```
var i: integer;
unit p: procedure ... end p;
handlers when znaleziony:
            writeln(" ciąg indeksów:");
            terminate
        end handlers;
begin
for i:=lower(A) to upper(A) do call p(0,i) od;
writeln(" taki ciąg nie istnieje");
last_will: writeln(" koniec ciągu indeksów")
end problem_plecakowy;
```

Zauważmy, że w czasie wykonywania pętli **for**, występującej w treści procedury **problem_plecakowy** procedura **p** może zasygnalizować wystąpienie wyjątku **znaleziony**. W trakcie obsługi tego wyjątku zostanie wypisana informacja „ciąg indeksów:", a następnie wykonana instrukcja **terminate**. Spowoduje ona wykonanie epilogów kończonych obiektów, zostaną więc wypisane znalezione indeksy, a następnie — przy kończeniu obiektu procedury **problem_plecakowy**, w której znajduje się moduł obsługi wyjątku — tekst „koniec ciągu indeksów". Informacja, że szukany ciąg indeksów nie istnieje, zostanie wypisana jedynie wówczas, gdy zostanie zakończone wykonanie pętli **for** bez zgłoszenia wyjątku.

Rozwiązanie 7.3.4

```
unit problem_plecakowy: procedure(A: array_of integer,
                                  v: integer);
    signal znaleziony;
    var i: integer;
    unit p: procedure(s,k: integer);
        var i: integer;
        begin
        if s+A(k)>v then return fi;
        if s+A(k)=v then raise znaleziony fi;
        s:=s+A(k);
        for i:=k to upper(A) do call p(s,i) od;
        last_will: writeln(k);
        end p;
    handlers when znaleziony: writeln(" ciąg indeksów:");
                terminate
            end handlers;
    begin
```



```
for i:=lower(A) to upper(A) do call p(0,i) od;
writeln(" taki ciąg indeksów nie istnieje");
last_will: writeln(" koniec ciągu indeksów")
end problem_plecakowy;
```

Bezpieczne dzielenie funkcji rzeczywistych — parametryzacja sygnałów 7.4

Sformułowanie problemu 7.4.1

Zaprojektuj bezpieczny algorytm dzielenia danych funkcji rzeczywistych. Przez bezpieczeństwo algorytmu rozumiemy konieczność wykrywania sytuacji, w których mogą wystąpić błędy wykonania programu i reagowanie na nie.

Dyskusja zadania 7.4.2

W naszym zadaniu jedynym błędem wykonania programu, jaki może być spowodowany samym dzieleniem funkcji rzeczywistych, jest błąd dzielenia przez zero. Z podobnym rodzajem błędu mieliśmy już do czynienia przy rozwiązywaniu układów równań liniowych (por. p. 7.2). Sytuacja występująca przy dzieleniu funkcji jest jednak odmienna, w tej bowiem sytuacji dzieleniu przez zero można nadać dobrze określone znaczenie. Możliwym rozwiązaniem jest np. przyjęcie, iż wynikiem jest granica ilorazu funkcji. Decyzja co do sposobu reakcji na dzielenie przez zero jest zależna od użytkownika, powinien on więc mieć możliwość modyfikacji wyniku dzielenia, zależnie od swoich potrzeb. Korzystając z konstrukcji Loglanu omówionych w rozdz. 1 - 6 użytkownik funkcji nie ma dostępu z zewnątrz do zmiennej **result**. Jest jednak możliwe uzyskanie tego dostępu przez parametryzację sygnałów. Rozwiązanie to omówimy w dalszej części rozdziału.

Podkreślmy, iż w naszym przykładzie omawiamy problem dzielenia przez zero jedynie dla uproszczenia rozważań. Można sobie wyobrazić wiele sytuacji, w których podobna modyfikacja wyniku funkcji jest bardzo pożytecznym i czytelnym narzędziem.

Omówienie rozwiązania 7.4.3

Podobnie jak w p. 7.2, będziemy dążyć do zgromadzenia wszystkich niezbędnych definicji w ramach jednego modułu. Podkreślmy jeszcze raz, iż

takie rozwiązanie umożliwia zdefiniowanie standardowej metody obsługi sygnału i, dzięki prefiksowaniu, ewentualną zmianę tej metody przez użytkownika. Określimy więc bardzo prosty język problemowy (z jedną operacją — dzielenia) o następującym schemacie:

```
unit dzielenie: class;
    ...
    unit dziel: function(a: real;
          function f(x:real): real;
          function g(x:real): real): real;
        (* wynikiem jest dzielenie f(a)/g(a) lub
          wysłanie odpowiedniego sygnału, gdy
          g(a)=0 *)
        end dziel;
 end dzielenie;
```

Zauważmy, że sygnały systemowe nie mają parametrów. Aby rozwiązać nasze zadanie wprowadzimy więc sygnał o nazwie **zero**. Ponieważ chcemy mieć możliwość zmiany zmiennej **result** występującej w funkcji **dziel**, wyposażymy ten sygnał w wyjściową zmienną **wynik**

```
signal zero(output wynik: real;...);
```

Funkcja **dziel** powinna przechwytywać systemowy sygnał **numerror**, w naszym przypadku informujący o dzieleniu przez zero, a następnie wysłać sygnał **zero**. Ważną informację w ustaleniu prawidłowego wyniku stanowią też funkcje **f** i **g**, a także wartość punktu **a**, dla którego był obliczany iloraz. Dlatego też sygnał **zero** wyposażymy dodatkowo w następujące parametry:

```
signal zero(output wynik: real; input wartość: real;
 function f(x:real): real;
 function g(x:real): real);
```

Jeśli użytkownik nie zdefiniuje swojego modułu obsługi sygnału **zero**, będzie to oznaczać, iż sytuacja błędna nie była przez niego przewidziana. Jako standardową reakcję przyjmiemy zatem wypisanie stosownej informacji i zakończenie obliczeń programu.

```
handlers when zero:
          writeln(" wystąpił błąd dzielenia",
              "wartości", f(wartość), "przez zero")
      end handlers;
```

Zwróćmy uwagę, iż funkcja **f** oraz zmienna **wartość**, występujące w instrukcji wypisania komunikatu, są dane jako parametry formalne sygnału **zero**.

Użytkownik może zdefiniować inną metodę obsługi sygnału **zero**, np.

```
pref dzielenie block
   unit granica: function(a: real;
                 function f(x:real):real;
                 function g(x:real):real): real;
      (* oblicza granicę ilorazu f(x)/g(x) przy x
        dążącym do a *) ...
      end granica;
 handlers when zero: wynik:=granica(wartość,f,g);
          (* wynik, f, g oraz wartość
            są parametrami sygnału zero *)
          return
      end handlers;
end;
```

Rozwiązanie 7.4.4

```
unit dzielenie: class;
   signal zero(output wynik: real; input wartość: real;
          function f(x:real): real;
          function g(x:real): real);
   unit dziel: function(a: real;
                 function f(x:real): real;
                 function g(x:real): real): real;
      handlers when numerror: raise zero(result,a,f,g);
          end handlers;
      begin result:=f(a)/g(a)
      end dziel;
   handlers when zero: writeln(" wystąpił błąd dzielenia",
              " wartości", f(wartość), "przez zero")
        end handlers;
   end dzielenie;
```

Podsumowanie 7.5

(1) Mechanizm obsługi sytuacji wyjątkowych umożliwia reakcję na wystąpienie zdarzenia wyjątkowego w czasie wykonania programu, np. błędu.

(2) Wystąpienie sytuacji wyjątkowej jest sygnalizowane przez wysłanie odpowiedniego sygnału za pomocą instrukcji **raise**. Sygnały są defi-

niowane w specyfikacjach **signal**. Reakcją na zgłoszenie sygnału jest wykonanie odpowiedniego modułu obsługi zadeklarowanego w specyfikacji **handlers**.

(3) Sygnały mogą być sparametryzowane. Lista parametrów formalnych jest podawana przy deklaracji sygnału, odpowiadająca zaś jej lista parametrów aktualnych w chwili zgłoszenia sygnału. Parametry są przekazywane do oraz z modułu obsługi sygnału.

(4) Moduł obsługi sygnału, który ma zostać wykonany, jest wyszukiwany wzdłuż łańcucha dynamicznego obiektu, w którym wystąpiła sygnalizacja. Reguły widoczności modułów obsługi sygnałów w modułach prefiksowanych są takie same, jak dla podprogramów wirtualnych.

(5) W module obsługi sygnału może wystąpić jedna z instrukcji określających sposób kontynuacji programu po obsłudze sytuacji wyjątkowej. Możliwy jest powrót do miejsca zgłoszenia sygnału (instrukcja **return**) lub nienormalne zakończenie wykonywania obiektów z aktywnego łańcucha współprogramu leżących między obiektem, w którym wystąpiła sygnalizacja a obiektem zawierającym moduł obsługi sygnału (włącznie z tym ostatnim — instrukcja **terminate** — lub bez niego — instrukcja **wind**). Standardowo jest wykonywana instrukcja **terminate**.

(6) Zakończenie wykonywania obiektu przez **terminate** lub **wind** jest związane z wykonaniem instrukcji końcowych przewidzianych na tę okazję. Instrukcje te są definiowane w klauzuli **last_will**.

(7) W Loglanie są zdefiniowane sygnały standardowe odpowiadające błędom występującym w czasie wykonania programu. Sygnały te są zgłaszane automatycznie w chwili wystąpienia błędu. Jest możliwe zdefiniowanie dla nich własnych modułów obsługi. W modułach tych nie może wystąpić instrukcja **return**.

Procesy współbieżne 8

Wprowadzenie 8.1

Podobnie jak wiele współcześnie rozwijanych języków programowania, Loglan umożliwia programowanie *procesów współbieżnych*. Zgodnie z koncepcją przyjętą w języku proces stanowi rozszerzenie pojęcia współprogramu. Obiekty współprogramów działają niezależnie od siebie, lecz w danej chwili tylko jeden obiekt może być aktywny. Natomiast w przypadku procesów, wznowienie dowolnego obiektu nie powoduje zawieszenia obiektu aktywnego. W danej chwili może być zatem wiele aktywnych obiektów procesów.

Procesy, jako rozszerzenie współprogramów, są też rozszerzeniem klas. Deklaracja procesu różni się syntaktycznie od deklaracji klasy jedynie użyciem słowa **process** zamiast **class** (por. p. 8.2.3).

Obiekty procesów są tworzone podobnie do obiektów klas i współprogramów — za pomocą operatora **new**. Sposób tworzenia nowego obiektu procesu jest taki sam, jak dla współprogramu. Nowo utworzony obiekt procesu po wykonaniu instrukcji inicjujących przechodzi w stan zawieszenia. Aby mógł kontynuować obliczenia musi zatem zostać wznowiony. Parametry procesu są przekazywane między tworzonym obiektem procesu a obiektem, który spowodował jego utworzenie, tak samo jak w przypadku tworzenia klas i współprogramów. Do lokalnych atrybutów obiektu procesu można odwoływać się korzystając z odległego dostępu.

Jak już wspomnieliśmy — proces jest także współprogramem. Oznacza to, że stosują się do niego operacje charakterystyczne dla współprogramów — **attach** oraz **detach**. Procesy mogą być prefiksowane oraz same mogą być prefiksami innych modułów (por. tab. 5.1, p. 5.1). Również z punktu widzenia obsługi sytuacji wyjątkowych proces jest traktowany podobnie jak współprogram. W szczególności stosuje się do niego ta sama definicja łańcucha dynamicznego obiektów.

Program główny może być traktowany nie tylko jako współprogram,

ale także jako proces. Jest to jedyny proces aktywny w chwili rozpoczęcia obliczeń programu. Dostępny jest pod nazwą **main**.

