Biblioteka
Inżynierii Oprogramowania

Redaktor serii BARBARA OSUCHOWSKA

Komitet Redakcyjny
ANDRZEJ J. BLIKLE
STANISŁAW GANCARCZYK
LEON ŁUKASZEWICZ
JAN MADEY
ANTONI MAZURKIEWICZ
ANDRZEJ SAŁWICKI
*przewodniczący* WŁADYSŁAW M. TURSKI
STANISŁAW WALIGÓRSKI
JAN WĘGLARZ
JAN ZABRODZKI

Siedemdziesiąta ósma pozycja serii
ukazującej się od roku 1978

Wykaz dotychczas wydanych tomów zamieszczono na końcu książki

Andrzej Szałas
Jolanta Warpechowska

# Loglan

Wydawnictwa Naukowo-Techniczne
Warszawa

ANDRZEJ SZAŁAS
Instytut Informatyki UW

JOLANTA WARPECHOWSKA
Instytut Informatyki UW

© Copyright
by Wydawnictwa Naukowo-Techniczne
Warszawa 1991

All rights reserved
*Printed in Poland*

ISBN 83-204-1295-1

W-2448/1

Biblioteka
Politechniki Białostockiej
1108075

# Loglan

## Loglan

English summary, see p. 173

## Логлан

Русское резюме смт. стр. 174

Książka zawiera opis podstawowych pojęć i konstrukcji języka Loglan 82. Wprowadzone w książce pojęcia klasy, prefiksowania i współprogramów są znane z Simuli-67. Koncepcja prefiksowania została jednak w istotny sposób uogólniona. Możliwość obsługi sytuacji wyjątkowych oraz programowanie procesów współbieżnych czyni język nowoczesnym i interesującym.
Duża liczba przykładów umożliwia poznanie nie tylko składni i semantyki, lecz także podstawowych metod programowania oraz zastosowań (np. symulacji) charakterystycznych dla Loglanu.
Książka jest przeznaczona dla programistów, pracowników nauki zajmujących się informatyką oraz studentów kierunków informatycznych.

681.3.06

Redaktor
ANNA KOPYT

Okładkę i układ typograficzny serii projektował
TADEUSZ PIETRZYK

Skład komputerowy
SPRINT sp. z o.o.

Tytuł dotowany przez
Ministra Edukacji Narodowej

# Spis treści


Przedmowa 9

Wstęp 12

1 Podstawowe konstrukcje języka 14

1.1 Struktura leksykalna 14
1.2 Typy pierwotne 15
1.3 Najprostsze deklaracje 19
1.4 Wprowadzenie do modułów i obiektów 20
1.5 Podstawowe instrukcje 22
1.6 Bloki 26
1.7 Podprogramy 26
1.8 Pliki 28

2 Tablice dynamiczne 33

2.1 Wprowadzenie 33
2.2 Iloczyn skalarny wektorów — tablice jednowymiarowe . . 34
2.3 Rozwiązywanie układu równań liniowych z macierzą trójkątną — wielowymiarowe tablice różnych kształtów . . . 36
2.4 Podsumowanie 39

3 Klasa jako typ danych 40

3.1 Wprowadzenie 40
3.2 Liczby zespolone — złożony typ danych 41
3.3 Binarne drzewa wyważone — rekurencyjne definiowanie typów danych 47
3.4 Podsumowanie 51

4 Parametryzacja modułów 52

4.1 Wprowadzenie 52
4.2 Szukanie miejsca zerowego funkcji — podprogram jako parametr modułu 53

4.3 Uniwersalna procedura sortująca — typ jako parametr modułu . . . . . 55
4.4 Kolejki — struktura danych sparametryzowana typem formalnym . . . . . 58
4.5 Podsumowanie . . . . . 60

5 Prefiksowanie — operacja rozszerzania modułów 62
5.1 Wprowadzenie . . . . . 62
5.2 Znajdowanie okręgu opisanego na trójkącie — język problemowy . . . . . 65
5.3 Sortowanie stogowe — łączenie instrukcji w modułach prefiksowanych . . . . . 70
5.4 Dalsze wiadomości o prefiksowaniu . . . . . 74
5.5 Sumowanie typów — operatory in, is, qua, this . . . . . 80
5.6 Zbiory rozmyte — użycie podprogramów wirtualnych . . 83
5.7 Tablice rozproszone — ochrona atrybutów przy prefiksowaniu . . . . . 88
5.8 Ogólny algorytm zachłanny i algorytm Kruskala — zastosowanie prefiksowania do budowy algorytmów abstrakcyjnych i ich konkretyzacji . . . . . 94
5.9 Podsumowanie . . . . . 101

6 Współprogramy 103
6.1 Wprowadzenie . . . . . 103
6.2 Symulacja licznika — podstawowe własności współprogramów . . . . . 106
6.3 Generowanie permutacji — zawieszanie działania algorytmów . . . . . 110
6.4 Scalanie ciągów liczb — korzystanie z częściowych wyników działania algorytmu . . . . . 112
6.5 Wieże z Hanoi — współprogramy a procedury rekurencyjne . . . . . 116
6.6 Podsumowanie . . . . . 119

7 Obsługa sytuacji wyjątkowych 121
7.1 Wprowadzenie . . . . . 121
7.2 Rozwiązywanie układu równań liniowych z macierzą trójkątną — błędy wykonania jako sytuacje wyjątkowe . . . 123
7.3 Problem plecakowy — epilog obiektu przy zakończeniu przez terminate lub wind . . . . . 128
7.4 Bezpieczne dzielenie funkcji rzeczywistych — parametryzacja sygnałów . . . . . 131
7.5 Podsumowanie . . . . . 133

8 Procesy wspólbieżne 135
8.1 Wprowadzenie . . . . . 135

8.2 Sortowanie szybkie — podstawowe własności procesów . . 137
8.3 Wzajemne wykluczanie operacji typów danych — wykorzystanie semaforów 141
8.4 Monitor — strukturalne mechanizmy współpracy między procesami 143
8.5 Rozwiązywanie dużych układów równań liniowych — zastosowanie współbieżności i prefiksowania 146
8.6 Podsumowanie 150

Dodatek A. Składnia Loglanu 82 152

Dodatek B. Procedury i funkcje standardowe (dla implementacji na IBM PC) 160

Dodatek C. Zgodność typów. Reguły zastępowania podprogramów formalnych i wirtualnych 162
C.1 Zgodność typów 162
C.2 Reguły zastępowania podprogramów formalnych i wirtualnych 163

Dodatek D. Implementacja procesów współbieżnych na IBM PC 165

Literatura 168

Skorowidz 170

# Przedmowa

Naszym celem jest przedstawienie najważniejszych mechanizmów języka Loglan. Ograniczona objętość książki nie pozwala na wprowadzenie czytelnika w zasady programowania od podstaw, zakładamy znajomość technik programowania w jednym z istniejących języków o strukturze blokowej. Znajomość Pascala lub Simuli-67, jakkolwiek niekonieczna, wydaje nam się doskonałą podstawą do pełnego zrozumienia koncepcji programistycznych opisywanych w tej książce. Toteż adresujemy ją przede wszystkim do szerokiego grona programistów-praktyków oraz do studentów kierunków informatycznych. Staraliśmy się, aby książka ta była przydatna zarówno dla tych, którzy chcą nauczyć się programowania w Loglanie, jak i tych, którzy zainteresowani są współczesnymi technikami i narzędziami programowania.

Bogactwo narzędzi programistycznych Loglanu, niejednokrotnie po raz pierwszy występujących razem w jednym języku, skłoniło nas do podjęcia próby przedstawienia nie tylko ich składni i semantyki, ale także naszkicowania związanych z nimi metod i technik programowania. Uważamy przy tym, że zarówno język programowania, jak i charakterystyczne dlań metody programowania najlepiej poznawać za pomocą przykładów. Dlatego też kolejne rozdziały, oprócz rozdz. 1 zawierającego powszechnie znane konstrukcje, są poświęcone wybranej konstrukcji lub technice programowania. Wyczerpujemy przy tym wszystkie konstrukcje języka i zarysowujemy najważniejsze techniki programowania. Są one wprowadzane w trakcie analizy przykładów, które dobieraliśmy tak, by za ich pomocą prezentacja była możliwie przystępna i wszechstronna. Staraliśmy się, aby były one proste, a przy tym — poza wstępnymi — niebanalne.

Książka ma strukturę warstwową. Rozpoczynamy ją od opisu podstawowych konstrukcji języka, które umożliwiają programowanie w przybliżeniu na poziomie Pascala. Wprowadzone pojęcia klas, prefiksowania i współprogramów dostarczają narzędzi znanych z Simuli-67, ale z bogatszą koncepcją prefiksowania. Następne rozdziały są poświęcone obsłudze sytuacji wyjątkowych oraz procesom współbieżnym. Ta, ostatnia już war-

stwa języka odpowiada poziomowi języka Ada. Całość książki kończą dodatki i uwagi bibliograficzne. Uważamy wprawdzie, że podręcznik języka programowania powinien zawierać wszystkie niezbędne informacje o tym języku, zamieszczamy jednak odnośniki do literatury, aby ułatwić dalsze poznawanie poruszanych zagadnień. Ograniczamy się przy tym przede wszystkim do powszechnie dostępnej literatury, wydanej w języku polskim.

Jak już wcześniej wspomnieliśmy, Loglan dostarcza bogatych mechanizmów definiowania typów danych. Do opisu tych typów zaadaptowaliśmy konwencję przyjętą powszechnie w tej dziedzinie. Typ danych nie jest określony jedynie przez zbiór wartości, jakie mogą przyjmować zmienne tego typu. Na przykład inne własności ma zbiór liczb naturalnych rozważany tylko z działaniem dodawania, a inne ten sam zbiór z działaniem mnożenia. Podobnie, para liczb rzeczywistych może określać liczbę zespoloną (jeśli odpowiednio określimy operacje dodawania i mnożenia), natężenie i napięcie prądu w przewodniku (jeśli określimy np. operację obliczania oporu przewodnika) itd. Tak więc dane różnych typów są rozróżnialne nie tylko przez ich wartości, lecz także przez operacje związane z typami (por. też p. 3.1). Toteż typ danych będziemy przedstawiać jako zbiór wartości wraz z określonymi na nim operacjami: funkcjami i relacjami. Będziemy przy tym podawać opis składni i semantyki funkcji i relacji. Częstą sytuacją w programowaniu jest występowanie pewnych typów danych jako „parametrów formalnych" innych typów. Na przykład definiując typ danych reprezentujący stosy powinniśmy definiować je w sposób ogólny, niezależny od typu elementów umieszczanych na stosach. W tym przypadku typ elementów będzie parametrem typu **stosy**. Podsumowując, typ danych będziemy opisywać jako skończony układ

$$T(E) = (D;\ f_1, f_2, \ldots, r_1, r_2, \ldots)$$

gdzie T jest nazwą typu, E, jeśli występuje, jest typem parametrycznym, $D$ — zbiorem wartości zwanym też *nośnikiem* typu, $f_1, f_2, \ldots$ — funkcjami oraz $r_1, r_2, \ldots$ — relacjami określonymi na wartościach ze zbioru $D$ (oraz $E$). Aby zwiększyć czytelność książki, typowi (T) oraz zbiorowi wartości (D) będziemy zwykle nadawać tę samą nazwę. Na przykład wspomniany już wcześniej typ danych **stosy** można opisać jako

$$\text{stosy}(E) = (\textit{stosy};\ \textbf{włóż}, \textbf{zdejmij}, \textbf{wierzchołek}, \textbf{pusty})$$

gdzie E jest typem elementów umieszczanych na stosach, nośnik typu jest zbiorem skończonych ciągów elementów należących do $E$, **włóż**, **zdejmij**, **wierzchołek** są funkcjami, zaś **pusty** — relacją. Liczbę parametrów funkcji i relacji specyfikujemy zgodnie z powszechnie przyjętą konwencją

włóż: *stosy* × *E* → *stosy*;
zdejmij: *stosy* → *stosy*;
wierzchołek: *stosy* → *E*;
pusty: *stosy* → *boolean*.

Powyższa specyfikacja oznacza, że

— włóż jest funkcją dwuargumentową o pierwszym argumencie typu stosy, drugim typu E i wyniku typu stosy;

— zdejmij i wierzchołek są funkcjami jednoargumentowymi, ich argumentem jest stos, wynikiem zaś stos (w przypadku zdejmij) lub element typu E (w przypadku wierzchołek);

— pusty jest jednoargumentową relacją o argumencie typu stosy. Taka relacja jest utożsamiana z funkcją o argumencie typu stosy i wyniku logicznym (typu boolean).

Mamy nadzieję, że ta konwencja zwiększy czytelność książki i uspójni prezentację różnorodnych przykładów.

Kończąc, chcielibyśmy złożyć podziękowania prof. A. Salwickiemu za inicjatywę i zachętę do napisania tej książki. Jesteśmy także wdzięczni drowi J. Nawrockiemu, którego szczegółowe uwagi pozwoliły nam usunąć różne niedociągnięcia i poprawić prezentację znacznych fragmentów książki. Nasza praca nad książką była wspierana przez program badań RP.I.09 Ministerstwa Edukacji Narodowej.

AUTORZY

Warszawa, w październiku 1989

# Wstęp

Język programowania Loglan-82 zaprojektowano i zrealizowano w Instytucie Informatyki Uniwersytetu Warszawskiego. Jest on uniwersalnym językiem programowania, którego podstawową cechą jest bogata struktura modularna. Głównymi pojęciami języka są klasy i prefiksowanie modułów. Pojęcia te, wprowadzone w języku Simula-67, wywarły znaczny wpływ na metodologię programowania i współczesne rozumienie pojęcia modułów. Wpływ ten zaznacza się wyraźnie w wielu tak popularnych obecnie językach programowania jak np. Modula, Smalltalk lub — do pewnego stopnia — Ada. Prefiksowanie modułów w Loglanie jest mechanizmem ogólniejszym niż w Simuli-67. Umożliwia ono osiąganie wielu różnorodnych celów, których praktyczne znaczenie jest ogromne, m. in.

— definiowanie typów danych i ich hierarchiczną organizację;
— rozwijanie oprogramowania metodą wstępującą i zstępującą z wykorzystaniem przez wiele modułów w wielu kontekstach wspólnych elementów — prefiksów;
— tworzenie języków problemowych jako osobnych pakietów programistycznych i proste, jednolite ich wykorzystanie.

Loglan jest językiem w pełni dynamicznym. Oznacza to, iż w czasie wykonania programu można tworzyć nowe egzemplarze modułów, zwane też obiektami. Zestaw narzędzi programistycznych występujących w Loglanie umiejscawia ten język wśród języków obiektowych. Zestaw ten obejmuje

— wygodne instrukcje strukturalne z nowoczesną składnią (wzorowaną na składni Algolu i Pascala);
— modularną strukturę z możliwościami rozszerzania i zagnieżdżania modułów;
— rekurencyjne procedury i funkcje;
— bogate możliwości parametryzacji modułów;
— tablice dynamiczne;

— klasy, współprogramy i procesy współbieżne;
— prefiksowanie modułów;
— mechanizmy ochrony atrybutów, zabezpieczające je przed niepowołanym użyciem;
— w pełni bezpieczne usuwanie zbędnych obiektów;
— obsługę sytuacji wyjątkowych.

Te nowoczesne narzędzia programowania, zdefiniowane w ramach jednego języka, tworzą bogate otoczenie, w którym technika prefiksowania jest bardzo skuteczna i umożliwia konstruowanie wielu warstw oprogramowania użytkowego w jednym języku programowania.

Prace nad językiem Loglan-82 rozpoczęto w drugiej połowie lat siedemdziesiątych pod wpływem badań konstrukcji programistycznych i rozwiązań przyjętych w Simuli-67. W roku 1977 powstał wstępny raport Loglanu. Od tego czasu, w wyniku prac implementacyjnych i doświadczeń programistycznych, język ulegał wielokrotnym zmianom. Duży wpływ na jego rozwój miały także aktualne, lecz dobrze już ugruntowane trendy występujące w badaniach nad językami programowania. W roku 1982, w Instytucie Informatyki Uniwersytetu Warszawskiego, uruchomiono kompilator języka na minikomputerze Mera-400, napisany w Fortranie. W następnym roku powstała znacznie oszczędniejsza wersja tego kompilatora napisana w języku wewnętrznym Mery. Do końca roku 1986 powstały interpretatory na maszynę Siemens (zgodną z IBM-370), Vax-780/VMS oraz IBM PC. Od roku 1985 trwają prace nad nową, udoskonaloną wersją języka (Loglan-84) i projektem środowiska programistycznego, którego język byłby składową.

# Podstawowe konstrukcje języka 1

## Struktura leksykalna 1.1

Zbiór *znaków podstawowych* składa się z liter od a do z, cyfr dziesiętnych od 0 do 9, *znaków specjalnych* ., :, ,, ;, _, =, /, +, *, -, <, >, (, ), ', " oraz znaku odstępu.

*Jednostkami leksykalnymi* języka są liczby, napisy, identyfikatory i ograniczniki. Postać liczb i napisów omówimy przy okazji prezentowania typów pierwotnych języka (por. p. 1.2.2, 1.2.3, 1.2.7).

*Identyfikatorami* nazywamy ciągi składające się z liter, cyfr i podkreśleń, zaczynające się od litery, np. x12, A3_B5. Pewne identyfikatory, zwane *słowami kluczowymi*, mają w języku ustalone znaczenie i nie mogą być używane przez programistę w sposób inny niż przewidziany przez twórców języka. W Loglanie słowami kluczowymi są następujące identyfikatory: and, and_if, array_of, attach, begin, block, call, case, class, close, const, copy, coroutine, detach, dim, div, do, downto, else, end, esac, exit, fi, for,function, get, hidden, handlers, if, in, inner, input, inout, is, kill, last_will, lock, main, mod, new, none, not, od, open, or, or_if, others, otherwise, output, pref, procedure, process, put, qua, raise, read, readln, repeat, return, signal, step, stop, taken, terminate, then, this, to, type, unit, unlock, var, virtual, wait, wind, when, while, write, writeln.

Jako *ograniczniki* w Loglanie są stosowane znaki ., :, ,, ;, =, /, +, *, -, <, >, (, ), symbole złożone =/=, <=, >=, := oraz znaki odstępu.

*Komentarze* w programach zaczynają się symbolem (*, a kończą *), np.:

```
(* to jest poprawny komentarz *)
```

# Typy pierwotne 1.2

## *Typ* boolean 1.2.1

Typ **boolean** reprezentuje wartości logiczne wraz ze spójnikami logicznymi oraz operacjami porównywania wartości. Definiujemy go następująco:

**boolean** = (*boolean*; **not**, **and**, **or**, **and_if**, **or_if**, **=**, **=/=**, **<**, **<=**, **>**, **>=**)

gdzie

(1) nośnikiem typu jest zbiór {**true**, **false**};
(2) **not**: *boolean* → *boolean*;
(3) **and**, **or**, **and_if**, **or_if**, **=**, **=/=**, **<**, **<=**, **>**, **>=**:
*boolean* × *boolean* → *boolean*.

Operatory **=** i **=/=** badają równość i różność podanych argumentów. Zatem **=** odpowiada logicznej równoważności, a **=/=** jej negacji. Zakładamy, że zbiór {**true**, **false**} jest uporządkowany, przy czym **true>false** (zatem **false<true**, **false<=true**, **true>=false**). Spójniki **not**, **and**, **or**, (**and_if**, **or_if**) odpowiadają negacji, koniunkcji i alternatywie. Zauważmy, że w Loglanie występują dwa spójniki określające koniunkcję (**and**, **and_if**) oraz dwa spójniki określające alternatywę (**or**, **or_if**). Różnią się one zarówno zakresem stosowania, jak i interpretacją obliczeniową. Spójniki **and** oraz **or** mogą występować we wszystkich wyrażeniach logicznych, podczas gdy **and_if** oraz **or_if** jedynie w warunkach instrukcji warunkowych. W celu obliczenia wartości wyrażenia logicznego złożonego ze spójników **or** i **and** oblicza się najpierw wartości wszystkich ich argumentów, a następnie, na ich podstawie, wartość całego wyrażenia. W przypadku wyrażenia **b and_if c**, jeżeli **b=true**, to wartość wyrażenia jest równa wartości **c**. Jeśli zaś **b=false**, to wartością całego wyrażenia jest **false**. Zatem różnica między **b and c** oraz **b and_if c** polega na tym, że spójnik **and_if** oblicza wartość **c** tylko pod warunkiem, że **b=true**. Symetryczna sytuacja występuje w przypadku spójnika **or_if**. W wyrażeniu **b or_if c** wartość **c** jest obliczana tylko wtedy, gdy **b=false**. Dla zilustrowania różnic rozważmy dwie instrukcje:

(1) **if x=/=0 and (y/x)=1 then x:=1 fi**;
(2) **if x=/=0 and_if (y/x)=1 then x:=1 fi**.

Wykonanie instrukcji (1) dla **x=0** spowoduje błąd dzielenia przez zero ze względu na konieczność obliczania wartości wyrażenia **y/x**. W instrukcji (2) błąd ten nie wystąpi, gdyż dla **x=0** nie dojdzie do obliczenia **y/x**.

Wartości logiczne są reprezentowane w programie przez stałe **true**, **false** oraz zmienne typu **boolean**. Zmienne te są inicjowane standardowo na **false**.

## *Typ* integer 1.2.2

Typ **integer** reprezentuje liczby całkowite wraz z operacjami na nich określonymi.

integer = (*integer*; -, +, *, div, mod, =, =/=, <, <=, >, >=)

gdzie

(1) nośnikiem typu jest zbiór liczb całkowitych (w każdej implementacji jest to oczywiście podzbiór zbioru liczb całkowitych, najczęściej zależny od arytmetyki konkretnej maszyny);
(2) -: *integer* ∪ *integer* × *integer* → *integer*;
(3) +, *: *integer* × *integer* → *integer*;
(4) div, mod: *integer* × (*integer* − {0}) → *integer*;
(5) =, =/=, <, <=, >, >=: *integer* × *integer* → *boolean*.

Zauważmy, że operacja minus (-) może być jedno lub dwuargumentowa. W pierwszym przypadku określa ona zmianę znaku, a w drugim odejmowanie. Funkcje +, *, div i mod oznaczają standardowe operacje dodawania, mnożenia, dzielenia całkowitego i reszty z dzielenia. Relacje (5) służą do porównywania liczb całkowitych.

Wartości typu **integer** są reprezentowane przez skończone ciągi cyfr dziesiętnych bądź przez stałe i zmienne typu **integer**. Zmienne te są inicjowane standardowo na 0.

## *Typ* real 1.2.3

Typ **real** reprezentuje liczby rzeczywiste wraz z określonymi na nich następującymi operacjami:

real = (*real*; -, +, *, /, =, =/=, <, <=, >, >=)

gdzie

(1) nośnikiem typu jest zbiór liczb rzeczywistych (w każdej implementacji jest to podzbiór zbioru liczb rzeczywistych, najczęściej zależny od arytmetyki konkretnej maszyny);
(2) -: *real* ∪ *real* × *real* → *real*;
(3) +, *: *real* × *real* → *real*;
(4) /: *real* × (*real* − {0}) → *real*;
(5) =, =/=, <, <=, >, >=: *real* × *real* → *boolean*.

Znaczenie funkcji i relacji (2)–(5) jest standardowe.

Wartości typu real są reprezentowane w programie przez stałe i zmienne typu real oraz wyrażenia postaci x.yEz, gdzie x, y, z są liczbami całkowitymi (przy czym y >= 0). Wartością wyrażenia x.yEz jest $x.y*10^z$. Jeśli z=0, to wyrażenie Ez można pominąć. Poprawnymi liczbami rzeczywistymi są np.:

```
888.01
3.51E1  (* = 35.1 *)
-100.5E-2  (* = -1.005 *).
```

Zmienne typu real są inicjowane standardowo na 0.0.

## *Zgodność typów* integer *i* real 1.2.4

Wartości typu integer i real mogą występować wspólnie w wyrażeniach, ponieważ zbiór liczb całkowitych jest podzbiorem zbioru liczb rzeczywistych. Przyjmuje się przy tym następujące zasady określania typu wyrażeń:

— dla każdej operacji op zdefiniowanej na wartościach obu typów wyrażenie x op y jest typu integer, jeśli oba argumenty są typu integer. W przeciwnym przypadku typem tego wyrażenia jest real;

— jeżeli wartość typu real występuje jako argument operacji właściwej dla typu integer (div, mod), to jest ona przed wykonaniem operacji przekształcana przez obcięcie części ułamkowej;

— jeżeli wartość typu integer występuje jako argument operacji zdefiniowanej dla wartości rzeczywistych (/), to jest ona przed wykonaniem operacji przekształcana w odpowiadającą jej wartość typu real (tj. dołączana jest część ułamkowa równa zero).

Na przykład:

```
28 div 3.4 = 28 div 3 = 9,
30.0/15 = 30.0/15.0 = 2.0.
```

## *Priorytety operacji arytmetycznych i logicznych* 1.2.5

Wartości wyrażeń arytmetycznych i logicznych są obliczane kolejno od lewej strony do prawej z zachowaniem następujących priorytetów:

(1) wyrażenia w nawiasach;
(2) *, /, div, mod;
(3) +, -;

(4) =, =/=, <, <=, >, >=;
(5) not;
(6) and;
(7) or.

Na przykład poprawne jest wyrażenie

```
n>=0 and n<100 or not n=0 and k=0
```

i jest ono równoważne wyrażeniu

```
((n>=0) and (n<100)) or ((not (n=0)) and (k=0)).
```

### *Typ* character 1.2.6

Typ character reprezentuje znaki dostępne na danym komputerze. Jest więc zależny od konkretnej implementacji. Przyjmujemy, że

**character** = (*character*; ord, chr, =, =/=, <, <=, >, >=)

gdzie

(1) nośnikiem typu jest zbiór znaków dostępnych na danym komputerze;
(2) ord: *character* → *integer*;
(3) chr: *integer* → *character*;
(4) =, =/=, <, <=, >, >=: *character* × *character* → *boolean*.

Wartością funkcji ord(c) jest liczba całkowita określająca kod znaku c. Funkcja chr jest odwrotna do ord. Wynikiem chr(i) jest znak o kodzie i. Jeżeli taki znak nie istnieje, to obliczenie chr(i) spowoduje błąd. Operacje porównywania znaków (4) są wyznaczone przez odpowiednie relacje typu integer zastosowane do kodów znaków.

Wartości typu character są reprezentowane w programie przez stałe i zmienne typu character bądź w postaci symboli znaków ujętych w apostrofy, np.: 'A', '7', '+'. Początkowa wartość zmiennych typu character jest zależna od konkretnej implementacji języka.

### *Typ* string 1.2.7

Typ string obejmuje zbiór napisów. Służy on do wygodnego reprezentowania napisów występujących w programie wielokrotnie. Definiujemy go następująco:

**string** = (*string*; =, =/=)

gdzie

(1) nośnikiem typu jest zbiór skończonych ciągów znaków;
(2) =, =/=: *string* × *string* → *boolean*.

Operacje (2) oznaczają równość bądź nierówność napisów, przy czym każdy wystepujący w programie napis jest traktowany jako inny od pozostałych. Na przykład

```
const napis = "to jest poprawny napis";
var n1, n2: string;
...
n1:=napis; n2:=napis;  (* n1=n2 *)
n1:=napis; n2:="to jest poprawny napis";  (* n1=/=n2 *).
```

Jak widać w powyższym przykładzie, napisy są reprezentowane w programie przez stałe i zmienne typu string bądź bezpośrednio w postaci ciągów znaków ujętych w cudzysłów. Początkowa wartość zmiennych napisowych jest zależna od konkretnej implementacji języka.

## Najprostsze deklaracje 1.3

### *Wprowadzenie* 1.3.1

Wszystkie wielkości występujące w programie, poza standardowymi, trzeba zadeklarować. Każdy moduł ma wydzieloną część deklaracyjną. Kolejność deklaracji nie jest istotna. Wyjątkiem od tej zasady są jedynie deklaracje stałych (por. p. 1.3.2).

W programie mogą być deklarowane:

(1) stałe;
(2) zmienne;
(3) podprogramy (procedury i funkcje);
(4) klasy;
(5) współprogramy;
(6) procesy współbieżne;
(7) sygnały;
(8) moduły obsługi sygnałów.

W tym punkcie omówimy tylko deklaracje stałych i zmiennych.

### *Deklaracje stałych* 1.3.2

*Stałe* są deklarowane za pomocą słowa kluczowego const. Na przykład

```
const pi=3.14, dwa_pi=2*pi;
```

jest równoważne

```
const pi=3.14; const dwa_pi=2*pi.
```

Zauważmy, że pojedyncze deklaracje w ramach jednej grupy deklaracji zaczynającej się słowem const są oddzielone przecinkami, natomiast różne grupy deklaracji muszą być oddzielone średnikiem.

Deklaracja stałej wiąże wartość stojącą po prawej stronie znaku równości z identyfikatorem stojącym po jego lewej stronie. W deklaracjach stałych mogą występować identyfikatory innych stałych pod warunkiem, że zostały zdefiniowane wcześniej w tekście programu. Jest to jedyna sytuacja, w której kolejność deklaracji jest istotna. Na przykład deklaracja

```
const dwa_pi=2*pi, pi=3.14;
```

nie jest poprawna.

### *Deklaracje zmiennych* 1.3.3

*Zmienne* są deklarowane za pomocą słowa kluczowego var, np.

```
var x: integer, y: real, c: character;
var s: string, b: boolean;
```

Deklaracje zmiennych wiążą z każdą zmienną jej typ. Oznacza to, że będą one reprezentować w programie tylko te wielkości, które mają typ zgodny z wyspecyfikowanym.

## Wprowadzenie do modułów i obiektów 1.4

Loglan jest językiem obiektowym. Oznacza to, że siła tego języka wynika przede wszystkim z bogatych możliwości operowania na obiektach tworzonych dynamicznie podczas wykonywania programu. Pojęcie *obiektu* jest uogólnieniem pojęć znanych z innych języków programowania: jednostki dynamicznej procedury oraz instancji bloku. Działanie programu polega na tworzeniu obiektów i wykonywaniu ich list instrukcji. Obiekty są tworzone według tekstowych wzorców, jakimi są moduły. *Moduł* definiuje lokalne dane i operacje obiektu, a także jego listę instrukcji. Moduł składa

się z nagłówka, części deklaracyjnej oraz listy instrukcji. Postać nagłówka jest różna dla różnych rodzajów nagłówków. Obiekty utworzone według wzorca danego modułu nazywamy jego egzemplarzami. W trakcie wykonywania programu może jednocześnie istnieć wiele egzemplarzy jednego modułu. Loglan oferuje następujące rodzaje modułów: bloki, procedury, funkcje, moduły obsługi sygnałów, klasy, współprogramy oraz procesy współbieżne. Klasy, współprogramy i procesy współbieżne poza tym, że są modułami, należą ponadto do *typów obiektowych*. Oprócz nich do typów obiektowych zalicza się tablice dynamiczne oraz pliki. Dokładniej omówimy każdy z tych typów i modułów w dalszej części książki. Tu zwróćmy uwagę przede wszystkim na najważniejsze cechy wspólne typów obiektowych.

Zbiorem wartości danego typu obiektowego jest pewien zbiór obiektów. W przypadku klas, współprogramów oraz procesów współbieżnych jest to zbiór egzemplarzy danego modułu. W przypadku tablic i plików jest to zbiór możliwych obiektów danej tablicy lub pliku. Zauważmy, że zawsze zbiór wartości typu obiektowego jest potencjalnie nieskończony, obiekty są tworzone dynamicznie. Deklaracja typu obiektowego opisuje zatem całą klasę obiektów. Z podobną sytuacją mamy niejednokrotnie do czynienia w codziennym życiu. Na przykład druk przekazu bankowego jest pewnego rodzaju wzorcem — specyfikuje dane, które należy zamieścić, aby przekaz mógł być zrealizowany. Definiuje więc typ obiektów — przekazów bankowych. Wypełnienie takiego druku powoduje powstanie nowego obiektu będącego konkretnym egzemplarzem przekazu bankowego. Zauważmy przy tym, że identyczność danych zawartych w obiektach o tej samej strukturze nie musi oznaczać identyczności tych obiektów — dwa przekazy bankowe wypełnione tymi samymi danymi są nadal dwoma różnymi przekazami.

Wartości typów obiektowych mogą być przypisywane zmiennym typów obiektowych. Zmienne te nazywamy *wskaźnikowymi*, gdyż ich wartościami są wskaźniki do obiektów.

Z rozróżnienia między deklaracjami i obiektami danego typu wynika konieczność występowania w języku możliwości deklarowania typów oraz mechanizmów tworzenia obiektów. W Loglanie każda deklaracja typu obiektowego obok atrybutów definiowanych przez użytkownika dostarcza standardowych operacji tworzenia nowego obiektu (**new**), jego kopiowania (**copy**) oraz usuwania z pamięci (**kill**). Istnieje także wyróżniony *obiekt pusty* **none** wspólny dla wszystkich typów obiektowych. Inicjalną wartością zmiennych każdego typu obiektowego jest **none**.

Wykonanie instrukcji **new T** powoduje powstanie nowego obiektu typu **T** (w przypadku plików i tablic zamiast **new** używa się instrukcji **open** oraz **array dim** — por. p. 1.8.1 oraz 2.1). Jest także możliwe zapamiętanie wskaźnika do nowo utworzonego obiektu. Służy temu inna

forma instrukcji new:

```
x:=new T
```

w wyniku której zmienna x wskazuje na nowo utworzony obiekt typu T.

