1) jeśli **G** jest małe (**G<4**), to wykorzystuje się mniej niż 3 bity; niewykorzystane bity w zasadzie mogą być użyte do zwiększenia długości wykładnika, co powoduje rozszerzenie zakresu

2) zakresy wykładnika dla różnych **G** kolidują ze sobą, tak więc, następujące dwie liczby:

010 011 x...x (G=2, E=3)
011 0011 x...x (G=3, E=3)

mają ten sam wykładnik; w celu uzyskania jednoznacznej reprezentacji liczb, wykładnik należy znormalizować przez przesunięcie w lewo i zmniejszenie **G**

3) znak całej liczby jest określony mniej znaczącym bitem pola **F** co jeszcze bardziej komplikuje porównywanie liczb, ponieważ nierówności

$G_1 > G_2$ i $E_1 > E_2$

nie gwarantują, że liczba $G_1E_1F_1$ jest większa od $G_2E_2F_2$; wynika stąd, że należy porównywać również F_1 i F_2, ale w tym wypadku porównywanie całkowitoliczbowe (bit po bicie) jest niemożliwe

4) bit występujący w sposób domyślny nie jest uwzględniany w trakcie obliczeń.

Powyższe wady (poza pierwszą) usuwa zapis liczb w postaci podzielonej o słowie SS_eGEF, gdzie **S** jest znakiem mantysy, a S_e znakiem wykładnika (proponowana konwencja: $S_e = 0$ dla liczb ujemnych, $S = 1$ dla nieujemnych). Symbol **G** jest g-bitowym polem określającym długość wykładnika jako **e=G+1**, a **E** — polem wykładnika zapisanym jako uzupełnienie dwójkowe. Efektywna wartość wykładnika jest zdefiniowana jako $E+(2^e-2)$, gdy $S_e=1$ i $E-(2^e-2)$ dla $S=0$. Symbol **F** oznacza znormalizowaną wartość mantysy, zapisaną jako uzupełnienie dwójkowe zawierające bit domyślny.

Przedstawiony format ma następujące zalety:

1) efektywna wartość wykładnika jest wyznaczona przez **G** i **E**, ponieważ **e = G+1**

2) nie jest możliwe wystąpienie takiego samego wykładnika dla różnych wartości **G**, np. dla **G = 1** wartość **e = G+1** wynosi 2; tak więc **e** jest dwubitowym polem przyjmującym wartości z zakresu od **—4 do 3**; w tym wypadku efektywna wartość wykładnika wynosi 2..5 (dla $S_e = 1$) i —6..—3 (dla $S_e=0$), co nie powoduje sprzeczności z przypadkami, gdyż G=0 lub G=2 (p. tabela).

Zależność między zakresem a wartością pola G (SEF — format tradycyjny: znak, wykładnik mantysa; ang. sign, exponent, fraction)

G	e	SEF	$\overline{E}-(2^e-2)$	$E+(2^e-2)$
0	1	—2..1	—2..—1	0..1
1	2	—4..3	—6..—3	2..5
2	3	—8..7	—14..—7	6..13
3	4	—16..15	—30..—15	14..29
4	5	—32..31	—62..—31	30..61
5	6	—64..63	—126..—63	62..125
6	7	—128..127	—254..—127	126..253
7	8	—256..255	—510..—255	254..509

3) efektywna długość mantysy wynosi **n—e—5=n—G—6** bitów — najmniejsza mantysa ma wartość **0,5**, a największa $1-2^{-(n-G-5)}$; gdy G=7, to wykładnik osiąga największą wartość i największą liczbą dodatnią, dającą przedstawić się w tej reprezentacji, jest $(1-2^{-(n-12)})*2^{509} \cong 10^{153,3}$ a najmniejszą $2^{-1}*2^{-510} \cong 10^{-153,9}$ (zakres liczb ujemnych jest podobny); ponieważ zakres liczb reprezentowanych w komputerze Cray-1 znacznie przekracza zakres osiągalny w proponowanym zapisie dla g=3, w dużych komputerach zakłada się użycie czterobitowego pola **G**, skąd wynika **e=16**, a ekstremalne wartości reprezentowanych liczb wynoszą: $(1-2^{-(n-20)})*2^{131069} \cong 10^{39456}$ i $2^{-1}*2^{-131070} \cong 10^{-39456}$

4) ponieważ zarówno **E** jak i **F** są liczbami zapisanymi jako uzupełnienia dwójkowe, można przeprowadzać stałoprzecinkowe porównywania liczb (bit po bicie).

Jeśli $\bar{x}$ oznacza zmiennoprzecinkową reprezentację liczby rzeczywistej **x**, to względny błąd można zapisać jako $\frac{|x-\bar{x}|}{\bar{x}}$ ponieważ największy błąd ma wartość połowy najmniej znaczącego bitu, prawdziwa jest następująca nierówność

$$|x-\bar{x}| \leq \frac{1}{2} 2^{-f} \cdot 2^{exp},$$

z której po dokonaniu kilku przekształceń

$$\frac{|x-\bar{x}|}{\bar{x}} \leq \frac{\frac{1}{2}2^{-f}\cdot 2^{exp}}{\bar{x}} = \frac{\frac{1}{2}2^{-f}\cdot 2^{exp}}{F\cdot 2^{exp}} = \frac{\frac{1}{2}2^{-f}}{F} \leq \frac{\frac{1}{2}2^{-f}}{\frac{1}{2}} = 2^{-f}$$

wynika, że zwiększenie **G** o 1 zmniejsza błąd względny o połowę, jednocześnie podwajając zakres (p. tabela).

Praktycznie rzadko zdarzają się sytuacje, w których zakres i dokładność mają takie samo znaczenie. W sytuacji, gdy można zrezygnować z dokładności na rzecz zakresu i odwrotnie, zastosowanie opisanej reprezentacji liczb dającej takie możliwości narzuca się samo.

[1]) Dokładniejszy opis tej reprezentacji przedstawiono w artykule: R. Morris, Tapered Floating Point, IEEE Transactions on Computers, Vol. C-20, p. 1578, 1971

Przy jej ocenie należy brać pod uwagę jedynie możliwości zaoszczędzenia pamięci, co czyniłoby ją bezużyteczną w chwili, gdy koszty pamięci mają tendencję zniżkową. Opisana postać zapisu liczb wykazuje swoistą niewrażliwość na te koszty wynikającą z następujących faktów:

1) w istniejących komputerach długość słowa i wielkość przestrzeni adresowej są określone niezależnie od kosztów

2) w komputerach przyszłości te dwa parametry będą determinowane raczej przez szybkość komputerów, wymiary fizyczne, długość rejestrów i szerokość magistrali niż przez koszt pamięci.

Reprezentacja o zmiennej podstawie

Definicja liczby zapisanej w systemie zmiennoprzecinkowym (F*B^E) zakłada, że podstawą **B** jest liczba 2. W celu zbadania możliwości zastosowania różnych podstaw użyto zmiennoprzecinkowego zapisu o postaci **SS$_e$GEF** i o podstawie $B = 2^{G+1}$. W tym wypadku pole **G** określa podstawę **B** nie wpływając na wartość wykładnika. Powyższa definicja dopuszcza podstawy będące potęgami liczby 2, choć nie ma żadnej praktycznej potrzeby stosowania podstaw o wartości 8, 32 itd. Jedyną zaletą takiego systemu jest zakres reprezentowanych liczb, ponieważ podstawa **B=256** dopuszcza stosowanie liczb z zakresu $[256^{-2^e}, 256^{2^e-1}]$.

Ponieważ postać reprezentacji zmiennoprzecinkowej nie ma wpływu na zakres i dokładność reprezentowanych liczb, to można rozważać najprostszą reprezentację typu: znak, wykładnik, mantysa. Wybór podstawy większej od 2 wpływa na liczbę w ten sposób, że zwiększa wielkość zakresu od 2^{2^e-1} do B^{2^e-1}. Jednakże, gdy chce się utrzymać ten sam zakres, pole wykładnika może być ograniczone do e' bitów, gdzie

$$2^{2^e-1} = B^{2^{e'}-1}$$

Podstawiając do tej równości $b = \log_2 B$ otrzymujemy

$$2^e - 1 = (2^{e'} - 1)\log_2 B = (2^{e'} - 1)b,$$

z czego wynika, że zmniejszenie pola cechy wynosi $e - e' = \log_2 b$ bitów. Zaoszczędzone bity mogą być dodane do mantysy w celu zwiększenia dokładności. Po przeprowadzeniu normalizacji, maksymalnie **b—1** zerowych bitów mantysy może zostać niewykorzystanych (średnio (b—1)/2 bitów zerowych). Ponieważ dodatkowo nie może być wykorzystany bit domyślny, efektywna długość mantysy skraca się o (1+(b—1)/2. Można stąd wnosić, że podstawa większa od 2, przy założeniu takiego samego zakresu, powoduje zmniejszenie efektywnej długości mantysy o $\log_2 b$ — [1+(b—1)/2] bitów. Jedyną zaletą stosowania podstawy **B>2** jest duży zakres, a także wygoda dokonywania szeregowych przesunięć przy zmiennoprzecinkowym dodawaniu.

Oprac. **MARIUSZ KUC**
na podstawie: BYTE, Vol. 10
No. 9, 1985

Z kraju

Zastosowanie mikrokomputerów szansą usprawnienia zarządzania gospodarką

Ośrodek Doradztwa i Treningu Kierowniczego (Spółdzielnia Pracy) spopularyzował na terenie całego kraju organizowane przez siebie imprezy pod nazwą FORUM INFORMACYJNE, przekazując w ten sposób szerokiemu gronu działaczy gospodarczych znaczny zasób aktualnej wiedzy ze sfery zarządzania oraz reformy gospodarczej.

W dniach 15—20 października 1984 r. i 6—10 maja 1985 r. ODiTK zorganizował dwie imprezy pod nazwą FORUM INFORMATYCZNE, w konwencji dotychczas organizowanych FORUM INFORMACYJNYCH, które uzyskały wysoką ocenę uczestników. Imprezy te wzbogacono prezentacją sprzętu komputerowego, łożą ekspertów oraz giełdą producentów i użytkowników.

FORUM INFORMATYCZNE postawiło sobie za cel doprowadzenie do świadomości szerokiej opinii działaczy gospodarczych potrzeby i możliwości wykorzystania szansy, jaką w warunkach polskich daje zastosowanie mikrokomputerów w sferze zarządzania. FORUM w zasadzie nie było adresowane do profesjonalnych informatyków, aczkolwiek ci spośród nich, którzy wzięli udział w imprezie, zyskali sporo poglądowych informacji, stanowiących źródło inspiracji w zakresie zastosowań mikrokomputerów. Organizatorzy pragnęli wskazać na pilną potrzebę zrewidowania poglądów na temat możliwości wykorzystania mikrokomputerów, w tym również do współpracy z dużymi komputerami. Nie jest bowiem tajemnicą, że w Polsce brak było dotychczas szerszego zastosowania terminali komputerowych, mogących współpracować interakcyjnie w bezpośrednim dostępie do dużej maszyny cyfrowej.

Na FORUM zaprezentowano przykład współpracy (przy wykorzystaniu dzierżawionego łącza telefonicznego) mikrokomputera IMP-85, produkcji firmy polonijnej IMPOL-2 z komputerem ICL 4/70 zainstalowanym w Zakładzie Informatyki Przemysłu Okrętowego. Analogiczna współpraca mikrokomputerów ELWRO 513/523, IMP-85/MK 4502, RTDS 8 lub ROBOTRON 5120/30 jest możliwa z komputerem ODRA 1305 pod nadzorem systemu operacyjnego GEORGE-3, a także z komputerami rodziny RIAD w systemie operacyjnym OS.

FORUM INFORMATYCZNE postawiło sobie za cel wskazanie jedynie sposobów i kierunków wykorzystania mikrokomputera do wspomagania procesów zarządzania, przez pokazanie poglądowych przykładów zastosowań. Ze względu na brak dostępu do odpowiednich informacji, nie można oczywiście omówić wszystkich aplikacji na terenie kraju. W polityce inwestowania w komputeryzację panuje obecnie chaos a na temat możliwości wykorzystania mikrokomputerów ukształtowało się wiele mitów oraz opinii mających charakter szumu informacyjnego. Na FORUM nie zamierzano lansować żadnego określonego producenta sprzętu, żadnego konkretnego komputera osobistego, ani żadnego wytwórcy oprogramowania. FORUM miało stanowić jedynie płaszczyznę zbliżenia różnych instytucji, aby w jak najlepszy sposób spożytkować nadarzającą się szansę wykorzystania mikroelektroniki do stworzenia dodatkowego narzędzia wspomagającego oraz trudniejszy proces kierowania działalnością przedsiębiorstw w reformowanej gospodarce. Jeżeli uczestnik FORUM został zainspirowany do wykorzystania mikrokomputera do rozwiązywania własnych problemów ekonomicznych, finansowych czy produkcyjnych, będzie to stanowić o końcowym sukcesie imprezy.