Podobnie jak w przypadku współprogramów jest zdefiniowany standardowy typ **process** zawierający wszystkie typy procesów. Zatem każdy obiekt będący procesem należy do typu **process**. Wartością wyrażenia **this process** jest wskaźnik do obiektu procesu, czyli procesu, w którego łańcuchu dynamicznym znajduje się obiekt obliczający to wyrażenie.

Zbędny obiekt procesu można usunąć z pamięci za pomocą operacji **kill**. Jest to możliwe tylko wówczas, gdy dany proces nie jest aktywny. Jeżeli proces nie został zakończony, to wraz z jego obiektem są usuwane z pamięci wszystkie obiekty z jego łańcucha dynamicznego.

W Loglanie zostały przyjęte następujące operacje na procesach:

— **resume(X)** — wznowienie działania procesu X bez zawieszania procesu wznawiającego;

— **stop** — zatrzymanie działania procesu wykonującego tę instrukcję;

— **wait** — zawieszenie obliczeń w procesie do chwili zakończenia działania dowolnego spośród utworzonych przezeń procesów; jeśli taki proces nie istnieje, wykonanie **wait** jest równoważne instrukcji pustej; operacja **wait** może być również traktowana jako funkcja — wówczas, obok efektu oczekiwania, jako jej wartość uzyskujemy wskaźnik do zakończonego procesu.

Gdy kilka procesów w sposób asynchroniczny korzysta ze wspólnych zasobów lub danych, jest niezbędna ochrona zasobów przed jednoczesną zmianą przez różne procesy. Powoduje to konieczność wprowadzenia pojęć służących do synchronizacji procesów. W Loglanie, odmiennie niż w większości współczesnych języków programowania, do synchronizacji służą semafory binarne. Za pomocą tych prostych narzędzi synchronizacji użytkownik może, korzystając z bogatych możliwości prefiksowania, zdefiniować różnorodne strukturalne narzędzia synchronizacji i narzucić wybrany protokół komunikacji między procesami.

Typ danych opisujący semafory binarne definiujemy następująco:

semaphore = (*semaphore*; **ts, lock, unlock**)

gdzie

(1) nośnikiem jest zbiór obiektów mogących przyjmować dwie wartości: semafor podniesiony oraz semafor opuszczony (wartości te jest wygodnie utożsamiać z wartościami logicznymi, przy czym **true** oznacza semafor opuszczony, a **false** —podniesiony;

(2) **ts**: *semaphore* → *boolean*;

(3) **lock, unlock**: *semaphore* → *semaphore*.

Wynikiem funkcji **ts(s)** jest **true**, jeśli semafor **s** jest opuszczony, **false** w przeciwnym przypadku. Skutkiem ubocznym działania **ts(s)** jest opuszczenie semafora bez względu na jego wartość wejściową.

Operacje **lock** i **unlock** są zdefiniowane następująco:

— jeśli semafor **s** jest podniesiony, to **lock(s)** oznacza jedynie jego opuszczenie; gdy semafor **s** jest opuszczony, proces wykonujący **lock** zostaje zawieszony aż do chwili, w której zostanie on wznowiony na skutek wykonania przez inny proces operacji **unlock**;

— wykonanie operacji **unlock(s)** powoduje podniesienie semafora **s** w przypadku, gdy żaden proces nie czeka na jej wykonanie; w przeciwnym razie jeden z czekających procesów zostaje wznowiony, a semafor pozostaje opuszczony.

Zakłada się, że operacje **ts**, **lock** i **unlock** są niepodzielne, tzn. w danej chwili na tym samym semaforze mogą być wykonywane przez co najwyżej jeden proces. Własność ta jest zapewniona przez implementację operacji semaforowych.

Dla wygody programisty wprowadzono także wariant instrukcji **stop**, który może zmienić wartość semafora. Jeżeli instrukcja ta występuje z parametrem, który jest semaforem, to **stop(s)** oznacza niepodzielne wykonanie **unlock(s)** oraz **stop**.

Zmienne semaforowe deklaruje się tak jak inne zmienne, podając **semaphore** jako ich typ. Standardową wartością początkową tych zmiennych jest semafor podniesiony.

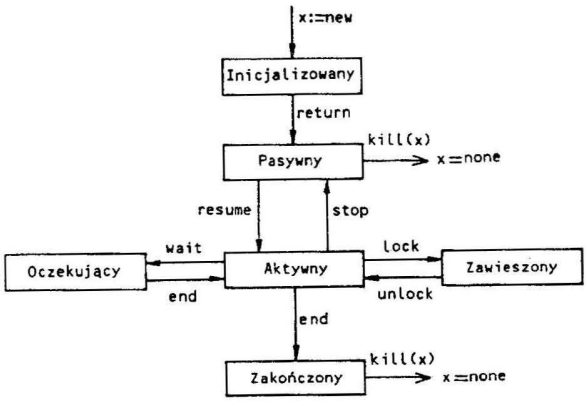
Na rysunku 8.1 przedstawiamy w sposób schematyczny wpływ operacji wykonywanych przez procesy na ich zachowanie. Zauważmy, że podobnie jak w przypadku współprogramowych operacji **attach** oraz **detach**, operacje związane ze zmianą stanu procesu dotyczą wykonującego je procesu — niekoniecznie tego, w którym tekstowo występują.

Opisany system procesów nie został dotychczas w pełni zrealizowany. Został natomiast opracowany i zrealizowany system procesów rozproszonych dla Loglanu. Jest on dostępny na sieciach mikrokomputerów IBM PC. System ten opisujemy w dodatku D.

Sortowanie szybkie — podstawowe własności procesów 8.2

Sformułowanie problemu 8.2.1

Dana jest tablica różnych liczb całkowitych. Posortuj elementy tej tablicy w porządku rosnącym.



RYS. 8.1 Schemat zmian stanów procesów współbieżnych

Dyskusja zadania 8.2.2

Z sortowaniem mieliśmy już do czynienia dwukrotnie — w p. 4.3 oraz 5.3. Dzięki zastosowaniu współbieżności można przyspieszyć proces sortowania. Zaadaptujemy w tym celu rozwiązanie sekwencyjne znane pod nazwą sortowania szybkiego (ang. quicksort). Załóżmy, że chcemy posortować tablicę A. Sortowanie szybkie polega na wykonaniu następującego (rekurencyjnego) algorytmu:

Wybierz pewien obiekt z tablicy A i nazwij go x.
Rozrzuć elementy tablicy A tak, aby po lewej stronie x znajdowały się elementy mniejsze od x, po prawej zaś elementy większe od x.
Posortuj (rekurencyjnie) elementy po lewej stronie x.
Posortuj (rekurencyjnie) elementy po prawej stronie x.

Ten algorytm można łatwo zrównoleglić — zamiast wykonywać sekwencyjnie dwie ostatnie fazy sortowania (rekurencyjne wołania) — można utworzyć dwa procesy, które wykonają te fazy współbieżnie. Ponieważ zrównoleglenie działania odbywa się na każdym poziomie rekurencyjnych wywołań, czas wykonania ulega znacznemu skróceniu. Zauważmy jednocześnie, że każdy z procesów operuje na innym fragmencie tablicy A, toteż problem równoczesnego dostępu do wspólnych zasobów nie musi być osobno rozwiązywany.

Rozwiązanie 8.2.3

Algorytm szybkiego sortowania zapiszemy jako proces:

```
unit sortowanie_szybkie: process(A:array_of integer,...);
 ... end sortowanie_szybkie;
```

Atrybutem procesu będzie procedura rozrzucająca elementy tablicy A. Rozrzucenia elementów tablicy dokonamy używając prostego, narzucającego się algorytmu. Niezbędne są także atrybuty procesu określające fragment tablicy, za posortowanie którego dany proces jest odpowiedzialny. Najwygodniej będzie zadeklarować je jako parametry procesu.

```
unit sortowanie_szybkie: process(A: array_of integer,l,p: integer);
```

Jako element x będziemy wybierać zawsze pierwszy element rozważanego fragmentu tablicy. Definicja procesu **sortowanie_szybkie** przyjmie więc taką oto postać:

```
unit sortowanie_szybkie: process (A: array_of integer,l,p: integer);
    ...
    begin
    return;
    rozrzuć elementy tablicy A przyjmując za x A(l);
    niech i będzie indeksem elementu x w tablicy
      uzyskanej po rozrzuceniu elementów;
    if l<i then
      lewy:=new sortowanie_szybkie(A,l,i-1);resume(lewy)
      fi;
    if i<p then
      prawy:=new sortowanie_szybkie(A,i,p);resume(prawy)
      fi;
    wait; wait; kill(lewy); kill(prawy)
end sortowanie_szybkie;
```

Zauważmy, że w końcowej sekwencji instrukcji tego procesu występuje dwukrotnie instrukcja **wait**. Oznacza ona oczekiwanie na zakończenie utworzonych wcześniej procesów **lewy** i **prawy** (w dowolnej kolejności). W szczególności proces, który spowoduje sortowanie tablicy będzie oczekiwał na zakończenie swoich procesów **lewy** i **prawy**; te z kolei na zakończenie swoich itd. W konsekwencji zostanie on wznowiony dopiero w chwili zakończenia sortowania całej tablicy. Operacje **kill** usuwają zbędne obiekty.

Rozważmy teraz przykład zastosowania procesu **sortowanie_szybkie**:

```
block
```

```
begin
...
resume(new sortowanie_szybkie
        (A,lower(A),upper(A));
wait; (* oczekiwanie na zakończenie sortowania *)
...
end;
```

W tym przykładzie został utworzony obiekt procesu sortującego (new), któremu przekazano sterowanie (resume). Następnie jest wykonywana instrukcja wait powodująca oczekiwanie na zakończenie sortowania. Wystąpienie tej instrukcji jest poprawne, ponieważ program główny jest też procesem (o nazwie main).

Program 8.2.4

```
unit sortowanie_szybkie: process(A: array_of integer,l, p:integer);
    var lewy, prawy: sortowanie_szybkie, i,j: integer;
    unit rozrzuć: procedure;
        var x,y: integer;
        begin
        i:=l; j:=p; x:=A(l);
        do
          while A(i)<x do i:=i+1 od;
          while x<A(j) do j:=j-1 od;
          if i<=j then
            y:=A(i); A(i):=A(j); a(j):=y;
            i:=i+1; j:=j-1 fi;
          if i>j then exit fi
          od
        end rozrzuć;
    begin
    return;
    call rozrzuć;
    if l<i then
      lewy:=new sortowanie_szybkie(A,l,i-1); resume(lewy)
      fi;
    if i<p then
      prawy:=new sortowanie_szybkie(A,i,p); resume(prawy)
      fi;
    wait; wait; kill(lewy); kill(prawy)
    end sortowanie_szybkie;
```


Wzajemne wykluczanie operacji typów danych — wykorzystanie semaforów 8.3

Sformułowanie problemu 8.3.1

Zaimplementuj ogólny mechanizm wzajemnego wykluczania operacji w strukturach danych.

Dyskusja zadania 8.3.2

Wzajemne wykluczanie jest istotnym problemem, pojawiającym się wraz z możliwością współdzielenia pewnej struktury danych przez procesy działające współbieżnie. Załóżmy na przykład, że kolejki omawiane w p. 4.4 mogą być wspólnym zasobem różnych procesów. W takiej sytuacji może się zdarzyć, iż w tej samej chwili jeden z procesów wykonuje operację usuń, a drugi operację pierwszy. Przypuśćmy, że kolejka jest jednoelementowa. Oczywiście operacje są wykonywane niezależnie od siebie. Może się więc zdarzyć, iż najpierw odbyło się sprawdzenie warunku początek=none w funkcji pierwszy (por. p. 4.4.4), a następnie całość procedury usuń. Do wykonania pozostała jeszcze instrukcja result:=początek.pierwszy_element. Jednak teraz początek=none, toteż wykonanie tej instrukcji spowoduje błąd wykonania programu, chociaż wcześniejsze sprawdzenie warunku miało zabezpieczyć właśnie przed tą możliwością.