Wykonanie instrukcji x:=copy(y) powoduje utworzenie nowego obiektu będącego kopią obiektu wskazywanego przez y. Wskaźnik do nowego obiektu jest przypisany zmiennej x.

Wyrażenia typu obiektowego można porównywać za pomocą relacji = oraz =/=, przy czym relacja = oznacza identyczność wskaźników, a nie zawartości obiektów. Na przykład

```
u:=new T; x:=new T; (* u=/=x *)
y:=x (* y=x *); z:=copy(x) (* z=/=x *)
```

Operacja kill(y) usuwa z pamięci obiekt wskazywany przez y. Po wykonaniu tej operacji wartością y będzie none. Zauważmy przy tym, że kill(y) różni się od podstawienia y:=none, to ostatnie bowiem nie powoduje usunięcia obiektu wskazywanego przez zmienną y, a jedynie zmienia wartość tej zmiennej. Warto podkreślić, że usunięcie obiektu za pomocą kill jest bezpieczne, bowiem wszystkie zmienne wskazujące na usuwany obiekt przyjmują wartość none.

## Podstawowe instrukcje 1.5

### *Instrukcja przypisania* 1.5.1

*Instrukcja przypisania* przyjmuje w Loglanie postać

```
x1,...,xn:=t
```

gdzie typ wyrażenia t jest zgodny z typami zmiennych x1,...,xn.
Wykonanie tej instrukcji polega na obliczeniu wartości t, a następnie przypisaniu tej wartości zmiennym x1,...,xn; np.

```
(* i=2 *) i,A(i):=i+3; (* i=5,A(2)=5 *)
(* i=2 *) i:=i+3; A(i):=i+3 (* i=5,A(5)=5 *).
```

### *Instrukcja wywołania procedury* 1.5.2

Procedurę wywołuje się za pomocą instrukcji call. Podaje się przy tym identyfikator procedury wraz z listą jej parametrów aktualnych, np.

```
call zamień(a,b)
```

Lista parametrów aktualnych musi być zgodna co do długości i typów argumentów z listą parametrów formalnych wywoływanej procedury (por. p. 1.7).

### *Instrukcje warunkowe* 1.5.3

*Instrukcjami warunkowymi* w Loglanie są dwie postacie instrukcji **if** oraz instrukcja **case**

```
(1) if b then l1 fi;
(2) if b then l1 else l2 fi;
(3) case w
        when w1,...,wn:l1;
        ...
        when z1,...,zk:lm;
        otherwise l
        esac;
```

gdzie **b** jest wyrażeniem typu **boolean**, **w** jest wyrażeniem typu **integer** lub **character**, **w1**, ..., **wn**, ..., **z1**, ..., **zk** są różnymi stałymi o typie zgodnym z typem wyrażenia **w**, natomiast **l**, **l1**, ..., **lm** są ciągami instrukcji.

Wykonanie instrukcji **if** polega na sprawdzeniu warunku **b** i w zależności od jego wartości na wykonaniu ciągu instrukcji **l1** (gdy wartością **b** jest **true**) lub, w wypadku drugiego wariantu instrukcji **if**, wykonaniu ciągu instrukcji **l2** (gdy wartością **b** jest **false**).

Wykonanie instrukcji **case** sprowadza się do wykonania jednego z jej składowych ciągów instrukcji w zależności od wartości wyrażenia **w**:

— jeśli wartością **w** jest jedna z wartości **w1**, ..., **wn**, to jest wykonywany ciąg **l1**;

...

— jeśli wartością **w** jest jedna z wartości **z1**, ..., **zk**, to jest wykonywany ciąg **lm**;

— jeśli wartością **w** nie jest żadna z wartości **w1**, ..., **wn**, ..., **z1**, ..., **zk**, to jest wykonywany ciąg **l**.

Klauzula **otherwise** nie musi wystąpić w instrukcji **case**. Jeśli wówczas wartością **w** nie jest żadna spośród **w1**, ..., **wn**, ..., **z1**, ..., **zk**, to jest sygnalizowany błąd wykonania programu. Na przykład poprawnymi instrukcjami **case** są

```
case i
    when 1,2,3,4,5: dzień:=roboczy;
    when 6,7: dzień:=wolny;
```

```
        otherwise call błąd
        esac;

    case c
        when 'a': call a;
        when 'b','x': call b
        esac;
```

*Instrukcje iteracji* 1.5.4

Najbardziej ogólną postacią *instrukcji iteracji* jest pętla postaci do...od. Wykonanie pętli do K od polega na powtarzaniu instrukcji K do chwili napotkania instrukcji umożliwiającej wyjście z pętli. Wyjście takie umożliwia instrukcja exit. Na przykład poniższy fragment programu

```
suma:=0; i:=0;
do i:=i+1;
  suma:=suma+i;
  if i=100 then exit fi
  od;
```

oblicza sumę kolejnych liczb naturalnych od 1 do 100. Obliczenie pętli kończy się w chwili, gdy $i = 100$. Wartością zmiennej suma jest wówczas $1 + 2 + \cdots + 100$.

Dla wygody programisty wprowadzono w Loglanie także dwie inne postacie instrukcji iteracji: while i for. Instrukcja while przyjmuje postać

```
while b do K od;
```

gdzie b jest wyrażeniem typu boolean, zaś K dowolnym ciągiem instrukcji. Jest ona równoważna pętli

```
do if not b then exit fi;
  K
  od;
```

Instrukcja for przyjmuje jedną z dwóch postaci:

```
(1) for i:=i1 step i2 to i3 do K od;
(2) for i:=i1 step i2 downto i3 do K od;
```

gdzie wyrażenia i1, i3 są oba typu integer lub character, wyrażenie i2 typu integer o dodatniej wartości, zaś i jest zmienną o typie zgodnym z typem i1. Postać (1) instrukcji for jest równoważna pętli

```
i:=i1;
do if i>i3 then exit fi;
  K;
  i:=i+i2 (* gdy i jest zmienną znakową, operacja +
          dotyczy kodu znaku reprezentowanego przez i,
          tzn. wykonywana jest instrukcja
          i := chr(ord(i)+i2) *)
  od;
```

W wypadku postaci (2) instrukcji **for**, wartość zmiennej **i** będzie zmniejszana o wartość **i2**. Wyjście z pętli nastąpi, gdy **i<i3**. Jeśli **i2** ma wartość **1**, klauzulę **step i2** można pominąć. Tak więc np. instrukcja

```
for i:=i1 downto i3 do K od;
```

równoważna jest instrukcji:

```
for i:=i1 step 1 downto i3 do K od;
```

Instrukcja **exit** jest szczególnym przypadkiem instrukcji **exit...exit repeat**. Instrukcja tej postaci może występować wewnątrz dowolnej pętli, przy czym jest ona określona, jeżeli **exit** występuje **j**-krotnie (**j>=0**), **repeat** występuje **k**-krotnie (**k=0** lub **k=1**) oraz jest ona zanurzona wewnątrz co najmniej **k+j** zagnieżdżeń pętli. Taka instrukcja oznacza zakończenie wykonywania **j** najbliżej otaczających ją instrukcji iteracji oraz, jeżeli **repeat** występuje, zaniechanie bieżącego powtórzenia (**j+1**)-szej iteracji i przejście do kolejnego jej powtórzenia. Zilustrujmy semantykę omawianej instrukcji przykładem. Załóżmy w tym celu, że mamy daną kwadratową macierz **n** × **n** o elementach dodatnich i chcemy znaleźć wiersz z minimalną sumą elementów. Na początek przyjmiemy, że wierszem o minimalnej sumie jest wiersz pierwszy

```
wiersz:=1; min_suma:=0;
for j:=1 to n do min_suma:=min_suma+A(1,j) od;
```

Potem będziemy liczyć sumy elementów w następnych wierszach. Sumę elementów **i**-tego wiersza liczymy tylko dopóty, dopóki nie przekroczy ona wartości minimalnej sumy

```
for i:= 2 to n do
   suma:=0;
   for j:=1 to n do
      suma:=suma+A(i,j);
      if min_suma<=suma then exit repeat fi
      od;
   wiersz:=i; min_suma:=suma
   od;
```

Zauważmy, że instrukcje wiersz:=i oraz min_suma:=suma wykonają się tylko wówczas, gdy suma elementów i-tego wiersza jest mniejsza niż wartość zmiennej min_suma.

## Bloki 1.6

*Blok* jest powszechnie znaną strukturą, zaczerpniętą z Algolu, która występuje w większości współczesnych języków programowania. Jest on jedynym modułem, którego deklaracja pokrywa się z wykonaniem. Dlatego też występuje w innych modułach w ciągu instrukcji. Blok składa się z lokalnych deklaracji i listy instrukcji. Wielkości zadeklarowane w bloku nie są widoczne na zewnątrz tego bloku. Program główny jest także blokiem. Jest to jedyny blok, który może mieć nazwę. Jeżeli program główny ma postać bloku bez nazwy, to jest dostępny pod nazwą main. Poniższy przykład wyjaśni zasady widoczności w strukturze blokowej.

```
block var x,y:integer; (* 1 *)
    ...
    block var x,z:integer; (* 2 *)
        begin
        (* W tym fragmencie programu jest widoczna
        zmienna y zadeklarowana w linii oznaczonej
        (1) oraz zmienne x, z zadeklarowane w linii
        oznaczonej (2). Nastąpiło zasłonięcie
        deklaracji x z bloku zewnętrznego przez
        deklarację x z bloku wewnetrznego *)
        end;
    (* W tym fragmencie programu są widoczne jedynie
    zmienne x, y zadeklarowane w linii (1). Zmienne z
    bloku wewnętrznego nie są widoczne. *)
end;
```

## Podprogramy 1.7

W Loglanie, podobnie jak w innych językach programowania, istnieją dwa rodzaje *podprogramów* — *procedury* i *funkcje*. Deklaracja podprogramu składa się z nagłówka, deklaracji oraz ciągu instrukcji. Nagłówek podprogramu wprowadza identyfikator podprogramu i specyfikuje listę *parametrów formalnych* oraz, w wypadku funkcji, typ wyniku. Na przykład procedura

```
unit zamień: procedure(inout x,y:real);
    var z:real;
    begin z:=x; x:=y; y:=z end zamień;
```

zamienia wartości dwóch zmiennych rzeczywistych. Zmienne te są dostarczone do procedury w chwili jej wywołania jako *parametry aktualne* (por. p. 1.5.2). Tryb ich przekazywania () oznacza, że w chwili wywołania procedury wartości jej parametrów aktualnych będą przypisane parametrom formalnym, a po wykonaniu ciągu instrukcji procedury nastąpi przypisanie wartości parametrów formalnych parametrom aktualnym. Parametry będące zmiennymi można również przekazywać w trybie **input** oraz **output**. Jeżeli parametr jest przekazywany w trybie , to w chwili wywołania podprogramu następuje przypisanie parametrowi formalnemu wartości parametru aktualnego. Jeżeli parametr jest przekazywany w trybie , to w chwili powrotu z podprogramu wartość parametru formalnego jest przypisywana parametrowi aktualnemu. Wynika stąd, że w przypadku trybu **inout** i **output** odpowiedni parametr aktualny, ze względu na końcowe przypisywanie wartości, musi być zmienną. W przypadku trybu **input** odpowiedni parametr aktualny może być wyrażeniem. Jeśli tryb przekazywania parametru nie jest wyspecyfikowany, przyjmuje się, iż jest nim **input**.

W omawianym przykładzie zmienna **z** jest zadeklarowana w procedurze jako lokalna. Zmienne lokalne podprogramu są inicjowane w sposób standardowy każdorazowo w chwili wywołania podprogramu. Po końcowym **end** podprogramu może zostać powtórzony identyfikator wprowadzony w nagłówku. Jest to dobry zwyczaj zwiększający czytelność programu i umożliwiający kompilatorowi kontrolę odpowiedniości początków i końców modułów.

W razie wystąpienia funkcji w nagłówku jest dodatkowo specyfikowany typ wyniku. Może być nim dowolny typ. Następująca funkcja:

```
unit silnia: function(n:integer): integer;
    var i:integer;
    begin
    if n<0 then return fi;
    result:=1;
    for i:=2 to n do result:=result*i od
    end silnia;
```

oblicza wartość n! dla zadanego parametru **n**. W funkcji **silnia** użyliśmy instrukcji . Może ona wystąpić również w treści procedur. Jej wykonanie oznacza zakończenie wykonywania podprogramu.

Każda funkcja obok zadeklarowanych zmiennych lokalnych i parametrów formalnych ma także dodatkową zmienną o typie wyniku funkcji.

Zmienna ta, podobnie jak każda inna zmienna lokalna, jest inicjowana standardowo w chwili wywołania funkcji. Można jej używać w treści funkcji, a ponadto, w chwili zakończenia wykonywania funkcji, jej wartością staje się bieżąca wartość zmiennej result.

Procedury i funkcje mogą być definiowane rekurencyjnie. Każde wystąpienie identyfikatora podprogramu w jego treści oznacza rekurencyjne wołanie tego podprogramu. Dla przykładu rozważmy rekurencyjną definicję funkcji silnia.

```
unit silnia: function(n:integer): integer;
    begin
    if n<0 then return fi;
    if n=0 or n=1 then result:=1
    else result:=n*silnia(n-1) fi
    end silnia;
```

Loglan oferuje znacznie większe możliwości parametryzacji podprogramów, niż przedstawiliśmy to w tym punkcie. Do zagadnienia tego wrócimy w rozdz. 4.

## Pliki 1.8

### *Wprowadzenie* 1.8.1

W tym rozdziale występują pewne różnice w stosunku do raportu Loglanu. System plików zaproponowany w raporcie nie został jednak wiernie zaimplementowany. Konkretne realizacje różnią się od siebie. Poniżej opisujemy więc najpowszechniej używaną implementację systemu plików Loglanu, opracowaną dla mikrokomputerów z rodziny IBM PC.

Typ *plikowy* należy do typów obiektowych. Służy on do komunikacji z pamięcią pomocniczą. Dostęp do tej pamięci jest sekwencyjny. Obiekt pliku reprezentuje nieograniczony ciąg bajtów. Sposób ich interpretacji zależy od rodzaju pliku. Z plikiem jest stowarzyszony wskaźnik bieżącej pozycji. Operacje wejścia/wyjścia dotyczą pozycji przezeń wskazywanej. Deklarując zmienne typu plikowego podaje się identyfikator file jako nazwę typu, np.

```
var f:file.
```

Mogą one być używane wszędzie tam, gdzie są dozwolone zmienne typów referencyjnych. Zmiennym typu plikowego nadaje się standardowo początkową wartość none.

Obiekty plików, podobnie jak obiekty innych typów referencyjnych, mogą być dynamicznie tworzone oraz usuwane z pamięci. Do tworzenia obiektów pliku służy instrukcja **open**. Parametrami tej instrukcji jest zmienna typu plikowego oraz specyfikacja rodzaju pliku. Specyfikacja ta może być jednym z identyfikatorów: **text**, **character**, **integer**, **real**. W zależności od rodzaju, pliki dzielimy na tekstowe i binarne. Elementy *pliku tekstowego* (specyfikacja **text**) interpretujemy jako znaki tekstu, natomiast elementy *pliku binarnego* (specyfikacja **character**, **integer** lub **real**) interpretujemy jako zapis binarny wartości odpowiedniego typu. Możliwy jest trzeci parametr instrukcji **open** będący wyrażeniem typu **array_of character** (por. p. 1.2.6 i 2.1), określający nazwę zewnętrzną pliku. W razie braku tego parametru plik jest traktowany jako tymczasowy i przestaje być dostępny z chwilą zakończenia wykonywania programu. W wyniku wykonania instrukcji **open** jest tworzony nowy obiekt pliku. Wskaźnik do tego obiektu jest pamiętany jako wartość zmiennej będącej pierwszym parametrem instrukcji. Jednocześnie jest otwierany plik w pamięci pomocniczej. Takie oto przykładowe instrukcje otwarcia plików

```
open(f1,integer);
open(f2,text,nazwa_pliku)
```

spowodują utworzenie dwóch nowych obiektów plików: tymczasowego pliku binarnego o elementach typu **integer** oraz pliku tekstowego, dostępnego pod nazwą zewnętrzną określoną przez tablicę **nazwa_pliku**.

Usuwanie obiektu pliku z pamięci jest dokonywane za pomocą instrukcji **kill(f)** (por. p. 1.4). Skutkiem ubocznym jest zamknięcie pliku zewnętrznego stowarzyszonego z danym obiektem. Dodatkowo są określone dwie standardowe procedury przygotowujące plik do komunikacji. Jedna z nich musi zostać wywołana przed wykonaniem pierwszej instrukcji wejścia/wyjścia. Procedura **reset(f)** przygotowuje plik do czytania, tzn. ustawia wskaźnik bieżącej pozycji na początek pliku. Procedura **rewrite(f)** przygotowuje plik do zapisu, czyli tworzy nowy pusty plik.

System plików może być rozszerzany w konkretnych realizacjach Loglanu (np. o pliki bezpośredniego dostępu). Ponadto, bardziej odpowiednie dla różnych zastosowań języki operacji plikowych można definiować za pomocą klas używając prefiksowania (por. rozd. 5).

### *Pliki binarne* 1.8.2

Dla zadanego typu elementów E (może nim być **character**, **integer** lub **real**) typ **pliki binarne** definiujemy następująco:

**pliki_binarne**(E) = (*pliki_binarne*; **get**, **put**, **eof**)

gdzie

(1) nośnikiem typu jest zbiór obiektów reprezentujących skończone ciągi elementów typu E;
(2) get: *pliki_binarne* → *pliki_binarne* × *E*;
(3) put: *pliki_binarne* × *E* → *pliki_binarne*;
(4) eof: *pliki_binarne* → *boolean*.

Operacja get jest realizowana jako instrukcja get(f,x), w której podaje się modyfikowany plik oraz zmmienną typu E. Wykonanie tej instrukcji powoduje odczytanie wartości elementu pliku f wskazywanego przez wskaźnik bieżącej pozycji i przypisanie tej wartości zmiennej x. Podobnie, operacja put przyjmuje postać put(f,w), gdzie f jest modyfikowanym plikiem, natomiast w wyrażeniem typu E. W wyniku wykonania tej operacji postać binarna wartości wyrażenia w zostanie zapisana na plik f. Po wykonaniu każdej z tych operacji wskaźnik bieżącej pozycji jest przesuwany o jedno miejsce naprzód. Funkcja eof(f) sprawdza, czy wskaźnik bieżącej pozycji znajduje się na końcu pliku. Dla wygody programisty wprowadzono syntaktyczny skrót wielu operacji wejścia/wyjścia

get(f, x1, ..., xn);
put(f, w1, ..., wn);

gdzie x1, ..., xn są zmiennymi typu E, natomiast w1, ..., wn są wyrażeniami typu E. W wyniku wykonania operacji get(f, x1, ..., xn) wartości n kolejnych elementów pliku f zostaną odczytane i przypisane zmiennym x1, ..., xn. Wskaźnik bieżącej pozycji pliku będzie wskazywał miejsce następne za ostatnio odczytanym. Operacja put(f, w1, ..., wn) spowoduje zapisanie na plik f kolejno wartości wyrażeń w1, ..., wn.

### *Pliki tekstowe* 1.8.3

Typ pliki tekstowe możemy opisać następująco:

pliki_tekstowe = (*pliki_tekstowe*; read, readln, write, writeln, eoln, eof)

gdzie

(1) nośnikiem typu jest zbiór reprezentujący skończone ciągi znaków;
(2) read: *pliki_tekstowe* → *pliki_tekstowe* × *character*;
(3) write: *pliki_tekstowe* × *character* → *pliki_tekstowe*;
(4) readln, writeln: *pliki_tekstowe* → *pliki_tekstowe*;
(5) eoln, eof: *pliki_tekstowe* → *boolean*.

Operacje read oraz write, które są odpowiednikami operacji get i put dla plików binarnych, przyjmują podobnie postać instrukcji

```
read(f,x);
write(f,w);
```

gdzie f jest modyfikowanym plikiem, a zmienna x i wyrażenie w są typu character. Operacje te powodują transmisję jednego znaku na plik f lub z pliku f. Dla wygody programisty wprowadzono też możliwość transmisji na plik tekstowy (z pliku tekstowego) elementów typów arytmetycznych: integer oraz real. W tym przypadku wykonanie operacji read(f,x) spowoduje odczytanie z pliku f najdłuższej pasującej sekwencji znaków jako wartości typu takiego, jak typ zmiennej x oraz przypisanie tej wartości zmiennej x. Wskaźnik bieżącej pozycji będzie wskazywał miejsce następujące po ostatnim odczytanym znaku. Podobnie wykonanie operacji write(f,w) spowoduje przekształcenie wartości wyrażenia zadanego jako parametr w odpowiadający jej ciąg znaków, a następnie zapisanie tego ciągu znaków na plik f. Operacja write(f,w) może być ponadto użyta do zapisania na plik f wartości typu napisowego. Format zapisanego na plik ciągu znaków może być określony w instrukcji

```
write(f, x:A1)
```

gdzie A1 jest wyrażeniem arytmetycznym typu integer. Wartość tego wyrażenia określa długość napisu reprezentującego wartość x, który ma być zapisany na plik. Jeżeli wartość A1 jest większa niż liczba znaków potrzebna do zapisania wartości x, to zapisana wartość jest poprzedzana odpowiednią liczbą spacji; jeżeli wartość A1 jest zbyt mała, to są obcinane najmniej znaczące cyfry. W przypadku zapisywania na plik tekstowy liczb rzeczywistych można również określić liczbę cyfr po kropce dziesiętnej. Specyfikuje się ją jako wartość wyrażenia A2 typu integer, np.

```
write(f, x:A1:A2)
```

Brak określonego formatu oznacza zastosowanie formatu standardowego (zależnego od konkretnej implementacji).

Operacje readln, writeln są związane z podziałem pliku tekstowego na linie. Operacja readln(f) powoduje pominięcie reszty bieżącej linii pliku f i przesunięcie wskaźnika bieżącej pozyji pliku na początek następnej linii. Operacja writeln(f) powoduje zapisanie na plik f znacznika końca linii (zamknięcie bieżącej linii).

Podobnie jak w przypadku plików binarnych możliwy jest skrótowy zapis wielu operacji wejścia/wyjścia:

```
read(f,x1,...,xn);
write(f,x1,...,xn);
```

Ponadto, jest możliwe syntaktyczne połączenie operacji read z readln oraz write z writeln. Operacja readln(f, x1, ..., xn) jest równoważna wykonaniu

read(f, x1, ..., xn); readln(f). Analogicznie, operacja writeln(f, w1, ..., wn) jest równoważna wykonaniu write(f, w1, ..., wn); writeln(f).

Znaczenie operacji eof jest takie samo jak dla plików binarnych, natomiast funkcja eoln(f) sprawdza, czy wskaźnik bieżącej pozycji znajduje się na końcu linii pliku f.

W języku występują dwa standardowe pliki oznaczające plik wejściowy i wyjściowy. Jeżeli operacja dotyczy pliku standardowego, to nie podaje się parametru określającego plik.

### *Przykład programu* 1.8.4

Przedstawiony program kopiuje bez zmian plik tekstowy o nazwie źródło na plik o nazwie wynik. W szczególności podział pliku na linie pozostaje bez zmiany.

```
block var f1,f2:file, ch:character;
    begin
    open(f1,text,unpack("źródło"));
    (* unpack jest funkcją standardową powodującą
       przekształcenie napisu będącego jej parametrem
       w tablicę znaków odpowiadającą temu napisowi,
       por. 1.2.7, 2.1, Dodatek B *)
    open(f2,text,unpack("wynik"));
    call reset(f1); call rewrite(f2);
    while not eof(f1) do
        while not eoln(f1) do
            read(f1,ch); write(f2,ch)
            od;
        readln(f1); writeln(f2);
        od;
     kill(f1); kill(f2)
     end.
```

Zauważmy, że ten program nie jest równoważny instrukcji f2:= copy(f1), taka instrukcja spowoduje jedynie powstanie nowego „wewnętrznego opisu" pliku f1 i przypisanie f2 wartości wskaźnika do tego opisu.

# Tablice dynamiczne 2

## Wprowadzenie 2.1

Jednym z najprostszych typów obiektowych jest typ *tablicowy*. Służy on do reprezentowania skończonych ciągów elementów dowolnego typu. W Loglanie wszystkie tablice są *tablicami dynamicznymi*. Oznacza to, że obiekty tablic są tworzone w czasie wykonywania programu. Możliwe jest zatem ustalenie zakresów indeksów dopiero w trakcie obliczeń, zależnie od aktualnego stanu programu. Przy deklaracjach zmiennych typu tablicowego należy podać typ elementów tablicy

```
var t: array_of E
```

Dla zadanego typu elementów E, typ tablicowy można opisać następująco:

tablice(E) = (*tablice*; array dim, lower, upper, =, =/=)

gdzie

(1) nośnikiem typu jest zbiór obiektów reprezentujących skończone ciągi zmiennych typu *E*;
(2) array dim: *integer* × *integer* → *tablice*;
(3) lower, upper: *tablice* − {none} → *integer*;
(4) =, =/=: *tablice* × *tablice* → *boolean*.

Operacja array dim służy do tworzenia obiektów tablic. Jej parametrami są dwie liczby całkowite określające dolną i górną granicę zakresu tablicy. W wyniku wykonania instrukcji array T dim (l:u) tworzy się nowy, dotychczas nie używany w programie, obiekt tablicy i wskaźnik do niego jest zapamiętany na zmiennej T. Nowo utworzony obiekt tablicy reprezentuje ciąg T(l), T(l+1), ..., T(u) zmiennych typu E. Zmienne te są inicjowane zgodnie ze standardem przyjętym dla typu E.

W trakcie obliczeń programu można sprawdzić, jakie są granice zakresu tablic. Do tego celu służą funkcje lower i upper. Dla tablicy utworzonej instrukcją array T dim (l:u) lower(T) = l, zaś upper(T) = u.

Ponieważ typ tablicowy jest również typem obiektowym, stosują się do niego operacje copy i kill. Zauważmy przy tym, iż elementy tablic, jako zmienne, są traktowane jako ich atrybuty, dlatego też operacja copy zastosowana do zmiennej tablicowej powoduje skopiowanie całej zawartości tablicy, wraz z wartością jej elementów (oczywiście w przypadku tablic wielowymiarowych zostaną skopiowane już tylko wskaźniki do tablic reprezentujących dalsze wymiary). Konwencja ta stosuje się także do operatora kill — jeśli wartościami elementów tablicy są wskaźniki, wskazywane przez nie obiekty nie są usuwane. Stała none określa pustą tablicę (por. p. 1.4). Jest ona też standardowo początkową wartością zmiennych typu tablicowego.

## Iloczyn skalarny wektorów — tablice jednowymiarowe 2.2

### *Sformułowanie problemu* 2.2.1

W pliku wejściowym są dane w kolejnych liniach:

— liczba naturalna dodatnia $n$;
— $n$ liczb całkowitych — wartości elementów pierwszego wektora;
— $n$ liczb całkowitych — wartości elementów drugiego wektora.

Oblicz i wypisz na plik wyjściowy wartość iloczynu skalarnego tych wektorów. Iloczyn skalarny wektorów $(a1, \ldots, an)$ i $(b1, \ldots, bn)$ definiujemy jako $a1 * b1 + a2 * b2 + \ldots + an * bn$.

### *Dyskusja zadania* 2.2.2

Na początku należy wczytać dane wejściowe. Załóżmy, że dane są poprawne, tzn. w pierwszej linii pliku wejściowego podana jest dodatnia liczba naturalna $n$, a następne dwie linie zawierają po $n$ liczb całkowitych.

Do rozwiązania zadania konieczne będzie stworzenie wewnętrznych reprezentacji obu wektorów. Reprezentacje te muszą umożliwiać dostęp do poszczególnych elementów. Na elementach tych powinny być określone działania arytmetyczne +, *. Algorytm zbudujemy za pomocą iteracji. W każdym jej kroku wartości bieżących elementów będą mnożone, a wynik dodawany do tego, co do tej pory zostało obliczone.

*Omówienie rozwiązania* 2.2.3

Dogodną reprezentacją wektora liczb całkowitych jest tablica o elementach typu integer. W Loglanie typ tablicowy definiuje się przy deklaracjach zmiennych typu tablicowego. Specyfikuje się wówczas typ elementów tablicy. Do rozwiązania naszego zadania będziemy potrzebować dwóch zmiennych typu tablicowego

```
var wekt1, wekt2: array_of integer;
```

Aby nadać im właściwe wartości początkowe, należy wykonać następujące instrukcje:

```
readln(n);
array wekt1 dim (1:n);
array wekt2 dim (1:n);
```

Do wartości elementów tablicy odwołujemy się używając zmiennych indeksowanych. Zmienna wekt1(i) oznacza i-ty element tablicy wekt1.

W kolejnych dwóch pętlach for wczytamy dane. Po przeczytaniu elementów pierwszego wektora przechodzimy do nowego wiersza

```
for i:=1 to n do read(wekt1(i)) od;
readln;
for i:=1 to n do read(wekt2(i)) od;
```

Również wartość iloczynu skalarnego obliczamy w pętli for. Do obliczenia tej wartości używamy zmiennej iloczyn. Programowe inicjowanie zmiennej nie jest konieczne, ponieważ zgodnie ze standardem dla typu integer przyjmuje ona wartość początkową 0

```
for i:=1 to n do iloczyn:=iloczyn+wekt1(i)*wekt2(i) od;
```

*Rozwiązanie* 2.2.4

```
block
    var wekt1, wekt2: array_of integer;
    var n, iloczyn, i: integer;
    begin
    readln(n); array wekt1 dim (1:n);array wekt2 dim (1:n);
    for i:=1 to n do read(wekt1(i)) od;
    readln;
    for i:=1 to n do read(wekt2(i)) od;
    for i:=1 to n do iloczyn:=iloczyn+wekt1(i)*wekt2(i) od;
    writeln(" Iloczyn skalarny = ",iloczyn)
    end;
```

# 2.3 Rozwiązywanie układu równań liniowych z macierzą trójkątną — wielowymiarowe tablice różnych kształtów

## 2.3.1 *Sformułowanie problemu*

Dana jest rzeczywista macierz trójkątna górna **A** rozmiaru $n \times n$ oraz $n$-elementowy wektor $\boldsymbol{b}$ liczb rzeczywistych. Napisać funkcję, której wynikiem jest taki wektor $\boldsymbol{x}$, że $\mathbf{A} * \boldsymbol{x} = \boldsymbol{b}$. Zakładamy przy tym, że wszystkie elementy diagonalne macierzy **A** są różne od zera.

## 2.3.2 *Dyskusja zadania*

Przepiszmy równanie $\mathbf{A} * \boldsymbol{x} = \boldsymbol{b}$ w formie czytelniejszej, pomijając te elementy macierzy, o których wiemy na pewno, że są zerami

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ & a_{22} & \dots & a_{2n} \\ & & \ddots & \\ & & & a_{nn} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Sprowadzając zadanie do układu równań, uzyskujemy

$$\begin{aligned} a_{11} * x_1 + a_{12} * x_2 + \dots + a_{1n} * x_n &= b_1 \\ a_{22} * x_2 + \dots + a_{2n} * x_n &= b_2 \\ \dots & \\ a_{nn} * x_n &= b_n \end{aligned}$$

Wartość niewiadomej $x_n$ wyznaczamy z ostatniego równania

(1) $x_n = b_n / a_{nn}$

Następnie, dla $k = n-1, \dots, 1$, po obliczeniu $x_n, x_{n-1}, \dots, x_{k+1}$, z $k$-tego równania układu obliczamy $x_k$

(2) $x_k = (b_k - (a_{k,k+1} * x_{k+1} + \dots + a_{kn} * x_n))/a_{kk}$

## 2.3.3 *Omówienie rozwiązania*

Ustalmy najpierw sposób reprezentacji danych. Wektor $\boldsymbol{b}$ może być reprezentowany przez tablicę o elementach typu real. Możemy zadeklarować zmienną tablicową i utworzyć obiekt tablicy odpowiedniej długości.

```
var b: array_of real;
...
array b dim (1:n);
```

Naturalną formą reprezentacji macierzy A jest tablica dwuwymiarowa, najlepiej o kształcie trójkątnym. Zauważmy, że w deklaracji zmiennej tablicowej typ elementu może być dowolnym typem, w szczególności znów tablicowym

```
var A: array_of array_of real;
```

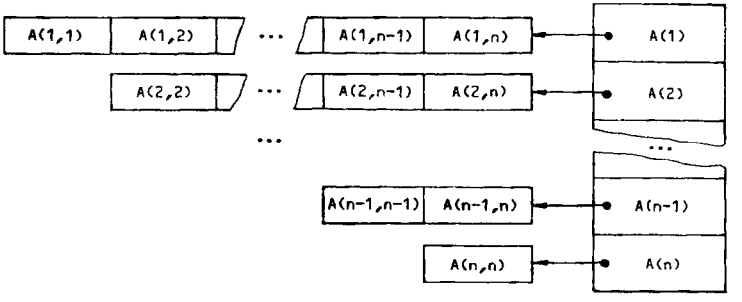
Wartością zmiennej A może być wskaźnik do tablicy, której elementami są wskaźniki do obiektów tablic liczb rzeczywistych. Jedną instrukcją tworzącą obiekt tablicy można utworzyć jedynie tablicę jednowymiarową

```
array A dim (1:n);
```

Zmienna A wskazuje na nowo utworzony obiekt tablicy mający n elementów typu **array_of real** o wartościach **none**. Obiekt każdego wiersza tablicy trzeba utworzyć osobno.

```
for i:=1 to n do array A(i) dim (i:n) od;
```

Każdy wiersz ma długość $n - i + 1$, zatem została utworzona tablica trójkątna górna (por. rys. 2.1). Poszczególne elementy tablicy mają wartość zero.