Na FORUM starano się dać początek rynkowi oprogramowania i rynkowi użytkownika. Brak jest bowiem w kraju koordynatora interesów wszystkich zainteresowanych stron. Przyjęto przy tym zasadę, że programy mikrokomputerowe w swej masie nie mogą być wyłącznie rozwiązaniami uniwersalnymi czy powszechnego zastosowania. Wręcz przeciwnie, oprogramowanie użytkowe powinno być dostosowane do określonych potrzeb użytkownika. Należy oczywiście wykorzystywać do tych celów gotowe i dostępne pakiety oprogramowania.

FORUM INFORMATYCZNE stworzyło uczestnikom możliwość kontaktu ze specjalistami o zróżnicowanych poglądach na sprzęt i oprogramowanie (pakiety programów, systemy operacyjne, języki programowania). Zorgani-

zowano kilkanaście stanowisk prezentacyjnych z pokazem tematycznych dziedzin zastosowań na mikrokomputerach różnych typów. W ciągu godzinnych seansów każdy z 250 uczestników FORUM miał możliwość zaznajomienia się z wybranymi przez siebie aplikacjami.

Przeglądu zastosowań dokonano w następujących dziedzinach tematycznych:
1) zagadnienia menadżerskie
2) kalkulacja cen i obliczanie rentowności
3) obliczanie zysku
4) obliczanie podatków, kosztów przerobu i sprzedaży
5) badanie czasu pracy
6) wariantowe podejmowanie decyzji
7) system informowania kierownictwa
8) usprawnianie prac sekretarsko-biurowych
9) sterowanie procesami produkcyjnymi
10) inne dziedziny zastosowań.

W tabeli 1. przedstawiono zestawienie przykładowych stanowisk z podziałem na dziedziny.

Niezwykle cenny w tego rodzaju imprezach był udział producentów sprzętu mikrokomputerowego: Zakładów Elektronicznych ELWRO, Krakowskiej Fabryki Aparatury Pomiarowej MERA-KFAP, Zakładów Urządzeń Komputerowych MERA-ELZAB w Zabrzu oraz przedstawicieli firm polonijnych IMPOL-2, IMPOL-1, EMIX i in. Stworzyło to okazję do nawiązania kontak-

Tabela 1. Zestawienie przykładowych stanowisk

Stanowisko	Temat	Dziedzina	Instytucja	Sprzęt
1	Zastosowanie systemu VisiCalc do wspomagania planowania w przedsiębiorstwie	7	Instytut Organizacji i Zarządzania Politechniki Wrocławskiej	LIDIA/APPLE
2	Opracowywanie gospodarki materiałowej	10	Zakład Techniki Biurowej PREDOM-ORG, Gdańsk	ROBOTRON 5120A
3	Obliczanie wynagrodzeń ajentów PZU Kalkulacja parametrów ekonomicznych przedsiębiorstwa TABPLAN — oprogramowanie narzędziowe BANK DANYCH — oprogramowanie narzędziowe	10 1 6 7	Computer Studio Kajkowski	ELWRO 513 LIDIA/APPLE
4	Fakturowanie i rozliczanie sprzedaży na przykładzie biura projektowego Mikrokomputer jako inteligentny terminal współpracujący z komputerem ICL 4/70 — system informacji bibliograficznej	4 7, 10	Zakład Informatyki Przemysłu Okrętowego, Gdańsk	IMP-85
5	Komputeryzacja systemu decyzyjnego	7	Fabryka Farb i Lakierów, Gdańsk	ODRA 1305
6	Wykorzystanie mikrokomputerów osobistych w kalkulacji, prowadzeniu kartoteki pracowników, obliczaniu płac,	2, 3, 4, 10	Przedsiębiorstwo Modernizacji Procesów Technologicznych NOWA-TECH, Katowice	ZX81 SINCLAIR SPECTRUM
7	Metody sieciowe PERT Programy kalkulacyjne	10 6	INFO-SIAT, Warszawa	SINCLAIR SPECTRUM
8	Kalkulacja cen	2	Akademia Ekonomiczna, Poznań	ZX81
9	Wariantowanie efektów ekonomicznych przedsiębiorstwa w warunkach reformy gospodarczej	6	Centrum Informatyki Gospodarki Morskiej, Gdańsk	DATAPOINT 66001
10	Pakiety do prac sekretarsko-biurowych	8	IMPOL 1, IMPOL 2, EMIX	APPLE IBM/PC
11	Kalkulacja cen w budownictwie	2	Zakład Elektronicznej Techniki Obliczeniowej ZOWAR, Warszawa	IMP-85
12	Mikrokomputer w zastosowaniach menedżerskich i badaniu czasu pracy	1,5	Ośrodek Doradztwa i Treningu Kierowniczego, Gdańsk	SINCLAIR SPECTRUM
13	System mikrokomputerowy automatyzacji procesów technologicznych w zastosowaniu do automatyzacji badań silników spalinowych	9	Instytut Elektroniki Politechniki Gdańskiej	mikroprocesor INTEL 8080
14	Możliwości fakturowania — gry telewizyjne	10	Zakłady Urządzeń Komputerowych MERA-ELZAB, Zabrze	MERITUM I
15	Mikrokomputer osobisty w zastosowaniach specjalistycznych — kartoteka silników, sprawozdawczość z gospodarki cieplnej	10	Ośrodek Doradztwa i Treningu Kierowniczego, Gdańsk, Huta KOŚCIUSZKO	SINCLAIR SPECTRUM ZX81

Tabela 2. Przykładowe zastosowanie mikrokomputerów

Ekonomika	Automatyka	Technika	Grafika	Dydaktyka
Kalkulacja kosztów Kalkulacja cen Obliczenia podatków Prowadzenie kartotek Zamówienia, kontrole dostaw Redagowanie tekstów Prowadzenie sekretariatu Obliczenia płac Fakturowanie dostaw sprzedaży Wariantowanie wyników ekonomicznych działalności Ewidencja i rozliczanie	Sterowanie procesami produkcyjnymi Sterowanie łącznością telefoniczną i teleksową Kontrola załadunku Kontrola procesów chemicznych Kontrola procesów fizycznych Pilotaż, kierowanie Kontrola i sterowanie dystrybucją energii elektrycznej Sterowanie magazynowaniem i składowaniem Sterowanie aparaturą	Obliczenia inżynierskie Obliczenia konstrukcyjne Optymalizacja Symulacja Informacja naukowo-techniczna	Modelowanie Rysunki techniczne Projektowanie wzorów użytkowych Wizualizacja wskaźników ekonomiczno-technicznych Prezentacja obrazów Statystyka	Konwersacja Nauka języków obcych Gry menedżerskie Testy kontrolne

tów handlowych oraz do przeprowadzenia wielu konsultacji w czasie trwania imprezy na stanowisku „loży ekspertów", gdzie prowadzono doradztwo informatyczne w zakresie oprogramowania, sprzętu oraz zastosowań.

FORUM INFORMATYCZNE było więc konwersatorium potencjalnych użytkowników z wytwórcami oprogramowania i producentami sprzętu mikrokomputerowego. Na podstawie jego przebiegu można sformułować kilka tez zalecanych do rozważenia przy kreowaniu rynku mikrokomputerowego.

Zasadniczą ideą wykorzystania mikrokomputera powinno być doprowadzenie go bezpośrednio na stanowisko pracy i umożliwienie współpracy w trybie konwersacyjnym (nie wymagającym znajomości informatyki) przy użyciu prostych specjalizowanych programów, uwzględniających indywidualne potrzeby użytkownika. Można do tego celu wykorzystywać oprogramowanie narzędziowe zawierające określone pakiety programów uniwersalnych.

Drugi zakres zastosowań to przykładowe wykorzystanie mikrokomputerów do autonomicznego (lokalnego) przetwarzania danych z równoczesną możliwością agregowania (kumulowania) danych w celu dalszego przetwarzania na dużych komputerach, jak ODRA 1300 lub RIAD. Tego rodzaju rozwiązanie może być stosowane w przypadku tworzenia na mikrokomputerach różnych zbiorów danych. Przekazując to zadanie dużemu komputerowi można odciążyć mikrokomputer od większości operacji związanych z utrzymaniem i rozwojem zbiorów. W takim przypadku jeden z mikrokomputerów może służyć jako koncentrator do zbierania danych i przesyłania ich do dużej maszyny cyfrowej. Jeszcze prostszym rozwiązaniem jest zebranie dyskietek z danymi, przewiezienie ich do ośrodka obliczeniowego i bezpośrednie wczytanie do pamięci zewnętrznych dużego komputera. Jeżeli natomiast nie występują tego rodzaju potrzeby, to należy preferować przetwarzanie autonomiczne wyłącznie na stanowiskach pracy.

W tabeli 2. przedstawiono przykładowe zastosowanie mikrokomputerów w ekonomice, automatyce, technice, grafice i dydaktyce.

STEFAN RAKOWSKI

Dolnośląski Oddział Polskiego Towarzystwa Informatycznego

Polskie Towarzystwo Informatyczne w piątym roku swojej działalności skupia już 800 członków. Większość spośród nich pochodzi z dużych ośrodków, szczególnie silnie związanych z informatyką. Jednym z takich ośrodków jest Wrocław. Tutaj też w końcu grudnia ubiegłego roku odbył się I Walny Zjazd Delegatów nowo powołanego Dolnośląskiego Oddziału PTI. Obszarem działania Oddziału są województwa: wrocławskie, opolskie, wałbrzyskie i jeleniogórskie.

Dotychczasowa działalność ponad 200 członków PTI z tego terenu skupiała się w kołach we Wrocławiu i Opolu. Ich działania zmierzają przede wszystkim do integracji środowiska, osiąganej przez utrwalenie związków nieformalnych, ułatwiających kontakty zawodowe oraz wymianę informacji, a także — do popularyzacji informatyki w społeczeństwie.

W ramach tych działań koło we Wrocławiu przygotowało już drugą edycję ogólnopolskiego konkursu PTI na najlepsze prace magisterskie (p. Informatyka, nr 4, 1985). W celu upowszechnienia najlepszych wdrożeń prac z dziedziny informatyki zainicjowano konkurs im. Jerzego Trybulskiego. Rezultatem tego konkursu będzie uhonorowanie nowoczesnych i użytecznych krajowych zastosowań informatyki. Jego wyniki będą ogłaszane na kolejnych zjazdach PTI.

Dynamiczne i efektywne działanie dolnośląskiego środowiska informatycznego sprzyja popularyzacji informatyki w społeczeństwie. Akcja — „Komputer w tornistrze", prowadzona wspólnie z redakcją „Wieczoru Wrocławia", oraz „Wakacje z komputerem" — są przykładem przemyślanych form nauczania dzieci i młodzieży posługiwania się komputerem. Nauka przez zabawę — a co więcej, mądre i poprawne nauczanie podstaw informatyki, zasługuje na uznanie.

Warto też wspomnieć o powstaniu, z inicjatywy środowiska wrocławskiego, Fundacji Edukacji Komputerowej, której celem — w oparciu o fundusze — jest wspomaganie rozwoju informatyki w szkołach i wśród młodzieży.

W trakcie obrad Zjazdu podsumowano dotychczasową działalność. Zwrócono uwagę na problemy dostępu do literatury fachowej, szczególnie poza ośrodkami akademickimi. Postulowano założenie nowych sekcji i klubów użytkowników komputerów, np. serii JS. Dyskutowano też nad formą szkoleń i kursów dokształcających w zakresie zastosowań i projektowania narzędzi informatycznych.

Prezesem Oddziału wybrano dr. inż. Zbigniewa Mazura z Centrum Obliczeniowego Politechniki Wrocławskiej.

(WI)

Pracownie komputerowe SEP i „Horyzontów Techniki"

W ramach działalności popularyzującej nową technikę wśród młodzieży i dorosłych, Stowarzyszenie Elektryków Polskich będzie prowadziło w swoich oddziałach wojewódzkich Pracownie Mikrokomputerowe. »Horyzonty Techniki« uruchamiają stałą rubrykę zawierającą informacje dotyczące prowadzonych pracowni oraz w miarę możliwości będą drukować materiały związane z zakresem ich działania. Funkcjonowanie poradni będzie też wspomagane przez publikacje w »Radioelektroniku«, dotyczące przede wszystkim konstruowania pomocniczego sprzętu oraz jego zastosowań. W miarę rozwoju pracowni można oczekiwać także współdziałania ze strony innych czasopism.

Zakres działania Pracowni Mikrokomputerowych obejmuje:

- zajęcia klubowe (odczyty wystawy) z mozliwością korzystania z klubowego sprzętu mikrokomputerowego oraz zgromadzonej w Pracowni literatury (książki, czasopisma, instrukcje obsługi, wydruki programów),
- poradnictwo i konsultacje dla posiadaczy sprzętu mikrokomputerowego, ułatwienia w wymianie programów dostosowanych do sprzętu typowego i nietypowego w Polsce,
- kursy i seminaria z zakresu zastosowań mikrokomputerów dostosowane do wiedzy osób nie związanych zawodowo z informatyką.