Aby uniknąć podobnej sytuacji, zakłada się najczęściej, że operacje zdefiniowane w strukturze danych wzajemnie wykluczają się w czasie — w danym momencie może wykonywać się co najwyżej jedna z nich. Jeśli w trakcie jej wykonywania pewien proces zażąda wykonania kolejnej operacji, jego obliczenia będą wstrzymane do chwili zakończenia aktualnie wykonywanej.

Prostym i skutecznym sposobem rozwiązania naszego zadania jest zastosowanie semaforów i prefiksowania. Zdefiniujemy klasę realizującą wzajemne wykluczanie z klasą entry, zdefiniowaną wewnątrz. Klasa realizująca wzajemne wykluczanie będzie prefiksem dla zabezpieczanej struktury danych, a klasa entry — prefiksem dla operacji, które mają się wzajemnie wykluczać.

Omówienie rozwiązania 8.3.3

Zastanówmy się najpierw, jak można uzyskać w Loglanie wzajemne wykluczanie działania poszczególnych fragmentów programu. Oczywiście

należy uszeregować ich wykonanie w czasie (nie determinując jednak kolejności ich wykonania). Taką synchronizację można osiągnąć stosując zmienne semaforowe. Przypuśćmy, że I oraz J są fragmentami programu, których wykonanie powinno się wzajemnie wykluczać. Związemy z nimi zmienną semaforową, a je same otoczymy „nawiasami" lock i unlock:

```
var s: semaphore;
...
lock(s); I; unlock(s);
...
lock(s); J; unlock(s);
```

W tej sytuacji instrukcje I oraz J rzeczywiście nie będą mogły wykonywać się jednocześnie.

Nasze rozważania sugerują postać listy instrukcji klasy entry: powinna ona otaczać instrukcje modułu prefiksowanego „nawiasami" lock, unlock. Skutek ten łatwo osiągamy dzięki instrukcji inner.

```
unit entry: class;
    begin
    lock(s);
    inner;(* treść operacji prefiksowanej typem entry *)
    unlock(s)
end entry;
```

Klasę, która realizuje wzajemne wykluczanie nazwiemy **wykluczanie**. Omówimy teraz przykład jej zastosowania. Podamy w tym celu definicję kolejek rozważanych w p. 4.4.

```
unit kolejki: wykluczanie class(type E);
    ...
    unit wstaw: entry procedure(e: E); ... end wstaw;
    unit usuń: entry procedure; ... end usuń;
    unit pierwszy: entry function: E; ... end pierwszy;
    end kolejki;
```

W tak zdefiniowanej klasie **kolejki** będzie się wykluczać wykonanie operacji **wstaw**, **usuń** oraz **pierwszy**. Gdyby natomiast wiele procesów mogło jednocześnie wykonywać pewne z operacji struktury danych, wystarczyłoby opuścić prefiks **entry**.

Zauważmy, że uzyskane przez nas rozwiązanie jest ogólne — klasa **wykluczanie** może prefiksować dowolny typ danych, w którym zapewnienie wzajemnego wykluczania jest niezbędne do poprawnego jego funkcjonowania.

Rozwiązanie 8.3.4

```
unit wykluczanie: class;
    hidden s;
    var s: semaphore;
    unit entry: class;
        begin
        lock(s);
        inner;
        unlock(s)
        end entry;
    end wykluczanie;
```

Monitor — strukturalne mechanizmy współpracy między procesami 8.4

Sformułowanie problemu 8.4.1

Zdefiniuj pojęcie *monitora*, tj. struktury danych rozszerzających klasę **wykluczanie** zdefiniowaną w poprzednim punkcie o kolejki procesów i dwie operacje: **delay** oraz **continue** określone na kolejkach procesów w następujący sposób:

— **delay(q)** oznacza zawieszenie procesu wykonującego tę instrukcję i wstawienie go do kolejki **q** z jednoczesnym umożliwieniem wykonywania operacji monitora przez inne procesy;

— **continue(q)** oznacza usunięcie pierwszego procesu z kolejki i jego uaktywnienie.

Dyskusja zadania 8.4.2

Monitor jest modułem, który gromadzi zmienne dzielone przez współpracujące procesy oraz wszystkie operacje działające na tych zmiennych. Kolejki procesów umożliwiają długoterminowe planowanie działania procesów. Na zmiennych monitora mogą być wykonywane operacje jedynie za pośrednictwem jego procedur. Podstawową cechą monitora jest wzajemne wykluczanie jego operacji.

Aby zdefiniować monitor skorzystamy z doświadczeń zebranych przy definiowaniu wzajemnego wykluczania. Użyjemy więc znów techniki prefiksowania. Wszystkie procedury i funkcje modułu definiowanego jako

monitor będą musiały być prefiksowane klasą **entry**, dane lokalne zaś powinny być wymienione na liście **hidden**.

Omówienie rozwiązania 8.4.3

Kolejki procesów zrealizujemy korzystając z definicji podanej w p. 4.4.4. Przekazując standardowy typ **process** zawierający wszystkie typy procesów jako argument klasy **kolejki** możemy utworzyć kolejkę dowolnych procesów, np.

```
q:=new kolejki(process).
```

Przypomnijmy teraz, że operator **this** zastosowany do typu **process** (**this process**) zwraca wskaźnik do bieżącego procesu. Zatem przekazanie argumentu **this process** do operacji **wstaw** kolejki **q** spowoduje wstawienie do niej wskaźnika do procesu, wykonującego tę operację.

W przypadku monitora klasa **entry** musi być bardziej rozbudowana niż analogiczna klasa zdefiniowana w p. 8.3.4. Niezbędne jest wyposażenie jej w operacje **delay** i **continue**.

Załóżymy jak poprzednio, że semaforem odpowiedzialnym za wzajemne wykluczanie będzie **s**. Wykonanie operacji **delay(q)** będzie polegać na wstawieniu aktywnego procesu do kolejki **q**, a następnie zatrzymaniu go z jednoczesnym wykonaniem **unlock(s)**, które udostępnia monitor innym procesom. Wykonanie operacji **continue(q)** będzie polegać na aktywowaniu pierwszego procesu z kolejki **q** i usunięciu go z niej. Jednocześnie, jeśli wykonanie operacji **continue** spowodowało wznowienie procesu, na zmienną logiczną **zajęty** zostanie podstawiona wartość **true** oznaczająca, że monitor nadal nie może być udostępniony pozostałym procesom.

Jako przykład użycia monitora rozważmy bufor komunikatów, służący do magazynowania wartości przesyłanych między procesami. Chcemy zdefiniować dwie operacje — odczytu i zapisu wartości. Zakładamy przy tym, że bufor ma ograniczoną pojemność, a zatem próba zapisu przy zapełnionym buforze, podobnie jak próba odczytu z pustego bufora powoduje zawieszenie procesu do chwili, w której wykonanie zapisu (odczytu) będzie możliwe. Przyjmiemy, że typem komunikatów jest **T**. Bufor zdefiniujemy jako monitor.

```
unit bufor: monitor class(type T; rozmiar: integer);
    hidden buf, licznik, we, wy, kolejka_czytających,
        kolejka piszących;
    var buf: array_of T,
        licznik, we, wy: integer,
```



```
    kolejka_czytających, kolejka_piszących: kolejki;
unit zapisz: entry procedure(t: T);
    begin
    if licznik=rozmiar then
        call delay(kolejka_piszących)
        fi;
    we:=we mod rozmiar +1; licznik:=licznik+1;
    buf(we):=t;
    call continue(kolejka_czytających)
    end zapisz;
unit odczytaj: entry function: T;
    begin
    if licznik=0 then
        call delay(kolejka_czytających) fi;
    wy:=wy mod rozmiar +1; licznik:=licznik-1;
    result:=buf(wy);
    call continue(kolejka_piszących);
    end odczytaj;
begin
array buf dim(1:rozmiar);
kolejka_czytających:=new kolejki(process);
kolejka_piszących:=new kolejki(process)
end bufor;
```

Rozwiązanie 8.4.4

```
unit monitor: class;
    hidden s;
    var s: semaphore;
    unit kolejki: class(type E);
        ...(* por. p. 4.4 *) ...
        end kolejki;
    unit entry: class;
        var zajęty: boolean;
        hidden zajęty;
        unit delay: procedure(q: kolejki);
            begin
            call q.wstaw(this process);
            stop(s)
            end delay;
        unit continue: procedure(q: kolejki);
            var p: process;
```

```
            begin
            if q.pierwszy=/=none then
               zajęty:=true; p:=q.pierwszy;
               call q.usuń; resume(p)
               fi
            end continue;
         begin
         lock(s);
         inner;
         if not zajęty then unlock(s) fi
         end entry;
      end monitor;
```

Rozwiązywanie dużych układów równań liniowych — zastosowanie współbieżności i prefiksowania 8.5

Sformułowanie problemu 8.5.1

Jest dana kwadratowa macierz rzeczywista $\mathbf{A}$ rozmiaru $n \times n$ oraz n-elementowy wektor $\boldsymbol{b}$ liczb rzeczywistych. Znajdź taki wektor $\boldsymbol{x}$, że $\mathbf{A} * \boldsymbol{x} = \boldsymbol{b}$. Zakładamy przy tym, że n jest bardzo duże (np. rzędu 10000).

Dyskusja zadania 8.5.2

W punkcie 2.3 omówiliśmy prostą metodę rozwiązywania układów równań liniowych. Zakładaliśmy wówczas, że macierz wejściowa jest macierzą trójkątną górną tzn. wszystkie jej elementy leżące nad główną przekątną są zerami. Metodę tę można oczywiście zmodyfikować tak, aby można było ją stosować do innych rodzajów macierzy. Zauważmy jednak, że nie rozwiązuje to naszego zadania, gdyż ze względu na duże rozmiary macierzy A niemożliwe jest zmieszczenie jej jako tablicy kwadratowej w pamięci wewnętrznej maszyny, a ponadto rozwiązanie staje się zbyt czasochłonne.

Innym sposobem rozwiązywania podobnych układów równań są metody iteracyjne. Ich zaletą jest możliwość stosunkowo szybkiego wyznaczenia przybliżenia rozwiązania z zadaną dokładnością, a ponadto nie wymagają one tak dużej pamięci wewnętrznej. Dodatkowe przyspieszenie działania algorytmu można uzyskać dzięki zastosowaniu obliczeń współbieżnych.

Przyjęte przez nas rozwiązanie nosi nazwę metody iteracji prostej. Dzięki zastosowaniu prefiksowania i funkcji wirtualnych zaprogramujemy

je w sposób ogólny. Poszczególne warianty metody iteracji prostej (np. metodę Jacobiego, Gaussa-Seidla, SOR i inne) będzie można uzyskiwać dzięki mechanizmowi prefiksowania jako konkretyzacje ogólnego algorytmu.

Metoda iteracji prostej polega na przejściu od danego układu równań $\mathbf{A} * \boldsymbol{x} = \boldsymbol{b}$ do układu mającego te same rozwiązania, lecz zmienioną postać: $\boldsymbol{x} = \mathbf{B} * \boldsymbol{x} + \boldsymbol{c}$. Znając nową postać, kolejne przybliżenia rozwiązania znajdujemy ze wzoru:

$$\boldsymbol{x}_{i+1} = \mathbf{B} * \boldsymbol{x}_i + \boldsymbol{c}$$

gdzie $\boldsymbol{x}_i$ oznacza i-te przybliżenie. Przyjmiemy, że x_0 jest dane przez użytkownika jako przybliżenie początkowe.