RYS. 2.1 Tablica reprezentująca macierz trójkątną górną

Zastanówmy się teraz nad nagłówkiem funkcji. Parametrami wejściowymi będą: rozmiar zadania $n$, macierz A oraz wektor $b$. Obliczony wektor $x$ będzie wynikiem funkcji.

```
unit rozwiąż: function (n:integer,
                        A:array_of array_of real,
                        b:array_of real): array_of real;
```

Wynik funkcji jest w jej treści reprezentowany przez standardową zmienną **result** (por. p. 1.7), której typem jest **array_of real**. Ponieważ początkową wartością zmiennej **result** jest **none**, należy utworzyć wynikową tablicę odpowiedniego rozmiaru

```
array result dim (1:n);
```

W trakcie obliczeń będą potrzebne wartości elementów tablicy A. Można odwoływać się do nich podając wartości obu indeksów. Można też skorzystać z faktu, iż tablica dwuwymiarowa jest jednowymiarową tablicą wskaźników do tablic jednowymiarowych, czyli A(i) oznacza wskaźnik do tablicy reprezentującej i-ty wiersz, zaś A(i)(j) oznacza j-ty element tej tablicy (a więc to samo co A(i,j)).

Wartość ostatniego elementu wektora wynikowego obliczymy wprost ze wzoru (1)

```
result(n):=b(n)/A(n,n);
```

Do obliczeń wartości pozostałych elementów będziemy używać zmiennej pomocniczej **suma**, reprezentrującej sumę występującą we wzorze (2)

```
for j:=k+1 to n do suma:=suma+A(k,j)*result(j) od;
```

Obliczenie to będzie powtórzone w pętli dla poszczególnych równań układu.

*Rozwiązanie* 2.3.4

```
unit rozwiąż: function (n:integer,
                        A:array_of array_of real,
                        b:array_of real): array_of real;
    var suma:real, k,j:integer;
    begin
    array result dim (1:n);
    result(n):=b(n)/A(n,n);
    for k:=n-1 downto 1 do
        suma:=0;
        for j:=k+1 to n do suma:=suma+A(k,j)*result(j) od;
        result(k):=(b(k)-suma)/A(k,k)
        od
    end rozwiąż;
```

# Podsumowanie 2.4

(1) Deklaracja zmiennej typu tablicowego wprowadza identyfikator tej zmiennej oraz specyfikuje typ jej elementów.

(2) Wartościami zmiennej typu tablicowego mogą być wskaźniki do obiektów tablic. Zmienna taka jest inicjowana standardowo wartością none oznaczającą pusty wskaźnik.

(3) Obiekty tablic są tworzone w trakcie obliczeń za pomocą instrukcji array dim. Podczas tworzenia obiektu są podawane zakresy indeksów tablicy.

(4) Instrukcja array dim tworzy tablicę jednowymiarową. Jeżeli tablica ma mieć więcej wymiarów, konieczne jest osobne utworzenie wszystkich jej składowych .

(5) Dostęp do elementów tablicy jest osiągalny za pośrednictwem zmiennych indeksowanych. Odwołanie Y(i)(j)(k) jest równoważne odwołaniu Y(i,j,k).

(6) Jeśli Y jest zmienną typu tablicowego oraz Y=/=none, to lower(Y), upper(Y) zwracają wartości dolnej i górnej granicy zakresu indeksów.

(7) Instrukcja przypisania może dotyczyć zmiennych tablicowych.

(8) Jeśli B jest tablicą jednowymiarową, to instrukcja A:=copy(B) powoduje, iż A wskazuje na nową tablicę będącą kopią B. Wartości elementów A są takie same jak wartości odpowiednich elementów B.

(9) Typ tablicowy może być typem wyniku funkcji.

# Klasa jako typ danych 3

## Wprowadzenie 3.1

Podstawowym pojęciem w Loglanie jest pojęcie klas. Klasy stanowią bogaty mechanizm definiowania typów obiektowych (por. p. 1.4). *Klasa* jest modułem, a zatem wzorcem, według którego są tworzone egzemplarze tej klasy. Daje ona znacznie szersze możliwości niż prostsze moduły jak bloki, procedury czy funkcje. Celem bloku (procedury, funkcji) jest wykonanie ciągu operacji zgodnie z podanym wzorcem. Po zakończeniu swoich akcji obiekt bloku (procedury, funkcji) jest automatycznie usuwany wraz ze swymi wewnętrznymi strukturami danych, podczas gdy obiekt klasy nadal istnieje i może być stosowany przez otoczenie. Klasy są istotnym uogólnieniem pojęcia rekordu znanego z Pascala czy Cobolu. Umożliwiają one definiowanie całego typu danych w ramach jednego modułu. W tym rozdziale przedstawimy klasę jako typ danych. Przypomnijmy przy tym, że typ danych to nie tylko zbiór pewnych wartości. Rozważmy bowiem słowa maszynowe. Definiują one zbiór wartości, które mogą być interpretowane na wiele różnych sposobów:

— jako liczby całkowite, jeśli wykonujemy na nich operacje charakterystyczne dla tych liczb;

— jako ciągi wartości typu boolean, jeśli wykonujemy na nich operacje logiczne;

— jako liczby rzeczywiste, znaki, napisy itd.

Zatem znaczenie typu danych w sposób istotny zależy od rodzaju operacji związanych z wartościami tego typu. Zgodnie więc z przyjętą konwencją (patrz Przedmowa), typ danych definiujemy jako zbiór wartości wraz z operacjami na nich określonymi.

W Loglanie klasy są najważniejszym narzędziem definiowania nowych typów danych. Deklaracja typu klasowego pokrywa się z deklaracją

klasy jako modułu i syntaktycznie przypomina deklarację procedury lub funkcji

```
unit T: class;
    ... (* atrybuty klasy *) ...
    end T;
```

Deklaracja taka wprowadza nowy typ obiektowy. Przypomnijmy (por. p. 1.4), że obok operacji zdefiniowanych przez programistę, do typów klasowych stosują się standardowe operacje typów obiektowych: **new** (tworzenia nowego obiektu), **kill** (usuwania obiektu), **copy** (kopiowania obiektu) oraz **=**, **=/=** (porównywania wskaźników). Istnieje także wyróżniony obiekt pusty **none**.

## Liczby zespolone — złożony typ danych 3.2

### *Sformułowanie problemu* 3.2.1

Zaprogramuj typ danych odpowiadający liczbom zespolonym. Przyjmujemy przy tym, iż

zespolone = (*zespolone*; re, im, moduł, sprzężona, plus, razy, równe)

gdzie

(1) nośnikiem typu jest zbiór par liczb rzeczywistych;
(2) re, im, moduł: *zespolone* → *real*;
(3) sprzężona: *zespolone* → *zespolone*;
(4) plus, razy: *zespolone* × *zespolone* → *zespolone*;
(5) równe: *zespolone* × *zespolone* → *boolean*.

Przypomnijmy, że operacje na liczbach zespolonych z1=(re1,im1) i z2=(re2,im2) są określone następująco:

```
re(z1)=re1, im(z1)=im1;
moduł(z1)=sqrt(re1*re1+im1*im1);
sprzężona(z1)=(re1,-im1);
plus(z1,z2)=(re1+re2,im1+im2);
razy(z1,z2)=(re1*re2-im1*im2,re1*im2+re2*im1);
równe(z1,z2)=(re1=re2 and im1=im2).
```

*Dyskusja zadania* 3.2.2

Do reprezentacji liczby zespolonej jest potrzebna struktura o dwu składowych typu real. Zastosujemy w tym celu konstrukcję klasy. Klasa może mieć własne atrybuty: dane lokalne oraz wewnętrzne procedury i funkcje operujące na tych danych. Jest też możliwe zainicjowanie danych instrukcjami klasy. Wróćmy teraz do naszego zadania. Pojawiają się dwie możliwości jego rozwiązania:

(1) Rozwiązanie wykorzystujące klasę jako rekord z możliwością niestandardowej inicjacji danych. Funkcje byłyby zrealizowane na zewnątrz klasy implementującej liczbę zespoloną. Zatem, na przykład dodawanie byłoby funkcją mającą jako dwa argumenty rekordy odpowiadające liczbom zespolonym i zwracającą jako wynik rekord odpowiadający ich sumie.

(2) Rozwiązanie w pełni wykorzystujące możliwości klas, które w swej istocie odzwierciedlają strukturę typu danych. Zatem, na przykład dodawanie będzie atrybutem obiektu klasy implementującej liczby zespolone. Aby dodać dwie liczby, będziemy wołać funkcję dodawania należącą do jednej z nich i zadawać drugą liczbę w postaci parametru.

W następnym punkcie przedstawimy drugie rozwiązanie.

*Omówienie rozwiązania* 3.2.3

Typ zespolone definiujemy jako typ klasowy

```
(1) unit zespolone: class;
        var re, im: real;
        end zespolone;
```

Typ klasowy może być sparametryzowany. Możemy zatem zadeklarować re oraz im jako parametry wejściowe klasy zespolone

```
(2) unit zespolone: class(re,im: real); end zespolone;
```

Deklarując zmienne typu klasowego podajemy nazwę klasy jako typ zmiennej

```
var z1,z2: zespolone;
```

Inicjalną wartością zmiennych typu klasowego jest none. W czasie obliczeń programu wartością zmiennej typu klasowego może być wskaźnik do konkretnego obiektu klasy.

Zanim w programie użyje się zmiennej odpowiadającej liczbie zespolonej, trzeba utworzyć odpowiedni obiekt klasy. Do tworzenia obiektów

klas służy operator **new**. W przypadku deklaracji (2), w wyniku wykonania instrukcji

```
z1:=new zespolone(0.0,1.5);
```

zostanie utworzony obiekt o wartościach atrybutów re=0.0 i im=1.5, a zmienna z1 będzie wskazywała ten obiekt. Jeżeli w klasie są zadeklarowane zmienne lokalne, to w czasie tworzenia obiektu są one inicjowane w sposób standardowy. Na przykład w przypadku deklaracji (1) po wykonaniu instrukcji

```
z1:=new zespolone;
```

zmienne lokalne obiektu wskazywanego przez z1 mają wartość zero.

Zmienne lokalne zadeklarowane w klasie oraz jej parametry są atrybutami obiektu klasy. Atrybuty te są dostępne z zewnątrz obiektu podobnie jak pola rekordu w Pascalu. Należy jedynie podać zmienną wskazującą obiekt oraz identyfikator atrybutu, np.

```
z1.re:=5.3; z1.im:=0.0;
```

Odwołania tego rodzaju nazywamy *odległym dostępem* do atrybutów obiektu. Spowodują one nadanie nowych wartości zmiennym lokalnym w przypadku deklaracji (1) lub parametrom w przypadku deklaracji (2).

Załóżmy, że mamy dwie zmienne z1, z2 typu **zespolone**. Zastanówmy się, jak wyrazić równość liczb zespolonych reprezentowanych przez te zmienne. Jak już zauważyliśmy wcześniej (por. p. 1.4), porównanie zmiennych z1 i z2 na ogół nie da poprawnego wyniku. Na przykład po wykonaniu instrukcji

```
z1:=new zespolone(1.0,0.0);
z2:=new zespolone(1.0,0.0);
```

wartości z1 i z2 są różne, bo zmienne te wskazują różne obiekty, chociaż o tych samych wartościach atrybutów. Aby sprawdzić, czy z1 i z2 reprezentują tę samą liczbę zespoloną, należy więc porównać wartości odpowiednich atrybutów.

Wartość zmiennej typu klasowego można zmienić za pomocą instrukcji przypisania, np.

```
z1:=new zespolone(1.0,0.0); z2:=z1;
```

Zmienna z2 uzyskuje wartość wskaźnika do tego samego obiektu, który wskazywała zmienna z1. Zatem po wykonaniu powyższych instrukcji z1=z2.

Moduł liczby zespolonej może być obliczany funkcją, której parametrem będzie dana liczba zespolona i której wynikiem będzie liczba rzeczywista.

```
unit moduł: function(z: zespolone): real;
    begin result:=sqrt(z.re*z.re + z.im*z.im) end moduł;
```

Zauważmy, że w takim przypadku wartość modułu danej liczby zespolonej będzie obliczana ilekroć o nią zapytamy. Tymczasem jest to wartość właściwa danej liczbie zespolonej, nie zmieniająca się w czasie. Toteż można zdefiniować moduł jako zmienną lokalną klasy i obliczać jej wartość raz, w chwili tworzenia obiektu klasy. Loglan umożliwia wygodne zrealizowanie tej koncepcji, ponieważ definicja klasy może zawierać listę instrukcji

```
unit zespolone: class(re,im: real);
    var moduł: real;
    begin moduł:=sqrt(re*re+im*im) end zespolone;
```

Lista instrukcji wykonuje się w chwili tworzenia obiektu klasy. Przed wykonaniem instrukcji klasy parametry wejściowe (**input**, **inout**) są przekazywane do jej obiektu. Zakończenie wykonywania instrukcji klasy następuje w wyniku napotkania instrukcji **return** lub końcowego **end** klasy. Po ich wykonaniu dokonuje się przekazanie parametrów wyjściowych (**output**, **inout**) oraz sterowania do obiektu modułu, w którym był wywołany operator **new**. Zatem po wykonaniu instrukcji

```
z:=new zespolone(3.0,4.0);
```

obiekt wskazywany przez **z** będzie miał następujące wartości atrybutów: z.re=3.0, z.im=4.0, z.moduł=5.0, bowiem 5.0=sqrt(3.0*3.0+4.0*4.0).

Działania arytmetyczne na liczbach zespolonych można zrealizować jako funkcje o parametrach typu **zespolone** dające wynik typu **zespolone**, np.

```
unit plus: function(z1,z2: zespolone): zespolone;
    begin result:=new zespolone(z1.re+z2.re,z1.im+z2.im)
    end plus;
```

Można też potraktować możliwość dodawania jako cechę właściwą liczbom zespolonym i zdefiniować operację dodawania wewnątrz definicji klasy realizującej liczbę zespoloną. W Loglanie jest możliwe zadeklarowanie procedur i funkcji wewnątrz deklaracji klasy. Zgodnie z zasadami widoczności przyjętymi w językach o strukturze blokowej (por. p. 1.6), z wnętrza takich procedur lub funkcji jest możliwy dostęp do wszystkich danych lokalnych klasy. Mogą one zatem operować na tych danych, np.

```
unit zespolone: class(re,im: real);
    var moduł: real;
    unit plus: function(z: zespolone): zespolone;
```

```
        begin result:=new zespolone(re+z.re,im+z.im)
        end plus;
    begin moduł:=sqrt(re*re+im*im) end zespolone.
```

Procedury i funkcje zadeklarowane w klasie są tak samo atrybutami obiektu klasy, jak zmienne. Można odwoływać się do nich z zewnątrz obiektu podając zmienną wskazującą obiekt oraz identyfikator procedury lub funkcji. Zdefiniowana funkcja **plus** ma tylko jeden parametr. Drugim jest liczba zespolona reprezentowana przez obiekt, którego atrybutem jest funkcja **plus**. Przyjmiemy zatem następującą wygodną konwencję:

> Jeśli jednym z parametrów podprogramu jest obiekt klasy, w której podprogram ten jest zadeklarowany, parametr ten opuszczamy. Jako wartość opuszczonego parameru przyjmujemy obiekt, z którego podprogram jest wywoływany.

Ilustracją tej konwencji jest np. rozważana wyżej funkcja **plus**. I tak w wyniku działania następujących instrukcji

```
var z1,z2,z3: zespolone;
  ...
z1:=new zespolone(4.5,-7.0);
z2:=new zespolone(-1.0,0.0);
z3:=z1.plus(z2);
```

zmiennej z3 zostanie przypisany wskaźnik do nowo utworzonego obiektu odpowiadającego sumie liczb reprezentowanych przez obiekty wskazywane przez zmienne z1 i z2.

Dostęp do atrybutów obiektu klasy spoza jej treści jest w praktycznym programowaniu niezbędny, jednak jeśli nie jest on w żaden sposób ograniczony, może spowodować niepożądane skutki. Np. po utworzeniu nowego obiektu

```
z:=new zespolone(3.0,4.0);
```

jego atrybuty będą miały poprawne wartości reprezentujące odpowiednio część rzeczywistą, urojoną i moduł liczby zespolonej (3.0, 4.0). Jeżeli teraz zostaną wykonane instrukcje

```
z.re:=0.0; z.moduł:=1.0;
```

to wartości atrybutów ulegną zmianie zaburzając własność, narzuconą przez specyfikację liczb zespolonych, **moduł=sqrt(re*re+im*im)** jako typu danych.

W Loglanie istnieje możliwość zabezpieczania atrybutu przed niepożądaną ingerencją z zewnątrz obiektu klasy. W tym celu należy na

początku deklaracji danej klasy podać nazwę atrybutu na liście specyfikacji close, np.

```
close re,im,moduł;
```

Atrybuty występujące w specyfikacji close nie są dostępne na zewnątrz obiektów. Daje to możliwość ukrycia szczegółów implementacyjnych typu danych i zwiększenia bezpieczeństwa programowania -- programista może tak zaprogramować typ danych, iż wszelkie doń odwołania są zgodne z przewidywanym przeznaczeniem tego typu.

W danej klasie może wystąpić wiele specyfikacji close. Skutek jest równoważny specyfikacji z łączną listą atrybutów, np. specyfikacja

```
close re,im; close moduł;
```

jest równoważna wcześniej rozważanej specyfikacji

```
close re,im,moduł;
```

Zauważmy, że ograniczenie jakie wprowadza specyfikacja close jest, w naszym przypadku, zbyt silne. Atrybuty klasy zespolone są zabezpieczone przed jakimkolwiek dostępem z zewnątrz, w szczególności nie jest możliwe odczytanie wartości zmiennych występujących w specyfikacji close. Loglan nie ma gotowego mechanizmu pozwalającego na selektywne ograniczanie dostępu do atrybutu, możemy jednak łatwo takie ograniczenie zaprogramować. Wystarczy atrybuty będące zmiennymi ukryć za pomocą specyfikacji close, a dodatkowo zadeklarować funkcje, dające w wyniku ich wartości, np.

```
unit zespolone: class(rev,imv: real);
    close rev,imv,modułv;
    var modułv: real;
    unit re: function: real; begin result:=rev end re;
    unit im: function: real; begin result:=imv end im;
    unit moduł: function: real;
        begin result:=modułv end;
    ...
    begin modułv:=sqrt(rev*rev+imv*imv) end zespolone;
```

*Rozwiązanie* 3.2.4

```
unit zespolone: class(rev,imv: real);
    close rev,imv,modułv;
    var modułv: real;
    (* funkcje zwracające wartości atrybutów *)
```

```
unit re: function: real; begin result:=rev end re;
unit im: function: real; begin result:=imv end im;
unit moduł: function: real;
    begin result:=modułv end moduł;
(* dodawanie, mnożenie i równość liczb zespolonych *)
unit plus: function(z: zespolone): zespolone;
    begin result:=new zespolone(rev+z.re,imv+z.im)
    end plus;
unit razy: function(z: zespolone): zespolone;
    begin
    result:=new zespolone(rev*z.re -imv*z.im,
                          rev*z.im+imv*z.re)
    end razy;
unit równe: function(z: zespolone): boolean;
    begin result:=re=z.re and im=z.im end równe;
begin modułv:=sqrt(rev*rev+imv*imv) end zespolone;
```

## 3.3 Binarne drzewa wyważone — rekurencyjne definiowanie typów danych

### 3.3.1 *Sformułowanie problemu*

Zaprogramuj typ danych reprezentujący drzewa binarne o wierzchołkach zawierających liczby całkowite, a następnie utwórz i wypisz binarne drzewo wyważone zawierające n podanych na wejściu liczb całkowitych. Przyjmij, że

drzewa_binarne = (*drzewa_binarne*; lewe, prawe, wartość)

gdzie

(1) nośnikiem jest zbiór drzew binarnych;
(2) lewe, prawe: *drzewa_binarne* – {none} → *drzewa_binarne*;
(3) wartość: *drzewa_binarne* – {none} → *integer*.

Drzewem binarnym nazywamy skończony zbiór elementów, zwanych wierzchołkami, który albo jest pusty, albo zawiera wierzchołek zwany korzeniem wraz z dwoma rozłącznymi drzewami binarnymi, zwanymi lewym i prawym poddrzewem.

Zakładamy, że operacje lewe i prawe mają standardowe znaczenie, tj. zwracają w wyniku odpowiednio lewe i prawe poddrzewo danego niepustego drzewa. Dla niepustego drzewa jest także określona funkcja wartość, której wynikiem jest liczba całkowita zapamiętana w korzeniu drzewa.

Drzewo jest wyważone, jeśli dla każdego wierzchołka, liczby wierzchołków zawartych w jego lewym i prawym poddrzewie różnią się co najwyżej o 1.

### *Dyskusja zadania* 3.3.2

Do reprezentacji drzewa należy użyć dynamicznej struktury danych. Rekurencyjna definicja drzew sugeruje, aby skorzystać z oferowanej przez Loglan możliwości rekurencyjnego definiowania typów danych. Tego rodzaju definicje są możliwe dzięki temu, że identyfikator klasy może występować wewnątrz jej deklaracji lokalnych.

Zauważmy jeszcze, że nasze zadanie można rozwiązać także za pomocą tablic dynamicznych. W takim przypadku stracilibyśmy jednak wiele z czytelności i elegancji przyjętego przez nas rozwiązania, niezbędny byłby bowiem jeszcze krok pośredni polegający na odpowiednim zakodowaniu drzewa w tablicy. Co więcej, zdefiniowanie rekurencyjnego typu danych umożliwia także podanie prostych i naturalnych algorytmów rekurencyjnych rozwiązujących postawiony problem.

### *Omówienie rozwiązania* 3.3.3

Typ danych drzewa_binarne reprezentuje klasa o następującej strukturze:

```
unit drzewa_binarne: class ...;
    var lewe,prawe: drzewa_binarne;
    var wartość: integer;
    ...
    end drzewa_binarne;
```

Zauważmy, że powyższy typ danych jest zdefiniowany rekurencyjnie: atrybuty lewe i prawe są definiowane za pomocą identyfikatora aktualnie definiowanej klasy drzewa_binarne.

Przedstawiona definicja z grubsza rozwiązywałaby pierwszą część problemu. Ponieważ jednak chodzi nam o rozwiązanie nieco ogólniejszego zadania, uzupełnimy zdefiniowaną strukturę o nowe atrybuty:

— parametr formalny definiujący rozmiar drzewa;
— procedurę buduj_drzewo tworzącą drzewo z danych na wejściu liczb całkowitych;
— procedurę wypisz_drzewo wypisującą zbudowane drzewo.

Procedura buduj_drzewo nie będzie widoczna na zewnątrz obiektów, posłuży jedynie do ich inicjacji w chwili tworzenia. Stosując się również do uwag o ochronie atrybutów, zawartych w p. 3.2.3, zmodyfikujemy treść

klasy **drzewa_binarne** tak, aby atrybuty **wartość**, **lewe** i **prawe** nie mogły być zmieniane z zewnątrz obiektów tej klasy. Uzyskana struktura danych będzie opisywać tylko drzewa binarne wyważone.

```
unit drzewa_binarne: class(n: integer);
    close n,lewe1,prawe1,buduj_drzewo;
    var lewe1,prawe1: drzewa_binarne, wartość: integer;
    unit lewe: function: drzewa_binarne; ... end lewe;
    unit prawe: function: drzewa_binarne; ... end prawe;
    unit buduj_drzewo: procedure; ... end buduj_drzewo;
    unit wypisz_drzewo: procedure;
        ... end wypisz_drzewo;
    begin ... call buduj_drzewo; ...
    end drzewa_binarne;
```

Procedurę **buduj_drzewo** zaprogramujemy zgodnie z następującym naturalnym algorytmem:

Użyj jeden wierzchołek na korzeń.
Zbuduj rekurencyjnie lewe poddrzewo, używając połowy wierzchołków.
Zbuduj rekurencyjnie prawe poddrzewo, używając pozostałych wierzchołków.

Procedura **wypisz_drzewo** dla uproszczenia będzie wypisywać jedynie wierzchołki drzewa, z pominięciem krawędzi, zachowując jednak jego graficzną strukturę (w porównaniu z typową reprezentacją graficzną drzew — z korzeniem na górze, a liśćmi na dole, uzyskany przez nas obraz będzie obrócony o 90 stopni w lewo). Również w tym przypadku algorytm będzie sformułowany rekurencyjnie. Ponieważ wygodne będzie wprowadzenie parametru formalnego **odstęp**, sterującego wydrukiem, w procedurze **wypisz_drzewo** zdefiniujemy wewnętrzną procedurę działającą zgodnie z następującym algorytmem:

Drukuj prawe poddrzewo z odstępem równym (**odstęp** + 1).
Drukuj wartość zapamiętaną w korzeniu z odstępem równym **odstęp**.
Drukuj lewe poddrzewo z odstępem równym (**odstęp** + 1).

Dzięki wprowadzeniu tej wewnętrznej procedury ukryliśmy przed użytkownikiem szczegóły implementacyjne, toteż nie musi on pamiętać o nieistotnych, z jego punktu widzenia, parametrach wewnętrznej procedury. Ponieważ procedura wypisywania drzewa jest bezparametrowa, zgodnie z konwencją opisaną w p. 3.2.3 oznacza to, iż wywołanie

```
call d.wypisz_drzewo
```

powinno spowodować wypisanie drzewa d. Aby było to możliwe, wewnątrz procedury wypisz_drzewo powinien być dostępny wskaźnik do drzewa d. Dany jest on w Loglanie jako wartość wyrażenia this drzewa_binarne. Ogólnie, jeśli wyrażenie this A występuje wewnątrz modułu A, to jego wartością jest wskaźnik do obiektu typu A, w którym wyrażenie to jest obliczane. Wyrażenie this omówimy w p. 5.4 i 5.5, w znacznie ogólniejszym kontekście.

*Rozwiązanie* 3.3.4

```
block
    var drzewo: drzewa_binarne;
    unit drzewa_binarne: class(n: integer);
        close n,lewe1,prawe1,buduj_drzewo;
        var lewe1,prawe1: drzewa_binarne,
            wartość: integer;
        unit lewe: function: drzewa_binarne;
            begin result:=lewe1 end lewe;
        unit prawe: function: drzewa_binarne;
            begin result:=prawe1 end prawe;
        unit buduj_drzewo: procedure;
            begin
            writeln(" podaj liczbę całkowitą");
            readln(wartość);
            if n div 2 > 0 then
              lewe1:=new drzewa_binarne(n div 2) fi;
            if n-(n div 2)-1 > 0 then
              prawe1:=new drzewa_binarne(n-(n div 2)-1)
              fi
            end buduj_drzewo;
        unit wypisz_drzewo: procedure;
            var i: integer;
            unit w: procedure(d:drzewa_binarne,
                              odstęp:integer);
                begin
                if d=/=none then
                  call w(d.prawe,odstęp+1);
                  write(" ");
                  for i:=1 to odstęp do write(" ") od;
                  writeln(wartość);
                  call w(d.lewe,odstęp+1)
                  fi
```

```
            end w;
        begin  call w(this drzewa_binarne,0)
        end wypisz_drzewo;
    begin if n>0 then call buduj_drzewo fi
    end drzewa_binarne;
begin
(* utworzenie drzewa *)
drzewo:=new drzewa_binarne(100);
(* wypisanie drzewa *)
call drzewo.wypisz_drzewo
end;
```

## Podsumowanie 3.4

**(1)** Klasę można traktować jako typ danych będący uogólnieniem rekordu dynamicznego znanego z języka Pascal. Identyfikatora klasy można użyć wszędzie tam, gdzie można użyć dowolnego innego typu. W szczególności można deklarować zmienne typu klasowego. Wartościami zmiennych typu klasowego mogą być wskaźniki do obiektów tablic lub wartość none odpowiadająca pustemu wskaźnikowi. Inicjalną wartością zmiennych klasowych jest none

**(2)** Klasa może być sparametryzowana, może zawierać lokalne dane oraz definicje operacji na tych danych. Może również zawierać listę instrukcji.

**(3)** Obiekty klasy są tworzone w czasie wykonania programu operatorem new. Podaje się wówczas parametry aktualne klasy. Utworzenie obiektu polega na zarezerwowaniu miejsca w pamięci na dane lokalne klasy, zainicjowaniu w sposób standardowy wartości tych danych oraz wykonaniu instrukcji klasy. Przed wykonaniem instrukcji zostają przekazane parametry wejściowe, a po wykonaniu instrukcji parametry wyjściowe.

**(4)** Zadeklarowane w klasie zmienne lokalne, parametry oraz procedury i funkcje są atrybutami obiektu klasy. Można się do nich odwoływać spoza obiektu korzystając z odległego dostępu.

**(5)** Można zabronić odległego dostępu do atrybutów obiektu klasy spoza tego obiektu. Służy temu specyfikacja close.

**(6)** Wartością wyrażenia this A występującego wewnątrz modułu A jest wskaźnik do obiektu typu A, w którego instrukcjach wyrażenie to jest obliczane.

# Parametryzacja modułów 4

## Wprowadzenie 4.1

W Loglanie wszystkie moduły, oprócz bloków, mogą mieć *parametry*. Dla wszystkich typów modułów sposób ich parametryzacji jest jednakowy. Istnieją trzy podstawowe rodzaje parametrów:

— zmienne;
— podprogramy;
— typy formalne.

Specyfikację parametrów rozpoczyna się od definicji ich rodzaju: **input**, **output**, **inout**, **procedure**, **function** lub **type**. Brak definicji rodzaju parametru oznacza domyślnie parametr **input**. Parametry specyfikowane jako **input**, **output** lub **inout** muszą być zmiennymi. Przypadek ten omówiliśmy już wcześniej, przy okazji omawiania podprogramów (por. p. 1.7).

Jeśli specyfikacja parametru rozpoczyna się od słowa **procedure** albo **function**, oznacza to, że parametrem będzie podprogram. Podprogramy będące parametrami mają podaną listę parametrów oraz, w przypadku funkcji, typ wyniku. Dla podprogramu będącego parametrem specyfikuje się jego uproszczoną listę parametrów. Uproszczenie polega na tym, że jeżeli taki podprogram ma również parametr funkcyjny lub proceduralny, to nagłówka tego parametru się nie podaje, np.

```
unit F: procedure(function G(x:real, procedure H): real);
```

W tym przykładzie parametrów procedury H nie wyspecyfikowaliśmy.

Przed wykonaniem instrukcji modułu, którego parametrem jest podprogram, zastępuje się go procedurą lub funkcją aktualną. Nagłówki podprogramów formalnych i aktualnych muszą być zgodne, np. identyczne. Dokładne reguły zgodności podano w Dodatku C.

Aby wyspecyfikować parametr będący typem, należy podać jego identyfikator poprzedzony słowem kluczowym **type**. Takiego typu (tzw. typu

formalnego) można następnie użyć do specyfikacji pozostałych parametrów modułu. W tym przypadku specyfikacja parametru będącego typem musi poprzedzać korzystające z niego specyfikacje innych parametrów, np.

```
unit sortuj: procedure(type T; A:array_of T;
                       function mniejsze(x,y:T):boolean);
```

Zastosowanie typów formalnych umożliwia definiowanie algorytmów oraz struktur danych na wysokim poziomie abstrakcji, niezależnie od konkretnych typów elementów. Istnieją jednak pewne ograniczenia użycia typów formalnych. Może on wystąpić jedynie w deklaracjach zmiennych oraz w definicjach innych typów. Zmienne typu formalnego mogą występować w instrukcjach przypisania, natomiast nie można ich użyć w sytuacji, gdy jest wymagana znajomość struktury wskazywanego przez nie obiektu. Na przykład nie można odwoływać się do konkretnych atrybutów typu formalnego, nie jest też możliwe tworzenie jego nowych obiektów. Przed wykonaniem instrukcji modułu mającego parametr będący typem musi on być zastąpiony typem aktualnym. Typ aktualny może być dowolnym typem obiektowym (por. p. 1.4) lub innym typem formalnym, nie może być natomiast typem pierwotnym.

## Szukanie miejsca zerowego funkcji — podprogram jako parametr modułu 4.2

### *Sformułowanie problemu* 4.2.1

Zaprogramuj algorytm szukania miejsca zerowego dowolnej funkcji $f$ ciągłej, rzeczywistej, określonej na przedziale $[a, b]$ i spełniającej warunek $f(a) * f(b) < 0$ .