Wyposażenie, przede wszystkim sprzęt komputerowy, byłoby kupowane przez członków zbiorowych SEP i przekazywane pracowniom. Bieżąca działalność pracowni byłaby oparta na zasadach samofinansowania, fundusze pochodziłyby z opłat wnoszonych przez osoby deklarujące stały udział w zajęciach klubowych, z opłat za porady i konsultacje oraz z opłat za kursy i seminaria. Dodatkowe dochody można by uzyskiwać ze sprzedaży nadbitek artykułów z »Horyzontów Techniki« i »Radioelektronika«.

Instruktorzy i prelegenci będą zatrudniani w ramach umów-zleceń. Ich wynagrodzenia byłyby pokrywane z funduszy pracowni. Eksperymentalnie proponuje się uruchomienie w 1986 r. dwóch takich pracowni: w Warszawie, w Klubie SIGMY przy ul. Mazowieckiej oraz we Wrocławiu. Obie pracownie będą działały pod patronatem, np. Zakładów ELWRO i zakładów ERA.

Oddziały wojewódzkie, które chciałyby zorganizować pracownie mikrokomputerowe, a widzą możliwości uzyskania, wypożyczenia lub korzystania ze sprzętu, proszone są o nawiązanie kontaktu z Podkomisją Popularyzacji Elektryki Centralnej Komisji Szkolnictwa Elektrycznego i Wydawnictw SEP (kol. J. Justat, kol. W. Rathman). Istnieje możliwość zorganizowania w 1986 r. większej liczby pracowni, poza proponowanymi.

Działalność Komitetu ds. Systemu CAMAC

Główne kierunki działalności Komitetu w 1985 r.:

- przedsięwzięcia organizacyjne zmierzające do umocnienia krajowej i międzynarodowej pozycji Komitetu,
- stymulowanie i ukierunkowanie rozwoju systemu CAMAC oraz wybór i wprowadzenie w Polsce perspektywicznego systemu modularnego,
- działalność międzynarodowa,
- organizacja Sympozjum CAMAC-86,
- działalność w zakresie informacji naukowo-technicznej.

Merytoryczna tematyka zainteresowań Komitetu, oprócz spraw dotyczących systemu CAHAC, coraz intensywniej wkracza w obszar perspektywicznych systemów modularnych, a w szczególności wyboru jednego z nich, w celu opracowania i uruchomienia produkcji w Polsce. Prezydium Komitetu wystąpiło z wnioskiem o przemianowanie dotychczasowej nazwy Komitetu na Polski Komitet ds. Systemu CAMAC oraz z wnioskiem o przyjęcie przemianowanego Komitetu na członka Komitetu ESONE. Należy się spodziewać, że formalności związane z przyjęciem zostaną załatwione na konferencji ESONE, która ma się odbyć w kwietniu 1986 r. w Warszawie łącznie z sympozjum CAMAC-86 (por. INFORMATYKA, nr 8'1985).

Podstawowa działalność merytoryczna Komitetu w 1985 r. koncentrowała się z jednej strony na rozwijaniu działań zmierzających do określenia perspektywy rozwojowej systemu CAMAC, zaś z drugiej strony — na zagadnieniu wyboru nowego systemu modularnego oraz zaprogramowaniu jego opracowania i uruchomienia produkcji w Polsce. Po przeprowadzeniu wielu prac studialnych i analiz za perspektywiczny uznano system CAMAC-S oparty na rozwiązaniu Multibus II firmy INTEL (por. INFORMATYKA, nr 6/1984). W ramach działalności merytorycznej Komitetu prowadzono również prace koordynacyjne, które obejmowały:

— opiniowanie kolejnych wersji planów koordynacyjnych opracowań systemu CAMAC i CAMAC-S (Multibus II),

— realizację prac objętych w latach 1981—1985 problemami węzłowymi i Programem Rządowym PR-8,

— opracowanie programu prac na lata 1986—1990 dotyczących systemu CAMAC i CAMAC-S.

Działalność międzynarodowa Komitetu dotyczyła współpracy z następującymi organizacjami:

- Komitet ESONE — udział w zebraniach technicznego Komitetu Koordynacyjnego i Komitetu Wykonawczego,
- Stała Komisja ds. Pokojowego Wykorzystania Energii Jądrowej RWPG — udział w pracach Sekcji ds. Elektroniki Jądrowej,
- Zjednoczenie „Interatominstrument",
- Międzynarodowa Komisja Elektrotechniczna IEC, Komitet Techniczny nr 45, grupa 3,
- Stała Grupa Robocza ds. Automatyzacji Badań Naukowych Akademia Nauk Krajów Socjalistycznych.

W ramach uczestnictwa przedstawicieli Komitetu ds. Systemu CAMAC w posiedzeniach wymienionych organizacji międzynarodowych podejmowano intensywne działania, aby system CAMAC-S (Multibus II) został uznany w krajach socjalistycznych za standard międzynarodowy. Propozycje polskie znalazły pozytywny oddźwięk i zostały uwzględnione w dokumentach międzynarodowych.

W 1985 r. prowadzono również działalność informacyjną, w ramach której:

— opublikowano artykuł nt. systemu Multibus II w czasopiśmie »Systemy mikroprocesorowe«,

— zorganizowano cykl seminariów nt. nowoczesnych systemów modularnych,

— utrzymano kontakty z grupą „Multibus II" organizacji IEEE oraz Komitetu ESONE i rozpowszechniono specyfikację systemu.

Działalność tegoroczna będzie koncentrowała się na stymulowaniu rozwoju nowoczesnych systemów modularnych i popularyzacji wiedzy o tych systemach.

J. ZAL.

Przetwarzanie współbieżne w Moduli-2

Jedną z głównych cech różniących Modulę-2 od Pascala, języka będącego jej poprzednikiem, jest ta, że Modula-2 posiada szczególne konstrukcje programowe pozwalające na opracowywanie programów współbieżnych czasu rzeczywistego. Dzięki temu można używać Moduli-2 do pisania systemów operacyjnych i innego oprogramowania niskiego poziomu, zamiast stosowanego tradycyjnie języka asemblera.

Współbieżność

Faktyczna współbieżność programów wykonywanych dokładnie w tym samym czasie jest zrozumiała, gdy mamy do czynienia z oddzielnymi komputerami. Występuje ona w procesorach rozłożonych, jak np. w systemach sterowania i innych zastosowaniach czasu rzeczywistego.

Natomiast, kiedy kilka osób pracuje przy jednym komputerze, współbieżność jest realizowana za pomocą przeplatania. Ten rodzaj programowania występuje w dużych systemach z podziałem czasu, w których wielu użytkowników może wykonywać swoje programy na pojedynczym komputerze, w tym samym czasie. Każdy użytkownik widzi działanie tylko swego programu, dzięki temu, że system operacyjny steruje przydziałem mocy obliczeniowej pomiędzy wszystkich użytkowników. Taki efekt przeplatania umożliwia pozorną współbieżność wielu procesów wykonywanych na jednym procesorze.

Niezależnie od tego, czy mamy do czynienia z prawdziwą współbieżnością czy nie, problemy związane z pisaniem programów tego rodzaju są takie same. Głównym zagadnieniem jest synchronizacja. Programy współbieżne muszą odnosić się do zdarzeń niezdeterminowanych, które mogą występować w dowolnym czasie.

Załóżmy, że istnieją dwa komputery połączone pewnym rodzajem łącza komunikacyjnego. Należy napisać program pozwalający dwóm osobom, siedzącym w dwóch różnych miejscach, na wymianę informacji przy użyciu klawiatury. Każdy komputer musi wykonywać następujący ciąg działań:

— po naciśnięciu klawisza komputera **A**, musi on określić kod odpowiedniego znaku i przesłać go do komputera **B**

— po nadejściu znaku z komputera **B**, komputer **A** musi wyświetlić go w następnym wolnym miejscu na swym monitorze.

Jeśli osoba siedząca przy klawiaturze przerwie naciskanie klawiszy, to jest oczywiste, że na wejściu łącza nie pojawią się żadne dane. Dwóch użytkowników może prowadzić dialog, polegający na zadawaniu pytań i udzielaniu odpowiedzi, tak że w określonej chwili tylko jeden z nich naciska klawisze.

Z drugiej zaś strony, obaj użytkownicy mogą używać klawiatury jednocześnie. W obu przypadkach program musi działać poprawnie. Takie niezdeterminowane działanie odróżnia programy czasu rzeczywistego od typowych programów aplikacyjnych, z ustalonymi funkcjami odczytu i zapisu.

Nie można przewidzieć z góry, ile odczytów (polegających na odbiorze sygnałów wejściowych z klawiatury) lub zapisów (tj. nadania sygnałów wyjściowych do łącza) będzie musiał wykonać program i w jakiej kolejności one następują.

Niezdeterminowana natura programowania współbieżnego prowadzi do znanego zjawiska, zwanego zakleszczeniem (ang. deadlock). Może ono, na przykład, powstać w sytuacji, gdy w systemie operacyjnym pracującym z podziałem czasu wykonywane są dwa programy, sterujące jedynie stacją pamięci taśmowej i specjalną drukarką. System operacyjny inicjuje program **A** obsługujący pamięć taśmową i po upływie określonego odcinka czasu zawiesza go inicjując program **B**. Program **B** bezpośrednio po rozpoczęciu uruchamia drukarkę i zostaje zawieszony. Kiedy program **A** rusza ponownie, próbuje bezskutecznie sterować drukarką, tak że jego działanie jest zawieszone do czasu zwolnienia drukarki przez program **B**. Kiedy program **B** zostaje reaktywowany, próbuje sterować pamięcią taśmową i również jest zawieszony do czasu jej zwolnienia przez program **A**. Programy są zakleszczone, gdyż żaden z nich nie może ruszyć dalej do chwili zwolnienia potrzebnych zasobów.

Kluczem do pisania udanych programów współbieżnych jest koordynacja. Programy te muszą współpracować lub oddziaływać na siebie w sposób konstruktywny — lub przynajmniej niedestrukcyjny. Modula-2 ma konstrukcje programowe pozwalające na zbieżność i umożliwiające koordynację.

Procesy współbieżne

Przetwarzanie współbieżne jest techniką umożliwiającą koordynację programów współbieżnych. Załóżmy, że należy napisać dwa programy do wykonywania procesów współbieżnych **Text** i **Disk**. Proces **Text** umożliwia wprowadzenie tekstu przez klawiaturę do pamięci dyskowej (może to być edytor), a proces **Disk** jest programem niskiego poziomu, sterującym odczytem i zapisem rekordów na dysku. Oddziaływanie wzajemne obu procesów jest następujące:

- **Text** wysyła zachętę do wprowadzenia informacji z klawiatury i umieszcza znaki w buforze aż do odebrania znaku końca rekordu
- **Text** sygnalizuje procesowi **Disk**, że rekord jest gotowy w buforze, po czym zawiesza się
- **Disk** odbiera rekord i zapisuje go do pamięci dyskowej
- **Disk** sygnalizuje procesowi **Text**, że rekord został zapisany, po czym zawiesza swoje działanie
- **Text** reaktywuje się od punktu zawieszenia i wysyła zachętę do wykonania nowego polecenia.

W ten sposób procesy **Text** i **Disk** współpracują, przekazując informację między sobą. Ta zależność między producentem **(Text)** a konsumentem **(Disk)** jest klasyczną zależnością między procesami współbieżnymi.

Można rozszerzyć pojęcie współbieżności na więcej niż dwa procesy. Załóżmy, że różne procesy powinny wymieniać komunikaty podczas wykonywania. Muszą one mieć zdolność do nadawania komunikatów do innych procesów i odbierania ich, czyli muszą być czymś w rodzaju elektronicznej poczty. Jeden specjalny proces spełnia rolę naczelnika poczty. Odbiera on i przechowuje komunikaty, a następnie rozdziela je stosownie do zadań procesów odbiorczych. Proces naczelny jest współbieżny z innymi procesami, które nadają lub odbierają komunikaty. Nadając komunikat proces zawiesza swoje działanie na czas, gdy proces naczelny umieszcza komunikat w buforze i reaktywuje się, gdy jest to zrobione. Odbierając komunikat proces zawiesza się aż proces naczelny wypełni komunikatem swój bufor, a następnie kontynuuje działanie.

Pisanie programów współbieżnych w Moduli-2

Do zrealizowania współbieżności potrzebne są dwa podstawowe mechanizmy: sposób identyfikacji i wykonywania programu, określający go jako proces, oraz metody sygnalizacji umożliwiające skoordynowanie działań dwóch procesów współbieżnych. Modula-2 ma odpowiednie konstrukcje, zapewniające realizację tych mechanizmów. W Moduli-2 programiści nie muszą odstępować od reguł języka, aby pisać programy współbieżne, ponieważ sam język umożliwia ten rodzaj programowania.