Warunkiem koniecznym i wystarczającym zbieżności powyższego ciągu przybliżeń jest, aby promień spektralny macierzy $\mathbf{B}$ był mniejszy od jedności. Przypomnijmy, że promieniem spektralnym macierzy $\mathbf{B}$ nazywamy największy moduł z jej wartości własnych, wartością własną zaś — liczbę rzeczywistą b o tej własności, że istnieje wektor $\boldsymbol{x}$ spełniający równość $\mathbf{B} * \boldsymbol{x} = b * \boldsymbol{x}$. W dalszej części tego rozdziału będziemy zakładać, że jest spełniony warunek zbieżności.

Przyjmiemy, że warunkiem zakończenia iteracji jest osiągnięcie stanu, w którym norma różnicy dwóch kolejnych przybliżeń rozwiązania jest nie większa niż pewna zadana wartość.

Jeżeli macierz $\mathbf{A}$ przedstawimy w postaci $\mathbf{A} = \mathbf{D} + \mathbf{L} + \mathbf{U}$, gdzie $\mathbf{D}$ jest macierzą diagonalną (tzn. o zerowych elementach poza przekątną), $\mathbf{L}$ i $\mathbf{U}$ są macierzami trójkątnymi (dolną i górną, odpowiednio) o zerowych elementach przekątniowych, to np.:

— w metodzie Jacobiego definiujemy:
$\mathbf{B} = -\mathbf{D}^{-1} * (\mathbf{L} + \mathbf{U})$, $\mathbf{C} = \mathbf{D}^{-1} * b$;

— w metodzie Gaussa-Seidla:
$\mathbf{B} = -(\mathbf{L} + \mathbf{D})^{-1} * \mathbf{U}$, $\mathbf{C} = (\mathbf{L} + \mathbf{D})^{-1} * b$

Aby powyższe rozważania miały sens, należy założyć, iż wszyskie elementy przekątniowe macierzy $\mathbf{A}$ są niezerowe.

Zauważmy, że do wyliczenia współrzędnych wektora $\boldsymbol{x}_{k+1}$, należy wyliczyć najpierw wszystkie współrzędne wektora $\boldsymbol{x}_k$, przy czym procesy obliczania i-tej i j-tej współrzędnej ($i =/= j$) są niezależne. Można je więc obliczać współbieżnie. Uwaga ta jest podstawą zastosowania w rozwiązaniu procesów współbieżnych.

Omówienie rozwiązania 8.5.3

Z każdą współrzędną związemy obliczający ją proces. Kolejne przybliżenie rozwiązania będzie zmienną dzieloną, gdyż wszystkie procesy jedynie odczytują (nie zmieniając) jego współrzędne. Również kolejne konstruowane rozwiązanie będzie zmienną dzieloną, ponieważ każdy proces ma dostęp do innej jego współrzędnej.

Problem synchronizacji działania procesów rozwiążemy za pomocą monitora zdefiniowanego w p. 8.4. Po wyliczeniu swojej współrzędnej każdy proces zgłosi się do monitora, w którym będzie badany warunek zakończenia iteracji. Kolejno zgłaszające się procesy będą wstawiane do kolejki. Jeśli po zgłoszeniu się wszystkich procesów warunek zakończenia iteracji jest spełniony, proces kończy obliczenia. W przeciwnym razie procesy są kolejno zwalniane z kolejki. Procedura zgłaszania się procesów do monitora przyjmie następującą postać:

```
unit zgłoś: entry procedure(i: integer);
    (* zgłoszenie procesu obliczjącego i-tą współrzędną;
      x jest nowo konstruowanym rozwiązaniem, zaś y
      rozwiązaniem znalezionym w poprzedniej iteracji *)
    begin
    norma:=norma + (x(i)-y(i))*(x(i)-y(i));
    if liczba procesów w kolejce =/= n-1 then
       zwiększ o 1 liczbę procesów w kolejce;
       call delay(q)
    else (* sprawdzenie warunku zakończenia iteracji *)
        if sqrt(norma)<=eps then koniec:=true
        else z:=x; x:=y; y:=z; norma:=0;
            liczba procesów w kolejce:=0 fi fi;
    call continue(q)
    end zgłoś;
```

Obliczanie i-tej współrzędnej kolejnego przybliżenia zależy od konkretnej metody oraz sposobu przechowywania macierzy (np. na pliku), nie może więc być zdefiniowane w sposób ogólny. Przyjmiemy zatem, że jest ono dane jako funkcja wirtualna, precyzowana później przez użytkownika na poziomie jego modułu.

```
unit virtual współrzędna: function(i: integer,
    poprzednie_przybliżenie: array_of real): real;
    end współrzędna;
```

Działanie procesów obliczających współrzędne przyjmie następującą postać:


```
unit proces_współrzędnej: process(i: integer);
    begin
    return;
    do y(i):=współrzędna(i,x);
      call mon.zgłoś(i);
      if koniec then exit fi
      od
    end proces_współrzędnej;
```

Algorytm rozwiązujący układ równań zakończy swoje działanie w momencie, gdy zakończą je wszystkie procesy obliczające współrzędne rozwiązania. Aby zaczekać na ten moment, w algorytmie umieścimy pętlę

```
while wait=/=none do od;
```

Skorzystamy przy tym z faktu, że wartością wait jest none tylko wtedy, gdy są zakończone wszystkie procesy utworzone przez proces wywołujący wait (por. p. 8.1).

Rozwiązanie 8.5.4

```
unit duży_układ: class(n: integer, eps: real,;
                  inout x: array_of real);
    (* n --- rozmiar macierzy, eps --- dokładność,
      x --- przybliżenie początkowe, a zarazem wynik *)
    unit monitor_układu: monitor class;
        (*definicja klasy monitor podana jest w p. 8.4.4*)
        var norma: real, q: kolejki,
            liczba_procesów_w_kolejce: integer;
        unit zgłoś: entry procedure(i: integer);
            begin
            norma:=norma + (x(i)-y(i))*(x(i)-y(i));
            if liczba_procesów_w_kolejce=/=n-1 then
              liczba_procesów_w_kolejce:=
                liczba_procesów_w_kolejce + 1;
              call delay(q)
            else if sqrt(norma)<=eps then koniec:=true
                else z:=x; x:=y; y:=z; norma:=0;
                    liczba_procesów_w_kolejce:=0 fi fi;
            call continue(q)
            end zgłoś;
        begin
        q:=new kolejki(proces_współrzędnej)
```

```
    end monitor_układu;
unit proces_współrzędnej: process(i: integer);
    begin
    return;
    do y(i):=współrzędna(i,x);
      call mon.zgłoś(i);
      if koniec then exit fi
      od
    end proces_współrzędnej;
unit virtual współrzędna: function(i: integer,
       poprzednie_przybliżenie: array_of real): real;
     end współrzędna;
var y, z: array_of real, mon: monitor_układu,
   koniec: boolean, j: integer,
   procesy: array_of proces_współrzędnej;
begin
array y dim(1:n); array procesy dim(1:n);
mon:=new monitor_układu;
for j:=1 to n do
   procesy(j):=new proces_współrzędnej(j);
   resume(procesy(j)) od;
while wait=/=none do od;
for j:=1 to n do kill(procesy(j)) od;
kill(procesy); kill(y); kill(mon)
end duży_układ;
```

Podsumowanie 8.6

(1) Loglan umożliwia programowanie procesów współbieżnych. W jednej chwili wykonywania programu może być wiele aktywnych procesów.

(2) Procesy mają wiele własności klas. Deklaracja procesu różni się od deklaracji klasy jedynie zamianą słowa kluczowego **class** na **process**. Obiekty procesów są tworzone za pomocą operatora **new**. Utworzony obiekt procesu przechodzi w stan zawieszenia. Aby mógł kontynuować obliczenia, musi zostać wznowiony przez inny proces. Podobnie jak w przypadku obiektów klas, jest możliwy odległy dostęp do atrybutów obiektu procesu.

(3) Proces jest także współprogramem. Podobnie jak współprogram definiuje swój łańcuch dynamiczny. Stosują się do niego operacje **attach** i **detach**.

(4) Jest zdefiniowany standardowy typ **process**. Do tego typu należą wszystkie obiekty procesów. Znaczeniem wyrażenia **this process** jest wskaźnik do obiektu procesu obliczającego to wyrażenie (tzw. bieżącego procesu).

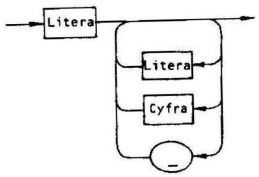
(5) Obiekt procesu może być aktywny lub zawieszony. Zdefiniowane są instrukcje zmieniające stan procesu. Instrukcja **resume(x)** powoduje wznowienie działania procesu x bez zawieszenia działania procesu wznawiającego. W wyniku wykonania instrukcji **stop** następuje zatrzymanie procesu ją wykonującego. Po wykonaniu instrukcji **wait** proces zawiesza swoje obliczenia do chwili zakończenia działania jakiegokolwiek utworzonego przezeń procesu. Operacja **wait** zastosowana jako funkcja zwraca wskaźnik do zakończonego procesu.

(6) Zakończony obiekt procesu jest równoważny zakończonemu obiektowi klasy. Może on zostać usunięty z pamięci za pomocą operatora **kill**. Także proces zawieszony może zostać usunięty z pamięci. W tym przypadku wraz z jego obiektem są usuwane z pamięci wszystkie obiekty z jego łańcucha dynamicznego.

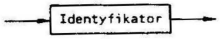
(7) Ze względu na możliwość jednoczesnego dostępu wielu procesów do wspólnych danych, Loglan oferuje mechanizmy synchronizacji zrealizowane za pomocą semaforów binarnych. Dla semaforów określone są operacje **lock**, **unlock** oraz funkcja **ts**. Jest też zdefiniowana operacja **stop(s)**, gdzie **s** jest zmienną semaforową. Polega ona na niepodzielnym wykonaniu **unlock(s)** oraz **stop**. Korzystając z bogatych możliwości prefiksowania użytkownik może zdefiniować własne strukturalne narzędzia synchronizacji oraz narzucić wybrany protokół komunikacji między procesami.