### *Dyskusja zadania* 4.2.2

Istnienie szukanego miejsca zerowego dla tak określonej funkcji wynika z podstawowych faktów analizy matematycznej. Algorytm, który opiszemy, nosi nazwę bisekcji. Najpierw badamy wartość funkcji w punkcie $x = (a + b)/2$, który jest środkiem przedziału $[a, b]$. Jeżeli $f(x) = 0$, to znaleźliśmy szukany punkt. Jeżeli natomiast $f(x) =/= 0$, to albo w punktach $a$ oraz $x$, albo w punktach $x$ oraz $b$ wartości funkcji $f$ mają różne znaki. Zawężamy przedział przeszukiwany do tej połowy, na której końcach funkcja przyjmuje wartości różnych znaków. Znów

badamy wartość funkcji w środku nowego przedziału, itd. Tą metodą możemy znaleźć dowolne przybliżenie szukanego punktu. Załóżmy, że poszukiwanie kończymy, gdy znajdziemy szukany punkt lub długość przeszukiwanego przedziału będzie mniejsza niż zadana wielkość *eps*. W drugim przypadku przyjmujemy, że rozwiązaniem jest środek przeszukiwanego przedziału.

### *Omówienie rozwiązania* 4.2.3

W Loglanie funkcja może być parametrem innego modułu, dlatego operację szukania miejsca zerowego możemy zaprogramować w sposób jednolity dla różnych funkcji. Algorytm szukania miejsca zerowego funkcji metodą bisekcji zaprogramujemy w postaci funkcji, która będzie miała trzy parametry wejściowe typu **real**: parametry **a**, **b** reprezentujące końce przedziału, **eps** reprezentujący dokładność oraz parametr funkcyjny odpowiadający funkcji $f$.

```
unit bisec: function(a,b,eps: real;
                     function f(x: real): real): real;
```

Jeśli na przykład szukamy miejsca zerowego funkcji $g(x) = x * x - 5$ w przedziale [0, 3] z dokładnością $1.E - 8$, to najpierw definiujemy funkcję $g$, a następnie wywołujemy funkcję **bisec** z odpowiednimi parametrami

```
var miejsce_zerowe: real;
unit g: function(x: real): real;
    begin result := x*x-5 end g;
...
miejsce_zerowe:=bisec(0,3,1.E-8,g);
```

### *Rozwiązanie* 4.2.4

```
unit bisec: function(a,b,eps: real;
                     function f(x: real): real): real;
    var długość: real, zn: integer;
    unit znak: function(x: real): integer;
        begin if x<0 then result:=-1 else
              if x=0 then result:=0 else result:=1 fi fi
        end znak;
    begin
    zn:=znak(f(a)); (*znak liczby*)
    if znak(f(b))=zn then (*błędne dane*) return fi;
    długość:=b-a;
```

```
    do długość:=długość/2; result:=a+długość;
      if f(result)=0 then return fi;
      if długość<eps then return fi;
      if znak(f(result))=zn
        then a:=result else b:=result fi
      od
    end bisec;
```

## Uniwersalna procedura sortująca — typ jako parametr modułu 4.3

### *Sformułowanie problemu* 4.3.1

Napisz uniwersalną procedurę sortującą elementy jednowymiarowej tablicy w porządku niemalejącym. Elementy tablicy należą do zbioru uporządkowanego liniowo relacją *mniejsze*.

### *Dyskusja zadania* 4.3.2

Zadanie można rozwiązać w Loglanie po przyjęciu typu elementu tablicy jako parametru procedury sortującej. Relację liniowego porządku na zbiorze wartości elementów podamy jako parametr będący funkcją logiczną porównującą dwie wartości typu takiego jak typ elementów tablicy. Samo sortowanie będzie się odbywać zgodnie z następującym prostym algorytmem. Najpierw umieszczamy na końcu tablicy element o największej wartości, następnie drugi co do wielkości na przedostatnim itd.; wreszcie pozostanie element najmniejszy — na pierwszym miejscu w tablicy. Zaprogramowanie bardziej wyrafinowanych i efektywniejszych algorytmów pozostawiamy czytelnikowi jako ćwiczenie. Odsyłamy też do p. 5.3 i 8.2.

### *Omówienie rozwiązania* 4.3.3

Parametr, który jest typem, umożliwia zdefiniowanie abstrakcyjnego algorytmu działającego niezależnie od dowolnego konkretnego typu aktualnego. Pozwala on również utworzyć strukturę danych zawierającą elementy typu formalnego. Nagłówek procedury sortującej przyjmie taką oto postać:

```
unit sortuj: procedure( type T; tab:array_of T;
                        function mniejsze(x,y:T):boolean);
```

Tablica tab jest przekazywana w trybie input, mimo że jej elementy mają być modyfikowane. Jest to możliwe, gdyż wprawdzie zmieni się zawartość obiektu tablicy, nie zmieni się jednak wskaźnik do tego obiektu, który jest pamiętany jako wartość zmiennej tab.

Algorytm sortowania tablicy zaprogramujemy w postaci dwu zagnieżdżonych pętli for. W pętli zewnętrznej będzie się zmieniać górne ograniczenie g indeksów tablicy, wskazujące, która jej część pozostała jeszcze do posortowania. W pętli wewnętrznej będzie wybierany z tej części tablicy największy element, który następnie będzie umieszczany na miejscu tab(g).

Procedurę sortuj można stosować do sortowania tablic o różnych typach elementów. Przypuśćmy, że tablica zawiera informacje o studentach

```
unit student: class( nr_albumu: integer);
    var imię, nazwisko: array_of character,
        średnia_ocen: integer;
    end student;
var kartoteka: array_of student;
```

Załóżmy, że obiekt tablicy kartoteka jest już utworzony i zawiera interesujące nas dane. Jeśli chcemy je uporządkować np. ze względu na średnią ocen (od największej do najmniejszej), musimy zdefiniować odpowiednią funkcję porządku.

```
unit poprzedza: function( S1,S2: student): boolean;
    begin result := S1.średnia_ocen >= S2.średnia_ocen
    end poprzedza;
```

Po wywołaniu uniwersalnej procedury sortującej z odpowiednimi parametrami aktualnymi

```
call sortuj(student, kartoteka, poprzedza);
```

otrzymujemy posortowaną kartotekę.

A oto inny przykład zastosowania procedury sortuj. Załóżmy, że mamy daną dwuwymiarową tablicę liczb całkowitych

```
var tablica: array_of array_of integer;
```

i chcemy przestawić jej wiersze tak, by na początku były wiersze najkrótsze, a na końcu — najdłuższe. Odpowiednia funkcja porządku jest następująca:

```
unit krótszy: function(w1,w2: array_of integer): boolean;
    begin
    result:=(upper(w1)-lower(w1))<=(upper(w2)-lower(w2))
    end krótszy;
```

Po wywołaniu procedury sortuj z następujacymi parametrami:

```
call sortuj(array_of integer, tablica, krótszy);
```

otrzymujemy żądane uporządkowanie wierszy tablicy.

Jak już wspomnieliśmy wcześniej, typem aktualnym może być jedynie typ obiektowy. W szczególności typy pierwotne nie są dozwolone w roli typów aktualnych. Gdybyśmy chcieli posortować wektor np. liczb rzeczywistych względem zwykłego porządku, wówczas musielibyśmy albo napisać procedurę sortującą od początku, albo „opakować" wartość typu real w klasę, płacąc nadmiarową pamięcią za uniwersalność rozwiązania. Zamiast sortowania tablicy o elementach typu real sortowalibyśmy wtedy tablicę o elementach odpowiedniego typu klasowego.

```
unit klasa_real: class( wart: real); end klasa_real;
var wektor: array_of klasa_real;
```

Porównanie dwóch elementów takiej tablicy polegałoby na porównaniu ich wartości

```
unit mn: function( x,y: klasa_real): boolean;
    begin result := x.wart <= y.wart end mn;
```

Wywołanie procedury sortującej przyjęłoby wówczas następującą postać:

```
call sortuj(klasa_real, wektor, mn);
```

*Rozwiązanie* 4.3.4

```
unit sortuj: procedure(type T; tab:array_of T;
                       function mniejsze(x,y:T): boolean);
    var  max:T, dol,gór,g,indmax:integer;
    begin
    if tab=none then return fi;
    dol:=lower(tab); gór:= upper(tab);
    for g:=gór downto dol+1 do
        max:=tab(dol);  indmax:=dol;
        for i:=dol+1 to g do
            if mniejsze(max,tab(i)) then
              indmax:=i;  max:=tab(i) fi
            od;
       tab(indmax):=tab(g);  tab(g):=max
       od
    end sortuj;
```

# Kolejki — struktura danych sparametryzowana typem formalnym 4.4

## *Sformułowanie problemu* 4.4.1

Zaimplementuj strukturę danych reprezentującą kolejki elementów dowolnego typu. Dla ustalonego typu elementów E strukturę tę definiujemy następująco:

kolejki(E) = (*kolejki*; wstaw, usuń, pierwszy)

gdzie

(1) nośnikiem typu jest zbiór skończonych ciągów elementów z E;
(2) wstaw: $E \times$ *kolejki* $\rightarrow$ *kolejki*;
(3) usuń: *kolejki* $\rightarrow$ *kolejki*;
(4) pierwszy: *kolejki* $\rightarrow E$.

Wynikiem wstaw(e, q) jest kolejka, która powstaje z kolejki q przez dołączenie na jej koniec elementu e. Wynikiem usuń(q) jest kolejka uzyskana z q przez usunięcie jej pierwszego elementu, funkcja zaś pierwszy dostarcza pierwszy element z kolejki. W przypadku pustej kolejki przyjmujemy, że operacja usuń jest równoważna instrukcji pustej, operacja zaś pierwszy zwraca wartość none.

## *Dyskusja zadania* 4.4.2

Możliwe są dwa sposoby rozwiązania postawionego problemu — za pomocą tablic dynamicznych lub struktur danych definiowanych rekurencyjnie. Ponieważ tablice nie mogą mieć parametrów formalnych, aby wypełnić warunki zadania, należałoby tablicę reprezentującą kolejkę otoczyć klasą. Rozwiązanie za pomocą rekurencyjnych struktur danych jest bardziej bezpośrednie i na nim właśnie się skupimy, tym bardziej że w tym przypadku kolejka zajmuje w każdej chwili dokładnie tyle pamięci, ile jest niezbędne. Ponadto, operacja wstaw jest znacznie bardziej efektywna przy przyjęciu takiego właśnie rozwiązania.

## *Omówienie rozwiązania* 4.4.3

Definicja typu danych kolejki narzuca strukturę klasy implementującej go

```
unit kolejki: class(type E);
    unit wstaw: procedure(e: E); ... end wstaw;
```

```
unit usuń: procedure; ... end usuń;
unit pierwszy: function: E; ... end pierwszy;
end kolejki;
```

Wprowadzenie typu elementów jako parametru formalnego umożliwiło spełnienie wymagania, że implementacja kolejek ma być od tego typu niezależna. Zwróćmy uwagę, że zgodnie z przyjętą przez nas konwencją (por. p. 3.2.3) pominęliśmy niektóre z parametrów funkcji występujących w typie danych **kolejki**. Wartościami brakujących parametrów są obiekty, z których są wywoływane dane funkcje. Podobnie funkcje **wstaw** i **usuń** zastąpiliśmy przez procedury. Oznacza to, że ich wynikiem jest obiekt, z którego są wywoływane. Takie podejście jest naturalnym rozszerzeniem wspomnianej konwencji.

Do rozwiązania naszego zadania brakuje nam teraz definicji wewnętrznej struktury reprezentującej elementy kolejek wraz ze wskaźnikami do ich następników w kolejce. Zauważmy, że kolejkę można zdefiniować tak oto rekurencyjnie: kolejka składa się z elementu pierwszego i pozostałości, która także jest kolejką. Ujmując tę definicję w formalizm języka Loglan, uzyskujemy

```
unit kolejka: class;
    var pierwszy_element: E,
        pozostała_część: kolejka
    end kolejka;
```

Ze względu na wygodę, przyjmiemy że atrybut **pierwszy_element** będzie parametrem formalnym klasy **kolejka**. Aby umożliwić efektywną implementację operacji występujących w klasie **kolejki**, wprowadzimy do niej dodatkowe atrybuty **początek** i **koniec** reprezentujące początek i koniec kolejki. Zastanówmy się teraz, w jaki sposób zrealizować poszczególne operacje. Dla przykładu rozważmy procedurę **wstaw**. Przypuśćmy, że do kolejki **q** chcemy wstawić element **e**

```
call q.wstaw(e);
```

Możliwe są dwa przypadki. Jeżeli kolejka **q** jest pusta (sytuację tę rozpoznamy sprawdzając, czy wartość zmiennej **początek** jest równa **none**), to wstawienie elementu **e** polega wówczas na podstawieniu

```
początek, koniec:= new kolejka(e)
```

**Początek** i **koniec** wskazują wtedy na jedyny element kolejki, którego wartością jest **e**. Jeśli kolejka **q** jest niepusta, na koniec kolejki należy wstawić element **e**

```
koniec, koniec.pozostała_część:= new kolejka(e)
```

*Rozwiązanie* 4.4.4

```
unit kolejki: class(type E);
    close początek, koniec;
    var początek, koniec: kolejka;
    unit kolejka: class(pierwszy_element: E);
        var pozostała_część: kolejka;
        end kolejka;
     unit wstaw: procedure(e: E);
         begin
         if początek=none then
           początek,koniec:=new kolejka(e)
         else koniec,koniec.pozostała_część :=
                          new kolejka(e) fi
         end wstaw;
     unit usuń: procedure;
         var p: kolejka;
         begin
         if początek=/=none then
           p:=początek;
           początek:=początek.pozostała_część;
           kill(p) fi
         end usuń;
     unit pierwszy: function: E;
         begin
         if początek=/=none then
           result:=początek.pierwszy_element fi
         end pierwszy;
    end kolejki;
```

## Podsumowanie 4.5

(1) Parametrami formalnymi modułów mogą być zmienne, podprogramy i typy.

(2) Procedura lub funkcja może być parametrem innego modułu. Funkcja, która jest parametrem, ma podaną listę parametrów oraz typ wyniku, procedura zaś — jedynie listę parametrów. W ogólnym przypadku podprogramu będącego parametrem specyfikuje się jego uproszczoną listę parametrów. Uproszczenie polega na tym, że jeżeli taka procedura lub funkcja ma również parametr funkcyjny lub proceduralny, to nagłówka tego parametru się nie podaje.

(3) Każde wywołanie podprogramu będącego parametrem oznacza wywołanie odpowiedniego podprogramu podanego jako parametr aktualny. Podprogram aktualny musi mieć nagłówek zgodny z nagłówkiem podprogramu formalnego. Dokładne reguły zgodności podajemy w Dodatku C.

(4) Moduł może mieć parametr formalny, który jest typem. Typem aktualnym zastępującym typ formalny może być dowolny typ obiektowy lub inny typ formalny. Typy proste nie mogą zastępować typów formalnych.

(5) Typu formalnego można użyć do deklaracji zmiennych, do definicji typów oraz do specyfikacji innych parametrów modułu. Specyfikacja typu formalnego powinna poprzedzać na liście parametrów inne specyfikacje wykorzystujące ten typ.

(6) Zmienne typu formalnego mogą być użyte w podstawieniach. Nie można tych zmiennych natomiast zastosować tam, gdzie jest wymagana znajomość struktury wskazywanego przez nie obiektu. W szczególności nie można odwoływać się do jego atrybutów. Nie można też tworzyć obiektów typu formalnego.

# Prefiksowanie — operacja rozszerzania modułów 5

## Wprowadzenie 5.1

W tym rozdziale przedstawiamy operację rozszerzania modułów zwaną prefiksowaniem. Pojęcie prefiksowania zostało wprowadzone po raz pierwszy w Simuli-67. Chociaż koncepcja klas i prefiksowania stwarzała programiście poważne ograniczenia, stanowiła jednak na tyle atrakcyjną i wartościową propozycję, iż wywarła znaczny wpływ na powstanie i rozwój wielu nowych języków programowania (jak Concurrent Pascal i Modula). Również w Loglanie pojęcia klasy i prefiksowania zajmują pozycję centralną. Operacja prefiksowania wprowadzona w tym języku eliminuje ograniczenia Simuli. Programista uzyskuje tu wygodne narzędzie, które — konsekwentnie stosowane — może znacznie obniżyć koszty produkcji oprogramowania.

*Prefiksowanie* jest operacją rozszerzania, która działa na modułach. Jej argument (*moduł rozszerzany*) może być klasą, współprogramem lub procesem. *Rozszerzenie* jest specyfikowane również w formie modułu: klasy, współprogramu, procesu, bloku, procedury lub funkcji. W nagłówku rozszerzenia jest podawany identyfikator rozszerzanego modułu, np. po założeniu, że uprzednio zdefiniowano moduły o następujących nagłówkach:

```
unit wyrażenie: class;
unit przesiewanie: class;
unit geometria: class;
```

możemy zdefiniować ich rozszerzenia

```
unit stała: wyrażenie class;
unit sortuj: przesiewanie procedure;
pref geometria block;
```

TABLICA 5.1 Wyniki operacji rozszerzenia modułów

| prefiks / rozszerzenie | unit A: class | unit A: coroutine | unit A: process |
|---|---|---|---|
| pref A block | blok | nielegalne | nielegalne |
| unit p: A procedure | procedura | nielegalne | nielegalne |
| unit f: A function | funkcja | nielegalne | nielegalne |
| unit B: A class | klasa typu B (podtypu A) | współprogram typu B (podtypu A) | proces typu B (podtypu A) |
| unit C: A coroutine | współprogram typu C (podtypu A) | współprogram typu C (podtypu A) | proces typu C (podtypu A) |
| unit P: A process | proces typu P (podtypu A) | proces typu P (podtypu A) | proces typu P (podtypu A) |

Mimo że rozszerzenie przyjmuje formę modułu, nie musi być jednak modułem w pełni zdefiniowanym. W treści rozszerzenia można bowiem korzystać z definicji określonych w środowisku syntaktycznym rozszerzanego modułu. Przez *środowisko syntaktyczne modułu* rozumiemy tu zbiór deklaracji widocznych z wnętrza tego modułu. Wynikiem operacji prefiksowania jest *moduł rozszerzony*, którego rodzaj zależy od rodzaju rozszerzanego modułu oraz od rodzaju rozszerzenia — por. tabl. 5.1. Moduł, który jest wynikiem operacji prefiksowania, jest dostępny pod nazwą rozszerzenia. Na przykład powyższe rozszerzenia definiują następujące moduły: klasę **stała**, procedurę **sortuj** oraz blok. Argument operacji prefiksowania będziemy też nazywać *modułem prefiksującym* lub *prefiksem*, jej wynik zaś *modułem prefiksowanym*.

Prefiksowanie polega na rozszerzeniu zarówno listy deklaracji, jak i listy instrukcji modułu prefiksującego. Lista parametrów formalnych modułu prefiksowanego jest połączoną listą paramerów formalnych modułu prefiksującego i rozszerzenia. Rozszerzony zbiór deklaracji modułu prefiksowanego jest sumą zbioru deklaracji prefiksu (mówimy, że moduł prefiksowany dziedziczy deklaracje prefiksu) oraz zbioru deklaracji rozszerzenia. Gdy ten sam identyfikator jest zadeklarowany zarówno w module prefiksującym, jak i w rozszerzeniu, zachodzi zasłonięcie deklaracji z prefiksu deklaracją z rozszerzenia. Rozszerzenie listy instrukcji w wyniku prefiksowania polega na tym, że w chwili utworzenia egzemplarza modułu prefiksowanego wykonują się zarówno instrukcje określone w prefiksie, jak i instrukcje określone w rozszerzeniu. Sposób połączenia list instrukcji wyznacza instrukcja **inner**, która występuje w module prefiksującym. Określa ona miejsce, w którym zostaną wykonane instrukcje rozszerzenia. Podczas tworzenia egzemplarza modułu prefiksowanego są wykonywane

instrukcje prefiksu aż do napotkania instrukcji inner. Wykonanie instrukcji inner polega na przejściu do wykonywania listy instrukcji rozszerzenia. Po wyczerpaniu tej listy następuje powrót do wykonywania instrukcji prefiksu, począwszy od instrukcji następującej po inner. Instrukcja inner może wystąpić w ciągu instrukcji modułu co najwyżej raz. Może być natomiast wykonywana wielokrotnie, ponieważ może występować wewnątrz instrukcji strukturalnej, w szczególności wewnątrz pętli. Jeżeli instrukcja inner w module nie występuje, przyjmuje się domyślnie, że została umieszczona na końcu listy instrukcji. W module, który nie jest prefiksem, instrukcja inner jest instrukcją pustą. W czasie wykonywania instrukcji modułu prefiksowanego instrukcje pochodzące z prefiksu są wykonywane w środowisku prefiksu, natomiast instrukcje pochodzące z rozszerzenia — w środowisku rozszerzenia wzbogaconym o odziedziczone z prefiksu deklaracje.

Rozważmy przykład, w którym zarówno moduł prefiksujący, jak i prefiksowany, są klasami. Załóżmy, że są dane następujące deklaracje:

```
(1) unit A: class(pfA); (* parametry formalne A *)
        dA; (* deklaracje A *)
        begin I1A; inner; I2A (* instrukcje A *) end A;
```

oraz

```
(2) unit B: A class(pfB); (* parametry formalne
                                rozszerzenia *)
        dB; (* deklaracje rozszerzenia *)
        begin I1B; inner; I2B (* instrukcje
              rozszerzenia *) end B;
```

Wynikiem operacji prefiksowania jest klasa B. Listą jej parametrów formalnych jest połączona lista parametrów klasy A oraz rozszerzenia B. Podczas tworzenia obiektu klasy B należy zatem podać odpowiednio duży zestaw parametrów aktualnych

```
new B(paA,paB);
```

Atrybutami klasy B są atrybuty zdefiniowane w rozszerzeniu (pfB,dB) oraz te spośród atrybutów odziedziczonych z prefiksu (pfA,dA), które nie zostały przedefiniowane w rozszerzeniu. Zwróćmy uwagę, że instrukcje inner mogą w ogólnym przypadku być zagnieżdżone wewnątrz instrukcji strukturalnej. W naszym przykładzie występują one na poziomie listy instrukcji. Podczas tworzenia obiektu klasy B wykonają się następujące instrukcje

```
I1A; I1B; I2B; I2A.
```

Gdyby instrukcji inner nie było w klasie A, wówczas zostałby wykonany ciąg instrukcji

I1A; I2A; I1B; I2B.

Wykonanie instrukcji inner z rozszerzenia B jest równoważne instrukcji pustej, gdyż jest tworzony obiekt klasy B, a klasa B nie pełni tu funkcji prefiksu. Instrukcje I1A, I2A są wykonywane w środowisku modułu (1), instrukcje I1B, I2B — w środowisku modułu (2), rozszerzonym o odziedziczone atrybuty (pfA,dA).

## Znajdowanie okręgu opisanego na trójkącie — język problemowy 5.2

### *Sformułowanie problemu* 5.2.1

Dla zadanego trójkąta znajdź okrąg opisany na tym trójkącie. Trójkąt jest dany w postaci trzech par liczb — współrzędnych wierzchołków trójkąta. Szukany okrąg może być zdefiniowany przez podanie współrzędnych jego środka oraz długości promienia.

### *Dyskusja zadania* 5.2.2

Środkiem okręgu opisanego na trójkącie jest punkt przecięcia symetralnych boków trójkąta, czyli prostych prostopadłych do boków trójkąta i przechodzących przez ich środki. Symetralne boków trójkąta można wyznaczyć przez punkty przecięcia okręgów o środkach w wierzchołkach trójkąta i odpowiednio dużym promieniu. Środek szukanego okręgu jest wyznaczony przez przecięcie dwu symetralnych boków trójkąta. Jego promień jest dany jako odległość środka okręgu od dowolnego wierzchołka trójkąta.

W naszym zadaniu mamy do czynienia z obiektami geometrycznymi. Językiem programowania najwygodniejszym do jego rozwiązania byłby zatem język, który ma zdefiniowane takie pojęcia, jak prosta, okrąg, trójkąt, punkt przecięcia prostych itd. W Loglanie pojęcia te nie są dane bezpośrednio. Można jednak podać ich definicje za pomocą klas i uzyskać w ten sposób pewien język problemowy. Języka tego, dzięki mechanizmowi prefiksowania, można używać przy rozwiązywaniu wielu rozmaitych problemów formułowanych w języku geometrii. Wystarczy w tym celu zamknąć moduły definiujące pojęcia geometryczne wewnątrz jednego modułu - klasy, a następnie prefiksować tą klasą dowolny moduł,

w którym będzie stosowany język geometrii. Dzięki własnościom prefiksowania zdefiniowane pojęcia geometryczne staną się w takim module bezpośrednio dostępne.

Z bardzo prostym językiem problemowym mieliśmy już do czynienia w p. 4.4. Rozważaliśmy tam klasę **kolejki**, która definiowała trzy operacje: **wstaw, usuń** i **pierwszy**. Operacje te możemy traktować jako nowe instrukcje, widoczne np. w bloku prefiksowanym tą klasą. Każdy definiowany typ danych jest więc pewnym językiem problemowym. Również na odwrót — na język problemowy można patrzeć jako na typ danych, który dostarcza nowych pojęć i operacji wygodnych do rozwiązania pewnej klasy problemów.

Stosując się do powyższych uwag, rozwiązanie zadania zaprogramujemy w sposób ogólny. Zdefiniujemy najpierw język geometrii planarnej, który będzie dany przez następujący typ danych:

**geometria** = (*Punkty*, *Proste*, *Okręgi*, *Trójkąty*;
**równy**, **odległość**, **równoległa**, **przecięcie**, **przecięcie'**)

gdzie

(1) *Punkty*, *Proste*, *Okręgi*, *Trójkąty* są odpowiednio zbiorami punktów, prostych, okręgów i trójkątów;
(2) **równy**: *Punkty* × *Punkty* → *boolean*;
(3) **odległość**: *Punkty* × *Punkty* → *real*;
(4) **równoległa**: *Proste* × *Proste* → *boolean*;
(5) **przecięcie**: *Proste* × *Proste* → *Punkty*;
(6) **przecięcie'**: *Okręgi* × *Okręgi* → *Proste*.

Relacja **równy** bada identyczność dwóch punktów na płaszczyźnie. Wynikiem funkcji **odległość** (p1, p2) jest liczba rzeczywista określająca odległość p1 i p2. Relacja **równoległa** bada równoległość dwóch prostych. Operacje **przecięcie** i **przecięcie'** wyznaczają punkt przecięcia dwóch prostych oraz prostą przechodzącą przez punkty przecięcia okręgów.

### *Omówienie rozwiązania* 5.2.3

Zdefiniujemy najpierw klasę **geometria** realizującą język geometrii planarnej. Pojęcia punktów, linii prostych, okręgów i trójkątów będą reprezentowane przez wewnętrzne klasy zdefiniowane w klasie **geometria**. Operacje na obiektach geometrycznych związemy z tymi obiektami, na których są one określone. I tak relacja **równy** i funkcja **odległość** będą atrybutami klasy **Punkty**, relacja **równoległa** i funkcja **przecięcie** będą atrybutami klasy **Proste**, funkcja **przecięcie'** zaś będzie atrybutem klasy **Okręgi**.

Punkt jest wyznaczony przez współrzędne na płaszczyźnie. Współrzędne te będą parametrami klasy

```
unit Punkty: class(x,y: real);
```

Wewnątrz klasy Punkty zdefiniujemy także właściwe operacje

```
unit równy: function(p: Punkty): boolean;
unit odległość: function(p: Punkty): real;
```

Prostą będziemy reprezentować przez współczynniki A, B, C jej równania liniowego $A * x + B * y + C = 0$.

```
unit Proste: class(A,B,C: real);
```

W instrukcjach klasy będziemy sprawdzać, czy dana prosta jest poprawnie zdefiniowana. Błąd jest sygnalizowany wówczas, gdy współczynniki A i B są równe zeru. Jeżeli współczynniki okażą się poprawne, dokonamy ich normalizacji. Podobnie jak w przypadku punktów, w klasie Proste będą zdefiniowane operacje dotyczące prostych. Przecięciem dwóch prostych nierównoległych będzie punkt, którego współrzędne są wyliczane standardowo z równań prostych. Jeśli proste są równoległe, to wynikiem będzie none.

Atrybutami okręgu będzie jego środek i promień

```
unit Okręgi: class(p: punkt, r: real);
```

Promień musi być dodatnią liczbą rzeczywistą. Punkt będący środkiem okręgu musi być określony (różny od none). Warunki te są sprawdzane wewnątrz instrukcji inicjowania obiektów klasy Okręgi. Zdefiniujemy także operację znajdowania punktów przecięcia dwóch okręgów. Jeśli punkty te istnieją, to jej wynikiem jest prosta przez nie wyznaczona. W przeciwnym razie wynikiem jest none.

Trójkąty będą zdefiniowane przez trzy punkty reprezentujące wierzchołki

```
unit Trójkąty: class;
    var w1, w2, w3: Punkty; ...
    end Trójkąty;
```

Wartości współrzędnych poszczególnych wierzchołków będą czytane z wejścia. Zbadamy przy tym, czy podawane punkty się nie pokrywają.

Zdefiniowaną klasę geometria wykorzystamy jako prefiks dla bloku rozwiązującego zadanie. Syntaktycznie nagłówek bloku prefiksowanego różni się od nagłówków innych modułów prefiksowanych

```
pref geometria block;
```

Ponieważ miejsce deklaracji bloku pokrywa się z miejscem utworzenia jego egzemplarza, w nagłówku bloku prefiksowanego należy podać listę parametrów aktualnych, jeżeli tylko prefiks ma jakieś parametry. Na przykład, jeżeli klasa **geometria** ma parametr **liczba_obiektów** typu **integer**, to nagłówek bloku może mieć następującą postać:

```
pref geometria(100) block;
```

Skutkiem prefiksowania bloku klasą **geometria** jest udostępnienie modułowi prefiksowanemu (blokowi) atrybutów zdefiniowanych w prefiksie (klasie **geometria**). Lista instrukcji bloku prefiksowanego pokrywa się z listą instrukcji bloku, gdyż w klasie **geometria** nie ma żadnych instrukcji. Po wczytaniu danych są tworzone trzy okręgi o środkach w wierzchołkach trójkąta i promieniu nie mniejszym niż długość każdego z boków. Każda z par tak zdefiniowanych okręgów przecina się w dwóch różnych punktach definiujących prostą. Wystarczy teraz znaleźć dwie takie proste (symetralne dwóch boków trójkąta). Ich przecięcie wyznacza środek okręgu opisanego na trójkącie.

*Rozwiązanie* 5.2.4

```
unit geometria: class;
    unit Punkty: class(x, y: real);
        unit równy: function(q: Punkty): boolean;
            begin
            if q=/=none then result:=x=q.x and y=q.y fi
            end równy;
        unit odległość: function(q: Punkty): real;
            begin
            if q=/=none then
              result := sqrt( (x-q.x)*(x-q.x) +
                              (y-q.y)*(y-q.y)) fi
            end odległość;
        end Punkty;
    unit Proste: class(A,B,C: real);
        unit równoległa: function(l: Proste): boolean;
            begin
            if l=/=none then result:=(A*l.B-B*l.A)=0 fi
            end równoległa;
        unit przecięcie: function(l: Proste): Punkty;
            var t: real;
            begin
            if l=/=none and_if not równoległa(l) then
```

```
            t:=1/(B*l.A-A*l.B);
            result:=new Punkty(-t*(B*l.C-C*l.B),
                               t*(A*l.C-C*l.A))
            fi
        end przecięcie;
    begin block var d: real; (* zmienna pomocnicza *)
            begin
            d:=sqrt(A*A+B*B);
            if d=0 then
              writeln(" nie ma takiej prostej")
            else d:=1/d; A:=A*d; B:=B*d; C:=C*d fi
            end
    end Proste;
unit Okręgi: class(p: Punkty, r: real);
    unit przecięcie: function(o: Okręgi): Proste;
        var r1, r2, A, B: real;
        begin
        if o=/=none then
          r1 := r*r - p.x*p.x - p.y*p.y;
          r2 := o.r*o.r - o.p.x*o.p.x - o.p.y*o.p.y;
          A := p.x - o.p.x; B := p.y - o.p.y;
          if sqrt(A*A+B*B)=/=0 then
            result := new Proste(A, B, (r1-r2)/2)
            fi
          fi
    end przecięcie;
    begin
    if p=none or r<=0 then
      writeln(" nie ma takiego okręgu") fi
    end Okręgi;
unit Trójkąty: class;
    var w1, w2, w3: Punkty;
    begin
    block var x,y: real;
        begin
        read(x,y); w1:=new Punkty(x,y);
        read(x,y); w2:=new Punkty(x,y);
        read(x,y); w3:=new Punkty(x,y);
        if w1.równe(w2) or_if w1.równe(w3)
            or_if w2.równe(w3) then
          writeln(" nie ma takiego trójkąta") fi
        end
    end Trójkąty;
```

```
    end geometria;
begin
pref geometria block
    var środek: Punkty,
        x,y,promień: real,
        o1,o2,o3: Okręgi,
        l1,l2: Proste;

    begin

    t:=new Trójkąty;
    promień:=t.w1.odległość(t.w2)+t.w2.odległość(t.w3);
    o1:=new okrąg(t.w1,promień);
    o2:=new okrąg(t.w2,promień);
    o3:=new okrąg(t.w3,promień);
    l1 := o1.przecięcie(o2);
    l2 := o2.przecięcie(o3);
    środek := l1.przecięcie(l2);
    promień:=środek.odległość(t.w1);
    writeln(" środek okręgu = (",
          środek.x, ",", środek.y, ")");
    writeln(" promień = ", promień)
    end
end;
```

## 5.3 Sortowanie stogowe — łączenie instrukcji w modułach prefiksowanych

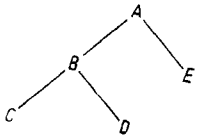
### 5.3.1 *Sformułowanie problemu*

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

### 5.3.2 *Dyskusja zadania*

Z sortowaniem tablic mieliśmy już do czynienia wcześniej, w p. 4.3, gdzie podaliśmy bardzo prosty algorytm sortowania. Teraz przedstawimy algorytm znacznie ciekawszy i bardziej efektywny, znany pod nazwą sortowania stogowego. W algorytmie tym podstawową strukturą danych jest stóg. Przypomnijmy, że stogiem nazywamy wyważone drzewo bi-

narne, którego każde poddrzewo spełnia następujący warunek: wartość elementu związanego z korzeniem jest większa niż wartość elementów znajdujących się w poddrzewach.
Przykład stogu pokazano na rys. 5.1.