Najważniejszą konstrukcją ułatwiającą programowanie współbieżne w Moduli-2 jest **PROCESS**, importowany z modułu **SYSTEM**. Zazwyczaj w językach programowania operuje się takimi typami jak **REAL** i **INTEGER**, natomiast proces występujący jako typ jest czymś nowym. Taka potrzeba jest oczywista: język operujący procesami musi umożliwiać odwoływanie się do nich w konkretny sposób.

W przetwarzaniu współbieżnym używa się zmiennych typu **PROCESS** do

komunikacji wzajemnej. Dlatego tworząc program współbieżny należy związać go z jedną z tych zmiennych. Podobnie, kiedy programy te przekazują sobie sterowanie, muszą używać do tego celu zmiennych typu **PROCESS.** Modula-2 zawiera dwie procedury, które to zapewniają **NEWPROCESS** i **TRANSFER,** obie importowane z modułu **SYSTEM.**

Procedura **NEWPROCESS,** która tworzy nowy proces współbieżny w systemie i wiąże go ze zmienną typu **PROCESS,** ma następujące wywołanie:

NEWPROCESS (p: PROC, a: ADDRESS, s: CARDINAL, VAR c: PROCESS);

W instrukcji tej **p** jest nazwą procedury zawierającej kod tworzący proces, **a** jest adresem obszaru pamięci służącej jako przestrzeń robocza dla procesu (przechowywane są w niej dane lokalne i kontekst), **s** jest rozmiarem tej przestrzeni, a **c** — zmienną typu **PROCESS,** której procedura **NEWPROCESS** przypisuje utworzony proces współbieżny.

TRANSFER jest procedurą, która przekazuje sterowanie od procesu wywołującego do wywołanego. Postać wywołania jest następująca:

TRANSFER (VAR thisprocess, coprocess: PROCESS);

W czasie wykonania procedury **TRANSFER** proces wywołujący zostaje zawieszony, a jego kontekst (wartości danych, stan licznika rozkazów itp.) przechowywany. Podczas kolejnego wykonania procedury **TRANSFER** przez proces wywoływany, proces wywołujący reaktywuje swoje działanie od miejsca następującego po poprzedniej instrukcji **TRANSFER.** Jest to istotna różnica między wywoływaniem procedur a przekazywaniem sterowania instrukcją **TRANSFER.** Przy każdym wywołaniu procedura jest wykonywana od początku, a pamięć lokalna jest przydzielana ponownie. Procesy współbieżne natomiast, zachowują kontekst przy każdym wywołaniu instrukcji **TRANSFER.**

Implementacje typu **PROCESS** i procedur **NEWPROCESS** i **TRANSFER** różnią się zależnie od rodzaju komputera i kompilatora. Nie należy się jednak tym przejmować. Mechanizmy abstrakcji w Moduli-2 umożliwiają operowanie zmiennymi typu **PROCESS** tak, że sprzężenia są niezależne od komputera. Jest to znaczny postęp w stosunku do programowania w języku asemblera, gdzie do operowania strukturami danych — reprezentującymi procesy — wymagana jest doskonała znajomość systemu operacyjnego.

Przykładowy program

Po omówieniu udogodnień do programowania procesów współbieżnych w Moduli-2, przedstawimy przykład programu. Na początku modułu **TermHandler** importowane są niezbędne struktury współbieżne z modułu **SYSTEM,** tj. typy **PROCESS, ADDRESS** i **WORD** oraz procedury **NEWPROCESS, TRANSFER, ADR** i **SIZE. ADDRESS** i **WORD** są elementarnymi typami danych, **ADR** jest funkcją udostępniającą adres początkowy zmiennej, a **SIZE** — funkcją udostępniającą rozmiar pamięci przydzielonej tej zmiennej. Zakłada się istnienie modułu o nazwie **SYSIO,** z którego importowane są procedury **GetChar** i **PutDisk.** W segmencie **CONST** zdefiniowano rozmiar bufora i terminator rekordu **CR** (ang. carriage return, powrót karetki). W segmencie **VAR** zdefiniowano bufor współdzielony **(buffer)** i zmienną współdzieloną oznaczającą liczbę znaków **(nchar),** ponadto zdefiniowano przestrzenie robocze dla dwóch tworzonych procesów współbieżnych **(wspT** i **wspD).**

Należy zapewnić obsługę transmisji z klawiatury na dysk dla kilku terminali — na przykład w systemie rejestrowania zamówień. Procedura **GetChar** przyjmuje numer terminala **(thisterm)** i udostępnia stan **(status= =TRUE** oznacza dostarczenie nowego znaku, a **status=FALSE** przeciwnie) oraz znak z klawiatury **(newchar).** Jeśli znak pojawi się po poprzednim wywołaniu **GetChar,** to zostanie udostępniony. W przeciwnym wypadku zmienna **status** zasygnalizuje błąd.

Ciało modułu **TermHandler** zaczyna się od utworzenia procesu **D,** po którym następuje utworzenie **nterm** procesów obsługujących terminale. Następnie w pętli wykonywane są współbieżnie procesy obsługi terminali. Każdy z nich zarządza swoim licznikiem znaków **(count)** i buforem **(localbuf).** Gdy proces wykrywa znak **CR** lub pełny bufor, to przypisuje wartości zmiennych **locabuf** i **count** zmiennym globalnym bufora **(buffer)** i licznika **(nchar)** i przekazuje sterowanie do procesu **D,** aby zarejestrować rekord. Choć częściej stosuje się współdzielenie wskaźników do zmiennych niż samych zmiennych przez procesy współbieżne, to w przedstawionym przykładzie zastosowano to drugie podejście w celu uproszczenia programu. Gdyby procedura **GetChar** oczekiwała na nadejście znaku, czyli gdyby nie miała parametru **status,** to pojedynczy terminal mógłby zawiesić wszystkie pozostałe, jeśli nie wyprowadzono by z niego żadnej informacji.

Co osiągnięto dzięki zastosowaniu przetwarzania współbieżnego w tym przykładzie?

Po pierwsze, gdy określony proces obsługi terminala przekazuje sterowanie procedurą **TRANSFER** z powrotem do procesu **C,** to jego kontekst jest przechowywany w przestrzeni roboczej. Dlatego przy następnym uaktywnieniu tego procesu wartości zmiennych **count** i **localbuf** są ustawiane właściwie. Nie byłoby to możliwe, gdyby użyto wywołań procedur zamiast procesów współbieżnych.

Po drugie, ponieważ każdy proces współbieżny utrzymuje swoje własne struktury danych, cały program jest dość prosty. Nie ma konieczności zarządzania tablicami buforów i liczników. Każdy proces obsługi ma pojedynczy bufor i licznik, a kod jest wykonywany **nterm** razy. Bardziej złożona wersja modułu **TermHandler** mogłaby uwzględniać priorytety niektórych terminali i posiadać możliwość ignorowania innych. W rzeczywistości, w przetwarzaniu współbieżnym programu **TermHandler** mógłby tworzyć na żądanie nowe procesy obsługi (zamiast operowania stałą ich liczbą) lub przekazywać sterowanie ściśle określonemu procesowi tylko w pewnych warunkach.

```
MODULE TermHandler;
  FROM SYSTEM IMPORT ADDRESS, PROCESS, NEWPROCESS, TRANSFER,
    WORD, ADR, SIZE;
  FROM SYSIO IMPORT GetChar, PutDisk;
  CONST bufsize = 80;
        nterm = 16;
        CR = 13C;
  TYPE buftype = ARRAY[0..bufsize - 1] OF CHAR;
  VAR buffer: buftype;
      nchar: INTEGER;
      wspT: ARRAY[0..nterm - 1],[1..200] OF WORD;
      wspD: ARRAY[1..200] OF WORD;
      D,C: PROCESS;
      T: ARRAY[0..nterm - 1] OF PROCESS;
      thisterm: INTEGER;
    PROCEDURE TextIn;
      VAR newchar: CHAR;
          status: BOOLEAN;
          localbuf: buftype;
          count: INTEGER;
      BEGIN
        count: = - 1;
        LOOP
          GetChar(thisterm,newchar,status);
          IF status THEN
            CASE newchar OF
              CR: nchar: = count;
                  buffer: = localbuf;
                  TRANSFER(T[thisterm],D);
                  count: = - 1
            ELSE
              INC(count);
              localbuf[nchar]: = newchar;
              IF count = bufsize - 1
                THEN
                  nchar: = count;
                  buffer: = localbuf;
                  TRANSFER(T[thisterm],D);
                  count: = - 1
                END
            END
          END;
          TRANSFER(T[thisterm],C)
        END
      END TextIn;
  PROCEDURE TextToDisk;
    BEGIN
      LOOP
        PutDisk(buffer,nchar);
        TRANSFER(D,T[thisterm])
      END
    END TextToDisk;
  BEGIN
    NEWPROCESS(TextToDisk,ADR(wspD),SIZE(wspD),D);
    FOR thisterm: = 0 TO nterm - 1 DO
      NEWPROCESS(TextIn,ADR(wspT[thisterm]),
                 SIZE(wspT[thisterm]),T[thisterm])
    END;
    thisterm: = 0;
    LOOP
      TRANSFER(C,T[thisterm]);
      thisterm: = thisterm + 1 MOD nterm
    END
  END TermHandler.
```

Przykład programowania współbieżnego w Moduli-2

* * *

Procesy współbieżne przetwarzają informację metodą przeplatanego wykonywania na pojedynczym komputerze. Jeżeli uda się konstruktywnie opisać ich współdziałanie, to konstrukcje Moduli-2 umożliwią skuteczne zaprogramowanie tej współpracy. Programowanie współbieżne dostarcza środków do przekazywania sterowania między programami, bez straty pożądanej kolejności wykonywania lub kontekstu.

Oprac. **MACIEJ ADAMCZYK**
na podst. **BYTE, Vol. 10, No. 4, 1985**

Ochrona programów przed kopiowaniem w komputerach osobistych

Problem ochrony programów przed kopiowaniem przez nieuprawnionych użytkowników pojawił się po rozpoczęciu seryjnej produkcji komputerów i mikrokomputerów. Napisanie programu o ogólnym zastosowaniu (np. systemu operacyjnego, translatora, edytora, pakietu wspomagającego projektowanie, obsługę arkuszy obrachunkowych itp.) wymaga dużego nakładu, liczonego niekiedy w dziesiątkach osobo-lat pracy.

Firma zajmująca się wytwarzaniem oprogramowania o powszechnym zastosowaniu musi sprzedać program tysiące razy, aby zapewnić rentowność produkcji. Istnieje zatem konieczność zabezpieczenia programu przed nielegalnym skopiowaniem i użytkowaniem go w innym systemie.

Najczęściej stosowaną dotąd metodą było zapisanie na stałe w pamięci komputera jego numeru fabrycznego (niekiedy listy programów sprzedawanych użytkownikowi), a następnie porównanie go z numerem zapisanym w programie, w początkowej fazie jego działania. Jeżeli program wykrył niezgodność, to zawieszał działanie, działał błędnie, a niekiedy niszczył siebie i inne dane zapisane w pamięci. W przypadku minikomputera znalezienie i odszyfrowanie sposobu ochrony programu jest zajęciem pracochłonnym, a w przypadku dużych systemów prawie niemożliwym. Metoda ta ma jednak ograniczony zakres stosowania. Producent oprogramowania musi znać dokładnie budowę sprzętu (numer fabryczny komputera i sposób jego odczytania przez program), przez co jest zmuszony do stałego kontaktu z jego producentem. Program musi być przed każdą sprzedażą odpowiednio wygenerowany, tak, aby uwzględniał właściwości sprzętu danego użytkownika. Nakład pracy, przeznaczony na wykonanie powyższych czynności, zwiększa się wraz z liczbą sprzedanych kopii, zmniejszając rentowność przedsięwzięcia. Ponadto, wytwórcy oprogramowania chcą chronić swoje produkty również na innych komputerach niż te, które są przystosowane do realizacji opisanego mechanizmu.

Problem ochrony programów wystąpił szczególnie wyraźnie po powstaniu rynku komputerów osobistych. Wielkość rynku, liczona w milionach użytkowników, pojawienie się firm zajmujących się dystrybucją sprzętu i oprogramowania, wszystko to spowodowało konieczność opracowania nowych metod zabezpieczenia programów przed kopiowaniem. Ochrona powinna być na tyle skuteczna, aby uniemożliwić skopiowanie programu zarówno przez użytkownika, jak również przez pośrednika, który chcąc powiększyć zyski dodaje oprogramowanie do zakupionego u niego sprzętu, zmniejszając tym samym dochody firmy będącej właścicielem programu. Celują w tym szczególnie małe przedsiębiorstwa, trudniące się często sprzedażą nielegalnie skopiowanych programów. Stosowane obecnie aktywne metody ochrony przed kopiowaniem programów, zapisanych na dyskach elastycznych, można podzielić na dwie zasadnicze grupy: niestandardowe sformatowanie dyskietki lub umieszczenie informacji w miejscach dyskietki, które podczas normalnej pracy nie są używane. Na oryginalnej dyskietce znajduje się informacja konieczna do działania programu, lecz jest ukryta tak, aby nie można jej skopiować na drugą dyskietkę. Najczęściej polega to na wytworzeniu specjalnego błędu, od którego wystąpienia program uzależnia dalsze działanie.