Dodatek A. Składnia Loglanu 82

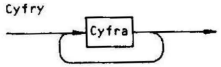
Identyfikator



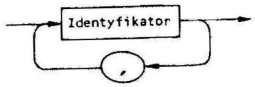
IdentStałej



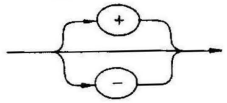
Cyfry



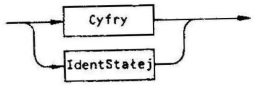
Identyfikatory



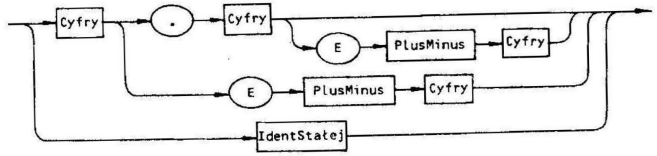
PlusMinus



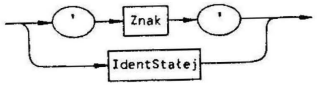
StałaCałkowita



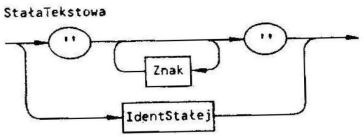
StałaRzeczywista



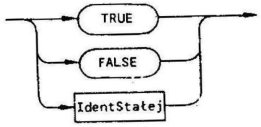
StałaZnakowa



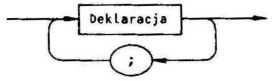
StałaTekstowa



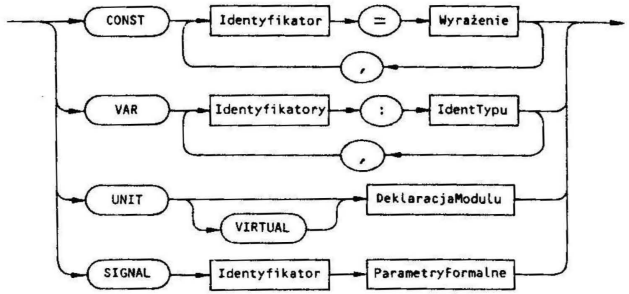
StałaLogiczna



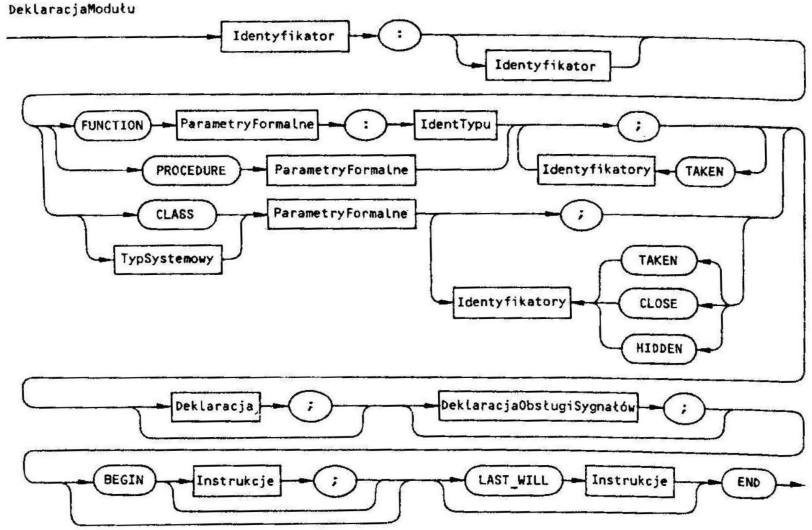
Dekralacje



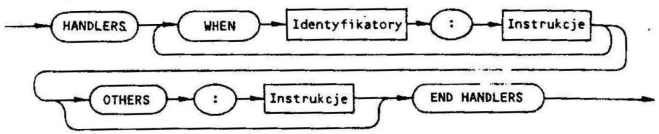
Deklaracje



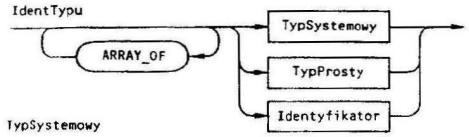
DeklaracjaModułu



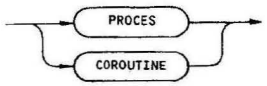
DeklaracjaObsługiSygnału



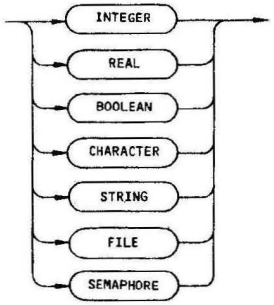
IdentTypu



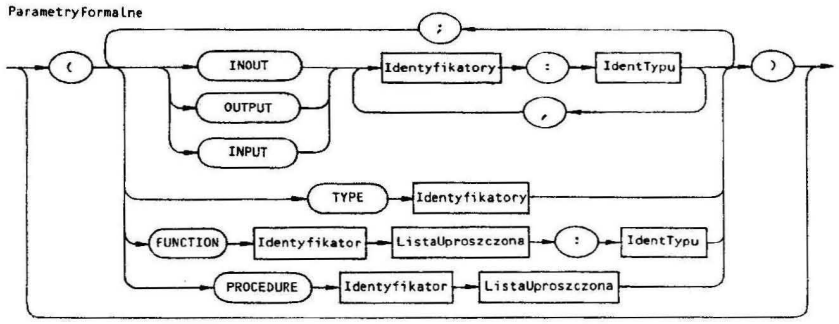
TypSystemowy



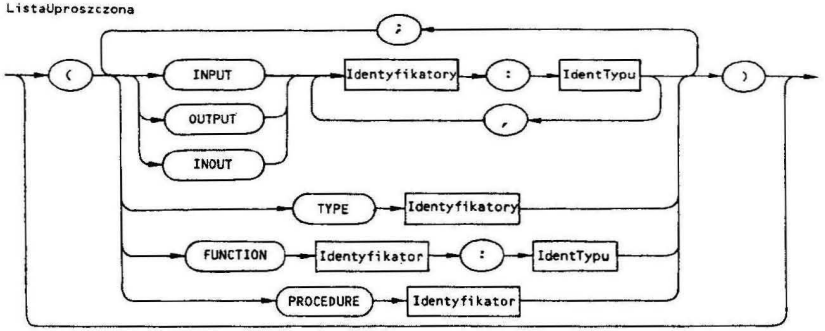
TypProsty



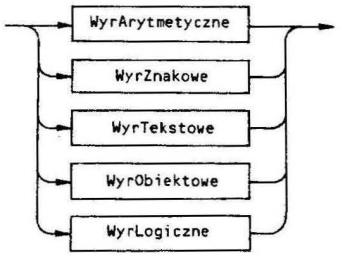
ParametryFormalne



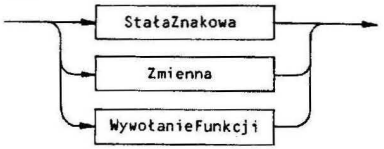
ListaUproszczona



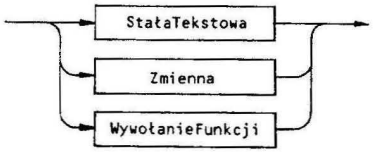
Wyrażenie



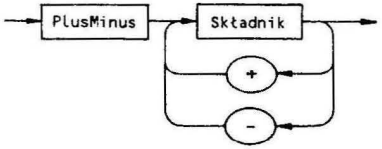
WyrZnakowe



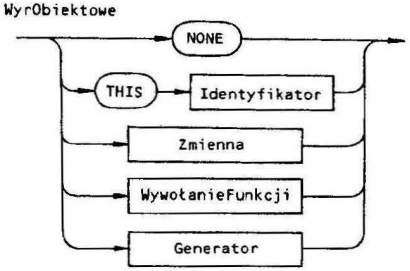
WyrTekstowe



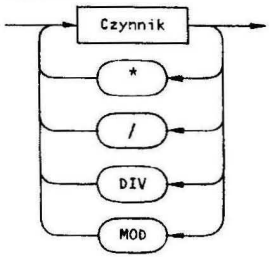
WyrArytmetyczne



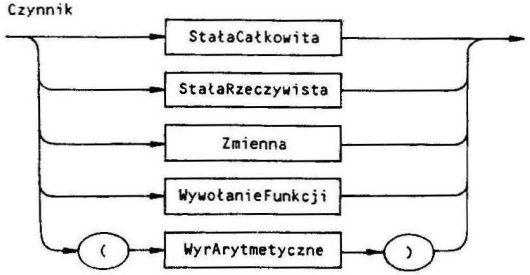
WyrObiektowe



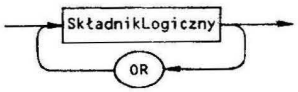
Składnik



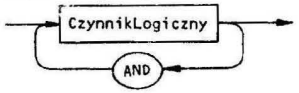
Czynnik



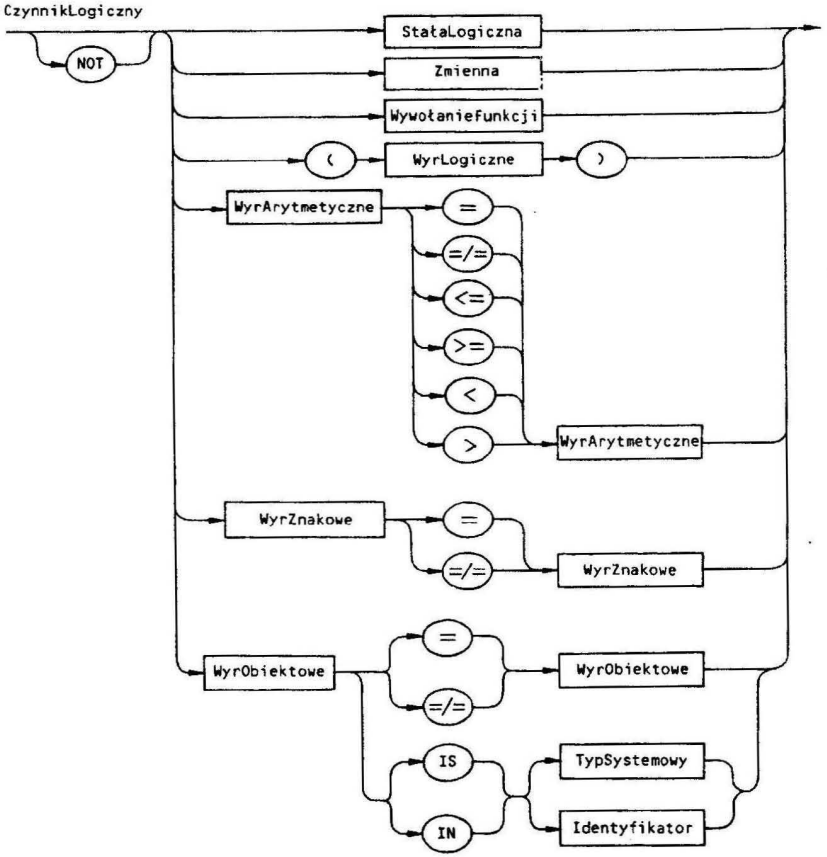
WyrLogiczne



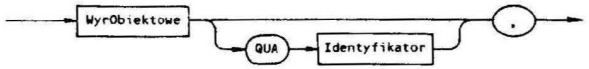
SkładnikLogiczny



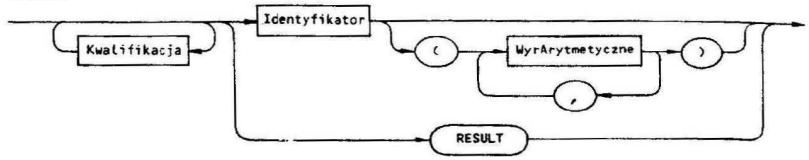
CzynnikLogiczny



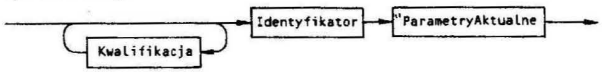
Kwalifikacja



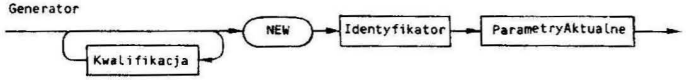
Zmienna



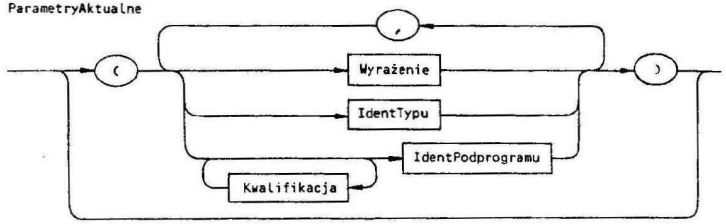
WywołanieFunkcji



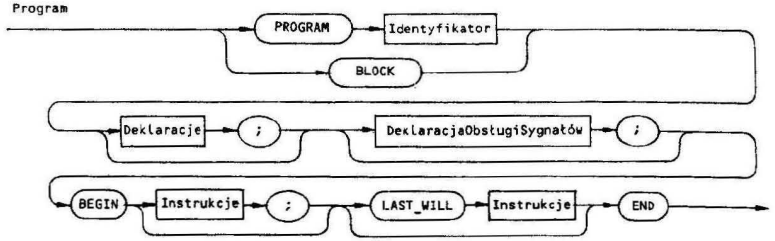
Generator



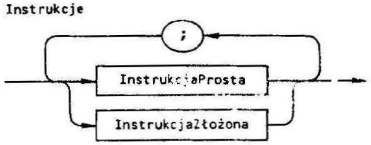
ParametryAktualne



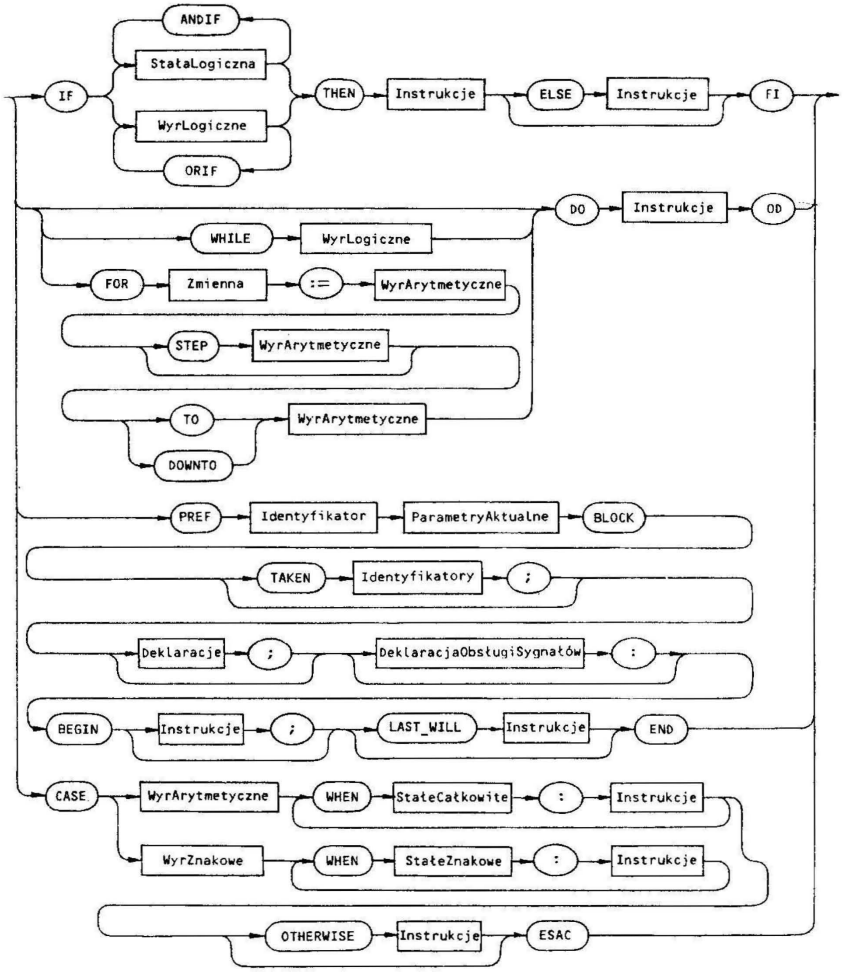
Program



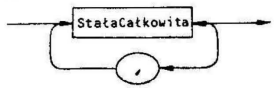
Instrukcje



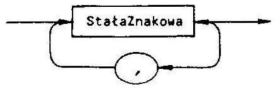
InstrukcjaZłożona



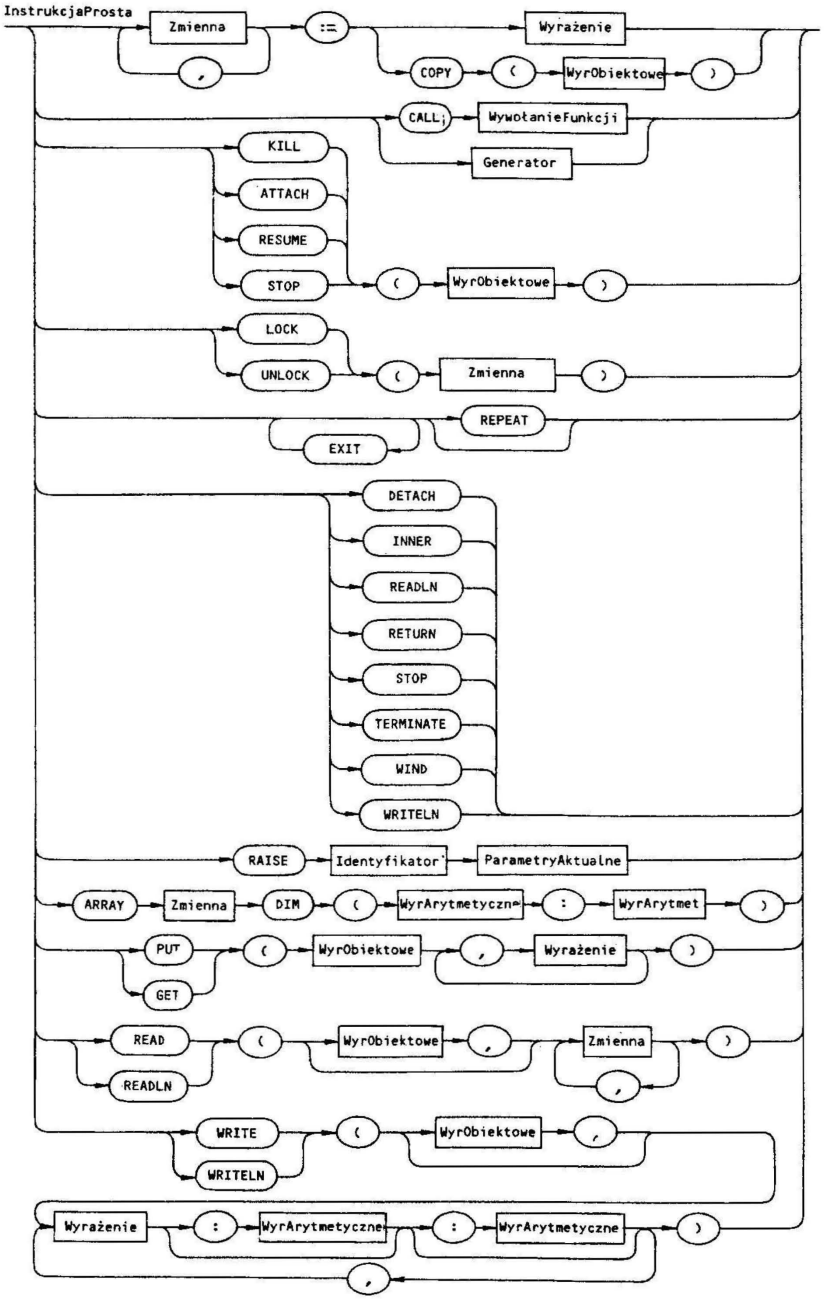
StałeCałkowite



StałeZnakowe



InstrukcjaProsta



Dodatek B. Procedury i funkcje standardowe (dla implementacji na IBM PC)

ENDRUN: procedure; — kończy obliczenie programu;

RANSET: procedure(x: real); — inicjalizuje generator liczb pseudolosowych (dla potrzeb funkcji RANDOM);

RANDOM: function: real; — zwraca liczbę pseudolosową z rozkładem równomiernym na przedziale (0,1);

SQRT: function(x: real): real; — oblicza pierwiastek kwadratowy z x;

SIN: function(x: real): real; — oblicza sinus parametru x;

COS: function(x: real): real; — oblicza cosinus parametru x;

TAN: function(x: reai): real; — oblicza tangens parametru x;

EXP: function(x: real): real; — oblicza e w potędze x;

LN: function(x: real): real; — oblicza logarytm naturalny z x;

ATAN: function(x: real): real; — oblicza arcus tangens parametru x;

ENTIER: function(x: real): integer; — oblicza część całkowitą parametru x;

ROUND: function(x: real): integer; — oblicza zaokrągloną wartość parametru x: ROUND(x) = ENTIER(x+0.5);

IMIN: function(x, y: integer): integer; — oblicza minimum z dwóch parametrów;

IMAX: function(x, y: integer): integer; — oblicza maksimum z dwóch parametrów;

IMIN3: function(x, y, z: integer): integer; — oblicza minimum z trzech parametrów;