RYS. 5.1 Przykład stogu

Aby posortować elementy ułożone w stóg, wystarczy wziąć jego korzeń jako element największy, usunąć go ze stogu wstawiając na koniec sortowanego ciągu, a następnie poprawić uzyskaną strukturę tak, aby znów była stogiem. Postępowanie to kontynuujemy dopóty, dopóki stóg nie jest pusty. Poprawianie struktury polega na uzupełnieniu brakującego elementu w korzeniu stogu. Zauważmy, że jedynymi kandydatami są elementy związane z korzeniem prawego i lewego poddrzewa, gdyż wszystkie inne elementy są od niech mniejsze. Z korzeniem stogu związujemy większy z tych dwóch elementów. Teraz brakuje elementu w korzeniu jednego z poddrzew. Jego uzupełnienia dokonujemy tą samą metodą. Postępowanie kontynuujemy dopóty, dopóki usunięty korzeń nie stanie się liściem.

### *Omówienie rozwiązania* 5.3.3

Stóg można oczywiście reprezentować za pomocą rekurencyjnego typu danych. Jednak, ze względu na oszczędność pamięci, lepsze będzie zakodowanie struktury drzewa w tablicy. Korzeń drzewa będzie zapamiętywany jako pierwszy element tablicy, kolejne jej elementy powiąże następująca zależność: lewy syn i-tego elementu tablicy jest zapamiętany jako element o indeksie 2i, zaś prawy syn — jako element o indeksie 2i+1. Na przykład pierwszym elementem tablicy reprezentującej drzewo przedstawione na rys. 5.3 jest 20, następnymi zaś jej elementami są: 11, 9, 7, 6, 3, 8.

Aby zrealizować operację poprawiania struktury stogu, użyjemy pewnego pomocniczego algorytmu, który umożliwia dołączenie do stogu nowego elementu. Załóżmy, że jest dany stóg zapamiętany w tablicy a na miejscach l+1, ..., p oraz że chcemy dołączyć nowy obiekt x tak, aby nowy stóg pamiętany był na miejscach l, ..., p. Element x wstawiamy do korzenia stogu, a następnie „przesiewamy" go przez wierzchołki mniejszych elementów stogu.

```
i:= l; j:= 2*i; x:= a(i);
while j<=p do
    if j<p and_if a(j)<a(j+1) then j:= j+1 fi;
    if x>=a(j) then exit fi;
    a(i):= a(j); i:= j; j:= 2*i
    od;
a(i):= x;
```

Powyższy fragment algorytmu nazwiemy **przesiewanie**. Zauważmy, że teraz można użyć tego fragmentu do konstrukcji algorytmu sortowania, gdy jest już dany stóg

```
p:= upper(a);
do x:=a(lower(a)); a(lower(a)):=a(p); a(p):=x; p:=p-1;
  if p=lower(A) then exit fi;
  ... (* teraz przesiewanie *)
  od;
```

Algorytm **przesiewanie** może być także zastosowany do budowy stogu. Elementy a(upper(a) div 2),..., a(upper(a)) są liśćmi stogu. Teraz w każdym kroku dołączamy do stogu elementy a(upper(a) div 2 - 1),..., a(1), ustawiając je na właściwych miejscach dzięki przesiewaniu

```
l:= (upper(a) div 2) + 1;
do l:= l-1;
  if l=lower(A) then exit fi;
  ... (* teraz przesiewanie *)
  od;
```

Obie fazy algorytmu zapiszemy w postaci procedur. Mają one zbliżoną strukturę, toteż wspólną ich część wydzielimy jako klasę **przesiewanie**. Klasa ta będzie prefiksować obie procedury.

```
unit przesiewanie: class; ... end przesiewanie;
unit twórz_stóg: przesiewanie procedure; ...
    end twórz_stóg;
unit sortuj: przesiewanie procedure; ... end sortuj;
```

Taką konstrukcję umożliwi nam łączenie list instrukcji w modułach prefiksowanych przy zastosowaniu instrukcji **inner**. Zauważmy, iż przesiewanie występuje w dwóch różnych kontekstach. Oba mają strukturę pętli **do** z innym warunkiem jej zakończenia (p=lower(a) oraz l=upper(a)). Aby móc wydzielić wspólną część algorytmu, wprowadzimy zmienną pomocniczą **koniec** typu **boolean** wskazującą, czy pętla ma się jeszcze wykonywać.

```
do
  [ koniec:=l=lower(a);l:=l-1 ]
  albo, zależnie czy tworzymy stóg, czy sortujemy:
  [ koniec:=p=lower(a); x:=a(lower(a));
    a(lower(a)):=a(p); a(p):=x; p:=p-1 ];
  if koniec then exit fi;
  ...(* teraz przesiewanie *)
  od;
```

Jeśli wspólny schemat algorytmu włączymy do algorytmu przesiewanie, przyjmie on następującą postać:

```
do ... (* miejsce na instrukcje zależne od algorytmu *)
  if koniec then exit fi;
  ... (* teraz dawny algorytm przesiewanie *)
  od;
```

Zauważmy, iż algorytm ten można zaprogramować jako instrukcje pewnej klasy przesiewanie, zastępując „miejsce na instrukcje zależne od algorytmu" słowem kluczowym inner. Jeśli taką klasą będziemy prefiksować dwa inne moduły (np. procedury), w miejscu inner zostaną wykonane odpowiednie instrukcje tych modułów, zgodnie ze sposobem łączenia list instrukcji przyjętym w Loglanie (por. p. 5.1). Algorytmy tworzenia stogu i sortowania przyjmą więc następującą postać:

```
unit twórz_stóg: przesiewanie procedure;
    begin koniec:= l=upper(a); l:=l-1 end twórz_stóg;
unit sortuj: przesiewanie procedure;
    begin
    koniec:= p=lower(a); x:=a(lower(a));
    a(lower(a)):=a(p); a(p):=x; p:=p-1
    end sortuj;
```

Inicjalizację zmiennych l oraz p (l:=upper(a) div 2 +1 oraz p:= upper(a)) można wykonać jednokrotnie na początku działania algorytmu.

Cały algorytm sortowania przyjmie teraz postać procedury

```
unit sortowanie_stogowe: procedure(a: array_of integer);
    ...
    begin
    l:=upper(a) div 2 +1; p:= upper(a);
    call twórz_stóg;
    call sortuj
end sortowanie_stogowe;
```

*Rozwiązanie* 5.3.4

```
unit sortowanie_stogowe: procedure(a: array_of integer);
    var i, j, l, p, x: integer;
    unit przesiewanie: class;
        var koniec: boolean;
        begin
        do inner;
          if koniec then exit fi;
          i:= l;  j:= 2*i;  x:= a(i);
          while j<=p do
               if j<p and_if a(j)<a(j+1) then
                  j:= j+1 fi;
               if x>=a(j) then exit fi;
               a(i):= a(j);  i:= j;  j:= 2*i
               od;
          a(i):= x
          od
        end przesiewanie;
    unit twórz_stóg: przesiewanie procedure;
        begin
        koniec:= l=lower(a); l:= l-1
        end twórz_stóg;
    unit sortuj: przesiewanie procedure;
        begin
        koniec:= p=lower(a); x:= a(lower(a));
        a(lower(a)):=a(p); a(p):=x; p:=p-1
        end sortuj;
    begin
    l:= upper(a) div 2 + 1;  p:= upper(a);
    call twórz_stóg;
    call sortuj
    end sortowanie_stogowe;
```

## Dalsze wiadomości o prefiksowaniu 5.4

Podstawowym zastosowaniem prefiksowania jest rozszerzanie typów danych o nowe atrybuty. Zauważmy, że w przypadku prefiksowania klas klasa prefiksowana ma własności klasy prefiksującej wzbogacone o dodatkowe własności zdefiniowane w rozszerzeniu. Typ danych definiowany klasą prefiksowaną jest zatem *podtypem* typu danych definiowanego klasą

prefiksującą. Narzucając bowiem dodatkowe własności, zawężamy zbiór elementów mających te własności. Mówimy też, że klasa B jest *podklasą* klasy A.

Ponieważ klasę prefiksowaną można znów użyć jako prefiks, użytkownik jest w stanie tworzyć *hierarchię klas* przez ich stopniowe rozszerzanie. Niech $C_1, C_2, \ldots, C_n$ będą klasami o tej własności, że $C_1$ nie ma prefiksu oraz dla k=2,...,n $C_k$ ma jako prefiks $C_{k-1}$. Wówczas $C_1, \ldots, C_k$ nazywamy *ciągiem prefiksowym* klasy $C_k$ oraz dla i>j $C_i$ nazywać będziemy *podklasą* $C_j$. Zwróćmy przy tym uwagę na ważne ograniczenie związane z prefiksowaniem — klasa nie może pojawić się w swoim ciągu prefiksowym więcej niż raz. Traktując klasy jako definicje typów danych, typy danych zaś jako zbiory elementów należących do typu, uzyskujemy następujące zawierania się zbiorów:

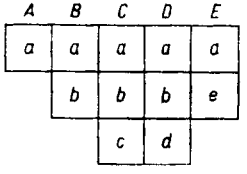
$$C_1 \supseteq C_2 \supseteq \ldots \supseteq C_k$$

Inaczej mówiąc, każdy obiekt klasy $C_i$ należy do typu definiowanego klasą $C_i$ oraz do wszystkich typów klasowych, dla których $C_i$ jest podklasą, a zatem $C_1, C_2, \ldots, C_{i-1}$.

Rozważmy następującą hierarchię klas:

```
unit A: class; ... end A;
unit B:A class; ... end B;
unit C:B class; ... end C;
unit D:B class; ... end D;
unit E:A class; ... end E;
```

Hierarchię tę można przedstawić jako drzewo, w którym x jest ojcem y, jeśli x prefiksuje y (por. rys. 5.1). Przyjmijmy, że małe litery a, b, c, d, e reprezentują atrybuty bezpośrednio zadeklarowane w klasach A, B, C, D, E. Strukturę obiektów odpowiednich klas ilustruje rys. 5.2.




RYS 5.2 Struktura obiektów przykładowej hierarchii typów danych

Wartością zmiennej typu klasowego A może być wskaźnik do dowolnego obiektu należącego do typu danych definiowanego klasą A. Obiekt

ten może być zatem nie tylko obiektem klasy A, lecz także obiektem dowolnej jej podklasy. Rozważmy instrukcję x:=w, gdzie x jest zmienną typu klasowego A. Instrukcja ta jest poprawna wówczas, gdy wartością wyrażenia w jest wskaźnik do obiektu klasy B, a A należy do ciągu prefiksowego B.

W niektórych zastosowaniach istotna jest znajomość dokładnego typu obiektu wskazywanego przez daną zmienną. Służą temu relacje in oraz is. Wyrażenie

```
x in A
```

ma wartość true wtedy i tylko wtedy, gdy zmienna x wskazuje obiekt należący do typu A (czyli obiekt klasy A lub jej podklasy). Wyrażenie

```
x is A
```

ma wartość true jedynie wówczas, gdy wartością zmiennej x jest wskaźnik do obiektu klasy A.

Wiemy już (por. p. 3.2.3), że do atrybutów obiektu można odwołać się spoza jego treści używając odległego dostępu. Podaje się przy tym oprócz nazwy atrybutu również zmienną wskazującą obiekt, o którego atrybut chodzi, np. x.atr. Aby takie odwołanie było poprawne, atrybut atr musi być zdefiniowany w ramach typu zmiennej x. Rozważmy sytuację, w której zmienna x typu A wskazuje na obiekt typu B, gdy B jest podklasą A. Przypuśćmy, że atrybut atr został zdefiniowany w klasie B. Zatem w klasie A nie jest on określony i odwołanie x.atr nie jest poprawne, mimo że obiekt wskazywany przez x ma taki atrybut. Aby umożliwić podobne odwołanie do atrybutu, została w Loglanie wprowadzona operacja qua chwilowej zmiany typu zmiennej wskazującej obiekt klasy. Wyrażenie

```
x qua B.atr
```

jest poprawne, gdyż typ zmiennej x został w zakresie tego wyrażenia zmieniony na B, a atrybut atr jest w klasie B określony. Taka zmiana typu zmiennej może wystąpić jedynie w wyrażeniach definiujących dostęp do atrybutów (przed kropką) i jej zakres ogranicza się do wyrażenia po kropce. Jeżeli x jest zmienną typu A, to wyrażenie x qua B jest poprawne w dwóch przypadkach: jeśli B jest podklasą klasy A lub jeżeli A jest podklasą klasy B. Zastosowanie zmiany typu zmiennej w pierwszym przypadku omówiliśmy powyżej. Zmianę typu zmiennej w drugim przypadku stosuje się wtedy, gdy chcemy odzyskać dostęp do atrybutu, który został zasłonięty przez ponowną deklarację w podklasie. Niech A będzie podklasą klasy B oraz niech atrybut atr będzie zadeklarowany zarówno w A, jak i w B. Niech zmienna x typu A wskazuje na obiekt klasy A. Wówczas

wyrażenie x.atr udostępnia atrybut zdefiniowany w klasie A, wyrażenie zaś x qua B.atr — zasłonięty atrybut zdefiniowany w klasie B.

Zauważmy, że użycie operatora qua umożliwia odwołanie do niedostępnego bezpośrednio atrybutu jedynie spoza zawierającego go obiektu. Analogicznym operatorem umożliwiającym dostęp do zasłoniętej deklaracji z jej środowiska jest operator this omawiany już w p. 3.3.3. Wyrażenie

```
this A.atr
```

jest poprawne, jeżeli istnieje moduł syntaktycznie otaczający miejsce wystąpienia tego wyrażenia, który w swoim ciągu prefiksowym ma moduł A zawierający atrybut atr. Wartością tego wyrażenia jest atrybut atr najbliższego egzemparza takiego modułu. Rozważmy np. deklaracje

```
unit A: class;
    var x: integer; ... end A;
```

oraz

```
unit B: A class;
    var x: real;
    begin  ... x  ... this A.x ...  end B;
```

Bezpośrednie odwołanie do atrybutu x w treści klasy B dotyczy zmiennej rzeczywistej zadeklarowanej w klasie B. Wyrażenie this A.x udostępnia zasłoniętą zmienną typu integer zdefiniowaną w klasie A.

Wyrażenie this A może występować samodzielnie, nie tylko w kontekście odległego dostępu do atrybutu. Jest ono poprawne, jeśli miejsce jego wystąpienia otacza syntaktycznie moduł, w którego ciągu prefiksowym występuje moduł A. Wartością tego wyrażenia jest wówczas wskaźnik do najbliższego egzemplarza takiego modułu. Pewnym odstępstwem od tej reguły są wyrażenia this coroutine oraz this process. Wyjątki te omawiamy w rozdz. 6 oraz 8.

Dzięki prefiksowaniu można budować hierarchie typów danych stopniowo wzbogacając je o nowe atrybuty. W szczególności hierarchia ta może dotyczyć stopnia określoności pewnego typu danych: w prefiksie nie wszystkie własności są znane, w module prefiksowanym programista precyzuje interesujące go własności. W ten sposób prefiks definiuje pewien abstrakcyjny typ danych, a moduł prefiksowany — pewną jego implementację. Zastanówmy się, czy jest możliwe podobne postępowanie w przypadku algorytmów, tzn. czy można zdefiniować abstrakcyjny algorytm używający nie zdefiniowanych do końca operacji, które następnie mogłyby być sprecyzowane (być może na wiele sposobów). Operacje są zwykle realizowane jako procedury lub funkcje. Przypuśćmy, że w klasie

A mamy zdefiniować abstrakcyjny algorytm korzystający z pewnej operacji p (zrealizowanej np. w postaci procedury), która w klasie A nie jest do końca sprecyzowana.

```
(1) unit A: class;
        unit p: procedure; ... end p;
        begin (* abstracyjny algorytm używający p *)
        ... call p; ...
        end A;
```

Przypuśćmy również, że operacja ta ma być sprecyzowana w podklasie B klasy A:

```
(2) unit B: A class;
        unit p: procedure; ... end p;
        begin ... end B;
```

Nastąpiło zasłonięcie poprzedniej deklaracji, zatem atrybutem obiektu klasy B jest procedura p o treści (2). Wszelkie odwołania do p w treści klasy B będą więc dotyczyły tej właściwej, w pełni zdefiniowanej procedury. Niestety, zgodnie z semantyką prefiksowania, instrukcje klasy A wykonują się w środowisku klasy A, zatem wywołanie call p w treści klasy A dotyczy procedury w niej zdefiniowanej, czyli (1). Naszym zamierzeniem było natomiast, by również w treści algorytmu zdefiniowanego w klasie A została wykorzystana w pełni zdefiniowana procedura p, czyli (2). Można to osiągnąć dzięki *podprogramom wirtualnym*. Syntaktycznie różnią się one od zwykłych podprogramów jedynie słowem **virtual** następującym po słowie **unit**, np.

```
unit virtual p: procedure;
```

Główną cechą podprogramów wirtualnych jest to, że deklaracja takiego podprogramu nie zasłania deklaracji z prefiksu, ale zastępuje ją we wszystkich do niej odwołaniach. Na przykład załóżmy, że w powyższym przykładzie procedura p została zadeklarowana w obu klasach A i B jako wirtualna. Zatem deklaracja z klasy B zastępuje deklarację z klasy A we wszystkich odwołaniach do procedury p, również z treści klasy A. Zauważmy, że w treści procedury wirtualnej p zdefiniowanej w klasie B mogą być wykorzystywane atrybuty zdefiniowane w klasie B, zastąpienie takie umożliwia zatem dostęp z treści klasy A (przez wywołanie call p) do wielkości zdefiniowanych w jej podklasie.

Podprogramy wirtualne mogą zatem realizować następujące zadania:

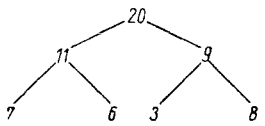
— stopniowe precyzowanie procedur i funkcji w kolejnych modułach ciągu prefiksowego;

— dostęp do atrybutów danego modułu z klasy występującej w jego ciągu prefiksowym;

— tworzenie ogólnych algorytmów i systemów, których szczegóły precyzuje użytkownik na poziomie swojego programu.

W celu zilustrowania podprogramów wirtualnych powróćmy do hierarchii klas zdefiniowanej uprzednio i przedstawionej na rys. 5.3. Załóżmy, że w każdej z klas został zadeklarowany podprogram wirtualny

```
unit virtual p: procedure;...end p;
```




RYS. 5.3 Przykładowa hierarchia klas

Oznaczmy przez IA (odpowiednio IB,...,IE) treść procedury p zadeklarowanej w klasie A (odpowiednio B,...,E). Wówczas wszelkie odwołania do p w obiekcie będącym egzemplarzem klasy A (klasy B,...,E) spowodują wykonanie instrukcji IA (odpowiednio IB,...,IE).

Aby zastąpienie podprogramów wirtualnych było możliwe, muszą one spełniać następujące warunki:

(1) muszą być tego samego rodzaju (albo procedury, albo funkcje);

(2) ich nagłówki muszą umożliwiać zastąpienie, np. być identyczne (dokładne reguły podane są w Dodatku C);

(3) muszą należeć do jednego ciągu podprogramów wirtualnych o tej samej nazwie, spełniających warunki (1) i (2), zadeklarowanych w następujących po sobie modułach spójnego fragmentu ciągu prefiksowego. Brak deklaracji podprogramu wirtualnego, ograniczenie dostępu specyfikacją hidden (por.p. 5.7.3) lub niespełnienie warunków (1) i (2) powoduje przerwanie ciągu podprogramów wirtualnych.

Rozważmy na przykład liniową hierarchię klas

```
unit A: class;
    unit virtual f: function(x: real): real; ... end f;
    ... end A;
unit B: A class;
    unit virtual f: function(x: real): real; ... end f;
    ... end B;
```

```
unit C: B class;
    unit f: function(x: real): real; ... end f;
      (* zwykła funkcja *)
    ... end C;
unit D: C class;
    unit virtual f: function(x: real): real; ... end f;
    ... end D;
unit E: D class;
    unit virtual f: function(x: real): real; ... end f;
    ... end E;
```

Zadeklarowanie w klasie C funkcji f jako zwykłej funkcji (nie wirtualnej) spowodowało utworzenie dwóch ciągów funkcji wirtualnych obejmujących klasy A i B oraz D i E. Podczas tworzenia obiektu klasy E odwołania do funkcji f z treści klas A i B dotyczą definicji z klasy B, odwołania zaś z treści klas D i E definicji z klasy E. Odwołania z treści klasy C dotyczą zwykłej funkcji zdefiniowanej w tej klasie.

Operacja prefiksowania przenosi się także w sposób naturalny na przypadek współprogramów i procesów. Tematyce tej poświęcimy więcej miejsca w dalszych rozdziałach książki.

## Sumowanie typów — operatory in, is, qua, this 5.5

### *Sformułowanie problemu* 5.5.1

Zaprojektuj następujące typy danych: człowiek, dorosły, dziecko, kobieta i mężczyzna tak, aby były spełnione następujące zależności:

człowiek = dorosły ∪ kobieta ∪ mężczyzna ∪ dziecko
dorosły = kobieta ∪ mężczyzna

gdzie ∪ oznacza sumę zbiorów obiektów.

### *Dyskusja zadania* 5.5.2

Tego typu problem pojawia się często przy projektowaniu baz danych. Mechanizm prefiksowania, jak już wcześniej zwróciliśmy na to uwagę, daje duże możliwości definiowania hierarchii typów danych. W naszym przypadku hierarchię taką można utworzyć następująco:

```
unit człowiek: class; ... end człowiek;
unit dorosły: człowiek class; ... end dorosły;
```

```
unit dziecko: człowiek class; ... end dziecko;
unit kobieta: dorosły class; ... end kobieta;
unit mężczyzna: dorosły class; ... end mężczyzna;
```

Powyższe definicje zapewniają następujące zawierania zbiorów obiektów:

człowiek $\supseteq$ dorosły $\cup$ kobieta $\cup$ mężczyzna $\cup$ dziecko
dorosły $\supseteq$ kobieta $\cup$ mężczyzna

nie zapewniają natomiast ich równości. Można bowiem dalej (celowo albo przez pomyłkę) rozszerzać powstałą hierarchię pojęć, np.

```
unit kot: człowiek class; ... end kot;
```

Po takim rozszerzeniu zbiór obiektów typu człowiek będzie zawierać także obiekty typu kot.

W Loglanie nie możemy syntaktycznie ograniczyć zakresu prefiksowania tak, aby uniemożliwić podobny skutek. Możemy jednak uniknąć tworzenia obiektów niepożądanych typów (np. typu kot), sprawdzając poprawność ich tworzenia w czasie wykonywania programu.

### *Omówienie rozwiązania* 5.5.3

Do rozwiązania zadania zastosujemy operatory this oraz is, uzupełniając definicje typów człowiek i dorosły odpowiednio o instrukcje

```
(1) if not( this człowiek is dorosły
          or this człowiek is dziecko
          or this człowiek is kobieta
          or this człowiek is mężczyzna) then
        writeln("błąd --- to nie człowiek");call endrun fi;
(2) if not( this dorosły is kobieta
          or this dorosły is mężczyzna) then
        writeln("błąd --- to nie dorosły"); call endrun fi;
```

Teraz przy tworzeniu obiektu typów człowiek i dorosły najpierw wykona się odpowiednie sprawdzenie, bowiem this człowiek (this dorosły) wskazuje na aktualnie tworzony obiekt typu człowiek (dorosły), przy użyciu zaś operatora is sprawdzamy, czy ten obiekt jest typu dorosły, dziecko, kobieta lub mężczyzna (kobieta lub mężczyzna — w przypadku typu dorosły). Gdyby tworzony był np. obiekt typu kot, wówczas zostałaby wypisana informacja o błędzie i program zakończyłby działanie.

Po dołączeniu instrukcji (2) do klasy dorosły możemy uprościć instrukcję (1) używając operatora in. Ponieważ typ dorosły jest teraz dokładnie sumą typów kobieta i mężczyzna, warunek

```
this człowiek is dorosły or this człowiek is kobieta or
  this człowiek is mężczyzna
```

jest równoważny warunkowi

```
this człowiek in dorosły.
```

Warunek instrukcji (1) może więc przyjąć następującą postać:

```
not(this człowiek in dorosły or this człowiek is dziecko).
```

Przy okazji omawiania powyższego przykładu zwróćmy jeszcze uwagę na zastosowanie operacji qua, niezbędnej przy bardziej zaawansowanym używaniu hierarchii typów danych. Przypuśćmy, że klasy występujące w zdefiniowanej hierarchii uzupełnimy o takie oto atrybuty:

```
unit człowiek: class;
    var imię, nazwisko: string, wiek: integer;
    end człowiek;
unit dziecko: człowiek class; end dziecko;
unit dorosły: człowiek class;
    var liczba_dzieci: integer; end dorosły;
unit kobieta: dorosły class; end kobieta;
unit mężczyzna: dorosły class;
    var kategoria_zdrowia: character; end mężczyzna;
```

Wyobraźmy sobie, że utworzyliśmy bazę danych gromadzącą dane o obiektach powyższych typów. Przyjmijmy dla uproszczenia, iż baza ta jest dana w postaci tablicy

```
var baza: array_of człowiek.
```

Załóżmy, że chcemy otrzymać dane o osobach dorosłych mających co najmniej jedno dziecko. Zadanie to możemy zapisać w postaci pętli

```
for i:= lower(baza) to upper(baza) do
    if baza(i) in dorosły then
      if baza(i) qua dorosły.liczba_dzieci>0 then
        writeln(baza(i).imię, baza(i).nazwisko,
          baza(i) qua dorosły.liczba_dzieci)
        fi
      fi
    od;
```

Zauważmy, iż badając warunek

baza(i) qua dorosły.liczba_dzieci>0

musieliśmy skorzystać z operatora qua chwilowej zmiany typu obiektu. Odwołanie baza(i).liczba_dzieci byłoby niepoprawne, ponieważ elementy tablicy baza mają typ człowiek, dla którego atrybut liczba_dzieci nie został zdefiniowany. Podobna sytuacja miała miejsce również w instrukcji wypisywania wartości atrybutu liczba_dzieci. Pozostałe atrybuty (imię, nazwisko) można było wypisać bez zmiany typu obiektu, ponieważ dla typu człowiek atrybuty te zostały zdefiniowane.

*Rozwiązanie* 5.5.4

```
unit człowiek: class;
    var imię, nazwisko: string, wiek: integer;
    begin
    if not(this człowiek in dorosły
          or this człowiek is dziecko) then
      writeln(" błąd --- to nie człowiek");call endrun
      fi;
    end człowiek;
unit dziecko: człowiek class; end dziecko;
unit dorosły: człowiek class;
    var liczba_dzieci: integer;
    begin
    if not( this dorosły is kobieta
          or this dorosły is mężczyzna) then
      writeln(" błąd --- to nie dorosły"); call endrun
      fi
    end dorosły;
unit kobieta: dorosły class; end kobieta;
unit mężczyzna: dorosły class;
    var kategoria_zdrowia: character; end mężczyzna;
```

## Zbiory rozmyte — użycie podprogramów wirtualnych 5.6

*Sformułowanie problemu* 5.6.1

Opracuj implementację struktury danych odpowiadającej zbiorom rozmytym. Dla ustalonego typu elementów E strukturę tę definiujemy następująco:

zbiory_rozmyte(E) = (*zbiory_rozmyte*; element, suma, iloczyn, dopełnienie)

gdzie

(1) nośnikiem typu jest zbiór wszystkich podzbiorów $E$;
(2) element: $E \times$ *zbiory_rozmyte* $\rightarrow [0,1]$;
(3) suma, iloczyn: *zbiory_rozmyte* $\times$ *zbiory_rozmyte* $\rightarrow$ *zbiory_rozmyte*;
(4) dopełnienie: *zbiory_rozmyte* $\rightarrow$ *zbiory_rozmyte*.

Z intuicyjnego punktu widzenia każdy zbiór rozmyty jest dany przez funkcję element, której znaczenie jest następujące: prawdopodobieństwo że element e należy do zbioru s jest równe wartości element(e,s). Operacje na zbiorach rozmytych definiujemy za pomocą funkcji element następująco:

element(e, suma(s,t)) = imax(element(e,s), element(e,t));
element(e, iloczyn(s,t)) = imin(element(e,s), element(e,t));
element(e, dopełnienie(s)) = 1 - element(e,s).

Warto zauważyć, że te operacje można definiować na wiele innych, istotnie różniących się sposobów. Jeśli np. prawdopodobieństwa należenia elementów do zbiorów są niezależne, można przyjąć następującą ich definicję:

element(e, iloczyn(s,t)) = element(e,s) * element(e,t);
element(e, suma(s,t)) = element(e,s) + element(e,t) -
element(e, iloczyn(s,t));
element(e, dopełnienie(s)) = 1 - element(e,s).

Zauważmy ponadto, że pojęcie zbioru rozmytego jest naturalnym rozszerzeniem klasycznego pojęcia zbioru, przyjmując bowiem, iż jedynymi wartościami funkcji element są liczby 0 oraz 1, uzyskujemy odpowiednik klasycznej definicji należenia elementu do zbioru oraz operacji na zbiorach.

## *Dyskusja zadania* 5.6.2

Jak już zauważyliśmy w trakcie formułowania problemu, każdy zbiór rozmyty jest wyznaczony przez funkcję element. Jest więc ona najważniejszym (i jedynym niezbędnym) atrybutem każdego zbioru rozmytego. Przeformułujmy nieco definicje operacji suma, iloczyn i dopełnienie:

— suma zbiorów rozmytych s i t jest zbiorem rozmytym, którego funkcja element określona jest jako imax(element(e,s), element(e,t))

— iloczyn zbiorów rozmytych s i t jest zbiorem rozmytym, którego funkcja **element** określona jest jako imin(element(e,s), element(e,t))

— dopełnienie zbioru rozmytego **s** jest zbiorem rozmytym, którego funkcja **element** określona jest jako 1 - element(e,t).

Dla każdego konkretnego zbioru rozmytego definiowanego bezpośrednio przez programistę trzeba podać funkcję **element**. Programista musi więc mieć możliwość zdefiniowania tej funkcji na poziomie swojego programu. Z drugiej strony powyższe definicje operacji na zbiorach wskazują, iż mamy do czynienia z pewną hierarchią typów — zarówno iloczyn, jak i suma czy dopełnienie, jest pewnym zbiorem rozmytym. Możemy więc spojrzeć na nie jak na podtypy typu reprezentującego zbiory rozmyte. Dlatego też najwygodniejszą metodą rozwiązania naszego zadania będzie użycie prefiksowania (umożliwiającego tworzenie hierarchii typów) oraz podprogramów wirtualnych (umożliwiających ich precyzowanie wewnątrz modułów prefiksowanych).

Oczywiście rozwiązanie takie, ze względu na użycie mechanizmu prefiksowania, w sposób automatyczny umożliwi użytkownikowi definiowanie operacji **suma**, **iloczyn** i **dopełnienie**.