Jako przykład stosowania niestandardowego formatowania ścieżki, w celu ukrycia danych za pośrednictwem umyślnie spowodowanych błędów, może służyć komputer osobisty C 64. Na początku każdego sektora umieszczone jest pole adresowe, składające się z serii synchronizującej (5 bajtów FF_H), znacznika pola adresowego (08_H), identyfikatora (numer ścieżki, numer sektora) oraz sumy kontrolnej. W celu ochrony przed kopiowaniem zmienia się wartość znacznika pola adresowego. Odczytana z dysku wartość znacznika jest porównywana z zawartością pamięci komputera (adres 39_H dla pola adresowego, 38_H dla pola danych). Niezgodność podczas próby kopiowania powoduje wyświetlenie na ekranie komunikatu „20 Read Error" lub „22 Read Error". Odczyt informacji jest możliwy jedynie po zmianie zawartości pamięci, na wartość odpowiadającą znacznikowi zapisanemu na dyskietce. Jeżeli w sektorach ze zmienionym polem adresowym znajdują się dane ważne dla programu, to nieumiejętnie skopiowany program nie będzie działał. Ochronę można uczynić jeszcze bardziej skuteczną, przez zmianę wartości znaczników w zależności od fazy działania programu lub numeru ścieżki, sektora.

Metodę niestandardowego formatowania ścieżki stosują również producenci oprogramowania dla komputera IBM PC. W systemie operacyjnym PC-DOS używa się sektorów o długości 512 bajtów. Przez zmianę parametrów wejściowych, dla podprogramu odpowiedzialnego za odczyt i zapis informacji na dyskietce (przerwanie INT 13 w module BIOS) można zapisać ścieżkę, na której sektory będą miały inną długość. Niezgodność formatu z przyjętym przez program DISCOPY uniemożliwi skopiowanie dyskietki i zasygnalizowanie błędu. Bardziej wyrafinowane metody wykorzystują fakt, że oprócz standardowych dla systemu PC-DOS dziewięciu sektorów, po 512 bajtów na ścieżce, można dodatkowo zamieścić jeszcze jeden sektor o długości 128 bajtów. Program DISCOPY nie wykryje podczas kopiowania żadnego błędu. Mimo tego skopiowany program nie będzie działał ze względu na brak danych, umieszczonych w niewidocznych dodatkowych sektorach.

Często stosowane jest generowanie błędów odczytu przez zmianę wartości sumy kontrolnej, umieszczonej na końcu pola adresowego i danych sektora. Przykładowo, w komputerze C 64 suma kontrolna jest obliczana jako sumu modulo dwa bajtów pola adresowego (danych). Zapisanie sektora ze zmienioną sumą kontrolną powoduje wyświetlenie komunikatu o błędzie „23 Read Error", sygnalizującego niezgodnośc sumy kontrolnej odczytanej z dysku — z obliczoną na podstawie odczytanych danych. Większość sterowników dysków elastycznych generuje sumę kontrolną przy zapisie i sprawdza automatycznie przy odczycie. W tym przypadku wygenerowanie błędu sumy kontrolnej jest zadaniem bardziej skomplikowanym. Polega to na zaprogramowaniu sterownika dysku do zapisu sektora o większej długości, tak aby zmienione bajty sumy kontrolnej sektora o nominalnej długości mogły być przesyłane na dysk jako dane. Po ich wysłaniu operacja zapisu zostaje natychmiast przerwana (przez wydanie odpowiedniego rozkazu do sterownika), pozostawiając na dysku jedynie dane odpowiadające sektorowi nominalnej długości i zmienioną sumę kontrolną.

Drugim, obok niestandardowego formatowania, sposobem ukrywania danych na dysku jest wykorzystanie właściwości mechanizmu dyskowego do pozycjonowania głowic nad obszarami nie używanymi podczas normalnej pracy, w celu zapisu chronionych danych lub wytworzenia błędu. Jako przykład zastosowania tej metody może posłużyć komputer VC 1541. Głowica dysku porusza się krokami równymi połowie odległości pomiędzy sąsiednimi ścieżkami. Po zmianie fragmentu systemu operacyjnego można przesunąć głowicę nad obszar w kształcie pierścienia, leżący między dwoma ścieżkami zapisanymi w normalny sposób. Na tym pierścieniu zapisuje się dane, które niszczą zawartość sąsiednich ścieżek, lecz dają się odczytać przez zmodyfikowany system operacyjny. Chroniony program sprawdza na początku działania zawartość określonego pierścienia, która dla niedokładnej kopii będzie nieokreślona.

Niekiedy wykorzystuje się możliwości mechanizmu dyskowego do pozycjonowania głowicy na większej liczbie

ścieżek niż używa system operacyjny. Na ścieżkach, zwykle niedostępnych (dla VC 1541 — powyżej numeru 35, dla IBM PC powyżej numeru 39) zapisuje się dane istotne dla działania programu, które nie mogą być skopiowane za pośrednictwem dyrektyw systemu operacyjnego. Niektórzy producenci oprogramowania (np. firma Ashton-Tate) zabezpieczają oryginalne dyskietki przez umyślne uszkodzenie nośnika (promieniem lasera) w ściśle określonym miejscu. Po rozpoczęciu działania program sprawdza, czy jest możliwy poprawny zapis w tym obszarze. Poprawny zapis świadczy o braku uszkodzenia, co oznacza, że dyskietka została skopiowana (program zawiesza działanie).

Lista wymienionych metod ochrony oprogramowania przed kradzieżą jest z pewnością niekompletna. Producenci chcąc chronić swoje interesy są zmuszani do wynajdowania wciąż nowych skuteczniejszych metod zabezpieczenia programów. Najmniej zadowoleni z takiego stanu rzeczy są użytkownicy, którzy narażeni są na niewygody związane z wykorzystaniem legalnie zakupionego programu. Nie mogąc zrobić samodzielnie kopii oryginalnej dyskietki, w przypadku jej uszkodzenia lub zużycia, właściciel zostaje pozbawiony możliwości dalszego korzystania z programu. Często uszkodzenie dyskietki powoduje utratę istotnych danych. Przeniesienie programu na dysk stały bywa kłopotliwe. Niektóre firmy (np. Microsoft) zezwalają na wykonanie ograniczonej liczby kopii oryginalnej dyskietki, za pośrednictwem znajdującego się w niej specjalnego programu kopiującego. Najczęściej dopuszcza się jednokrotne skopiowanie na dysk stały.

Ochrona programów przed kopiowaniem utrudnia ich wykorzystywanie, naraża użytkownika na utratę danych, spowalnia działanie programów i powoduje zwiększenie ich rozmiarów. Z drugiej strony, osoby i przedsiębiorstwa czerpiące zyski z nielegalnego ropowszechniania oprogramowania mają niezbędne środki do szybkiego rozszyfrowania nowej metody ochrony programu. Stosowane metody zabezpieczają więc program przed kopiowaniem przez zwykłego użytkownika, który go zakupi, natomiast nie stanowią większego utrudnienia dla profesjonalistów, zajmujących się nielegalnym rozpowszechnianiem oprogramowania.

Niektórzy producenci wycofują się z ochrony programów przed kopiowaniem. Udostępniają nawet specjalne oprogramowanie służące do skopiowania większości zabezpieczonych dyskietek. Programy te wytwarzają wierną kopię dyskietki, której ochrona polega na niestandardowym sformatowaniu lub na zapisaniu informacji na dodatkowych ścieżkach. Autorzy programów kopiujących zawierają w nich możliwie dużą liczbę algorytmów przeciwdziałających ukrywaniu danych. Podczas swego działania program kopiujący sprawdza algorytmy ochrony, zastosowane na oryginalnej dyskietce, a w przypadku ich rozpoznania, powtarza je przy zapisie danych na kopii. Przykładem programów kopiujących dyskietki dla komputera IBM PC są: COPYIIPC, COPYWRITE, COPYPC, PCOPIER. Istnieje nawet możliwość zakupu programu kopiującego w ramach prenumeraty. Otrzymując ciągle nowe wersje programów kopiujących, rozpoznających coraz nowsze algorytmy ochrony, użytkownik może skopiować każdy program w 2—3 miesiące po jego pojawieniu się na rynku.

Zamiast stosowania aktywnych sposobów ochrony programów, producenci zaczynają coraz częściej stosować metody bierne. Na dyskietce zapisuje się numer fabryczny lub nazwisko i adres użytkownika, które pojawiają się na wszystkich wydrukach sporządzanych podczas pracy programu. Na tej podstawie można określić, czy dyskietka pochodzi z legalnego źródła. Ułatwia to egzekwowanie praw autorskich na drodze sądowej i uzyskanie odszkodowania od osoby lub firmy, która nielegalnie powielała oprogramowanie. Liczba procesów sądowych i dochodzeń wszczętych przeciwko przedsiębiorstwom bezprawnie powielającym i sprzedającym oprogramowanie ciągle wzrasta.

STANISŁAW CHMIELEWSKI

Metastabilność systemów VME i MULTIBUS II

Projektanci systemów VME i MULTIBUS II mogą niespodziewanie wpaść w pułapkę. Zdarza się, że doprowadzają oni do wystąpienia zjawiska zwanego metastabilnością asynchroniczną, a wówczas niezawodność systemu znacznie maleje. Kilku opracowaniom opartym o VME udowodniono już zawodność; przy opracowaniach opartych o MULTIBUS II grozi to samo.

Metastabilność asynchroniczna jest przypadkowym, trudnym do przewidzenia stanem układu, który pojawia się w systemach zestawionych z asynchronicznych elementów (nie zsynchronizowanych wspólnym zegarem). Metastabilność występuje na wyjściach układów synchronizujących, które koordynują wzajemne oddziaływanie pomiędzy elementami systemu. Jeśli to przypadkowe niestabilne zachowanie nie zostanie wzięte pod uwagę przez projektanta, to system staje się nieobliczalny i zawodny. Prawdopodobieństwo wystąpienia stanu metastabilnego jest proporcjonalne do liczby i szybkości różnych zegarów w systemie. Ryzyko metastabilności wzrasta w systemach pracujących z szybkim zegarem, używających wielu zegarów oraz używających wielu procesorów (ang. masters).

Magistrala o strukturze asynchronicznej, taka jak VME, jest szczególnie podatna na metastabilność asynchroniczną, ale systemy oparte o MULTIBUS II też nie są całkowicie odporne. MULTIBUS II sam w sobie jest synchroniczny i nie narażony na metastabilność. Jeżeli jednak projektant dołączy do magistrali procesor lub jakieś urządzenie pracujące z innym zegarem, to system staje się asynchroniczny i może stać się metastabilny.

Oczywiście można zaprojektować niezawodny system asynchroniczny, ale tylko wówczas, gdy problemy metastabilności zostaną wzięte pod uwagę. Metastabilność jest dość dobrze opisana w literaturze, lecz wielu projektantów żyje w nieświadomości problemu. Dawniej, gdy systemy były powolne i proste, utrata synchronizacji i błąd z powodu metastabilności rzadko wywoływały poważne skutki. We współczesnych bardzo szybkich systemach o wieloprocesorowych architekturach, przy niezwykle małych tolerancjach czasowych, nieuwzględnienie metastabilności może mieć poważne konsekwencje. Na rynku znajdują się już wyroby mające z tego powodu znaczne wady.

Przerzutniki źródłem metastabilności

Metastabilność pojawia się przy synchronizacji pracujących asynchronicznie elementów systemu. Aby dwa takie elementy mogły porozumieć się, muszą być wcześniej zsynchronizowane. Kiedy jednostka centralna z zegarem 8MHz wysyła dane na 10MHz magistralę, układ synchronizujący musi próbkować 8MHz cykle z częstotliwością 10 MHz. Układ synchronizujący, zwykle przerzutnik, staje się źródłem metastabilności. Jak każde urządzenie, które ma nie mniej niż dwa stabilne wyjścia, może zachowywać się metastabilnie przy różnych sytuacjach na wejściu. W szczególności, dla układu synchronizującego w postaci przerzutnika taki stan zachodzi, kiedy na wejście danych oraz wejście zegarowe jednocześnie docierają przednie zbocza dwu synchronizowanych sygnałów.

Prawdopodobieństwo tego, że którekolwiek dwa asynchroniczne przednie zbocza przypadkowo zbiegną się w czasie, jest niewielkie. Wzrasta ono jednak ze wzrostem częstotliwości obu sygnałów. Jeśli zdarzy się taki wypadek, to wyjście przerzutnika będzie w stanie nieustalonym (metastabilnym) przez

ok. 50 ns dla urządzeń TTL [1]) a nieco krócej dla ECL [2]). W tym czasie system może traktować stan wyjścia przerzutnika przypadkowo, tj. jak poziom wysoki lub niski.

Istnienie systemów metastabilnych opartych o standard VME nie wyklucza niezawodnych rozwiązań systemów asynchronicznych. Metoda polega na zaprojektowaniu systemu tak, aby sygnały na wyjściach układów synchronizujących były nieważne w czasie stabilizowania się (50 ns dla układów TTL).