IMAX3: function(x, y, z: integer): integer; — oblicza maksimum z trzech parametrów;

ISHIFT: function(x, k: integer): integer; — logicznie przesuwa x o k bitów; jeśli k jest dodatnie w lewo, w przeciwnym razie w prawo;

IAND: function(n, k: integer): integer; — daje w wyniku iloczyn logiczny parametrów (na wszystkich bitach);

IOR: function(n, k: integer): integer; — zwraca w wyniku sumę logiczną parametrów (na wszystkich bitach);

XOR: function(n, k: integer): integer; — zwraca w wyniku sumę rozłączną parametrów (na wszystkich bitach);

INOT: function(n: integer): integer; — daje w wyniku dopełnienie logiczne parametru (na wszystkich bitach);

ORD: function(c: char): integer; — daje w wyniku liczbę reprezentującą znak c;

CHR: function(n: integer): char; — zwraca w wyniku znak reprezentowany przez parametr (spełnione są następujące zależności: CHR(ORD(c)) = c oraz ORD(CHR(n)) = n);

UNPACK: function(s: string): array_of char; — zwraca odsyłacz do nowego obiektu tablicy zawierającego znaki z napisu s;

MEMAVAIL: function: integer; — daje w wyniku rozmiar dostępnej pamięci dla danego procesu (w słowach);

TIME: function: integer; — podaje czas (w sekundach) zużyty przez bieżący proces;

RESET: procedure(f: file); — ustawia wskaźnik pliku na początek i przygotowuje plik do czytania;

REWRITE: procedure(f: file); — ustawia wskaźnik pliku na początek oraz przygotowuje plik do zapisu; plik staje się pusty (eof(f)=true);

UNLINK: procedure(f: file); — zamyka i usuwa plik f.

Dodatek C. Zgodność typów. Reguły zastępowania podprogramów formalnych i wirtualnych

Zgodność typów C.1

Typ T nazwiemy dynamicznie zgodnym (w skrócie: zgodnym) z typem S, jeżeli przypisanie zmiennej x typu T wartości typu S jest poprawne (przypisanie takie może mieć miejsce albo podczas wykonywania instrukcji przypisania, albo w wyniku przekazania parametru będącego zmienną).

Typ T jest zgodny z typem S w następujących przypadkach:

(1) oba typy T i S są typami prostymi oraz zachodzi jeden z dwóch warunków:
(a) T = S;
(b) T, S ∈ {integer, real};
(2) typ T jest klasą, współprogramem lub procesem, S zaś jest klasą, współprogramem lub procesem zawartym w typie T (tzn. moduł T należy do ciągu prefiksowego modułu S);
(3) T i S są tablicami o tej samej liczbie wymiarów i tym samym typie elementów, nie będącym ani typem formalnym, ani tablicowym.

Poza dynamiczną zgodnością typów, która jest stwierdzana w czasie wykonania mamy określoną również statyczną zgodność typów odnoszącą się do czasu kompilacji. Dwa typy T i S są statycznie zgodne, jeżeli w czasie kompilacji nie sposób jest stwierdzić, że przypisanie x:=w zmiennej x typu T wyrażenia w typu S jest niepoprawne. Dane zestawienie określa warunki statycznej zgodności dla poszczególnych typów (niespełnienie żadnego z następujących warunków spowoduje błąd wykrywalny już w fazie kompilacji):

(1) T i S są typami prostymi oraz: T = S lub T, S ∈ {integer, real};
(2) T i S są klasami, współprogramami lub procesami oraz typ T jest zawarty w typie S lub typ S jest zawarty w typie T;
(3) jeden z typów T, S jest formalny, a drugi — formalny lub klasowy;

(4) T = array_of^i T1, S = array_of^j S1, gdzie T1, S1 nie są tablicowe, i,j>=0, i>0 lub j>0, a ponadto zachodzi jeden z przypadków:
(a) i=j, T1=S1;
(b) i=j, jeden z typów T1, S1 jest formalny, a drugi formalny lub klasowy;
(c) i>j, S1 — typ formalny;
(d) i<j, T1 — typ formalny.