### *Omówienie rozwiązania* 5.6.3

Zbiór rozmyty zdefiniujemy jako klasę:

```
unit zbiór_rozmyty: class;
    unit virtual element: function(e: E): real;
        end element;
    end zbiór_rozmyty;
```

Przykładowa operacja sumowania zbiorów rozmytych jest definiowana następująco:

```
unit suma: zbiór_rozmyty class(s,t: zbiór_rozmyty);
    unit virtual element: function(e: E): real;
        begin result:=imax(s element(e),t.element(e))
        end element;
    end suma;
```

W tej definicji klasa **suma** jest podklasą klasy **zbiór_rozmyty**, dziedziczy więc jej atrybut **element**. Atrybut ten jest w klasie **suma** precyzowany i rzeczywiście odpowiada funkcji **element** właściwej dla sumy zbiorów s i t. Kontynuując rozważania, jako implementację struktury danych **zbiory_rozmyte** uzyskamy klasę o następującym schemacie:

```
unit zbiory_rozmyte: class(type E);
    unit zbiór_rozmyty: class;
        unit virtual element: function(e: E): real;
            end element;
        end zbiór_rozmyty;
    unit suma: zbiór_rozmyty class ... end suma;
    unit iloczyn: zbiór_rozmyty class ... end iloczyn;
    unit dopełnienie: zbiór_rozmyty class ...
        end dopełnienie;
    end zbiory_rozmyte;
```

Rozważmy pewne zastosowanie tak zdefiniowanej klasy.

```
block
    unit elementy: class(a,b,c: real); end elementy;
    pref zbiory_rozmyte(elementy) block
        (* wewnątrz tego bloku są dostępne atrybuty
          zbiorów rozmytych o elementach typu
          elementy *)
        unit S: zbiór_rozmyty class;
            unit virtual element: function
                                (e: elementy): real;
                begin
                result:=e.a+e.b*(e.a-e.c);
                if result<0 or result>1 then
                  result:=0 fi
                end element;
            end S;
        var P,Q,R,T: zbiór_rozmyty; var e:elementy;
        begin
        P:=new S; Q:=new S;
        R:=new iloczyn(P, new dopełnienie(Q));
        T:=new suma(P, new suma(Q,R));
        e:=new elementy(1.0,0.1,3.0);
        write(P.element(e),Q.element(e),R.element(e),
          T.element(e):10:5)
        end;
```

Przeanalizujmy działanie tego fragmentu programu. W wyniku prefiksowania bloku klasą wszystkie wielkości zadeklarowane w klasie prefiksującej stają się bezpośrednio dostępne w bloku. Wykonanie takiego bloku polega na utworzeniu jego obiektu i wykonaniu jego listy instrukcji. Lista ta jest połączeniem listy instrukcji klasy prefiksującej (w naszym przypadku jest to lista pusta) i listy instrukcji bloku. Instrukcje klasy pre-

fiksującej są wykonywane w środowisku tej klasy. Dostępne są wielkości zadeklarowane w klasie i jej otoczeniu syntaktycznym. Instrukcje bloku wykonują się w połączonym środowisku bloku i klasy. W środowisku tym są określone wszystkie wielkości zdefiniowane w bloku i jego środowisku oraz te spośród dostępnych w klasie (zadeklarowanych w niej samej lub jej środowisku), które nie zostały zasłonięte ponowną deklaracją w bloku.

Wykonanie bloku rozpocznie się od wykonania instrukcji P:=new S oraz Q:=new S. Zmienne P i Q będą wskazywać dwa takie same zbiory rozmyte (określone tą samą funkcją element). Kolejne instrukcje spowodują powstanie bardziej złożonych zbiorów wskazywanych przez zmienne R oraz T, przy czym

```
R.element(e) = imin(P.element(e), 1 − Q.element(e)),
T.element(e) = imax(P.element(e), imax(Q.element(e), R.element(e)));
```

Po wykonaniu instrukcji zostaną zatem wypisane liczby:

```
0.8 0.8 0.2 0.8.
```

Gdyby użytkownik chciał zmienić definicję operacji iloczynu i sumy tak, jak to wspomnieliśmy w p. 5.6.1, może to uczynić w następujący sposób:

```
unit nowy_iloczyn: iloczyn class;
    unit virtual element: function(e:E): real;
        begin result:=s.element(e)*t.element(e)
        end element;
    end nowy_iloczyn;
unit nowa_suma: suma class;
    unit virtual element: function(e:E): real;
        begin
        result:=s.element(e)*t.element(e) -
                s.element(e)*t.element(e)
        end element;
    end suma;
```

Gdyby teraz te deklaracje umieścić wśród deklaracji poprzednio rozważanego bloku, jego zaś instrukcje zastąpić następującymi:

```
begin
 P:=new S; Q:= new S;
 R:=new nowy_iloczyn(P, new dopełnienie(Q));
 T:=new nowa_suma(P, new nowa_suma(Q,R));
 e:=new elementy(1.0, 0.1, 3.0);
 write(P.element(e), Q.element(e), R.element(e),
      T.element(e):10:5)
 end;
```

w wyniku jego działania zostałyby wypisane liczby

0.8 0.8 0.16 0.9664.

*Rozwiązanie* 5.6.4

```
unit zbiory_rozmyte: class(type E);
    unit zbiór_rozmyty: class;
        unit virtual element: function(e: E): real;
            end element;
    end zbiór_rozmyty;
    unit suma: zbiór_rozmyty class(s,t: zbiór_rozmyty);
        unit virtual element: function(e: E): real;
            begin result:=imax(s.element(e),t.element(e))
            end element;
        end suma;
    unit iloczyn: zbiór_rozmyty class(s,t: zbiór_rozmyty);
        unit virtual element: function(e: E): real;
            begin result:=imin(s.element(e),t.element(e))
            end element;
        end iloczyn;
    unit dopełnienie: zbiór_rozmyty class
          (s: zbiór_rozmyty);
        unit virtual element: function(e: E): real;
            begin result:=1-s.element(e) end element;
        end dopełnienie;
    end zbiory_rozmyte;
```

## Tablice rozproszone — ochrona atrybutów przy prefiksowaniu 5.7

*Sformułowanie problemu* 5.7.1

Zaimplementuj strukturę danych realizującą tablice rozproszone. Implementacja ma być niezależna od typu elementów umieszczanych w tablicy. Ma ona także umożliwiać użytkownikowi definiowanie funkcji rozpraszającej na poziomie jego programu. Dla danego typu elementów E typ **tablice_rozproszone** definiujemy następująco:

**tablice_rozproszone**(E) = (*tablice_rozproszone*; **wstaw**, **sprawdź**)

gdzie

(1) nośnikiem typu jest zbiór zbiorów elementów z $E$;
(2) **wstaw**: $E \times$ *tablice_rozproszone* $\rightarrow$ *tablice_rozproszone*;
(3) **sprawdź**: $E \times$ *tablice_rozproszone* $\rightarrow$ *boolean*.

Wynikiem funkcji **wstaw**(**e**,**s**) jest tablica reprezentująca sumę teoriomnogościową **s** oraz {**e**}. Relacja **sprawdź**(**e**,**s**) przyjmuje wartość **true** jedynie wówczas, gdy element **e** występuje w tablicy **s**.

### *Dyskusja zadania* 5.7.2

Implementacja tablic rozproszonych sprowadza się do rozwiązania następujących dwóch zadań:

(1) znajdź odpowiednią funkcję odwzorowującą zbiór E w zbiór indeksów tablicy (tzw. funkcję rozpraszającą);
(2) rozwiąż problem kolizji pojawiający się w przypadku, gdy funkcja, o której mowa w punkcie (1) nie jest różnowartościowa, a chcemy umieścić w tablicy dwa różne elementy, dla których wartość tej funkcji jest identyczna.

Zauważmy, że na ogół mamy do czynienia z sytuacją, w której moc zbioru E znacznie przewyższa moc zbioru indeksów tablicy. Aby zminimalizować średnią liczbę kolizji, należy tak dobrać funkcję rozpraszającą, aby moce jej przeciwobrazów jak najmniej różniły się od siebie. Oczywiście wybór takiej funkcji jest ściśle zależny od typu elementów, dlatego też użytkownik musi podawać własną funkcję rozpraszającą.

Najprostszym, a zarazem jednym z najbardziej efektywnych rozwiązań problemu kolizji jest umieszczanie elementów o identycznym adresie wyznaczonym przez funkcję rozpraszającą w kolejce związanej z tym adresem.

### *Omówienie rozwiązania* 5.7.3

Implementacja tablic rozproszonych powinna mieć jako parametr typ elementów, przyjmiemy więc, że jest on typem formalnym. Rozmiar tablicy jest także zależny od konkretnego zastosowania, toteż będzie on również parametrem formalnym. Realizacja tablic rozproszonych przyjmie zatem następującą postać:

```
unit tablice_rozproszone: class(type E; rozmiar:integer);
    var tablica: array_of ...;
    ...
    unit wstaw: procedure(e: E); ... end wstaw;
    unit sprawdź: function(e: E): boolean; ...
        end sprawdź;
    begin
    array tablica dim(0:rozmiar)
    end tablice_rozproszone;
```

Użytkownik może podawać funkcję rozpraszającą po zdefiniowaniu tej funkcji jako parametru formalnego, albo po przyjęciu, że jest ona atrybutem wirtualnym. Ponieważ to drugie rozwiązanie jest bardziej naturalne przy zastosowaniu techniki prefiksowania, przyjmiemy je w naszej implementacji. Wewnątrz klasy **tablice_rozproszone** pojawi się zatem następująca deklaracja:

```
unit virtual funkcja_rozpraszająca: function
                                (e: E): integer;
    end funkcja_rozpraszająca;
```

W celu rozwiązania problemu kolizji skorzystamy z definicji kolejek podanej w p. 4.4. Zauważmy przy tym, że zaprogramowanie operacji **sprawdź** będzie wymagać rozszerzenia struktury danych **kolejki** o operację sprawdzania, czy dany element występuje w kolejce. Rozszerzona struktura będzie zdefiniowana następująco:

**kolejki_spr**(E) = (*kolejki_spr*; **wstaw, usuń, pierwszy, spr**)

gdzie

(1) nośnik typu oraz operacje **wstaw**, **usuń**, **pierwszy** są określone jak dla typu kolejki (por. p. 4.4.1);
(2) **spr**: $E \times$ *kolejki_spr* $\rightarrow$ *boolean*.

Wynikiem **spr(e,q)** jest **true** wtedy i tylko wtedy, gdy element **e** występuje w kolejce **q**.

Zdefiniowanie operacji **spr** będzie wymagać od użytkownika podania funkcji

**równe**: $E \times E \rightarrow$ *boolean*

która sprawdza równość elementów ze zbioru $E$. Funkcja ta będzie również atrybutem wirtualnym klasy **tablice_rozproszone**.

Rozszerzenie typu danych **kolejki** uzyskamy przy pomocy prefiksowania.

```
unit kolejki_spr: kolejki class;
    ...
    unit spr: function(e: E): boolean;
        ...
        end spr;
    end kolejki_spr;
```

Zauważmy, że wewnątrz klasy **kolejki_spr** są widoczne wszystkie atrybuty klasy **kolejki**, podczas gdy do naszych celów jest niezbędna jedynie operacja **wstaw** oraz zmienna **początek** i typ **kolejka**. Aby pozostałe atrybuty były niewidoczne, można wewnątrz klasy **kolejki_spr** użyć specyfikacji **taken**

```
taken wstaw, początek, kolejka;
```

Ogólnie, specyfikacja

```
taken a1,...,an;
```

umieszczona w module prefiksowanym powoduje, iż spośród atrybutów prefiksu są w nim widoczne jedynie atrybuty **a1,...,an**. Jeżeli specyfikacja **taken** występuje w module wielokrotnie, jest równoważna jednej specyfikacji z sumaryczną listą atrybutów. Analogiczny skutek uzyskujemy używając specyfikacji **hidden** w module prefiksującym

```
hidden a1,...,ak;
```

Uniemożliwia ona dostęp do atrybutów **a1,...,ak** z wewnątrz modułu prefiksowanego. Podobnie jak w przypadku **taken**, specyfikacja **hidden** może wystąpić wielokrotnie. Wynik jest równoważny jednej specyfikacji z łączną listą atrybutów.

Gdy w prefiksie występuje specyfikacja **hidden**, a w module prefiksowanym **taken**, to w module prfiksowanym są dostępne te atrybuty prefiksu, które są wymienione na liście **taken**, a nie występują na liście **hidden**.

Zauważmy, że możliwość ochrony atrybutów jest przydatna zwłaszcza przy tworzeniu dużych programów, w których niebezpieczeństwo niekontrolowanego dostępu do atrybutów znacznie wzrasta. W tym przypadku niepożądanej zmianie mógłyby ulec np. atrybut **koniec** klasy **kolejki**, dotychczas zabezpieczony jedynie przed dostępem z zewnątrz obiektu. Podobnie będą chronione wszystkie atrybuty klasy **tablice_rozproszone** poza **wstaw** i **sprawdź**. Podejście to daje także dokładną zgodność implementacji ze specyfikacją struktury danych.

Przykładowym zastosowaniem klasy **tablice_rozproszone** jest następujący blok:

```
block const rozmiar=1023;
    unit element: class(i: integer); end element;
    unit tablice_rozproszone: class ...
        end tablice_rozproszone;
    var e1, e2, e3: element;
    begin
    pref tablice_rozproszone(element, rozmiar) block
        unit virtual równe: function
                (e1, e2: element): boolean;
            begin result:=e1.i=e2.i end równe;
        unit virtual funkcja_rozpraszająca: function
                                (e: element): integer;
            begin result:=e.i mod (rozmiar+1)
            end funkcja_rozpraszająca;
        begin
        e1:=new element(1); e2:=new element(1025);
        call wstaw(e1); call wstaw(e2);
          (* kolizja e1 i e2 *)
        ...
        if sprawdź(e1) then e3:=new element(100)
        else e3:=e1 fi;
        call wstaw(e3);
        ...
        end
    end;
```

Zauważmy, że w tym przykładzie atrybuty **funkcja_rozpraszająca** oraz **równe** są sprecyzowane dopiero w bloku prefiksowanym, a więc bezpośrednio przed ich użyciem.

*Rozwiązanie* 5.7.4

```
unit tablice_rozproszone: class(type E; rozmiar:integer);
    close tablica;
    hidden tablica, kolejki, kolejki_spr;
    var tablica: array_of kolejki_spr;
    unit virtual funkcja_rozpraszająca: function
                                    (e: E): integer;
        end funkcja_rozpraszająca;
    unit virtual równe: function(e1, e2: E): boolean;
        end równe;
    unit kolejki: class(type E);
        (* treść tej klasy została podana w p. 4.4;
```

```
        definicja ta nie musi wystąpić wewnątrz klasy
        tablice_rozproszone --- wystarczy, by pojawiła
        się w programie w miejscu widocznym stąd
        syntaktycznie *)
    ...
    end kolejki;
unit kolejki_spr: kolejki class;
    taken wstaw, początek, kolejka;
    unit spr: function(e: E): boolean;
        var pom: kolejka;
        begin
        pom:=początek;
        while pom=/=none do
            if równe(pom.pierwszy_element,e) then
              result:=true; return
            else pom:=pom.pozostała_część fi
            od
        end spr;
    end kolejki_spr;
unit wstaw: procedure(e: E);
    begin
    if not sprawdź(e) then
      call tablica(funkcja_rozpraszająca(e)).wstaw(e)
      fi
    end wstaw;
unit sprawdź: function(e: E): boolean;
    begin
    result:=tablica(funkcja_rozpraszająca(e)).spr(e)
    end sprawdź;
begin
array tablica dim(0:rozmiar)
end tablice_rozproszone;
```

# Ogólny algorytm zachłanny i algorytm Kruskala — zastosowanie prefiksowania do budowy algorytmów abstrakcyjnych i ich konkretyzacji 5.8

## *Sformułowanie problemu* 5.8.1

Zaimplementuj ogólny algorytm zachłanny, a następnie zastosuj ten algorytm do znajdowania lasu, o najmniejszym koszcie, rozpinającego dany graf nieskierowany.

## *Dyskusja zadania* 5.8.2

Algorytmy zachłanne służą do rozwiązywania pewnej klasy zagadnień optymalizacyjnych. Podstawowym pojęciem w teorii algorytmów zachłannych są matroidy. Parę $(E, J)$, gdzie $E$ jest zbiorem skończonym, a $J$ pewnym niepustym zbiorem podzbiorów $E$ nazywamy *matroidem*, gdy są spełnione następujące warunki:

— jeśli $A \in J$ i $B \subseteq A$, to $B \in J$;
— dla dowolnych $A, B \in J$, jeśli $B$ ma o jeden element więcej niż $A$, to istnieje $e \in B - A$ taki, że $A \cup \{e\} \in J$.

Załóżmy teraz, że jest dana funkcja $w: E \rightarrow real$ o tej własności, że $w(e) > 0$ dla $e \in E$. Wartość $w(e)$ nazywamy kosztem elementu $e$. Funkcję tę rozszerzamy na podzbiory $E$ przyjmując, że koszt podzbioru $A \subseteq E$, $w(A)$ jest dany wzorem: $w(A) = \sum_{e \in A} w(e)$. Ogólna postać algorytmu zachłannego jest następująca:

```
posortuj E według malejących kosztów tak, by
    E = (e_1, ..., e_n), gdzie w(e_1) >= w(e_2) >= ... >= w(e_n);
S := ∅;
dla i od 1 do n powtarzaj:
    jeśli S ∪ {e_i} ∈ J to S := S ∪ {e_i}.
```

Przy założeniu, że $(E,J)$ jest matroidem, w wyniku działania tego algorytmu $S$ jest zbiorem o największym koszcie należącym do $J$.

Przypomnijmy teraz pojęcia niezbędne do zbudowania algorytmu Kruskala znajdowania lasu, o najmniejszym koszcie, rozpinającego dany graf. Parę $(E,V)$ nazywamy grafem nieskierowanym (lub krócej — grafem), gdy $V$ jest skończonym zbiorem zwanym zbiorem wierzchołków, zaś $E \subseteq \{\{x, y\}: x, y \in V, x =/= y\}$ jest zbiorem krawędzi grafu. Cyklem w grafie nazywamy ciąg wierzchołków $v_1, v_2, \ldots, v_k$ taki, że dla $i < k$

$\{v_i, v_{i+1}\} \in E$ oraz $\{v_k, v_1\} \in E$. Las definiujemy jako graf bez cykli. Lasem rozpinającym dany graf $G$ nazywamy dowolny las zawierający wszystkie wierzchołki $G$. Dla dowolnego grafu $G = (V, E)$ definiujemy $M(G)$ jako parę $(E, \{A \subseteq E$: graf $(V, A)$ jest lasem$\})$. Nietrudno zauważyć, że $M(G)$ jest matroidem. Zatem algorytm zachłanny zastosowany do tego matroidu znajduje las rozpinający $G$ o największym koszcie. Aby znaleźć las rozpinający o najmniejszym koszcie, wystarczy zmienić definicję relacji >= występującej w fazie sortowania elementów ze zbioru $E$ w algorytmie zachłannym na przeciwną.

### *Omówienie rozwiązania* 5.8.3

Ogólny algorytm zachłanny zdefiniujemy jako algorytm abstrakcyjny tzn. operujący na pojęciach abstrakcyjnych, których znaczenie jest precyzowane dopiero w programie użytkownika. Najbardziej naturalnym sposobem realizacji takiego podejścia jest prefiksowanie, toteż algorytm zachłanny zdefiniujemy jako klasę. Parametrami tej klasy będzie typ elementów zbioru $E$, zbiór $E$ oraz funkcja kosztu $w$. Przyjmiemy, że zbiór $E$ będzie dany w postaci tablicy zawierającej wszystkie jego elementy

```
unit algorytm_zachłanny: class( type T; E: array_of T;
                                function w(t: T): real);
  ...
  end algorytm_zachłanny;
```

W definicji algorytmu zachłannego (p. 5.8.2) występują abstrakcyjne pojęcia $\emptyset$, $S \cup \{e_i\}$, $S \cup \{e_i\} \in J$. Ponieważ dany jest typ elementów $T$, pojęcie zbioru pustego i sumy zbiorów możemy implementować już na poziomie algorytmu abstrakcyjnego, np. za pomocą kolejek (por. p. 4.4). Zbiorowi pustemu będzie odpowiadać oczywiście pusta kolejka. Ponieważ do zbioru zawsze dołączamy tylko jeden element, więc sumę tę możemy implementować za pomocą operacji **wstaw** określonej dla kolejek w p. 4.4. Pozostaje abstrakcyjna, wymagająca sprecyzowania, operacja sprawdzania czy dany zbiór należy do rodziny $J$. Jest ona oczywiście ściśle zależna od definicji rodziny $J$ i nie może być zrealizowana w sposób ogólny, niezależny od konkretnych zastosowań. Również, jak to wynika z dyskusji zadania przedstawionej w poprzednim punkcie, w sposób elastyczny powinna być zdefiniowana relacja porównywania elementów zbioru $E$. Obie te operacje zrealizujemy jako podprogramy wirtualne.

Fazę sortowania algorytmu zachłannego zrealizujemy za pomocą uniwersalnej procedury sortującej zdefiniowanej w p. 4.3. Deklaracja klasy **algorytm_zachłanny** przyjmie więc następującą postać:

```
unit algorytm_zachłanny: class(type T; E: array_of T;
                               function w(e:T): real);
    unit kolejki: class(type E);
        ... (* por. p. 4.4.4 *) ...
        end kolejki;
    unit sortuj: procedure(type T; tab: array_of T;
                    function mniejsze(x,y:T): boolean);
        ... (* por. p. 4.3.4 *) ...
        end sortuj;
    unit virtual większe: function(e1,e2: T): boolean;
        begin result:= w(e1) >= w(e2) end większe;
    unit virtual warunek: function: boolean;
        end warunek;
    var S: kolejki, i: integer;
    begin
    inner;
    call sortuj(T,E,większe);
    S:=new kolejki(T);
    for i:=lower(E) to upper(E) do
       if warunek then call S.wstaw(E(i)) fi
       od
    end algorytm_zachłanny;
```

Instrukcja inner występuje na początku listy instrukcji algorytmu zachłannego. Daje to możliwość inicjalizacji struktur danych użytkownika przed wykonaniem algorytmu.

Zauważmy, że jeśli użytkownik zastosuje algorytm zachłanny w sposób typowy, tzn. bez konieczności zmiany relacji, względem której sortowany jest zbiór $E$, nie będzie musiał podawać swojej definicji tej relacji. W tym przypadku treść funkcji **większe** zostanie wzięta z klasy **algorytm zachłanny**, gdzie jest ona określona poprawnie dla typowych zastosowań. W przypadku algorytmu Kruskala będziemy musieli podać nową definicję funkcji

```
unit virtual większe: function(e1,e2: T): boolean;
 begin result:= w(e1) <= w(e2) end większe;
```

Zastanówmy się teraz, jak reprezentować graf. Dla naszych celów najwygodniejsze jest przedstawienie grafu za pomocą tablicy zawierajacej wszystkie krawędzie grafu. Załóżmy przy tym, że wierzchołki grafu są ponumerowane liczbami od 1 do n dla pewnej zadanej liczby całkowitej n>0. Atrybutami krawędzi są wierzchołki ją tworzące oraz wartość funkcji kosztu.

```
unit krawędź: class;
    var i1,i2: integer, w: real;
    begin
    writeln(" podaj wierzchołki tworzące krawędź");
    readln(i1,i2);
    writeln(" podaj wartość funkcji kosztu");
    readln(w)
    end krawędź;
```

W tej definicji założyliśmy, że użytkownik będzie podawał poprawne dane, tzn. i1,i2,w>0.

Graf będzie dany za pomocą tablicy:

```
var graf: array_of krawędź;
```

Tablicę tę utworzymy następująco:

```
writeln(" podaj liczbę krawędzi grafu");
readln(liczba_krawędzi);
array graf dim(1:liczba_krawędzi);
for j:=1 to liczba_krawędzi do
   graf(j):=new krawędź
   od;
```

Wynikiem funkcji w jest atrybut krawędzi, reprezentujący jej koszt

```
unit w: function(e: krawędź): real;
    begin result:=e.w end w;
```

Do rozwiązania pozostaje nam jeszcze problem implementacji funkcji wirtualnej warunek. W trakcie działania algorytmu Kruskala bierzemy kolejno krawędzie grafu i sprawdzamy, czy po dołożeniu aktualnie analizowanej krawędzi dotychczas uzyskany graf nadal pozostanie lasem. Należy zatem pamiętać osobno każdy spójny fragment lasu i dołączać krawędź tylko wówczas, gdy jej wierzchołki leżą w różnych spójnych fragmentach, w przeciwnym bowiem razie powstałby cykl.

Naturalną strukturą danych do rozwiązania tego problemu są zbiory. Dokładniej, potrzebujemy struktury danych zdefiniowanej dla danej liczby naturalnej n w sposób następujący:

$$\mathsf{Find_Union}(n) = (\mathit{Find_Union};\ \mathsf{Find}, \mathsf{Union})$$

gdzie

(1) nośnikiem jest rodzina podzbiorów zbioru $\{1,2,\ldots,n\}$;
(2) Find: $\{1,2,\ldots,n\} \to \mathit{Find_Union}$;
(3) Union: $\mathit{Find_Union} \times \mathit{Find_Union} \to \mathit{Find_Union}$.

Wynikiem operacji Find(i) jest zbiór, do którego element i należy. Operacja Union usuwa z rodziny zbiory będące jej argumentami, dołączając w ich miejsce zbiór będący sumą argumentów. Zauważmy, że jeśli przed wykonaniem operacji Union rodzina zbiorów wyznaczała podział zbioru, $\{1,2,\ldots,n\}$ to będzie tak również po jej wykonaniu. Zauważmy też, że operacja Find jest dobrze określona przy założeniu, że każda liczba ze zbioru $\{1,2,\ldots,n\}$ jest elementem dokładnie jednego zbioru. Musimy zatem zapewnić, aby rozważana rodzina zbiorów wyznaczała podział zbioru $\{1,2,\ldots,n\}$.

W naszym problemie zbiorowi $\{1,2,\ldots,n\}$ będzie odpowiadać zbiór wierzchołków grafu, zbiorom podziału zaś będą odpowiadać spójne fragmenty lasu. Inaczej mówiąc, dwa wierzchołki należą do tego samego zbioru, jeśli należą do jednego spójnego fragmentu lasu. Zauważmy ponadto, że w naszym problemie początkowa rodzina zbiorów zawiera jednoelementowe zbiory: $\{1\},\{2\},\ldots,\{n\}$, bowiem algorytm Kruskala startuje od lasu nie zawierającego żadnych krawędzi. Algorytm startuje więc od podziału zbioru $\{1,2,\ldots,n\}$. Ponieważ wykonanie operacji Find oraz Union nie zmienia własności, iż rozważana rodzina zbiorów jest podziałem, warunek poprawności operacji Find jest spełniony.
Ze względu na liczne zastosowania, struktura danych Find_Union jest przedstawiana w wielu źródłach (patrz Literatura), nie będziemy więc omawiać jej subtelności. Przyjmiemy prostą implementację drzewową. Drzewo będzie implementowane za pomocą tablicy ojciec, przy czym wartością ojciec(i) jest ojciec elementu i w drzewie. Zbiór będzie identyfikowany przez korzeń drzewa reprezentującego ten zbiór. Schemat definicji klasy Find_Union jest następujący:

```
unit Find_Union: class(n: integer);
    close ojciec;
    var ojciec: array_of integer;
    unit Find: function(x: integer): integer; ...
        end Find;
    unit Union: function(y,z: integer): integer; ...
        end Union;
    begin ... end Find_Union;
```

Parametrem aktualnym Find_Union będzie liczba wierzchołków w grafie. Definicja funkcji warunek przyjmie teraz postać

```
var las: Find_Union;
unit virtual warunek: function: boolean;
    var k,l: integer;
    begin
    k:= las.Find(E(i) qua krawędź.i1);
```

```
        l:= las.Find(E(i) qua krawędź.i2);
        if k=/=l then (* wierzchołki krawędzi
                         leżą w różnych zbiorach *)
          result:=true; l:=las.Union(k,l) fi
        end warunek;
```

Zauważmy jeszcze, że była niezbędna zmiana typu E(i) za pomocą operatora **qua**, elementy tablicy E były bowiem typu formalnego T, toteż nie można było bezpośrednio korzystać ze znajomości struktury obiektów zapamiętanych w E.

*Rozwiązanie* 5.8.4

```
block
    unit krawędź: class;
        var i1,i2: integer, w: real;
        begin
        writeln(" podaj wierzchołki tworzące krawędź");
        readln(i1,i2);
        writeln(" podaj wartość funkcji kosztu");
        readln(w)
        end krawędź;
    unit w: function(e: krawędź): real;
        begin result:=e.w end w;
    unit algorytm_zachłanny: class(type T; E: array_of T;
                                   function w(e: T): real);
        unit kolejki: class(type E);
            ... (* por. p. 4.4.4 *) ...
            end kolejki;
        unit sortuj: procedure(type T; tab: array_of T;
                               function mniejsze(x,y:T):boolean);
            ... (* por. p. 4.3.4 *) ...
            end sortuj;
        unit virtual większe: function(e1,e2: T): boolean;
            begin result:= w(e1) >= w(e2) end większe;
        unit virtual warunek: function: boolean;
            end warunek;
        var S: kolejki, i: integer;
        begin
        inner;
        call sortuj(T,E,większe);
        S:=new kolejki(T);
        for i:=lower(E) to upper(E) do
```

```
        if warunek then call S.wstaw(E(i)) fi
        od
    end algorytm_zachłanny;

begin
pref algorytm_zachłanny(krawędź, w) block
    (* algorytm Kruskala *)
    unit Find_Union: class(n: integer);
        close ojciec;
        var ojciec: array_of integer;
        unit Find: function(x: integer): integer;
            begin
            result:=x;
            while ojciec(result)=/=0 do
                result := ojciec(result)
                od
            end Find;
        unit Union: function(y,z: integer): integer;
            begin
            result, ojciec(z) := y
            end Union;
        begin
        array ojciec dim(1:n)
        end Find_Union;
    var las: Find_Union;
    unit virtual większe: function(e1,e2: T): boolean;
        begin result:= w(e1) <= w(e2) end większe;
    unit virtual warunek: function: boolean;
        var k,l: integer;
        begin
        k:= las.Find(E(i) qua krawędź.i1);
        l:= las.Find(E(i) qua krawędź.i2);
        if k=/=l then (* wierzchołki krawędzi
                    leżą w różnych zbiorach *)
          result:=true; l:=las.Union(k,l) fi
        end warunek;
    begin
    writeln(" podaj liczbę wierzchołków");
    readln(liczba_wierzchołków);
    writeln(" podaj liczbę krawędzi grafu");
    readln(liczba_krawędzi);
    array graf dim(1:liczba_krawędzi);
    for j:=1 to liczba_krawędzi do
```

```
        graf(j):=new krawędź od;
    las:=new Find_Union(liczba_wierzchołków)
    (* teraz wykonują się instrukcje z prefiksu,
       wyznaczające na zmiennej S szukany las
       rozpinający *)
    end
end;
```

## Podsumowanie 5.9

(1) Prefiksowanie jest operacją rozszerzania działającą na modułach. Modułem rozszerzanym (inaczej: modułem prefiksującym lub prefiksem) może być klasa, współprogram (por. rozdz. 6) lub proces (por. rozdz. 8). Rozszerzenie jest również specyfikowane w formie modułu. Rodzaj modułu wynikowego operacji rozszerzania (tzw. modułu prefiksowanego) zależy od rodzajów prefiksu i rozszerzenia (patrz tabl. 5.1).

(2) Obiekt modułu prefiksowanego dziedziczy wszystkie atrybuty prefiksu, które nie zostały zasłonięte ponowną deklaracją, o ile dostęp do nich nie jest ograniczony specyfikacją hidden lub taken. W module prefiksowanym są dostępne tylko te atrybuty prefiksu, które zostały wymienione w specyfikacji taken tego modułu oraz nie zostały wymienione w specyfikacji hidden prefiksu. Brak specyfikacji ograniczającej oznacza brak ograniczenia.

(3) Podczas tworzenia egzemplarza modułu prefiksowanego jest wykonywana połączona lista instrukcji prefiksu i rozszerzenia: w ciągu instrukcji prefiksu wykonanie instrukcji inner jest zastąpione wykonaniem listy instrukcji rozszerzenia. Instrukcja inner może wystąpić w treści modułu co najwyżej raz, ale może być wykonywana wielokrotnie. Jeśli instrukcja inner nie występuje, przyjmuje się ją domyślnie na końcu listy instrukcji. Jeżeli moduł nie jest użyty jako prefiks, to instrukcja inner jest równoważna instrukcji pustej. Instrukcje prefiksu są wykonywane w środowisku prefiksu, instrukcje rozszerzenia zaś — w jego środowisku wzbogaconym o odziedziczone atrybuty prefiksu.

(4) Jeśli klasa A prefiksuje klasę B, to B nazywamy podklasą klasy A.

(5) Ponieważ klasa prefiksowana może znów zostać użyta jako prefiks, użytkownik może tworzyć hierarchie typów danych przez stopniowe ich rozszerzanie. Obiekt klasy A należy do typu definiowanego tą klasą oraz do wszystkich typów klasowych, dla których A jest podklasą.

(6) Wartością zmiennej typu klasowego A może być wskaźnik do dowolnego obiektu należącego do typu danych definiowanego klasą A.

Obiekt ten może być zatem nie tylko obiektem klasy A, ale również dowolnej jej podklasy.

(7) Możliwe jest sprawdzenie, jakiego typu jest obiekt wskazywany przez daną zmienną. Służą temu relacje in oraz is. Wyrażenie x in A ma wartość true wtedy i tylko wtedy, gdy zmienna x wskazuje na obiekt należący do typu A (czyli obiekt klasy A lub jej podklasy). Wyrażenie x is A ma wartość true jedynie wówczas, gdy wartością zmiennej x jest wskaźnik do obiektu klasy A.

(8) Użycie operatora qua umożliwia w pewnych sytuacjach odległy dostęp do atrybutu obiektu (gdy typ obiektu jest podklasą typu zmiennej wskazującej obiekt lub gdy atrybut został zasłonięty — por. p. 5.4).

(9) Wyrażenie this A jest poprawne, jeżeli istnieje moduł syntaktycznie otaczający miejsce wystąpienia tego wyrażenia, który ma w swoim ciągu prefiksowym moduł A. Wartością tego wyrażenia jest wskaźnik do najbliższego obiektu takiego modułu.

(10) Deklaracja podprogramu wirtualnego o tej samej nazwie nie zasłania deklaracji z prefiksu, ale zastępuje ją we wszystkich do niej odwołaniach.