Znaczenie problemu dla producentów

Choć można uporać się z metastabilnością, wciąż jeszcze sprawia ona kłopoty nieświadomym inżynierom, po części z powodu niechęci firm Motorola i Intel do publikowania informacji na ten temat. Prawdopodobnie żadna z firm nie chce dać konkurencji atutu w walce o upowszechnienie standardu magistrali. Zarówno John Black — szef handlowy Motoroli do spraw VME, jak i John Beaston — szef handlowy Intela do spraw MULTIBUS II, utrzymują, że metastabilność jest problemem interface'ów i nie należy rozważać jej przy specyfikacji magistrali.

Na poziomie technicznym przypuszczalnie mają rację, lecz jeden czy dwa paragrafy każdej specyfikacji będą wymagać dużego nakładu pracy projektanta, zanim dojdzie on samodzielnie do dobrego rozwiązania problemu. Firma Motorola zastanawia się nad rozwiązaniem, lecz Intel nie uwzględnia problemu na poziomie specyfikacji! Co więcej, twierdzi, że problemy, które narosły dla produktów VME, w produktach MULTIBUS II nie występują.

Firma Intel utrzymuje, że użytkownicy MULTIBUS I już dawno nauczyli się rozwiązywać problemy niestabilności i nie powinno im sprawiać kłopotu opracowanie odpowiedniego interface'u dla MULTIBUS II. Z drugiej strony wiadomo, że nie zbudowano jeszcze dostatecznej liczby systemów opartych o MULTIBUS II, aby stwierdzić, jak skutecznie radzą sobie użytkownicy z metastabilnością. Zaleta systemów opartych o MULTIBUS I polegała na tym, że nie były one na tyle skomplikowane, aby ulegać metastabilności i zaledwie kilku użytkowników w ogóle zauważyło problem. Użytkownicy MULTIBUS II mogą napotkać o wiele bardziej skomplikowane problemy. Zależnie od tego czy będą, czy nie będą świadomi metastabilności, mogą wpaść w tę samą pułapkę co użytkownicy VME.

Problematyka systemów wieloprocesorowych

Pierwszą „ofiarą" metastabilności produktów VME stał się John Willis, szef wieloprocesorowego projektu Rapid Bus opracowywanego w Carnegie Mellon Robotics Laboratory (Pittsburg, stan Pensylwania). Willis zaplanował, że użyje gotowych pakietów VM02 jako podstawy wieloprocesorowego systemu z procesorami MC68000, ewentualnie odcinając zbędne części tych pakietów. Tak zrobiony system, zawierający tylko dwa pakiety VM02 (8 MHz) może zapętlić się w ciągu 4—10 minut. Tymczasem specyfikacja Motoroli przewiduje stabilną (niezawodną) pracę 16 pakietów VM02 w konfiguracji wieloprocesorowej.

W tym przypadku udało się zlokalizować źródła błędów w pakiecie VM02: dwuwejściowy arbiter, arbiter zgłoszeń magistrali (ang. bus request), oraz sterownik zgłoszeń (ang. bus requester). Najbardziej kłopotliwy okazał się dwuwejściowy arbiter, który steruje dostępem do każdej dwuwejściowej pamięci pakietu UM02. Dwuwejściowy arbiter decyduje czy procesor MC68000 z jednego pakietu VM02 może otrzymać dostęp do pamięci, czy też otrzyma go przez magistralę VME procesor z drugiego pakietu.

Metastabilność powstaje w układzie synchronizującym arbitra, który odpowiada za synchronizację dwóch źródeł zgłoszeń magistrali z własnym zegarem. (Zegar arbitra jest synchroniczny z zegarem MC68000). Ponieważ arbiter podejmuje swoje decyzje w czasie ok. 20 ns, to wyjście układu synchronizującego ma tylko 20 ns na ustalenie się stanu stabilnego, a potrzebuje co najmniej 50 ns dla niezawodnej pracy. Arbiter zgłoszeń magistrali (ang. bus grant) na pakietach VM02 jest zawodny z tej samej przyczyny — usiłuje wykonać arbitraż w ciągu zaledwie 20 ns.

Funkcją sterownika zgłoszeń (ang. bus requester) jest generowanie zgłoszeń magistrali (ang. bus requests) i próbkowanie sygnału przyznania magistrali (ang. bus grant). Każdy procesor (ang. master) w systemie VME ma swój sterownik zgłoszeń, a różne sterowniki zgłoszeń są połączone w pierścień (ang. daisy chain), tak że sygnał przyznania magistrali, generowany przez arbiter, przechodzi szeregowo przez wszystkie sterowniki (ang. requesters). Każdy sterownik decyduje, czy przyjąć sygnał przyznania i uzyskać dostęp do magistrali, czy też przepuścić ten sygnał do następnego sterownika w pierścieniu. Jeśli procesor (ang. master) związany z danym sterownikiem właśnie wysłał zgłoszenie, sterownik winien przechwycić sygnał przyznania magistrali. Jeśli procesor nie wystawia żądania, sygnał powinien przejść do następnego sterownika.

Willis twierdzi, że takie rozwiązanie jest zawodne z powodu braku synchronizacji pomiędzy sygnałem zgłoszenia i sygnałem przyznania magistrali. Wykazuje on, że układ, który w sterowniku synchronizuje zgłoszenia i przydział magistrali, nie ma dostatecznej ilości czasu na ustabilizowanie się. Narastające zbocze sygnału zgłoszenia na jednym z wejść układu synchronizującego pojawia się równocześnie z narastającym zboczem sygnału przyznania na innych z wejść, powodując nieprzewidziany stan sterownika, co grozi przyznaniem magistrali dwu procesorom na raz.

Argumenty te wspiera Dave Barr z firmy Indocomp (Drayton Palms, stan Michigan), projektant zajmujący się systemami wieloprocesorowymi opartymi o VME. W trakcie testu, w którym dwa procesory cyklicznie zapisywały dane do pamięci globalnej (również przez magistralę VME), zauważono, że kolizje na magistrali między żądającymi dostępu procesorami powodowały zapis nieprawidłowych danych. Aby tego uniknąć, odrzucono proponowaną przez VME strategię arbitrażu magistrali na rzecz systemu synchronicznego, używając do koordynacji zgłoszeń i przydziału jednego zegara 4MHz. To rozwiązanie ogranicza szybkość arbitrażu, ale zapewnia niezawodność systemu. Jego autor mógłby zmodyfikować logikę arbitrażu proponowaną przez VME, ale nie zrobił tego, wskutek nieznajomości zagadnień metastabilności.

Oprac. ANNA CZARNECKA

[1]) TTL — ang. Transistor-Transistor Logic.
[2]) ECL — ang. Emitter Coupled Logic

System ekspertowy w przemyśle samochodowym

Firma Renault zademonstrowała ostatnio system ekspertowy umożliwiający szybką i efektywną diagnostykę skrzyń biegów. System został zaprojektowany i wykonany przez firmę Cap Gemini Sogeti w oparciu o mikrokomputer IBM PC. Zapowiedziano, że systemy takie zostaną wkrótce zainstalowane we wszystkich warsztatach i stacjach obsługi posiadających autoryzację firmy Renault.

System ekspertowy zapewnia pracę w trybach: doradczym, archiwacji, uaktualniania. Z funkcji doradczych korzystają mechanicy i obsługa warsztatowa. Tryb archiwacji służy do gromadzenia danych o usterkach technicznych pojawiających się w skrzyniach biegów. Dane te przekazywane są następnie do centralnego działu sprzedaży firmy. Tryb uaktualniania umożliwia wzbogacenie bazy wiedzy systemu ekspertowego o informacje wynikające z nowych doświadczeń eksploatacyjnych.

Firma Renault twierdzi, że zrealizowany system jest jednym z pierwszych praktycznych zastosowań techniki sztucznej inteligencji, które trafiają do tak szerokiego kręgu odbiorców.

M. M.

Źródło: System International, December 1985

Nowa generacja 32-bitowych systemów wieloprocesorowych z wykorzystaniem magistrali MULTIBUS

Ostatnio firma Intel Corp. wprowadziła na rynek sześć pakietów w systemie MULTIBUS II. Stanowią one kompletny zestaw złożony z płytki centralnego procesora i pamięci, systemu operacyjnego, programu uruchomieniowego, kasety i magistrali.

MULTIBUS II jest otwartym 32-bitowym standardem opracowanym dla systemów wieloprocesorowych (por. INFORMATYKA, nr 6 '1984). W przeciwieństwie do MULTIBUS I (INFORMATYKA, nr 1 '1983) oraz VME (INFORMATYKA nr 6 '1984) zapewnia pracę synchroniczną i zdecentralizowany arbitraż, ma multipleksowane linie danych i adresów, akceptuje przerwania programowe i adresowanie geograficzne.

Pełny system MULTIBUS II ma kilka magistrali. Uniwersalna magistrala równoległa PSB o szybkości 40 MB/s umożliwia przesyłanie danych oraz komunikację między procesorami. Magistrala lokalna LBX II pozwala na rozszerzenie pamięci do 64 MB, poza arbitrażem. Uzupełniająca magistrala SBX rozszerza system w obrębie pakietu SBC (ang. Single Board Computer).

Na pakiecie centralnego procesora iSBC 286/100 znajduje się mikroprocesor 80286, podstawka dla koprocesora arytmetycznego 80287, interface LBX II dla pamięci RAM, rozbudowane sterowanie we-wy DMA, sterowanie przekazywaniem komunikatów i arbitrem magistrali, mikrokontroler samotestu i dwa programowane kanały szeregowego we-wy. Wymienny port równoległy zawiera interface SCSI, (ang. Small Computer Systems Interface), interface drukarki Centronics lub interface definiowany przez użytkownika. Ponadto dostępne jest złącze SBX dla dodatkowych pakietów.

Istnieją cztery typy płytek pamięci z kontrolą parzystości (iSBC MEM/312, 310, 320 i 340) o pojemności odpowiednio $^1/_2$, 1, 2 i 4 MB. Każda 32-bitowa płytka ma 8 KB pamięci podręcznej (ang. cache), podwójne porty dla magistrali PSB oraz LBX II, mikrokontroler samotestu i układ automatycznego inicjowania — po włączeniu zasilania.

Moduł iSBC CSM/001 realizuje centralne sterowanie systemu przez magistralę PSB, a ponadto zawiera zegar dzienny, mikrokontroler samotestu i uniwersalny interface do innych magistrali.

Przeznaczona dla MULTIBUS II wersja systemu operacyjnego czasu rzeczywistego iRMX86 (Release6) zapewnia obsługę przerwań w postaci komunikatów, automatyczną konfigurację pamięci i samoczynne testowanie stanu. Obsługuje protokół SCSI, magistralę uzupełniającą SBX, rozbudowane sterowanie we-wy DMA i sterowniki komunikacji szeregowej. Monitor uruchomieniowy iSDM286 (ang. system debugging monitor) umożliwia programistom sprawdzenie własnych rozwiązań w systemach MULTIBUS I lub II. Do pracy wymaga on terminala względnie pełnego systemu uruchomieniowego Intellec.

Wytwarzane są kasety na 6 lub 9 stanowisk z magistralą LBX na płycie tylnej i z jedną lub dwoma płytkami pamięci. Zestaw ćwiczebny (ang. evaluation kit) zawiera płytkę procesora, płytkę pamięci o pojemności $^1/_2$ MB, centralny moduł obsługi, program uruchomieniowy, dziewięciostanowiskową kasetę i płytę tylną z trzema złączami LBX.

Koszt poszczególnych modułów jest dość duży. Mikrokomputer iSBC 286/100 kosztuje 3125 dolarów, a ceny pamięci iSBC MEM zależnie od pojemności wynoszą od 2250 do 8095 dolarów. Centralny moduł obsługi iSBC CSM/001 kosztuje 995 dolarów. Jednorazowy koszt licencji na system operacyjny i iRMX86-MBII — dla użytkowników rozpoczynających pracę z zestawem — wynosi 6500 dolarów, a dla obecnie używających — 2000 dolarów. Monitor uruchomieniowy iSDM 286 kosztuje dodatkowo 2500 dolarów.