Reguły zastępowania podprogramów formalnych i wirtualnych C.2

W Loglanie są możliwe dwie sytuacje, w których dochodzi do zastąpienia jednego podprogramu drugim:

(1) w przypadku podprogramów formalnych — wówczas podprogram aktualny zastępuje podprogram formalny;
(2) w przypadku podprogramów wirtualnych — wtedy podprogram zdefiniowany w module prefiksowanym zastępuje podprogram zdefiniowany w prefiksie.

Aby takie zastąpienie mogło mieć miejsce, nagłówki podprogramu zastępującego i zastępowanego muszą spełniać następujące warunki:

(1) procedura może być zastąpiona tylko przez procedurę, funkcja zaś przez funkcję;
(2) liczba parametrów podprogramu zastępującego i zastępowanego musi być taka sama, a ich odpowiedniość jest ustalana pozycyjnie;
(3) parametr będący typem formalnym może być zastąpiony jedynie przez typ formalny;
(4) procedura formalna może być zastąpiona tylko przez procedurę formalną, funkcja formalna tylko przez funkcję formalną; ten warunek dotyczący parametrow proceduralnych i funkcyjnych wystarcza w razie zastępowania podprogramów formalnych, ponieważ wtedy nagłówki ich podprogramów formalnych nie są specyfikowane; jeżeli są zastępowane podprogramy wirtualne, dodatkowo żąda się, by nagłówki odpowiednich ich parametrów proceduralnych i funkcyjnych umożliwiały zastąpienie;
(5) *typem statycznie określonym* nazwijmy taki typ, który nie jest ani typem formalnym, ani typem tablicowym o elementach typu formalnego; parametr będący zmienną typu statycznie określonego może być zastąpiony przez parametr zgodnego z nim typu statycznie określonego;
(6) w razie parametru będącego zmienną typu formalnego (tablicowego o elementach typu formalnego) rozróżniamy dwie sytuacje:

(a) gdy ten typ formalny jest wprowadzony w tym samym nagłówku (nazwijmy go *typem formalnym własnym nagłówka*);

(b) przypadek przeciwny (nazwijmy ten typ *typem formalnym zewnętrznym*):

— parametr będący zmienną typu formalnego własnego nagłówka (tablicowego o elementach typu formalnego własnego) może być zastąpiony tylko przez parametr, będący zmienną odpowiadającego mu pozycyjnie typu formalnego własnego (odpowiednio: tablicowego);

— parametr będący zmienną typu formalnego zewnętrznego może być zastąpiony przez parametr będący zmienną typu formalnego zewnętrznego lub typu obiektowego;

(7) W przypadku zastępowania funkcji wymagania dotyczące typu wyniku funkcji zastępującej i zastępowanej są dość skomplikowane i różne w przypadku zastępowania funkcji wirtualnych oraz funkcji formalnych:

(a) dla funkcji wirtualnych:

— jeżeli typem zastępowanym jest typ formalny własny nagłówka (patrz (6)) lub tablicowy nad formalnym własnym, to typem zastępującym musi być odpowiadający mu pozycyjnie typ formalny własny (tablicowy nad odpowiednim formalnym własnym);

— jeżeli typem zastępowanym jest typ prosty, formalny zewnętrzny lub tablicowy (nie będący typem tablicowym nad formalnym własnym), to typem zastępującym musi być dokładnie ten sam typ;

— jeżeli typem zastępowanym jest klasa, współprogram lub proces A, to typem zastępującym musi być klasa, współprogram lub proces B zawarty w typie A (tzn. A należy do ciągu prefiksowego B);

(b) dla funkcji formalnych:

— jeżeli typem formalnym jest typ statycznie określony, to wymagania dotyczące typu zastępującego są jak w przypadku zastępowania funkcji wirtualnych;

— jeżeli typem zastępowanym jest typ formalny lub tablicowy o elementach typu formalnego, to wymagania dotyczące typu zastępującego są takie same jak dla typów parametrów będących zmiennymi.

Dodatek D. Implementacja procesów współbieżnych na IBM PC

Dodatek ten opisuje system procesów rozproszonych dla języka Loglan. Został on zaprojektowany i zrealizowany przez Bolesława Ciesielskiego — absolwenta Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego w ramach jego pracy magisterskiej. W opisie zostały wykorzystane fragmenty wspomnianej pracy.

W języku programowania Loglan zawarte są mechanizmy umożliwiające tworzenie i synchronizację procesów współbieżnych (por. rozdz. 8). Istniejący w Loglanie mechanizm synchronizacji procesów, oparty na semaforach binarnych, nie nadaje się jednak do realizacji w systemie rozproszonym. Semafory są bowiem z definicji zmiennymi dzielonymi i jako takie mają sens tylko w systemach ze wspólną pamięcią. Przedstawiony poniżej mechanizm synchronizacji procesów rozproszonych został zrealizowany w istniejącym interpretatorze Loglanu 82 i sieci lokalnej komputerów IBM PC. Procesy mogą komunikować się i synchronizować przez tzw. *obce wołanie procedury*. Jest to zmodyfikowane odległe wołanie procedury w innym procesie (por. odległy dostęp — p.3.2.3). W obcym wołaniu procedury uczestniczy zarówno proces wołający, jak i wołany. Jest to podstawowa różnica w stosunku do odległego wołania procedury, gdzie jest ona wykonywana przez proces wołający bez żadnego zaangażowania ze strony procesu wołanego. Dzięki temu, że w obcym wołaniu procedury biorą udział oba procesy, można stworzyć synchroniczny mechanizm komunikacji procesów oparty na właśnie takim sposobie wołania procedur. Obce wołanie procedury jest inicjowane przez proces wołający, tzn. wykonujący instrukcję

```
call X.p(lpa)
```

gdzie X jest wskaźnikiem do procesu wołanego, p — jego procedurą, lpa zaś jest listą parametrów aktualnych procedury. Po przekazaniu parametrów wejściowych proces wołający zawiesza się w oczekiwaniu na zakończenie obcego wołania. Procedura jest wykonywana przez proces wołany. Aby mógł on rozpocząć wykonywanie procedury, muszą być

spełnione pewne warunki: proces ten nie może być w żaden sposób zawieszony (wyjątek stanowi instrukcja **accept** opisana poniżej), wołana procedura zaś musi być w nim odblokowana. Dla każdego obiektu procesu zdefiniowana jest *maska procedur* będąca podzbiorem wszystkich procedur zadeklarowanych w tym procesie. Procedura jest *odblokowana* w procesie, jeśli należy do jego maski procedur, w przeciwnym razie jest w nim *zablokowana*. W nowo utworzonym obiekcie procesu maska procedur jest pusta.

Gdy zostaną spełnione warunki umożliwiające rozpoczęcie wykonywania procedury, proces wołany zostaje przerwany i zaczyna wykonywać wołaną procedurę (z parametrami przekazanymi przez proces wołający). Przy wejściu do procedury wszystkie procedury w procesie wołanym zostają zablokowane. Po zakończeniu wykonywania procedury maska procedur procesu wołanego zostaje odtworzona do stanu sprzed wywołania i proces wołany jest wznowiony w punkcie przerwania. Proces wołający odczytuje parametry wyjściowe i wznawia działanie począwszy od instrukcji następującej po obcym wołaniu procedury. Ten podstawowy mechanizm jest uzupełniony o instrukcje pozwalające zmieniać maskę procedur. Instrukcje

enable $p_1, \ldots, p_n$;

oraz

disable $p_1, \ldots, p_n$;

powodują odpowiednio odblokowanie lub zablokowanie procedur o identyfikatorach $p_1, \ldots, p_n$ w bieżącym procesie. Instrukcja

return enable $p_1, \ldots, p_n$ **disable** $q_1, \ldots, q_m$;

pozwala zmienić maskę procedur procesu wołanego w treści procedury. W wyniku wykonania tej instrukcji następuje zakończenie wykonywania procedury, a maska procedur procesu wołanego jest modyfikowana, bezpośrednio po jej odtworzeniu do stanu sprzed wywołania, przez wykonanie instrukcji

enable $p_1, \ldots, p_n$; **disable** $q_1, \ldots, q_m$;

Dodatkowo przewidziana jest instrukcja

accept $p_1, \ldots, p_n$;

która dołącza do maski (odblokowuje) procedury $p_1, \ldots, p_n$ oraz zawiesza wykonujący ją proces w oczekiwaniu na obce wywołanie którejś z aktualnie odblokowanych procedur (być może różnej od $p_1, \ldots, p_n$). Po zakończeniu wykonywania procedury maska procedur jest odtwarzana do

stanu sprzed instrukcji **accept**, a wykonywanie procesu kontynuowane. Dzięki instrukcji **accept** można uniknąć aktywnego czekania na obce wołanie jednej spośród wskazanych procedur. Zauważmy też, że jeśli nie używamy instrukcji **enable** oraz **disable**, to proces wołany może obsłużyć obce wołanie jedynie podczas wykonywania instrukcji **accept**. W innych momentach wszystkie procedury są zablokowane. Zatem sama instrukcja **accept** definiuje w pełni synchroniczny mechanizm komunikacji procesów. Pozostałe konstrukcje zostały wprowadzone w celu umożliwienia asynchronicznych protokołów komunikacji.

W razie deklarowania procesów rozproszonych musimy zapewnić, aby pierwszym ich parametrem (uwzględniając także ciąg prefiksowy) była zmienna typu **integer**. Określa ona numer komputera w sieci, na którym ma być wykonywany dany proces. Zauważmy, że wiele procesów może być wykonywanych przez ten sam komputer. W szczególności, jeśli wszystkie procesy mają być wykonywane na jednym komputerze, parametr określający jego numer musi być równy zero.

Ponadto, ze względów implementacyjnych wprowadzono pewne ograniczenia. Na przykład, wszystkie procesy muszą być zadeklarowane bezpośrednio w programie głównym, a w konsekwencji nie ma możliwości zagnieżdżania procesów. Odległy dostęp do lokalnych atrybutów procesu poza obcym wołaniem procedury nie jest możliwy. Jedynymi zmiennymi programu głównego dostępnymi dla innych procesów są zmienne wskazujące procesy.

Literatura

Podstawą naszej książki jest raport języka Loglan 82 [3]. Loglan 82 należy do linii rozwojowej języków programowania Algol 60 [13], Pascal [6, 17] i Simula 67 [12]. Opis języków Ada i Modula, oferujących wiele rozwiązań alternatywnych dla konstrukcji programistycznych występujących w Loglanie, czytelnik znajdzie np. w książkach [15, 18]. Zastosowana konwencja notacyjna dotycząca prezentacji typów danych jest powszechnie przyjęta w literaturze. Omówienie podobnych konwencji znaleźć można np. w książkach [2, 11].

Pojęcie klas, prefiksowania i współprogramów wywodzi się z języka Simula 67 [12]. Wyczerpujące omówienie współprogramów można znaleźć w pracy [10]. Współprogramy są przydatne przede wszystkim do symulacji procesów dyskretnych. Loglan — w przeciwieństwie do Simuli 67 — nie dostarcza gotowego pojęcia procesów symulacyjnych i operacji na tych procesach. Jednak klasa SIMULATION rozwiązująca ten problem (por. [12]) może być zaprogramowana w Loglanie jako język problemowy. Jej wersja dla języka Loglan została podana w pracy [16].

Metody sortowania prezentowane w naszej książce, wraz z wieloma innymi, są omówione bardziej wyczerpująco np. w książkach [1, 17]. W pierwszej z nich można ponadto znaleźć omówienie typu danych Find-Union wraz z jego zastosowaniami. Problematyka ta jest prezentowana także w książce [2]. Również wiele przedstawionych przez nas typów danych jest omówionych w [1, 2, 17].