(11) Aby zastąpienie poprogramów wirtualnych było możliwe, muszą być spełnione następujące warunki:

(a) podprogramy muszą być tego samego rodzaju (albo procedury, albo funkcje);

(b) nagłówki podprogramów muszą umożliwiać zastąpienie, np. być identyczne (dokładne reguły są podane w Dodatku C);

(c) podprogramy muszą należeć do jednego ciągu podprogramów wirtualnych o tej samej nazwie spełniających warunki (a) i (b), zadeklarowanych w kolejnych modułach spójnego fragmentu ciągu prefiksowego; brak deklaracji podprogramu wirtualnego, ograniczenie dostępu specyfikacją hidden lub niespełnienie warunków (a) i (b) powoduje przerwanie ciągu podprogramów wirtualnych.

(12) Klasa może definiować język problemowy. W modułach prefiksowanych tą klasą są dostępne pojęcia tego języka.

# Współprogramy 6

## Wprowadzenie 6.1

Ważnym narzędziem programistycznym są *współprogramy*. Ich podstawowym zastosowaniem jest modelowanie systemów składających się ze współistniejących w tym samym czasie, współpracujących ze sobą obiektów. Współprogramy są mechanizmem sekwencyjnym — w danej chwili może być aktywny co najwyżej jeden obiekt współprogramu. Systemy programowane z użyciem współprogramów nazywamy zatem quasi-równoległymi, w odróżnieniu od systemów równoległych, które omówimy w dalszej części książki.

Obok modelowania zjawisk współbieżnych, system współprogramów jest stosowany przy szczególnie eleganckim zapisie takich algorytmów, w których w sposób naturalny można wyodrębnić wzajemnie przeplatające się fazy. Współprogramy dają także bogate możliwości udostępniania częściowych wyników działania algorytmów.

Z punktu widzenia programisty system współprogramów w Loglanie może być uważany za rozszerzenie systemu klas; współprogramy mają te same własności co klasy, a dodatkowo są wyposażone w możliwość przekazywania sterowania między ich obiektami.

Deklaracja współprogramu przypomina deklarację klasy, przy czym słowo **class** zastąpione jest słowem **coroutine** (por. p. 6.2.3).

Obiekty współprogramów są tworzone za pomocą operatora **new**. Podobnie jak w przypadku klas, zostaje zawieszone wykonanie obiektu wykonującego tę operację, następuje transmisja parametrów wejściowych do tworzonego obiektu współprogramu, jego lokalne atrybuty są inicjowane w sposób standardowy i zaczynają wykonywać się jego instrukcje. Mówimy, że obiekt ten znajduje się w stanie inicjalizacji. Faza inicjalizacji obiektu współprogramu trwa aż, do napotkania w jego ciągu instrukcji **return** (lub do wyczerpania się instrukcji współprogramu, wrazie jej braku). W tym momencie są przekazywane parametry wyjściowe i sterowanie z

obiektu współprogramu do obiektu, który spowodował jego utworzenie. Wykonanie obiektu współprogramu zostaje zawieszone — obiekt przechodzi ze stanu inicjalizacji w stan zawieszenia.

Podobnie jak jest w przypadku obiektu klasy, do atrybutów obiektu współprogramu można odwoływać się używając odległego dostępu (por. p. 3.2). Sposób ochrony atrybutów przed niepowołanym dostępem jest również taki sam jak dla klas (por. p. 3.2 oraz 5.7.3). Podstawową różnicą między obiektem współprogramu, a obiektem klasy jest to, że utworzony obiekt klasy pełni jedynie funkcję bierną — jako rekord zawierający pewne dane. Tymczasem obiekt współprogramu, jeżeli tylko lista jego instrukcji nie została wyczerpana, może zostać uaktywniony.

Uaktywnienie współprogramu następuje w wyniku wykonania jednej z instrukcji przekazywania sterowania: **attach** lub **detach**. Jeśli **w** jest wyrażeniem obiektowym wskazującym na obiekt współprogramu, to wykonanie instrukcji **attach(w)** spowoduje przekazanie sterowania do wskazywanego obiektu. Instrukcja **detach** jest bezparametrowa. Zwraca ona sterowanie do obiektu współprogramu, który ostatnio aktywował obiekt wykonujący tę instrukcję za pomocą instrukcji **attach**. Przerwanie akcji współprogramu może nastąpić w dowolnej chwili jego wykonywania, np. w trakcie wykonywania innej jednostki wołanej przez współprogram. Gdy sterowanie zostanie zwrócone, wykonanie instrukcji rozpocznie się od miejsca ostatniego ich przerwania.

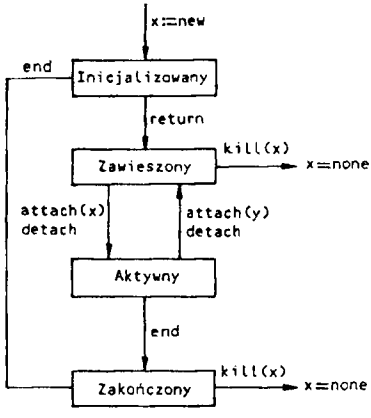
Widzimy zatem, że obiekt współprogramu, który zakończył fazę inicjalizacji znajduje się na przemian w stanach zawieszenia i aktywności. Jednocześnie, ponieważ uaktywnienie współprogramu wiąże się z zawieszeniem obecnie aktywnego, jest spełniony warunek, że w danej chwili dokładnie jeden obiekt współprogramu jest aktywny.

Gdy lista instrukcji współprogramu zostanie wyczerpana (sterowanie osiągnie instrukcję **end** kończącą współprogram), obiekt współprogramu staje się zakończony. Jednocześnie jest wykonywana instrukcja **detach**. Zakończony obiekt współprogramu jest niedostępny dla operacji przekazywania sterowania. Istnieje on, podobnie jak klasa, jako zbiór danych i operacji, do których odwoływać się można za pomocą odległego dostępu.

Podobnie jak w przypadku wszystkich typów obiektowych, zakończony obiekt współprogramu można usunąć z pamięci wykonując instrukcję **kill** (por. p. 1.4). Zawieszony obiekt współprogramu można także usunąć z pamięci. Wiąże się to wówczas z usunięciem wraz z nim wszystkich obiektów, których utworzenie i wykonywanie spowodował bezpośrednio lub pośrednio, czyli obiektów z tzw. łańcucha dynamicznego współprogramu.

Rysunek 6.1 ilustruje zmiany stanów obiektu współprogramu wynikłe z wykonania opisanych instrukcji.

Aby opisać działanie współprogramów pomiędzy ich utworzeniem




RYS. 6.1 Schemat zmian stanów współprogramów

a zakończeniem, wprowadźmy pojęcie *łańcucha dynamicznego współprogramu*. Aktywny współprogram wykonuje się, aż do chwili napotkania instrukcji **attach**, **detach**. Wynika stąd, że tworzenie obiektów innych modułów nie ma wpływu na zmianę aktywnego współprogramu. Z każdym więc obiektem współprogramu może być związany łańcuch obiektów $X_1, \ldots, X_k$ o tej własności, że $X_1$ jest obiektem współprogramu, zaś dla $i < k$ wykonanie $X_i$ spowodowało utworzenie (i wykonywanie) $X_{i+1}$.

Program główny jest też współprogramem. Jest on dostępny pod nazwą **main**.

W danej chwili wykonywania programu dokładnie jeden współprogram jest aktywny. Konfigurację dynamiczną programu można sobie zatem wyobrazić jako pewną liczbę rozłącznych łańcuchów dynamicznych, z których dokładnie jeden odpowiada współprogramowi aktywnemu (będziemy go nazywać łańcuchem aktywnym w odróżnieniu od pozostałych — zawieszonych). Konfiguracja początkowa składa się z obiektu programu głównego, który jest jedynym obiektem aktywnego łańcucha współprogramu o nazwie **main**.

Wszystkie instrukcje przekazywania sterowania wykonywane w obiektach z łańcucha dynamicznego danego współprogramu odnoszą się do tego właśnie współprogramu. Obiekt współprogramu, który jest w stanie inicjalizacji nie jest jeszcze w pełni utworzony i nie stanowi samodzielnego łańcucha. Jest on natomiast częścią łańcucha dynamicznego współprogramu, który spowodował jego utworzenie, podobnie jak inne obiekty tworzone przez ten współprogram. Wynika stąd, że instrukcje **attach** i **detach** wykonywane przez współprogram w fazie inicjalizacji odnoszą się nie do tego współprogramu, lecz do obiektu będącego

początkiem zawierającego go łańcucha dynamicznego. Do chwili wykonania instrukcji **return** inicjalizowany obiekt nie różni się od obiektu klasy. Przejście w stan zawieszenia po wykonaniu tej instrukcji jest związane z usunięciem obiektu współprogramu z aktywnego łańcucha dynamicznego i utworzeniem nowego łańcucha zawierającego tylko ten obiekt.

Ponieważ współprogram, podobnie jak klasa, może zostać użyty jako prefiks dla innego modułu (por. p. 5.1), ma on wszelkie własności klas związane z prefiksowaniem, w szczególności może wystąpić jako argument następujących operacji: **in**, **is**, **qua**, **this** (por. p. 5.4).

Dodatkowo w przypadku współprogramów jest zdefiniowany standardowy typ **coroutine** obejmujący wszystkie współprogramy (każdy obiekt współprogramu należy do tego typu). Zatem poza zawieraniem się typów wynikającym z prefiksowania, wszystkie typy będące współprogramami są zawarte w typie **coroutine**. Typ **coroutine** nie ma specyficznych atrybutów. Najczęściej jest używany w kontekście wyrażenia **this coroutine**, którego wartością jest wskaźnik do aktywnego obiektu współprogramu.

## Symulacja licznika — podstawowe własności współprogramów 6.2

### *Sformułowanie problemu* 6.2.1

Napisz program, który modeluje działanie licznika. Licznik składa się z N kół reprezentujących kolejne cyfry rozwinięcia dziesiętnego liczby wskazywanej przez licznik. Każde koło może znajdować się w pozycji odpowiadającej cyfrze $0, 1, \ldots, 9$. Licznik otrzymuje z otoczenia impulsy, na które reaguje obrotem koła odpowiadającego jednostkom. Jeśli koło to znajduje się w pozycji odpowiadającej cyfrze 9 — zeruje się, a impuls przekazuje następnemu kołu. Podobnie działają pozostałe koła poza ostatnim, które nie przekazuje dalej impulsu.

### *Dyskusja zadania* 6.2.2

W postawionym zadaniu mamy do czynienia z typowym przykładem systemu quasi-równoległego. Obiektami w tym systemie są koła wzajemnie na siebie wpływające oraz otoczenie przekazujące im impulsy. Najwłaściwszym mechanizmem Loglanu, służącym do rozwiązania zadania są współprogramy. Umożliwiają one znalezienie strukturalnego i eleganckiego rozwiązania: każdy obiekt będzie opisywał działanie jednej ze

składowych naszego systemu quasi-równoległego — otoczenia lub jednego z kół.

### *Omówienie rozwiązania* 6.2.3

Opiszmy najpierw zachowanie kół. Każde koło będzie działać zgodnie z następującym algorytmem:

Jeśli cyfrą wskazywaną przez koło jest 9 — zmień swoją cyfrę na 0, przekaż impuls następnemu kołu; po powrocie sterowania przekaż je obiektowi, który nadesłał impuls.

W przeciwnym razie — zwiększ cyfrę o 1 i zwróć sterowanie obiektowi, który je przekazał.

Zauważmy, że jedynie ostatnie koło będzie musiało mieć nieco inne działanie — trzeba określić czym jest dlań „następne koło". Ponieważ w tej sytuacji następne koło nie istnieje, przyjmiemy, że ostatnie koło będzie pozostawiać sterowanie samemu sobie. Następnym kołem będzie więc dla niego ono samo.

Z algorytmu wynika, że atrybutem koła musi być wskazywana przezeń cyfra oraz wskaźnik do następnego koła. Zadeklarujmy współprogram reprezentujący koło

```
unit koło: coroutine;
    var cyfra: integer, następne: koło;
    ...
    end koło;
```

Cały licznik zadeklarujemy jako współprogram, którego parametrem jest liczba kół N, zaś jedynym atrybutem — wskaźnik do pierwszego koła, bowiem tylko jemu są przekazywane bezpośrednio impulsy. Aby zabezpieczyć zmienną wskazującą pierwsze koło przed niepowołanym użyciem, umieścimy ją w specyfikacji close. A oto schemat deklaracji licznika:

```
unit licznik: coroutine(N: integer);
    close pierwsze;
    unit koło: coroutine; ... end koło;
    var pierwsze: koło;
    begin
     (* tworzenie kół *) ...
    return; (* oddzielenie obiektu jako równoprawnego
            współprogramu *)
    do
       (* przekazywanie impusów *);
```

```
        attach(pierwsze);
        detach
        od
    end licznik;
```

Ponieważ zmienna wskazująca na pierwsze koło jest niedostępna spoza obiektu reprezentującego licznik, jedyną możliwością przekazania impulsu z zewnątrz jest przekazanie temu obiektowi sterowania. Ten z kolei przekaże je do obiektu reprezentującego pierwsze koło (**attach(pierwsze)**), a następnie, po jego odzyskaniu, przekaże z powrotem modułowi, który wysłał impuls (**detach**).

Pozostaje nam jeszcze uzupełnienie algorytmu tworzenia kół oraz ich działania. Tworzenie kół zaczniemy od ostatniego, które — jak już wcześniej wspomnieliśmy — musi być traktowane nieco inaczej niż pozostałe. Wskaźnik do tego koła zapamiętamy w zmiennej pomocniczej **pom**.

```
pom:=new koło;
pom.następne:=pom;
```

Inne koła tworzymy według identycznego wzorca.

```
for i:=1 to N-1 do
    pierwsze:=new koło;
    pierwsze.następne:=pom;
    pom:=pierwsze
    od;
```

Algorytm działania koła jest zgodny z wcześniej omówionym.

```
do
  if cyfra=9 then
     cyfra:=0; (* zmień swoją cyfrę na 0 *)
     attach(następne); (* przekaż impuls następnemu *)
     detach (* przekaż sterowanie obiektowi, który
               nadesłał impuls *)
  else cyfra:=cyfra+1; (* zwiększ cyfrę o 1 *)
       detach (* przekaż sterowanie obiektowi, który
                 nadesłał impuls *) fi
od;
```

Zauważmy, że w tym rozwiązaniu przekazanie impulsu jest zawsze przekazaniem sterowania. Co więcej — uzyskane rozwiązanie jest w pełni strukturalne — struktura obiektów odpowiada dokładnie strukturze modelowanej rzeczywistości.

*Rozwiązanie* 6.2.4

```
block
    unit licznik: coroutine(N: integer);
        close pierwsze;
        unit koło: coroutine;
            var cyfra: integer,
                następne: koło;
            begin
            return; (* oddzielenie obiektu jako
                       równoprawnego współprogramu *)
            do if cyfra=9 then
                  cyfra:=0; attach(następne); detach
               else cyfra:=cyfra+1; detach fi
               od
            end koło;
        var pierwsze: koło;
        begin
        block var pom: koło, i: integer;
              begin
              pom:=new koło; pom.następne:=pom;
              for i:=1 to N-1 do
                  pierwsze:=new koło;
                  pierwsze.następne:=pom;
                  pom:=pierwsze
                  od
              end;
        return;
        do attach(pierwsze);
          detach
          od
        end licznik;
    var l: licznik;
    begin
    l:=new licznik(100);
    do ... attach(l) ... od
    end;
```

# Generowanie permutacji — zawieszanie działania algorytmów 6.3

## *Sformułowanie problemu* 6.3.1

Zaprogramuj algorytm generujący kolejne permutacje liczb całkowitych $1, \ldots, n$, który umożliwia wykorzystanie każdej permutacji osobno.

## *Dyskusja zadania* 6.3.2

Typową metodą używaną do rozwiązania naszego zadania jest zdefiniowanie procedury, która daną tablicę reprezentującą permutację przekształca na następną permutację (porządek określony na permutacjach jest zależny od konkretnego algorytmu). Bardziej naturalne i prostsze koncepcyjnie jest jednak inne podejście — stworzenie procedury generującej wszystkie permutacje i zawieszanie działania tej procedury każdorazowo po utworzeniu nowej permutacji. Sam mechanizm procedur i funkcji występujący w językach programowania (również w Loglanie), nie daje możliwości chwilowego zawieszania ich działania i przekazania sterowania do innych modułów. Tymczasem podobny skutek można łatwo uzyskać stosując współprogramy.

Zastanówmy się teraz nad algorytmem generowania wszystkich permutacji liczb $1, \ldots, n$. Zdefiniujemy go rekurencyjnie. Przypuśćmy, że wiemy jak generować wszystkie permutacje liczb $p_1, \ldots, p_{k-1}$. Aby uzyskać wszystkie permutacje liczb $p_1, \ldots, p_k$ wystarczy użyć tego algorytmu do wszystkich kombinacji (k-1)-elementowych spośród $p_1, \ldots, p_k$.

## *Omówienie rozwiązania* 6.3.3

Permutacje obliczymy za pomocą współprogramu o nazwie **permutacja**. Kolejne permutacje liczb $1, \ldots, n$ będą pamiętane w tablicy **p(1:n)**. Tablica ta będzie atrybutem omawianego współprogramu. Każde przekazanie sterowania do obiektu tego współprogramu spowoduje utworzenie nowej permutacji. Wprowadzimy także nowy atrybut o nazwie **wszystkie** będący zmienną typu **boolean**. Wartością tej zmiennej będzie **true**, jeśli zostały już utworzone wszystkie permutacje. Wewnątrz współprogramu **permutacja** zdefiniujemy także procedurę **generator** realizującą wspomniany w poprzednim punkcie algorytm generowania permutacji.

```
unit permutacja: coroutine(n: integer);
    var wszystkie: boolean,
```

```
    p: array_of integer,
    q: integer;
unit generator: procedure(k: integer);
    ...end generator;
begin
array p dim(1:n);
for q:=1 to n do p(q):=q od;
return;
call generator(n);
wszystkie:=true
end permutacja;
```

Zastanówmy się jeszcze nad realizacją algorytmu generowania permutacji. Generowanie permutacji p(1),...,p(k-1) uzyskamy przez rekurencyjne wywołanie

```
call generator(k-1);
```

Aby uzyskać wszystkie permutacje, trzeba to wywołanie powtórzyć k-krotnie, w każdym zaś wywołaniu trzeba wykluczyć p(i) (1<=i<=k) spośród elementów p(1),...,p(k). W ten sposób algorytm zastosujemy do wszystkich kombinacji (k-1)-elementowych spośród p(1),...,p(k). Aby zrealizować to wykluczanie założymy, że algorytm zwraca po wywołaniu niezmieniony ciąg p(1),...,p(k-1). Algorytm przyjmie wówczas taką oto postać:

```
call generator(k-1);
for i:=1 to k-1 do
    q:=p(i); p(i):=p(k); p(k):=q;
    call generator(k-1)
    od;
```

Zauważmy. że przy tak zdefiniowanym algorytmie po wykonaniu instrukcji call generator(k-1) elementy p(1),...,p(k) będą przesunięte cyklicznie o jedno miejsce w prawo. Aby nasze założenie, że tablica p pozostaje niezmieniona było słuszne, musimy przywrócić jej poprzednią postać.

```
q:=p(1);
for i:=1 to k-1 do p(i):=p(i+1) od;
p(k):=q;
```

Nowa permutacja będzie utworzona zawsze, gdy procedura generator zostanie wywołana z parametrem k równym 1. Sterowanie jest zwracane do modułu korzystającego z permutacji, przez wykonanie instrukcji detach.

*Rozwiązanie* 6.3.4

```
unit permutacja: coroutine(n: integer);
    var wszystkie: boolean,
        p: array_of integer,
        q: integer;
    unit generator: procedure(k: integer);
        var i: integer;
        begin
        if k=1 then detach
        else call generator(k-1);
            for i:=1 to k-1 do
                q:=p(i); p(i):=p(k); p(k):=q;
                call generator(k-1)
                od;
            q:=p(1);
            for i:=1 to k-1 do p(i):=p(i+1) od;
            p(k):=q;
            fi
        end generator;
    begin
    array p dim(1:n);
    for q:=1 to n do p(q):=q od;
    return;
    call generator(n);
    wszystkie:=true
    end permutacja;
```

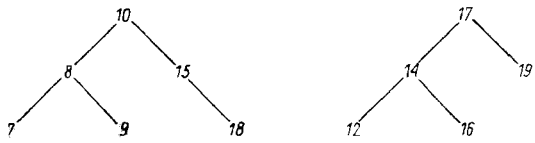
## Scalanie ciągów liczb — korzystanie z częściowych wyników działania algorytmu 6.4

*Sformułowanie problemu* 6.4.1

Dane jest n ciągów liczb całkowitych reprezentowanych przez binarne drzewa poszukiwań. Wypisz w porządku niemalejącym wszystkie elementy występujące w tych ciągach. Elementy występujące w więcej niż jednym drzewie mają być wypisane wielokrotnie.

*Dyskusja zadania* 6.4.2

Binarnym drzewem poszukiwań nazywamy drzewo binarne, w którym każde poddrzewo spełnia warunek, że elementy w jego lewym poddrzewie są mniejsze od elementu reprezentowanego przez korzeń, korzeń zaś jest mniejszy od wszystkich elementów prawego poddrzewa. Przykład binarnych drzew poszukiwań jest przedstawiony na rys. 6.2.




Rys. 6.2 Przykłady drzew binarnych poszukiwań

Przy założeniu drzewowej reprezentacji ciągów narzucającym się rozwiązaniem naszego zadania jest następujący algorytm:

Dopóki drzewa nie są puste wykonuj:
znajdź elementy minimalne drzew **t1**, ..., **tn**; wyznacz **t** będące drzewem o najmniejszym ze znalezionych elementów minimalnych;
wypisz element minimalny drzewa t;
usuń wypisany element z drzewa t.

Wadą tego algorytmu jest konieczność wielokrotnego przeglądania każdego drzewa w celu znalezienia kolejnych minimalnych elementów. Tymczasem istotne jest nie tyle znajdowanie elementu minimalnego w drzewie, co podawanie kolejno elementów drzewa w porządku niemalejącym. Przyjrzyjmy się strukturze drzewa poszukiwań — porządek jego elementów można odtworzyć przechodząc je za pomocą następującej metody rekurencyjnej (tzw. porządek infiksowy):

Odwiedź lewe poddrzewo.
Kolejnym elementem w porządku niemalejącym jest teraz korzeń.
Odwiedź prawe poddrzewo.

Zatem do znalezienia kolejnego co do wielkości elementu nie jest potrzebne każdorazowe przechodzenie drzewa od początku i usuwanie elementu minimalnego. Można zamiast tego korzystać z częściowych wyników algorytmu przechodzenia drzewa, zawieszając jego wykonywanie

do chwili, w której będzie potrzebny kolejny element. Podobnie jak w p. 6.3, zawieszenie algorytmu osiągniemy za pomocą współprogramów. Przyjęty przez nas algorytm przyjmie więc postać:

Znajdź elementy minimalne drzew t1, ..., tn.
Dopóki nie zostały wyczerpane elementy wszystkich drzew wykonuj
    niech t będzie drzewem o najmniejszym z bieżących elementów;
    wypisz bieżący element drzewa t;
    przejdź do kolejnego co do wielkości elementu w drzewie t (jeśli istnieje).

### *Omówienie rozwiązania* 6.4.3

Element drzewa binarnego zdefiniujemy jako typ klasowy.

```
unit drzewo: class(wartość: integer);
    var lewe, prawe: drzewo;
    end drzewo;
```

Dla każdego drzewa binarnego zdefiniujemy współprogram typu **ciąg** dostarczający kolejnych elementów drzewa będącego jego parametrem. Wyczerpanie się elementów drzewa jest sygnalizowane nadaniem zmiennej **koniec** wartości **true**. Ponadto z każdym ciągiem związemy zmienną x typu **integer** pełniącą funkcję skrzynki kontaktowej. W skrzynce tej są umieszczane wartości kolejnych elementów drzewa.

Procedura przechodzenia drzewa będzie także atrybutem współprogramu **ciąg**. Zgodnie z wcześniejszymi uwagami przyjmie ona następującą postać:

```
unit przejdź_drzewo: procedure(r: wierzchołek);
    begin
    if r=/=none then
      call przejdź_drzewo(r.lewe);
      x:=r.wartość; detach;
      call przejdź_drzewo(r.prawe) fi
    end przejdź_drzewo;
```

Cały algorytm scalania ciągów liczb zdefiniujemy jako procedurę, której parametrem jest liczba wejściowych drzew oraz tablica zawierająca te drzewa. Wszystkie współprogramy typu **ciąg** są zgromadzone w tablicy **ciągi**. Ciągów tych jest w każdej chwili tyle, ile pozostało niepustych drzew.

```
unit scalanie: procedure(n:integer, las:array_of drzewo);
    var ciągi: array_of ciąg;
```

```
...
begin
zainicjuj tablicę ciągi;
do j:=liczba niepustych drzew;
  if j=0 then exit fi;
  min:=numer drzewa zawierającego najmniejszy
       spośród bieżących elementów;
  wypisz najmniejszy bieżący element;
  attach(ciągi(min))
  od
end scalanie;
```

*Rozwiązanie* 6.4.4

```
unit drzewo: class(wartość: integer);
    var lewe, prawe: drzewo;
    end drzewo;
unit scalanie: procedure(n: integer; las: array_of drzewo);
    (* wypisuje uporządkowany ciąg elementów zawartych w
      n drzewach z tablicy las *)
    unit ciąg: coroutine( korzeń: drzewo);
        var x: integer, koniec:boolean;
        unit przejdź_drzewo: procedure(r: wierzchołek);
            begin
            if r=/=none then
            call przejdź_drzewo(r.lewe);
            x:=r.wartość; detach;
            call przejdź_drzewo(r.prawe)
            fi
            end przejdź_drzewo;
        begin
        return;
        call przejdź_drzewo(korzeń);
        koniec:=true
        end ciąg;
    var ciągi: array_of ciąg,
       n,i,j,k,min: integer;
    begin
    if las=none then return fi;
    array ciągi dim(1:n);
    j:=1;
    for i:=1 to n do
```

```
    ciągi(j):=new ciąg(las(i)); (* utworzenie
      obiektu współprogramu dla kolejnego drzewa *)
    attach(ciągi(j)); (* jeśli drzewo jest niepuste,
      to w skrzynce kontaktowej jest najmniejszy
      element tego drzewa, a w przeciwnym razie
      zmienna koniec ma wartość true *)
    if not ciągi(j).koniec then j:=j+1 fi
    od;
  j:=j-1;
  (* wybór minimalnego bieżącego elementu *)
  do if j=0 then exit fi;
    min:=1;
    for i:=2 to j do
      if ciągi(i).x<ciągi(min).x then min:=i fi
      od;
    write(ciągi(min.x)); attach(ciągi(min));
    if ciągi(min).koniec then
      ciągi(min):=ciągi(j); j:=j-1 fi
    od
  end scalanie;
```

## Wieże z Hanoi — współprogramy a procedury rekurencyjne 6.5

### *Sformułowanie problemu* 6.5.1

Dane są trzy wieże utworzone z krążków. Pierwsza z nich zawiera n krążków o różnych średnicach ułożonych w stos w ten sposób, że żaden krążek nie leży na krążku o mniejszej średnicy. Dwie pozostałe wieże są początkowo puste. Przenieś krążki z pierwszej wieży na trzecią (używając drugiej) tak, aby spełnione były następujące warunki:

— jednorazowo można przenosić tylko jeden krążek;

— żaden krążek nie może zostać położony na krążku o mniejszej średnicy.

### *Dyskusja zadania* 6.5.2

Najczęściej spotykanym rozwiązaniem naszego zadania jest procedura rekurencyjna oparta na następującym algorytmie:

Jeśli n=1 to przeniesienie pierwszej wieży na trzecią polega na przeniesieniu (jedynego) krążka.
W przeciwnym razie
przenosimy (n-1) krążków z pierwszej wieży na drugą (używając trzeciej);
następnie największy krążek z pierwszej na trzecią;
wreszcie (n-1) krążków z drugiej na trzecią (używając pierwszej).

Nasze rozwiązanie będzie również oparte na tym algorytmie. Zamiast procedur rekurencyjnych zastosujemy jednak współprogramy. Zauważmy, że procedury rekurencyjne są realizowane przez stos. Każdemu wywołaniu procedury odpowiada utworzenie nowego pola na stosie, zawierającego dane lokalne dla tego wywołania. W naszym zadaniu utworzymy jednorazowo odpowiednią liczbę obiektów współprogramu, których działanie będzie odzwierciedlać zachowanie procedury rekurencyjnej. Przedstawione przez nas rozwiązanie jest bardziej efektywne — wielokrotne tworzenie nowych obiektów procedur rekurencyjnych jest zastąpione jednorazowym utworzeniem mniejszej liczby obiektów współprogramu.

### *Omówienie rozwiązania* 6.5.3

Przyjrzyjmy się najpierw schematowi działania następującej procedury rekurencyjnej:

```
unit Hanoi: procedure(i, z, na: integer);
    (* przenosi i krążków z wieży z na wieżę na *)
    var k: integer;
    begin
    k:=numer wieży pomocniczej;
    if i>1 then call Hanoi(i-1,z,k) fi;
    przenieś krążek z wieży z na wieżę na;
    if i>1 then call Hanoi(i-1,k,na) fi
    end Hanoi;
```

Aby móc obsłużyć wywołania rekurencyjne, zdefiniujemy współprogram Hanoi, którego obiekty będą odpowiadać możliwym wywołaniom tej procedury. Zauważmy przy tym, iż parametry aktualne procedury Hanoi zmieniają się w sposób następujący:

— i przyjmuje wartości od 1 do n;
— z oraz na przyjmują wartości od 1 do 3, przy czym zawsze z=/=na.

Utworzymy trójwymiarową tablicę obiektów współprogramu Hanoi:

```
var obiekty: array_of array_of array_of Hanoi;
...
array obiekty dim(1:n);
for j:=1 to n do
   array obiekty(j) dim (1:3);
   for l:=1 to 3 do
      array obiekty(j,l) dim(1:3);
      for m:=1 to 3 do
         if l=/=m then
            obiekty(j,l,m):=new Hanoi(j,l,m) fi
         od
      od
   od;
```

Wywołaniu procedury Hanoi(i,j,k) odpowiada wznowienie (i,j,k)–tego obiektu współprogramu Hanoi. Zgodnie z tą konwencją zakończeniu działania procedury będzie odpowiadać przekazanie sterowania do obiektu, który aktywował obiekt aktualnie wykonywany.

```
unit Hanoi: coroutine(i, z, na: integer);
    var k: integer;
    begin
    return;
    do k:=numer wieży pomocniczej;
      if i>1 then attach(obiekty(i-1,z,k)) fi;
      przenieś krążek z wieży z na wieżę na;
      if i>1 then attach(obiekty(i-1,k,na)) fi;
      detach
      od
    end Hanoi;
```

Pętla do...od występująca w tym współprogramie ma za zadanie umożliwić wielokrotne wznowienie obiektów odpowiadające wielokrotnym wywołaniom procedury rekurencyjnej. Pętla ta jest nieskończona — nie ma z niej wyjścia za pomocą instrukcji exit. Taka sytuacja jest typowa dla współprogramów pełniących funkcję usługową. Mimo potencjalnej możliwości działania w nieskończoność, ich wykonanie jest powtarzane tylko dopóty, dopóki moduł je aktywujący nie zostanie zakończony.

Zauważmy na koniec, że numer wieży pomocniczej można obliczyć ze wzoru

```
numer wieży pomocniczej = 6-(z+na)
```

Opisane przez nas rozwiązanie można zastosować także we wszystkich sytuacjach, w których występuje wielokrotne wywoływanie proce-

dury rekurencyjnej z tymi samymi parametrami. Wystarczy wówczas z każdym takim wywołaniem związać odpowiedni obiekt współprogramu i zamienić wywołanie na wznowienie tego obiektu.

*Rozwiązanie* 6.5.4

```
unit wieże_z_Hanoi: class(n: integer);
    unit Hanoi: coroutine(i, z, na: integer);
        var k: integer;
        begin
        return;
        do
          k:=6-(z+na);
          if i>1 then attach(obiekty(i-1,z,k)) fi;
          writeln(" przenieś krążek z wieży",z,
                  " na wieżę",na);
          if i>1 then attach(obiekty(i-1,k,na)) fi;
          detach
          od
        end Hanoi;
    var obiekty: array_of array_of array_of Hanoi,
       j,l,m: integer;
    begin
    array obiekty dim (1:n);
    for j:=1 to n do
       array obiekty(j) dim(1:3);
       for l:=1 to 3 do
          array obiekty(j,l) dim(1:3);
          for m:=1 to 3 do
             if l=/=m then
                obiekty(j,l,m):=new Hanoi(j,l,m) fi
             od
          od
       od;
    attach(obiekty(n,1,3))
    end wieże_z_Hanoi;
```

## Podsumowanie 6.6

(1) Współprogramy są mechanizmem sekwencyjnym służącym do modelowania systemów quasi-równoległych. W danej chwili działania

programu może być aktywny co najwyżej jeden współprogram.

(2) Każdy obiekt współprogramu tworzy własny łańcuch dynamiczny składający się z obiektów, których utworzenie i wykonywanie spowodował bezpośrednio lub pośrednio. Konfiguracja dynamiczna w każdej chwili wykonywania programu składa się z pewnej liczby takich łańcuchów, z których dokładnie jeden jest aktywny.