Oprac. **ANNA CZARNECKA**

Zmierzch mikroprocesorów 8-bitowych

Wszystkie znaki na niebie i na ziemi wskazują na powolne wypieranie na rynku brytyjskim mikrokomputerów 8-bitowych przez bardziej zaawansowane konstrukcje. Najnowszym przejawem tego trendu jest ostatnia decyzja brytyjskiego Ministerstwa Handlu i Przemysłu (DIT — Department of Trade and Industry), popierająca system MSDOS. Jest to system operacyjny stosowany głównie w mikrokomputerach 16-bitowych, będący przyjętym w praktyce standardem przemysłowym. Nieco może naciągając porównanie można powiedzieć, że jest on dla mikrokomputerów tym, czym dla minikomputerów jest obecnie system UNIX. System MSDOS w szczególności wypiera z rynku CP/M i jego pochodne. Znaczenie decyzji DTI polega na tym, że stanowi ona oficjalne zalecenie dla wszystkich instytucji korzystających z pomocy finansowej rządu. Oznacza to m.in., że instytucje publiczne, takie jak szkoły, uniwersytety, urzędy itp. nie będą już (w zasadzie) mogły kupować mikrokomputerów, które nie są wyposażone w system MSDOS, co w praktyce wyklucza głównie mikrokomputery 8-bitowe. Jest to jednocześnie zdecydowana zmiana kursu, gdyż przez ostatnie pięć lat obowiązywało zalecenie kupowania właśnie mikrokomputerów 8-bitowych: szczególnie dużym poparciem cieszył się model BBC firmy ACORN. W tej chwili np. praktycznie wszystkie szkoły średnie i podstawowe dysponują średnio kilkoma mikrokomputerami tego typu. Pionierski etap wprowadzania nowej techniki do szkół został więc zakończony i decyzja DTI zmierza do ulepszenia istniejącego parku mikrokomputerowego. Oznacza to jednocześnie początek końca zestawów typu mikrokomputer-magnetofon kasetowy, wypierany przez bardziej wydajne zestawy mikrokomputer-dyskietki.

Decyzja uderza najbardziej w głównych producentów, tj. firmy ACORN (mikrokomputer BBC) oraz SINCLAIR RESEARCH (modele SPECTRUM + i QL). Do tej pory 85% mikrokomputerów zainstalowanych w szkołach stanowił właśnie BBC. W bardziej korzystnym położeniu znalazły się więc te firmy, które już wcześniej zdecydowały się na zastosowanie w swoich produktach standardu MSDOS. Dotyczy to głównie dwóch producentów: RML, który dzięki modelowi NIMBUS opanował 13% rynku szkolnego, oraz ACT, którego 16-bitowy APRICOT jest bardzo popularny na uniwersytetach, m.in. ze względu na dobrą grafikę.

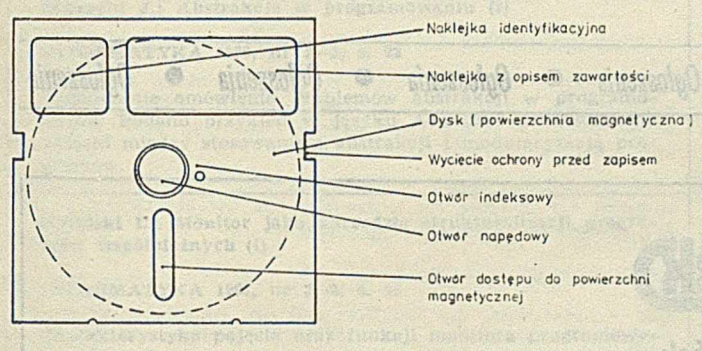
Główną zaletą systemu MSDOS (oczywiście oprócz zdolności obsługiwania dysków) jest łatwość stosowania tak wyposażonych mikrokomputerów jako inteligentnych terminali większych maszyn, oraz możliwość pracy w sieci. Niewątpliwą zaletą jest także popularność tego standardu wśród użytkowników i producentów. (RKK)

Terminy związane z pamięciami dyskowymi

W „Minisłowniku terminów informatycznych" opublikowanym przez Gdańskie Koło Polskiego Towarzystwa Informatycznego [1]) zawarto definicje kilkudziesięciu podstawowych terminów informatycznych, z których wyodrębniliśmy grupę związaną z pamięciami dyskowymi. W bieżącym numerze przedrukowujemy odpowiednie definicje, uzupełniające je o kilka terminów pokrewnych, opracowanych na podstawie normy ISO-2382/XII, które dla odróżnienia zaznaczono gwiazdką (Red.).

CYLINDER *) — na dysku magnetycznym zbiór wszystkich ścieżek o tej samej nominalnej odległości od osi, wokół której obraca się dysk.

DYSK ELASTYCZNY (DYSKIETKA) — dysk magnetyczny z miękkiego tworzywa, zamknięty w kopertę z papieru lub plastiku. Dyski elastyczne różnią się wymiarami (8 cali, 5 1/4 cala, 3 1/2 cala), sposobem zapisu (SS, DS, SD, DD), formatem oraz organizacją. Wymiary są cechą charakterystyczną samego dysku, sposób zapisu i format zależą od napędu i systemu operacyjnego, a organizacja dysku jest określona przez system operacyjny. Typowe parametry obecnie stosowanych dysków elastycznych są następujące:



Wymiar (cale)	Sposób zapisu	Liczba ścieżek	Pojemność KB
8	SS,SD	77	250
5 1/4	DS,DD	80	500
3 1/2	DS,DD	160	1

Na rys. przedstawiono dyskietkę o wymiarze 5 1/4 cala.

DYSK (MAGNETYCZNY) *) — płaska okrągła płyta o magnesowalnej warstwie powierzchniowej, na której można przechowywać dane przez ich rejestrację magnetyczną.

DYSK SZTYWNY — dysk magnetyczny wykonany ze sztywnego materiału. Dysk sztywny może być wymienny lub zespolony z napędem, z którym stanowi nierozłączną całość. Jest stosowany jako pamięć zewnętrzna o znacznej pojemności (od 3 MB do ponad 100 MB).

DYSK WINCHESTER *) — dysk sztywny niewymienny o specjalnej konstrukcji, umożliwiającej ustawienie głowicy podczas lotu na wysokości ułamka mikrometra, dzięki czemu uzyskuje się znaczną gęstość zapisu. Szczelne zamknięcie głowicy wraz z dyskami we wspólnej komorze zmniejsza dostęp zanieczyszczeń, wskutek czego dyski Winchester są niezawodne (por. Informatyka, nr 6, 1983, str. 26—28).

FORMAT — ustalony sposób podziału powierzchni dysku na ścieżki i sektory. Zależnie od rodzaju dysku i napędu, sposób identyfikacji, liczba ścieżek i sektorów może znacznie się różnić. Wyróżnia się format zmienny (ang. soft), który może być częściowo zmieniany przez system operacyjny w trakcie formatowania, oraz format stały (ang. hard), który nie może być zmieniany przez użytkownika. Dyski o formacie zmiennym posiadają jeden otwór indeksowy, a dyski do formatu stałego — wiele otworów. Dyski elastyczne mają na ogół format zmienny, a dyski sztywne — format stały.

FORMATOWANIE — operacja wyznaczania ścieżek i sektorów na dysku. Zazwyczaj formatowanie jest realizowane przez producenta, choć możliwe jest często formatowanie przez użytkownika. Formatowanie niszczy całą informację zapisaną na dysku.

GŁOWICA MAGNETYCZNA *) — elektromagnes, który może wykonywać funkcje odczytu, zapisu lub wymazywania danych na nośniku magnetycznym.

JEDNOSTKA PAMIĘCI DYSKOWEJ *) — urządzenie zawierające napęd dyskowy, głowice magnetyczne i odpowiednie sterowanie.

NAPĘD — element pamięci dyskowej, stanowiący mechanizm służący do poruszania dysku i sterowania jego ruchem. Napęd przez otwór napędowy wprawia dysk w ruch obrotowy, a przez otwór dostępu do powierzchni realizuje operacje odczytu i zapisu sektora. Niezawodność napędu decyduje o jakości całej pamięci dyskowej. Do rodzaju napędu musi być dostosowany wymiar dysku i sposób zapisu, a także jego format.

ORGANIZACJA DYSKU — sposób wyróżniania na powierzchni dysku obszarów fizycznych odpowiadających jednostkom logicznym, jak pliki, rekordy itp. Sposób organizacji dysku zależy od systemu operacyjnego i nawet dla mikrokomputerów tego samego typu może być różny. Zgodność organizacji dysków dwóch mikrokomputerów jest warunkiem umożliwiającym łatwe przenoszenie oprogramowania.

OTWÓR INDEKSOWY — niewielki okrągły otwór w powierzchni dysku, pasujący do odpowiedniego otworu w kopercie, zależnie od formatu dysku określający początek ścieżki lub sektora.

PAMIĘĆ DYSKOWA — pamięć zewnętrzna, w której dane są przechowywane przez magnesowanie odpowiednich miejsc na powierzchni dysku. Pamięć dyskowa ma zazwyczaj dużą pojemność. Zależnie od sposobu wykonania dysku, dzieli się na pamięć na dyskach elastycznych oraz na dyskach sztywnych. W wypadku pamięci na dyskach elastycznych dysk jest wymienny, a dla pamięci na dyskach sztywnych dysk jest najczęściej na stałe połączony z napędem.

SEKTOR — fragment ścieżki dysku, która może być zapisywana bądź odczytywana przez głowicę magnetyczną. Zależnie od formatu dysku, pojemność sektora wynosi od 128 B do 1024 B.

SPOSÓB ZAPISU — sposób, odwzorowania informacji przez namagnesowanie powierzchni dysku w pamięci dyskowej. Wyróżnia się zapis po jednej stronie dysku (ang. single sided, SS) oraz po obu stronach (ang. double sided, DS).

[1]) Cofta P.: Materiały pomocnicze seminarium „Mikrokomputery w zarządzaniu", cz. I, „Minisłownik terminów informatycznych". Polskie Towarzystwo Informatyczne, Gdańsk, 1985

Ponadto zapis może być realizowany z tzw. pojedynczą gęstością (ang. single density, SD) lub podwójną gęstością (ang. double density, DD). W przybliżeniu, całkowita pojemność dysku, zapisanego sposobem SS i SD jest czterokrotnie mniejsza od pojemności dysku zapisanego jako DS i DD. Możliwość użycia odpowiedniego sposobu zapisu zależy od rodzaju napędu i dysku.

ŚCIEŻKA — koncentryczny, odpowiednio oznaczony okrąg na powierzchni dysku, niewidoczny gołym okiem, wyznaczający miejsce, gdzie zapisywane są dane. Liczba ścieżek jest cechą napędu i waha się od 40 do 160. Ścieżka jest podzielona na pewną liczbę sektorów, zależnie od formatu dysku.

PIOTR COFTA

Wirth N.: Zasady wieloprogramowości i ich implementacja w Moduli-2

INFORMATYKA 1986, nr 2—3, s. 1

Charakterystyka różnych zasad realizacji wieloprogramowości oraz szczegółowe omówienie sposobu ich implementacji w języku Modula-2, z podaniem odpowiednich przykładów.

Karczmarczuk J.: Język programowania Icon (1)

INFORMATYKA 1986, nr 2—3, s. 11

Pierwsza część charakterystyki języka programowania Icon, przeznaczonego głównie do analizy i przetwarzania złożonych tekstów oraz rozwiązywania problemów z dziedziny sztucznej inteligencji.

Bielecki J.: Zaawansowane konstrukcje języka C

INFORMATYKA 1986, nr 2—3, s. 14

Charakterystyka zasad konstruowania wyrażeń w języku C oraz omówienie zestawu programów zawierających zaawansowane pod względem składniowym i semantycznym konstrukcje.

Szkaradnik Z.: Operacje zmiennoprzecinkowe w języku Forth

INFORMATYKA 1986, nr 2—3, s. 17

Rozszerzenie translatora języka Forth o pakiet operacji zmiennoprzecinkowych, zrealizowane przez autora na mikrokomputerze Mera 60.

Zakręcki J.: Abstrakcje w programowaniu (2)

INFORMATYKA 1986, nr 2—3, s. 22

Zakończenie omówienia problemów abstrakcji w programowaniu. Podano przykład w języku Ada oraz przedstawiono związki między stosowaniem abstrakcji i modularyzacją programów.

Kotulski L.: Monitor jako narzędzie strukturalizacji programów współbieżnych (1)

INFORMATYKA 1986, nr 2—3, s. 24

Charakterystyka pojęcia oraz funkcji monitora programowego. Szczegółowo omówiono rozwiązania zapewniające strukturalizację programów współbieżnych.

Kleiber M., Leśny M., Szuniewicz R.: Komputery osobiste w zastosowaniach profesjonalnych (1)

INFORMATYKA 1986, nr 2—3, s. 27

Pierwsza część charakterystyki komputerów osobistych z punktu widzenia ich cech użytkowych. Omówiono cechy sprzętu mikrokomputerowego oraz oprogramowanie użytkowe, obejmujące programy redagowania tekstów oraz obliczeń tablicowych.

Stokłosa J.: Abraham Stern — pierwszy polski konstruktor maszyn arytmetycznych

INFORMATYKA 1986, nr 2—3, s. 31

Zwięzła charakterystyka działalności oraz istoty osiągnięć polskiego pioniera budowy maszyn liczących z początku XIX wieku.

Postół M.: Modyfikacja procesu kompilacji PASCALA/MT+

INFORMATYKA 1986, nr 2—3, s. 32

Propozycja usprawnienia procesu kompilacji programów w języku PASCAL/MT+, przeznaczonych dla mikrokomputerów specjalizowanych, np. sterowników.