Czytelników zainteresowanych tematyką zbiorów rozmytych, algorytmów zachłannych, tablic rozproszonych i rozwiązywania dużych układów równań liniowych odsyłamy odpowiednio do książek [5], [8], [9] oraz [14].

Literatura związana z programowaniem współbieżnym jest bardzo bogata. Omówienie tej problematyki, wraz z dokładnym opisem narzędzi synchronizacyjnych — w tym monitorów, przedstawione jest np. w podręcznikach [4] oraz [7]. Problemy semantyki obliczeń współbieżnych inspirowane pracami nad Loglanem 82 są poruszane obszernie w książce [11].

Przedstawione źródła, obok szerokiego omówienia poruszanej przez nas tematyki, zawierają także bogatą bibliografię, do której odsyłamy Czytelników głębiej zainteresowanych poszczególnymi problemami.

1. AHO A.V., HOPCROFT J.E., ULLMAN J.D.: *Projektowanie i analiza algorytmów komputerowych*. Warszawa, PWN 1983.
2. BANACHOWSKI L., KRECZMAR A.: *Elementy analizy algorytmów*. Warszawa, WNT 1983.
3. BARTOL W.M. i inni: *Report on the Loglan 82 programming language*. Warszawa-Łódź, PWN 1984.

4. BRINCH HANSEN P.: *Podstawy systemów operacyjnych*. Warszawa, WNT 1979.
5. CZOGAŁA E., PEDRYCZ W.: *Elementy i metody zbiorów rozmytych*. Gliwice, Wyd. Politechniki Śląskiej 1983.
6. IGLEWSKI M., MADEJ J., MATWIN S.: *Pascal. Język wzorcowy. Pascal 6000*. Wyd. 3, Warszawa, WNT 1984.
7. ISZKOWSKI W., MANIECKI M.: *Programowanie współbieżne*. Warszawa, WNT 1982.
8. JAGIELSKI R.: *Tablice rozproszone*. Warszawa, WNT 1982.
9. LIPSKI W.: *Kombinatoryka dla programistów*. Wyd. 2. Warszawa, WNT 1989.
10. MARTIN C.D.: *Coroutines. LNCS 95*. Berlin, Heidelberg, Springer-Verlag 1980.
11. MIRKOWSKA G., SALWICKI A.: *Algorithmic Logic*. Warszawa, PWN 1987.
12. OKTABA H., RATAJCZAK W.: *Simula 67*. Warszawa, WNT 1980.
13. PASZKOWSKI S.: *Język Algol 60*. Wyd. 8. Warszawa, PWN 1977.
14. JANKOWSCY J. i M.: *Przegląd metod i algorytmów numerycznych. Cz. 1*, Wyd. 2. Warszawa, WNT 1988.
15. PYLE I.C.: *Ada*. Warszawa, WNT 1986.
16. RATAJCZAK W., SZCZEPAŃSKA-WASERSZTRUM D.: *Data structure for simulation purposes in Loglan 77*. Prace IPI PAN 373. Warszawa 1979.
17. WIRTH N.: *Algorytmy + struktury danych = programy*. Wyd. 2. Warszawa, WNT 1989.
18. WIRTH N.: *Modula 2*. Warszawa, WNT 1987.

Skorowidz

U

W

Z

Loglan

Summary

Loglan 82 is a general purpose object-oriented language. The execution of Loglan program consists in creating objects and executing their statements. Objects are instances of modules. There are several kinds of moduls: blocks, procedures, functions, coroutines, concurrent processes and exception handlers. Modules can be both nested and extended by means of prefixing operation. They can be parametrized with data, operations and types. Class is a basic notion in Loglan. It enables encapsulating data and operations on them into one module. Any module can by prefixed with a class. Coroutine and process are special cases of class. They provide specific control transfer operactions that enable multiplexed and concurrent execution. Class is not only a module, but also defines data type. This type contains all objects being an instance of the given class. Other object types are arrays and files. Their objects are created dynamically, as well. Exepction handling provides facilities for dealing with run-time errors and other exeptional situations.

The book presents syntax and semantics of Loglan together with selected, most important methods of programming in Loglan. Both the language and methology are introduced mainly by examples of solving problems. All examples are presented uniformly: first we define and discuss the problem, then we explain the chosen solution, in the end we present the ready solution in Loglan.

The book is intended for programmers, system designers, computer scientists and students.

Логлан

Резюме

Логлан — это универсальный, объектно-ориентированный язык. Выполнение программы на Логлане состоит в создании объектов и выполнении их команд. Объекты являются экземплярами модулей. Имеется несколько типов модулей: блоки, процедуры, функции, классы, сопрограммы, параллельные процессы и обработчики особых ситуации. Модули могут быть вложены один в другой и расширены с помощью префиксирования. Они могут быть параметризованы данными, операциями и типами.

Класс — это основное поняте в Логлане. Оно даёт бозможность обединить в одно целое данные и выполняемые на них операции. Любой модуль может быть префиксован классом. Сопрограмма и процесс это специальные случаи класса. Они доставлывают особые операции передачи управления, благодария которым бозможно мультиплексированное и параллельное выполнение программ.

Класс — это не только модуль, но тоже определение типа данных. К этому типу принадлежат все объекты являющиеся экземплярами данного класса. Другие объектные типы это блоки и фаилы. Их объекты тоже создаются динамически. Обработка особых ситуаций дает бозможность справиться с ошибками времени быполнения и другими особыми ситуациями.

Книга представляет синтаксис и семантику Логлана вместе с выбранными методами програмирования на нём.

Язык и методы прежде всего введены примерами решения задачь. Все примеры представляются в одном порядке: с начала мы задаём и дискутируем решения, а потом мы даем готобое решение проблемы на Логлане.

Книга предназначена для програмиетов, проектантов систем обработки информации и студентов занимающихся вычислительной техникой.

Biblioteka Inżynierii Oprogramowania

Wykaz wydanych książek

S. ALAGIĆ, M. A. ARBIB — Projektowanie programów poprawnych i dobrze zbudowanych
I. O. ANGELL — Wprowadzenie do grafiki komputerowej, wyd. 1 i 2
R. L. BABER — O oprogramowaniu inaczej
L. BANACHOWSKI, A. KRECZMAR — Elementy analizy algorytmów, wyd. 1 i 2
L. BANACHOWSKI, A. KRECZMAR, W. RYTTER — Analiza algorytmów i struktur danych, wyd. 1 i 2
M. BEN-ARI — Podstawy programowania współbieżnego
J. BIELECKI — System VSAM. Zasady stosowania w języku PL/I
J. BŁAŻEWICZ — Złożoność obliczeniowa problemów kombinatorycznych
L. BOLC, M. CICHY, L. RÓŻAŃSKA — Przetwarzanie języka naturalnego
S. BORAK, J. KLACZAK, S. KORCZAK, Z. PŁOSKI — System operacyjny George 3, wyd. 1 i 2 rozszerzone
J. M. BRADY — Informatyka teoretyczna w ujęciu programistycznym
K. L. CLARK, F. G. MCCABE — Micro-Prolog
M. DĄBROWSKI, K. LAUS-MĄCZYŃSKA — Metody wyszukiwania i klasyfikacji informacji
C. DELOBEL, M. ADIBA — Relacyjne bazy danych
P. DEMBIŃSKI, J. MAŁUSZYŃSKI — Matematyczne metody definiowania języków programowania
J. DEMINET — System operacyjny RSX-11
E. W. DIJKSTRA — Umiejętność programowania, wyd. 1 i 2
S. GASIK, P. KULCZYCKI, K. PIASECKI, J. WITASZEK — PL/(F)
P. GIZBERT-STUDNICKI, J. KARCZMARCZUK — Snobol4
M. GŁOWACKI — Systemy operacyjne DOS i OS
M. J. C. GORDON — Denotacyjny opis języków programowania
R. E. GRISWOLD, M. T. GRISWOLD — Icon
A. N. HABERMAN, D. E. PERRY — Ada dla zaawansowanych
L. J. HOFFMAN — Poufność w systemach informatycznych
M. IGLEWSKI, J. MADEY, S. MATWIN — Pascal. Język wzorcowy. Pascal 6000, wyd. 2
M. IGLEWSKI, J. MADEY, S. MATWIN — Pascal. Język wzorcowy. Pascal 360, wyd. 3 zmienione i 4
W. ISZKOWSKI, M. MANIECKI — Programowanie współbieżne
R. JAGIELSKI — Tablice rozproszone
A. P. JERSZOW — Wprowadzenie do teorii programowania
C. B. JONES — Konstruowanie oprogramowania metodą systematyczną

A. KASSUR, P. PERKOWSKI — Obliczeniowe aspekty projektowania układów elektronicznych
B. W. KERNIGHAN, P. J. PLAUGER — Narzędzia programistyczne w Pascalu
B. W. KERNIGHAN, D. M. RITCHIE — Język C, wyd. 1 i 2
F. KLUŹNIAK, S. SZPAKOWICZ — Prolog
H. KOPETZ — Niezawodność oprogramowania
R. KOWALSKI — Logika w rozwiązywaniu zadań
W. LIPSKI — Kombinatoryka dla programistów, wyd. 1 i 2
J. MARTINEK — Lisp. Opis, realizacja i zastosowania
G. J. MYERS — Projektowanie niezawodnego oprogramowania
L. NIEMCZYCKI — Oprogramowanie teleprzetwarzania maszyn Jednolitego Systemu
H. OKTABA, W. RATAJCZAK — Simula 67
J. OLSZEWSKI — Projektowanie struktur systemów operacyjnych
W. PACHELSKI — Fortran dla maszyn Odra serii 1300
W. PACHELSKI — Fortran IV dla maszyn Jednolitego Systemu, wyd. 1 i 2 poprawione i rozszerzone
T. PAVLIDIS — Grafika i przetwarzanie obrazów. Algorytmy
T. PERKOWSKI — Technika symulacji cyfrowej
Przegląd metod i algorytmów numerycznych, cz. 1 — J. i M. JANKOWSCY, wyd. 1 i 2 poprawione
Przegląd metod i algorytmów numerycznych, cz. 2 — M. DRYJA, J. i M. JANKOWSCY, wyd. 1 i 2 poprawione
I. C. PYLE — Ada
W. REISIG — Sieci Petriego. Wprowadzenie
P. P. SILVESTER — System operacyjny Unix
B. SZAFRAŃSKI, W. SKURZAK, W. SZYPUŁA — System operacyjny RT-11
A. SZAŁAS, J. WARPECHOWSKA — Loglan
D. VAN TASSEL — Praktyka programowania, wyd. 1 i 2 rozszerzone
D. C. TSICHRITZIS, F. H. LOCHOVSKY — Modele danych
W. M. TURSKI — Metodologia programowania, wyd. 1 i 2 rozszerzone
J. TYSZER — Symulacja cyfrowa
E. CH. TYUGU — Programowanie z bazą wiedzy
J. D. ULLMAN — Systemy baz danych
W. M. WAITE, G. GOOS — Konstrukcja kompilatorów
J. WAŁASEK — Konwersacyjne otoczenie programowe Pascala
N. WIRTH — Modula 2, wyd. 1 i 2 poprawione
N. WIRTH — Wstęp do programowania systematycznego, wyd. 1 i 2
R. WIT — Metody programowania nieliniowego. Minimalizacja funkcji gładkich
K. ZORYCHTA, W. OGRYCZAK — Programowanie liniowe i całkowitoliczbowe

WNT Warszawa 1991.
Wydanie I. Format B5.
Ark. wyd. 11,6. Ark. druk. 11,0.
Symbol Et/82366/MEN
Bielskie Zakłady Graficzne, zam. 1766/91