Виртх Н.: Принципы мультипрограммирования и их реализация на языке Modula-2

INFORMATYKA 1986, № 2—3, стр. 1

Характеристика принципов мультипрограммирования и подробное обсуждение способов их реализации на языке Modula-2, решения проиллюстрированы примерами.

Качмарчук Я.: Язык программирования Icon (1)

INFORMATYKA 1986, № 2—3, стр. 11

Первая часть характеристики языка программирования Icon, предназначенного прежде всего для анализа и обработки сложных текстов и решения проблем в области искусственного интеллекта.

Белецки Я.: Сложные выражения на языке С

INFORMATYKA 1986, № 2—3, стр. 14

Характеристика принципов сложения выражения на языке С. Обсуждено состав программ содержащих выражения сложные с точки зрения семантики.

Шкарадник З.: Операции с плавающей запятой на языке Forth

INFORMATYKA 1986, № 2—3, стр. 17

Транслятор языка Forth дополнено пакетом для реализации операций с плавающей запятой. Проект был реализован автором на микро-ЭВМ Мега-60.

Закрецки Я.: Абстракты в программировании (2)

INFORMATYKA 1986, № 2—3, стр. 22

Завершение характеристики проблем абстрактов в программировании. Приведено пример разработанный на языке Ada для иллюстрации связи между модулями программ и абстрактами.

Котульски Л.: Монитор как инструмент разработки одновременных, структурных программ

INFORMATYKA 1986, № 2—3, стр. 24

Характеристика понятия и функций программы-монитора. Подробно обсуждены решения обеспечивающие возможность разработки одновременных, структурных программ.

Клейбер М., Лесны М., Шуневич Р.: Персональные компьютеры для профессиональных применений (1)

INFORMATYKA 1986, № 2—3, стр. 27

Первая часть характеристики персональных компьютеров с точки зрения свойств их применения. Обсуждено свойства микро-ЭВМ, их программное обеспечение и в том числе программу редактирования текстов и матричных расчетов.

Стоклоса Я.: Абрагам Стерн — польский пионер конструктор вычислительных машин

INFORMATYKA 1986, № 2—3, стр. 31

Краткая характеристика деятельности и достижений польского пионера конструктора вычислительных машин начала XIX века.

Постул М.: Модификация процесса компиляции Pascal/MT+

INFORMATYKA 1986, № 2—3, стр. 32

Предложение усовершенствования процесса компиляции программ на языке Pascal/MT+ разработанных для специализированных мини-ЭВМ.

Wirth N.: Multiprogramming principles and their implementation in Modula-2

INFORMATYKA 1986, No. 2—3, p. 1

Characteristics of different multiprogramming realisation principles and detailed discussion of their implementation method in Modula-2 with presentation of adequate examples.

Karczmarczuk J.: Icon programming language (1)

INFORMATYKA 1986, No. 2—3, p. 11

First part of characteristics of Icon programming language destined mainly for analysing and complex text processing, as well as for artificial intelligence problems solving.

Bielecki J.: Advanced constructions of C language

INFORMATYKA 1986, No. 2—3, p. 14

Characteristics of constructing principles of C language expressions and discussion of programs set, which includes advanced syntax and semantic constructions.

Szkaradnik Z.: Floating-point operations in Forth

INFORMATYKA 1986, No. 2—3, p. 17

Extension of Forth translator for floating-point operations package, which is realised by the author on Mera 60 microcomputer.

Zakręcki J.: Abstractions in programming (2)

INFORMATYKA 1986, No. 2—3, p. 22

Termination of presentation on abstraction in programming problems. An example in Ada and relationship between abstraction application and programs modularization are discussed.

Kotulski L.: Monitor as a tool for concurrent programs structuralization

INFORMATYKA 1986, No. 2—3, p. 24

Characteristics of programming monitor idea and functions. Solutions, which assure concurrent programs structuralization, are discussed in details.

Kleiber M., Leśny M., Szuniewicz R.: Personal computer in professional applications (1)

INFORMATYKA 1986, No. 2—3, p. 27

First part of personal computer characteristics from application features point of view. Microcomputer hardware features and application software, which includes text editing and table calculation programs, are presented.

Stokłosa J.: Abraham Stern — the first polish arithmetic machines designer

INFORMATYKA 1986, No. 2—3, p. 31

Concise characteristics of activity and achievements of polish pioneer in building computer at the begin of the XIX century.

Postół M.: Modification of PASCAL/MT+ compilation

INFORMATYKA 1986, No. 2—3, p. 32

Proposal for improving compilation process of Pascal/MT+ programs, which are destined for specialized microcomputers, for example controlers.

Wirth N.: Multiprogrammierungsgrundsätze und ihre Implementierung in Modula-2

INFORMATYKA 1986, Nr. 2—3, S. 1

Eine Charakteristik der verschiedenen Realisierungsgrundsätze von Multiprogrammierung und detaillierte Besprechung ihrer Implementierung in Modula-2 mit Angabe von entsprechenden Beispielen.

Karczmarczuk J.: Icon-Programmiersprache (1)

INFORMATYKA 1986, Nr. 2—3, S. 11

Erster Teil einer Charakteristik von Icon-Programmiersprache, die hauptsächlich für Analyse und Verarbeitung von zusammengetzten Texte, sowie für Lösung von Problemen aus dem Bereich der künstlichen Intelligenz, erarbeitet wurde.

Bielecki J.: Fortgeschrittene Konstruktionen der C-Sprache

INFORMATYKA 1986, Nr. 2—3, S. 14

Eine Charakteristik von Konstruirungsgrunsätzen der Ausdrücke in der C-Sprache und eine Beschreibung von Programmensatz mit syntaktisch und semantisch fortgeschrittenen Konstruktionen.

Szkaradnik Z.: Gleitkommaoperationen in Forth

INFORMATYKA 1986, Nr. 2—3, S. 17

Eine Erweiterung des Forth-Übersetzers um Gleitkommaoperationenpaket, die vom Autor auf Mera 60 Mikrorechner realisiert wurde.

Zakręcki J.: Abstraktionen in Programmierung (2)

INFORMATYKA 1986, Nr. 2—3, S. 22

Beendigung einer Besprechung von Abstraktionen in Programmierung. Es wurde ein Beispiel in Ada angegeben, sowie Beziehungen zwischen Abstraktionsanwendung und Programmemodularisation vorgestellt.

Kotulski L.: Monitor als Hilfsmittel zur Strukturalisierung der parallel ablaufenden Programme (1)

INFORMATYKA 1986, Nr. 2—3, S. 24

Eine Charakteristik des Begriffes und der Funktionen eines Programmonitors. Es wurden detaillierte Lösungen, die eine Strukturalisierung der parallel ablaufenden Programme ermöglichen, besprochen.

Kleiber M., Leśny M., Szuniewicz R.: Personal Computer in professionellen Anwendungen (1)

INFORMATYKA 1986, Nr. 2—3, S. 27

Erster Teil einer Charakteristik von Personal Computern aus Gesichtspunkt ihrer Nutzmerkmale. Es wurden Hardware- und Anwendungssoftwaremerkmale, die Textredigierungs- und Tabellenberechnungsprogramme umfasst, besprochen.

Stokłosa J.: Abraham Stern — erster polnischer Konstrukteur von arithmetischen Maschinen

INFORMATYKA 1986, Nr. 2—3, S. 31

Kurzgefasste Charakteristik von Tätigkeit und Errungenschaften des polnischen Pioniers des Rechenmaschinenbaues aus den Anfängen des 19. Jahrhunderts.

Postół M.: Modifikation des PASCAL/MT+ Kompilationsprozesses

INFORMATYKA 1986, Nr. 2—3, S. 32

Ein Vorschlag für Rationalisierung des Kompilationsprozesses von den in Pascal/MT+ geschriebenen Programmen, die für spezialisierte Mikrorechner, z.B. Steueranlagen, verwendet werden.

CENTRUM INFORMATYKI GOSPODARKI MORSKIEJ

Przedsiębiorstwo Państwowe

Poleca po cenach konkurencyjnych półprzewodnikowe pamięci operacyjne do komputerów:

ODRA 1305 — do 512 K słów
ODRA 1325 — do 64 K słów
MERA 9150 — do 64 K słów
ICL 2900 — do 256 K słów

Wyroby nasze zainstalowane w wielu przedsiębiorstwach na terenie kraju charakteryzują się:

— niezawodnością działania
— wysokim poziomem technologii
— wysoką jakością wykonania
— nowoczesnością rozwiązań opartych na własnych patentach.

zapewnia:

- **natychmiastową realizację zamówień,**
- **dwuletnią gwarancję,**
- **dostępność części przez 10 lat,**
- **serwis techniczny w ciągu 24 godzin,**
- **dokumentację i szkolenie.**

Nasz adres:

ul. Heweliusza 11
80-890 Gdańsk
tel.: 31-83-57
teleks: 051 2951

EO/88/K/86

COMPUTER STUDIO KAJKOWSCY

ul. Balladyny 3B, 81-524 Gdynia
Tel.: 29-00-18, 24-01-50

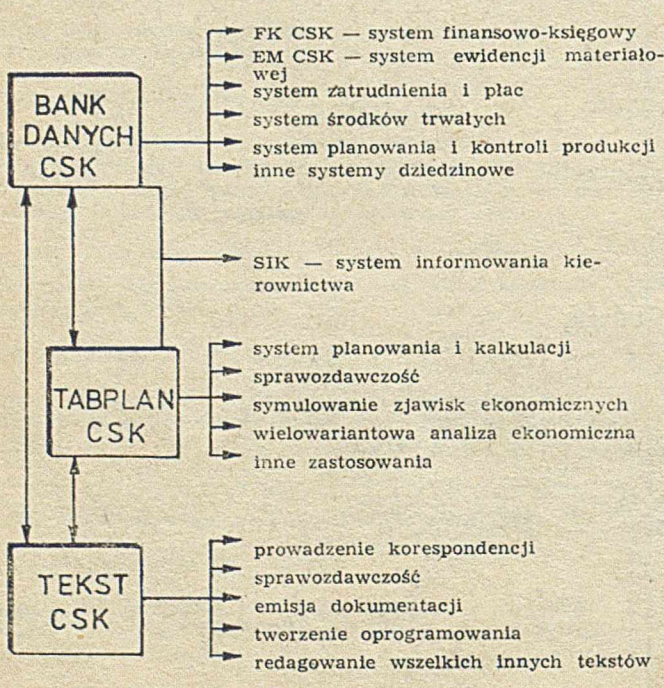
Oprogramowanie z CSK dla potrzeb zarządzania

(dla mikrokomputerów 8- i 16-bitowych)

Oferta programowa CSK obejmuje programy, które służą do automatyzacji przetwarzania danych we wszystkich dziedzinach funkcjonowania przedsiębiorstw:

- do prowadzenia korespondencji, ewidencji i rozliczeń finansowych,
- do wspomagania decyzji na szczeblu dyrektora.

Miejsce oprogramowania CSK w zautomatyzowanym systemie przetwarzania danych w przedsiębiorstwie przedstawia rysunek.



PROFESJONALNE OPROGRAMOWANIE MIKROKOMPUTERÓW

CSK oferuje następujące oprogramowanie użytkowe i systemowe dla mikrokomputerów 8- i 16-bitowych

Nazwa programu	8-bitowe	16-bitowe
Oprogramowanie użytkowe		
BANK-DANYCH CSK — system zarządzania bazą danych	+	+
TABPLAN CSK — komputerowy arkusz kalkulacyjny	+	+
TEKST CSK — pakiet redagowania tekstów	+	+
TRANSCOM CSK — program komunikacji z ODRĄ	+	+
TRANSCOM/M CSK — program komunikacji między mikrokomputerami	+	+
BANK-CSK — graficzny system komunikacji z bazą danych	—	+
BGRAF CSK — pakiet grafiki prezentacyjnej	—	+
FK CSK — system finansowo-księgowy	+	+
EM CSK — system ewidencji materiałowej	+	+
PL TEKST CSK — pakiet redagowania tekstów (polskie znaki)	—	+
Oprogramowanie systemowe		
SOMIK — rozbudowa systemu operacyjnego CP/M 2.0	+	—
W/SYS CSK — system operacyjny wielozadaniowy/wielostanowiskowy	+	+
GSK CSK — pakiet procedur graficznych wg normy GKS	—	+

+ tak; — nie

Oprogramowanie CSK może być eksploatowane na **mikrokomputerach 8-bitowych:**

- ELWRO seria 500 i 600
- ROBOTRON 5110/20/30, 1715
- MK 4101/02
- ComPAN
- IMP-85
- innych (z systemem CP/M)

mikrokomputerach 16-bitowych:

— LIDIA II/XT
— MAZOWIA
— ELWRO 800
— M24 (Olivetti)
— innych (zgodnych z IBM PC XT i AT).

Cena obejmuje: dyskietki z programem (wraz z kopią), jeden egzemplarz dokumentacji użytkowej. Termin realizacji — 7 dni od daty otrzymania zamówienia.
CSK organizuje kursy użytkowania oprogramowania.

EO/81/K/96