(3) Współprogramy mają te same własności co klasy, dodatkowo umożliwiają przekazywanie sterowania. Po przekazaniu sterowania współprogram zawiesza swoje działanie. Przekazanie sterowania następuje w wyniku wykonania instrukcji **attach** lub **detach**. Instrukcja **attach(w)** powoduje przekazanie sterowania do obiektu współprogramu wskazywanego przez wartość wyrażenia wskaźnikowego **w**. Bezparametrowa instrukcja **detach** powoduje przekazanie sterowania do obiektu współprogramu, który ostatnio aktywował dany współprogram w wyniku wykonania instrukcji **attach**. Obiekt współprogramu, do którego zostało przekazane sterowanie, kontynuuje wykonanie instrukcji od miejsca ostatniego przerwania.

(4) Instrukcje przekazania sterowania dotyczą obiektu współprogramu stanowiącego początek łańcucha dynamicznego zawierającego obiekt je wykonujący.

(5) Obiekt współprogramu w fazie inicjalizacji (od chwili rozpoczęcia wykonywania instrukcji do napotkania instrukcji **return** lub — w razie jej braku — do wyczerpania się listy instrukcji) jest równoważny klasie. W szczególności wykonywane przezeń instrukcje przekazania sterowania dotyczą obiektu współprogramu stanowiącego początek łańcucha dynamicznego zawierającego dany obiekt. Po wykonaniu instrukcji **return** obiekt przechodzi w stan zawieszenia. Od tej chwili tworzy też własny łańcuch dynamiczny.

(6) Przekazywanie parametrów wiąże się z fazą inicjalizacji. Przekazanie danych wraz ze sterowaniem jest możliwe jedynie z zastosowaniem odległego dostępu.

(7) Wraz z zakończeniem wykonywania współprogramu (**end**) następuje przekazanie sterowania, podobnie jak w wyniku wykonania instrukcji **detach**. Zakończony obiekt współprogramu jest równoważny zakończonemu obiektowi klasy. Można odwoływać się do jego atrybutów z zastosowaniem odległego dostępu. Można go usunąć z pamięci przez wykonanie instrukcji **kill**. Instrukcja **kill** może być także stosowana do zawieszonych obiektów współprogramów.

(8) Jest zdefiniowany standardowy typ **coroutine** zawierający wszelkie typy współprogramów (każdy obiekt współprogramu należy do typu **coroutine**). Typ ten nie ma żadnych specyficznych atrybutów. Najczęściej jest on używany w kontekście wyrażenia **this coroutine**, którego wartością jest wskaźnik do obiektu współprogramu aktywnego w danej chwili.

# Obsługa sytuacji wyjątkowych 7

## Wprowadzenie 7.1

Podobnie jak większość projektowanych obecnie języków programowania, Loglan umożliwia obsługę sytuacji wyjątkowych. Przez *sytuację wyjątkową* rozumiemy wystąpienie w czasie wykonania programu zdarzenia wymagającego specjalnej reakcji. Zdarzeniem takim może być np. błąd spowodowany dzieleniem przez zero, a także takie zdarzenia nie będące błędami, które programista uzna za wyjątkowe, np. napotkanie końca pliku w czasie jego kopiowania itp.

W Loglanie do obsługi sytuacji wyjątkowych służy mechanizm *sygnałów*. Wystąpienie sytuacji wyjątkowej jest zgłaszane przez wysłanie sygnału odpowiedniego dla tej sytuacji. Moduł, w którym wystąpiła sytuacja wyjątkowa może nie być odpowiedni do jej obsługi. Na przykład, jeśli argumentem funkcji obliczającej pierwiastek kwadratowy jest liczba ujemna, zdarzenie to można uznać za wyjątkowe. Obsługa takiego wyjątku wewnątrz funkcji obliczającej pierwiastek nie jest zazwyczaj dobrym rozwiązaniem, bowiem błąd został spowodowany wcześniej — w module wywołującym tę funkcję, w module, który ten moduł wywołał lub jeszcze wcześniej. Dlatego też w Loglanie przyjęto, iż sygnały są przekazywane wzdłuż łańcucha dynamicznego kończącego się obiektem, w którym wystąpiła sygnalizacja sytuacji wyjątkowej (pojęcie łańcucha dynamicznego zdefiniowaliśmy w p. 6.1). Sygnał jest przekazywany do najbliższego w łańcuchu dynamicznym egzemplarza modułu, który zawiera odpowiedni *moduł obsługi sygnału*.

Metoda przekazywania sygnału jest następująca. Załóżmy, że sygnał **s** został wysłany z egzemplarza modułu **M**. Jeśli w module **M** jest zdefiniowany moduł obsługi sygnału **s**, to jest wykonywany ten właśnie moduł obsługi. W przeciwnym razie:

— jeśli **M** jest klasą, blokiem, funkcją lub procedurą, to sygnał **s** jest przekazywany do egzemplarza modułu wołającego **M** (czyli do modułu,

w którym była wykonana odpowiednia instrukcja **new**, **block**, **call** lub wywołanie funkcji);

— jeśli **M** jest modułem obsługi sygnału, to sygnał **s** jest przekazywany do egzemplarza modułu zawierającego jego deklarację;

— jeśli **M** jest samodzielnym współprogramem, to cały program zostaje zakończony.

Moduł obsługi sygnału można porównać z procedurą (jak to pokażemy w p. 7.2 — w przypadku prefiksowania z procedurą wirtualną). Instrukcje modułu obsługi sygnału są wykonywane w środowisku modułu zawierającego ten moduł obsługi, zmodyfikowanym o parametry formalne sygnału. Wśród tych instrukcji są dostępne trzy specyficzne instrukcje określające sposób kontynuacji programu po obsłudze wyjątku:

— **return** — oznacza normalną kontynuację, czyli przekazanie sterowania do egzemplarza modułu, który wysłał sygnał i wznowienie wykonywania tego egzemplarza począwszy od instrukcji następującej bezpośrednio po tej, która spowodowała wysłanie sygnału;

— **wind** — powoduje zakończenie wykonywania egzemplarza modułu, który wysłał sygnał oraz wszystkich poprzedzających ją w łańcuchu dynamicznym, aż do egzemplarza modułu zawierającego moduł obsługi sygnału (bez tego obiektu);

— **terminate** — powoduje zakończenie wykonania wszystkich obiektów, które kończy instrukcja **wind** wraz z egzemplarzem modułu, w którym nastąpiła obsługa sygnału.

Jeżeli żadna z tych instrukcji nie wystąpi w module obsługi, przyjmuje się, że jest on kończony instrukcją **terminate**.

W modułach kończonych w sposób wymuszony przez wykonanie instrukcji **wind** lub **terminate** istnieje możliwość wykonania *instrukcji końcowych* przewidzianych specjalnie na tę okazję. Te instrukcje omówimy w p. 7.3.

W Loglanie są także zdefiniowane *sygnały standardowe* odpowiadające błędom występującym w czasie wykonania programu. Sygnały te są wysyłane automatycznie w chwili wystąpienia błędu. Standardowa obsługa takiego sygnału polega na zakończeniu wykonywania programu. Jest jednak możliwe zdefiniowanie własnych modułów obsługi również w tej sytuacji. W modułach obsługi sygnałów systemowych nie może wystąpić instrukcja **return**. Loglan definiuje następujące sygnały standardowe:

— **accerror** — oznacza dostęp do atrybutów nieistniejącego obiektu albo błąd w wyrażeniu **x qua A** (gdy wartością **x** jest **none** lub obiekt wskazywany przez **x** nie należy do typu **A**);

— memerror — oznacza brak pamięci do utworzenia nowego obiektu;
— numerror — oznacza błąd numeryczny (np. przekroczenie zakresu liczb, dzielenie przez zero itp.);
— logerror — oznacza dowolny błąd wykrywany w trakcie wykonania programu związany z użyciem instrukcji niezgodnym z przeznaczeniem (np. przekazanie sterowania w sposób niezgodny z regułami Loglanu, próba usunięcia obiektu aktywnego itp.);
— conerror — oznacza, iż został przekroczony zakres indeksów tablicy lub, że zakresy tablicy są niepoprawne;
— syserror — oznacza dowolny rodzaj błędu sygnalizowanego przez system operacyjny (np. błąd wejścia/wyjścia, błąd parzystości itp.).

W konkretnych implementacjach języka mogą wystąpić także inne rodzaje sygnałów standardowych.

## Rozwiązywanie układu równań liniowych z macierzą trójkątną — błędy wykonania jako sytuacje wyjątkowe 7.2

### *Sformułowanie problemu* 7.2.1

Dana jest rzeczywista macierz trójkątna górna $\mathbf{A}$ rozmiaru $n \times n$ oraz $n$-elementowy wektor $\boldsymbol{b}$ liczb rzeczywistych. Napisz funkcję, której wynikiem jest taki wektor $\boldsymbol{x}$, że $\mathbf{A} * \boldsymbol{x} = \boldsymbol{b}$.

### *Dyskusja zadania* 7.2.2

Z problemem rozwiązywania układu równań tego typu mieliśmy już do czynienia w p. 2.3. Zakładaliśmy tam jednak, że wszystkie elementy przekątniowe macierzy $\mathbf{A}$ są różne od zera. Pojawia się naturalne pytanie: co należy uczynić, gdy procedura zostanie wywołana z parametrem aktualnym nie spełniającym tego warunku? W językach programowania, które nie zawierają mechanizmu obsługi sytuacji wyjątkowych, najczęstszym rozwiązaniem jest sprawdzenie, czy elementy przekątniowe są różne od zera i ewentualna sygnalizacja błędu np. przez zwrócenie specjalnej wartości funkcji. Rozwiązanie takie rodzi oczywiście problemy metodologiczne związane z wyróżnieniem specjalnej wartości, sprawdzaniem przez użytkownika, czy w wyniku wywołania funkcji nie została ta wartość otrzymana itd. My zaprezentujemy rozwiązanie, które korzysta z loglanowego mechanizmu sygnałów. Zauważmy, że podanie nieprawidłowych danych spowoduje błąd dzielenia przez zero (por. p. 2.3.4). W zasadzie można byłoby uznać, że jest to wystarczająca reakcja na błąd,

jeśli tylko jest znany kontekst, w którym następuje wywołanie. Jednak jeśli funkcja rozwiązująca układ równań ma być wykorzystywana przez wiele programów, to kontekst ten nie jest możliwy do przewidzenia. Dlatego też zdefiniujemy osobny sygnał, jednoznacznie wskazujący na rodzaj popełnionego błędu.

Z metodologicznego punktu widzenia takie rozwiązanie jest znacznie lepsze — użytkownik zdefiniowanej przez nas funkcji może być pewien, że ewentualny błąd dzielenia przez zero nie jest sygnalizowany przez tę funkcję. Nie musi on w ogóle wiedzieć, że podanie nieprawidłowych danych może spowodować taki błąd.

### *Omówienie rozwiązania* 7.2.3

Aby zgromadzić wszystkie niezbędne definicje w ramach jednego modułu założymy, że funkcja rozwiązująca układ równań jest zawarta w klasie o nazwie **układ_równań**. Użytkownik, który chce stosować tę funkcję będzie musiał prefiksować tą klasą swój moduł.

Zacznijmy od zdefiniowania odpowiedniego sygnału. W Loglanie sygnały są definiowane za pomocą klauzuli **signal**, np.

```
signal zero_na_przekątnej.
```

Ogólna postać tej klauzuli jest następująca:

```
signal s1(p1), s2(p2) ,..., sk(pk);
```

gdzie **s1**,...,**sk** są nazwami deklarowanych sygnałów, zaś **p1**,...,**pk** — ich parametrami formalnymi.

Sygnały standardowe nie muszą być deklarowane.

Sygnały są wysyłane za pomocą instrukcji **raise si(ai)**, gdzie **si** jest nazwą sygnału, zaś **ai** — listą parametrów aktualnych, odpowiadających parametrom formalnym **pi**.

Zastanówmy się, jak powinna wyglądać deklaracja funkcji rozwiązującej układ równań. Zgodnie z uwagami zawartymi w dyskusji zadania powinna ona przechwytywać sygnał oznaczający dzielenie przez zero i w czasie obsługi tego sygnału wysyłać zdefinowany wcześniej sygnał **zero_na_przekątnej**. Musimy zatem zadeklarować odpowiedni moduł obsługi sygnału **numerror** (odpowiadającego w naszej sytuacji dzieleniu przez zero).

W Loglanie deklaracja modułów obsługi sygnałów może wystąpić w części deklaracyjnej dowolnego modułu, przy czym — jeśli występuje — musi kończyć część deklaracyjna. Deklaracja modułów obsługi przyjmuje następującą postać:

```
handlers when s1: l1;
        ...
        when sk: lk;
        others l
        end handlers;
```

gdzie s1,...,sk są nazwami sygnałów (wyspecyfikowanych za pomocą klauzuli **signal**), zaś l1,...,lk, l są ciągami instrukcji. Wewnątrz listy instrukcji li (1<=i<=k) mogą wystąpić odwołania do parametrów formalnych sygnału si. Parametry aktualne są ustalane analogicznie do wołania procedur z tym, że zamiast instrukcji **call** występuje instrukcja **raise**. Specyfikacja **others** oznacza wszystkie sygnały różne od s1,...,sk. Jeśli zatem przesłany sygnał nie jest żadnym z s1,...,sk oraz występuje specyfikacja **others**, to jest wykonywana lista instrukcji l.

Wracając do naszego zadania zauważmy, że moduł obsługi sygnału **numerror** wewnątrz funkcji rozwiązującej układ równań przyjmie taką oto postać:

```
handlers when numerror: raise zero_na _przekątnej;
        end handlers;
```

Zgodnie z uwagami zawartymi w p. 7.1, brak instrukcji kończącej wykonanie modułu obsługi sygnału jest równoważny wystąpieniu instrukcji **terminate**. Zdefiniowany przez nas moduł jest więc równoważny modułowi

```
when numerror: raise zero_na_przekątnej; terminate.
```

Definiowana klasa przyjmie następującą postać:

```
unit układ_równań: class;
    close rozwiąż;
    signal zero_na_przekątnej;
    unit rozwiąż: function ...;
        var suma: real, i,j: integer;
        handlers when numerror: ...;
                end handlers;
        begin ... end rozwiąż;
    ...
    end układ_równań;
```

Po uzupełnieniu treści funkcji **rozwiąż** (por. p. 2.3.4) klasa **układ_równań** rozwiązuje postawiony przez nas problem. Dzięki połączeniu obsługi sytuacji wyjątkowych z prefiksowaniem, możemy jednak wzbogacić powyższe rozwiązanie. Możemy mianowicie zdefiniować w klasie **układ_równań** moduł obsługi sygnału **zero_na_przekątnej**, przewidując typowe zakończenie obliczeń. Jak już wspomnieliśmy w p. 7.1, przy prefiksowaniu moduły

obsługi sygnałów zachowują się podobnie jak procedury wirtualne. Oznacza to, iż użytkownik chcący obsłużyć sygnał **zero_na_przekątnej** w sposób niestandardowy może zadeklarować własny moduł obsługi, redefiniując moduł z klasy **układ_równań**.

Standardową reakcją będzie wypisanie komunikatu o błędzie i zakończenie obliczeń. Klasę **układ_równań** uzupełnimy więc o specyfikację

```
handlers when zero_na_przekątnej:
        writeln(" zero na przekątnej macierzy");
        call endrun;
    end handlers;
```

Przyjrzyjmy się teraz przykładowym zastosowaniom klasy **układ_równań**.

(1)
```
pref układ_równań block
...
var b: boolean;
handlers when zero_na_przekątnej:
        b:=false;
        writeln("złe dane",
                "podaj nową macierz");
        wind;
    end handlers;
begin
do b:=true;
  ... (* wczytanie macierzy *) ...
  r:=rozwiąż ...
  if b then exit fi
  od;
...
end;
```

(2)
```
pref układ_równań block
...
var b: boolean;
begin
do b:=true;
  ... (* wczytanie macierzy *) ...
  r:=rozwiąż ...
  if b then exit fi
  od;
...
end;
```

W bloku prefiksowanym (1), modułem obsługi wyjątku zero_na_przekątnej będzie moduł zdefiniowany w tym bloku. Zatem w razie błędnych danych zostanie wypisany komunikat „złe dane, podaj nową macierz" i pętla **do...od** będzie się wykonywać aż do chwili, w której zostaną podane prawidłowe dane. Natomiast w bloku prefiksowanym (2), moduł obsługi wyjątku zero_na_przekątnej nie został zredefiniowany. Dlatego , gdy poda się złe dane, wykona się moduł zdefiniowany w klasie **układ_równań**. Zostanie wypisany komunikat „zero na przekątnej macierzy" i program zakończy działanie.

Zwróćmy uwagę na ważną cechę Loglanu, która wynika z tego przykładu. Z jednej strony połączenie obsługi sytuacji wyjątkowych i prefiksowania stwarza bogate możliwości definiowania sytuacji wyjątkowych w językach problemowych wraz ze standardowymi metodami ich obsługi, z drugiej zaś strony umożliwia użytkownikom tych języków definiowanie obsługi odmiennej, odpowiadającej ich specyficznym potrzebom. W naszym przykładzie takim bardzo prostym językiem problemowym był język zawierający jako jedyną operację rozwiązywanie układu równań.

*Rozwiązanie* 7.2.4

```
unit układ_równań: class;
    close rozwiąż;
    signal zero_na_przekątnej;
    unit rozwiąż: function(n: integer,
          A: array_of array_of real, b: array_of real):
          array_of real;
        var suma: real, i,j: integer;
        handlers when numerror:
                  raise zero_na_przekątnej;
              end handlers;
        begin
        ... (* por. p. 2.3.4 *) ...
        end rozwiąż;
    handlers when zero_na_przekątnej:
              (* standardowa obsługa *)
              writeln(" zero na przekątnej macierzy");
              call endrun
          end handlers;
    end układ_równań;
```

## Problem plecakowy — epilog obiektu przy zakończeniu przez terminate lub wind 7.3

### *Sformułowanie problemu* 7.3.1

Dla danej tablicy różnych liczb całkowitych dodatnich i liczby całkowitej v wypisz ciąg indeksów i1, i2, ..., ik o własności A(i1)+A(i2)+...+A(ik) = v, lub odpowiednią informację, jeśli taki ciąg nie istnieje.

### *Dyskusja zadania* 7.3.2

Dla uproszczenia rozumowania założymy, że tablica A jest posortowana w ten sposób, że dla każdego i (lower(A)<=i<upper(A)) mamy A(i)<A(i+1). Przedstawiamy rozwiązanie rekurencyjne. Można je sformułować w sposób naturalny, używając pomocniczej procedury p o dwóch parametrach wejściowych s oraz k, będących liczbami całkowitymi. Wywołanie p(s,k) oznacza poszukiwanie ciągu indeksów, począwszy od k-tego, przy założeniu, że suma dotychczas przyjętych elementów jest równa s. Schemat procedury p jest następujący:

jeśli s + A(k) > v, to dotychczas znaleziona wartość jest zbyt duża i należy sprawdzić inne możliwości wycofując się z poprzednich wyborów;
w przeciwnym razie
jeśli s + A(k) = v, to ciąg indeksów został znaleziony; wypisujemy go i zatrzymujemy algorytm;
przyjmujemy, że k jest kolejnym indeksem,
zwiększamy s i rekurencyjnie sprawdzamy, czy dla tego wyboru istnieje dalszy ciąg indeksów.

Zauważmy, że ostatnia akcja wykona się tylko wówczas, gdy warunek s + A(k) = v kończący obliczenia nie został spełniony.

Mając procedurę p, można z łatwością zapisać algorytm, który rozwiązuje nasze zadanie. Należy w tym celu sprawdzić dla każdego k (lower(A)<=k<=upper(A)), czy procedura p znajdzie szukany ciąg indeksów startując z s = 0. Do rozwiązania pozostaje jeszcze problem wypisania ciągu indeksów w razie ich znalezienia. Korzystając tylko z procedur rekurencyjnych musielibyśmy zapamiętywać znalezione indeksy, usuwając niektóre z nich w razie wycofywania się z ewentualnych nieudanych wyborów. Tymczasem, aktualnie rozpatrywany ciąg indeksów jest dany jako ciąg wartości parametru k w kolejnych rekurencyjnych wołaniach procedury p. Jest więc on zawarty w łańcuchu dynamicznym. Suge-

ruje to, by znalezienie właściwego ciągu indeksów potraktować jako sytuację wyjątkową, której wystąpienie spowoduje zakończenie wykonywania obiektów łańcucha dynamicznego. W tej sytuacji można zdefiniować instrukcje końcowe, które zostałyby wykonane po zasygnalizowaniu sytuacji wyjątkowej. W naszym zadaniu taką instrukcją końcową będzie po prostu wypisanie wartości k. Zauważmy, że ze względu na sposób „zwijania" łańcucha dynamicznego (począwszy od ostatnio wykonywanego obiektu do obiektu, w którym jest obsługiwany wyjątek), indeksy zostaną wypisane w porządku malejącym.

### *Omówienie rozwiązania* 7.3.3

Procedurę p można uściślić w następujący sposób:

```
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;
    ...
    end p;
```

Musimy jeszcze zdefiniować instrukcje końcowe procedury p na wypadek zaistnienia sytuacji wyjątkowej. W Loglanie definiuje się je klauzulą last_will. Postać tej klauzuli jest następująca:

```
last_will: I;
```

gdzie I jest dowolnym ciągiem instrukcji, wśród których nie może wystąpić inner. Klauzula ta może wystąpić jedynie przed końcowym end modułu. Ciąg instrukcji I jest wykonywany tylko w razie kończenia działania obiektu spowodowanego wykonaniem instrukcji terminate lub wind. W modułach prefiksowanych są wykonywane kolejno instrukcje końcowe pochodzące z modułów ciągu prefiksowego.

Instrukcje końcowe procedury p zdefiniujemy za pomocą klauzuli

```
last_will: writeln(k);
```

Procedura rozwiązująca postawiony problem przyjmie taką oto postać:

```
unit problem_plecakowy: procedure(A: array_of integer,
                                  v: integer);
    signal znaleziony;
```

```
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

### *Sformulowanie 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.

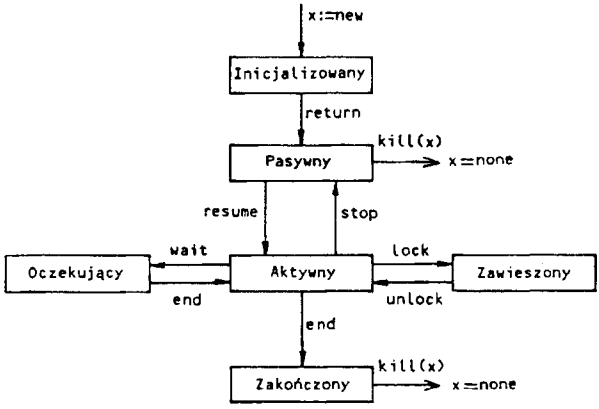
Na rysunku 8.1 przedstawiamy w sposób schematyczny wpływ operacji wykonywanych przez procesy na ich zachowanie. Zauważmy, że podobnie jak w przypadku współprogramowych operacji attach oraz detach, operacje związane ze zmianą stanu procesu dotyczą wykonującego je procesu — niekoniecznie tego, w którym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

## 8.5 Rozwiązywanie dużych układów równań liniowych — zastosowanie współbieżności i prefiksowania

### 8.5.1 *Sformułowanie problemu*

Jest dana kwadratowa macierz rzeczywista **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).

### 8.5.2 *Dyskusja zadania*

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: $\boldsymbol{x} = \mathbf{B} * \boldsymbol{x} + \mathbf{c}$. Znając nową postać, kolejne przybliżenia rozwiązania znajdujemy ze wzoru:

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

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 ($i \neq j$) 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**.

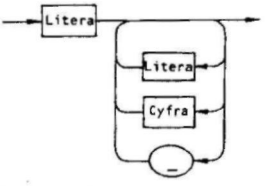
(4) Jest zdefiniowany standardowy typ **process**. Do tego typu należą wszystkie obiekty procesów. Znaczeniem wyrażenia **this process** jest wskaźnik do obiektu procesu obliczającego to wyrażenie (tzw. bieżącego procesu).

(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.

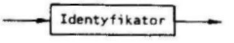
(6) Zakończony obiekt procesu jest równoważny zakończonemu obiektowi klasy. Może on zostać usunięty z pamięci za pomocą operatora **kill**. Także proces zawieszony może zostać usunięty z pamięci. W tym przypadku wraz z jego obiektem są usuwane z pamięci wszystkie obiekty z jego łańcucha dynamicznego.

(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.

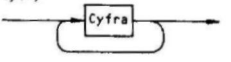
Identyfikator



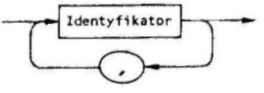

IdentStałej



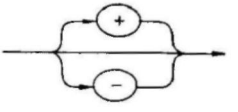

Cyfry



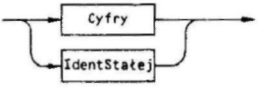

Identyfikatory



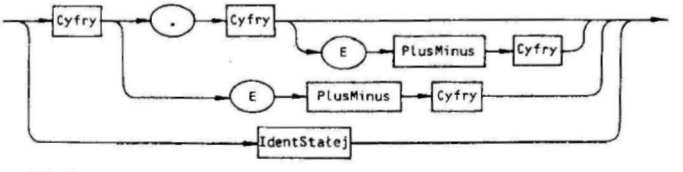

PlusMinus



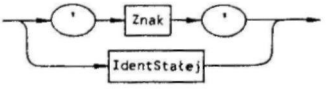

StałaCałkowita



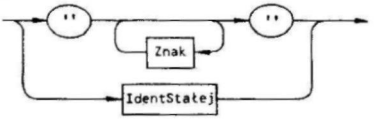

StałaRzeczywista



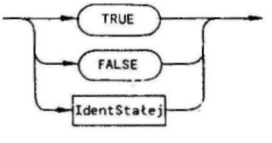

StałaZnakowa



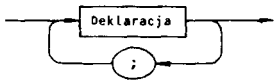

StałaTekstowa



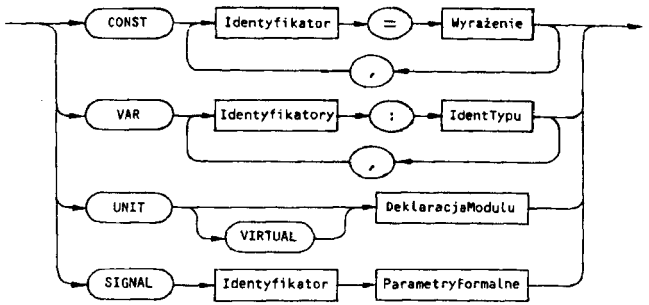

StałaLogiczna



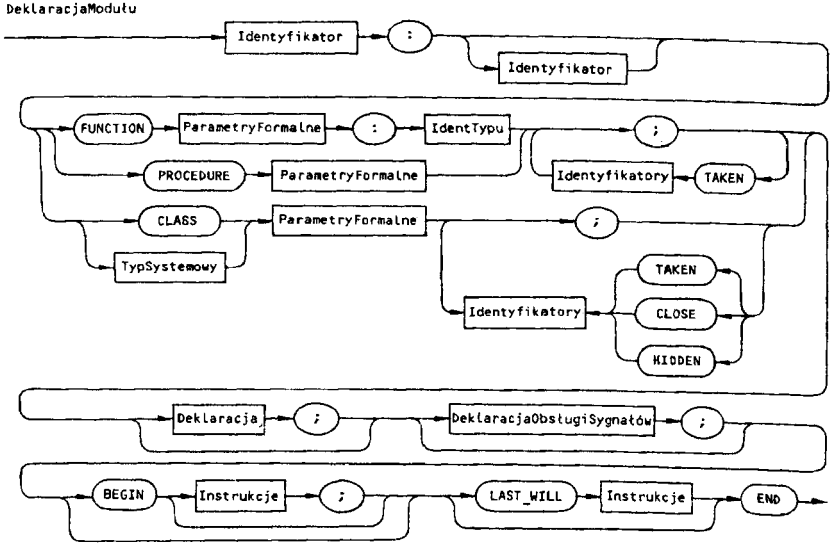
Dekralacje



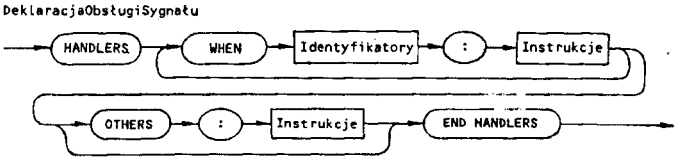

Deklaracje



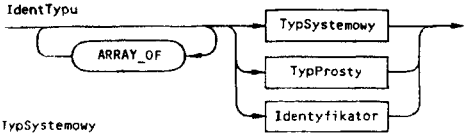

DeklaracjaModułu



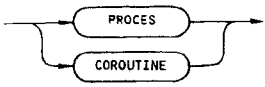

DeklaracjaObsługiSygnału



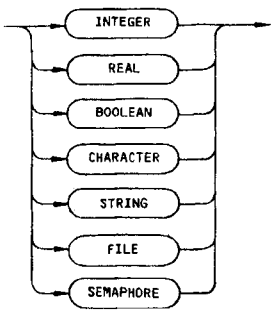
IdentTypu



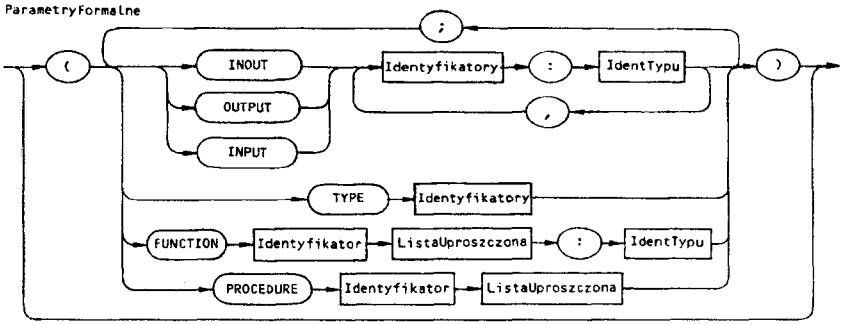

TypSystemowy



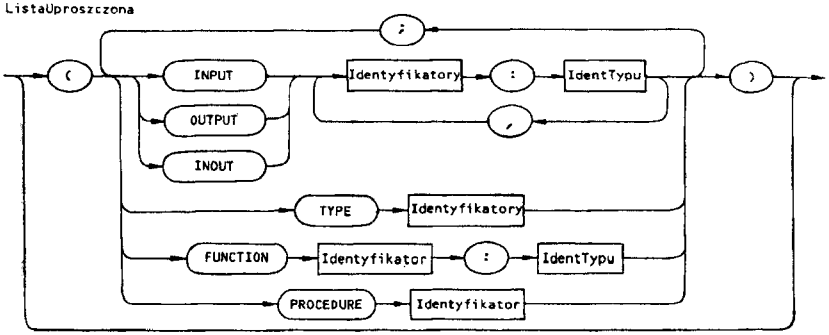

TypProsty



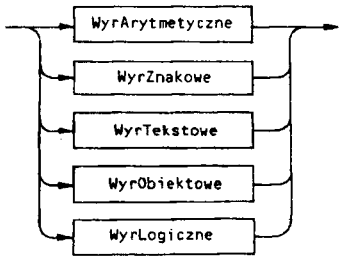

ParametryFormalne



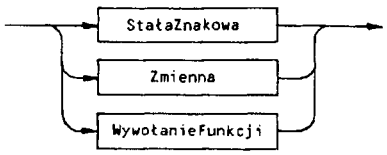

ListaUproszczona



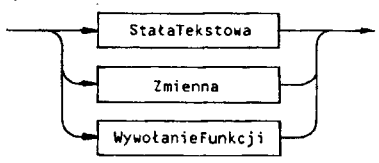
Wyrażenie



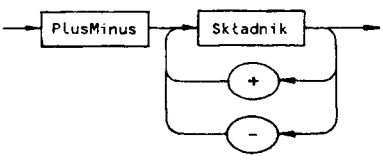

WyrZnakowe



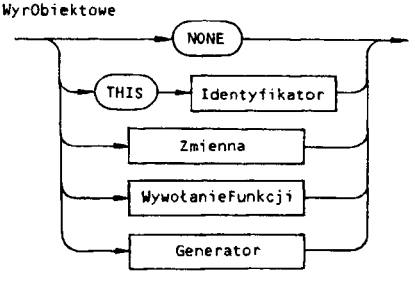

WyrTekstowe



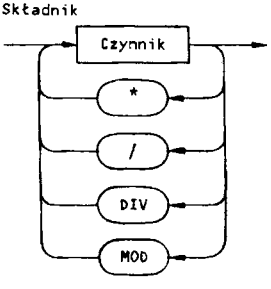

WyrArytmetyczne



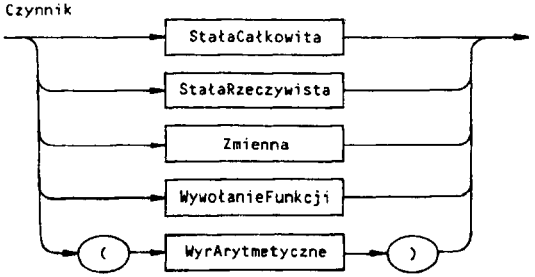

WyrObiektowe

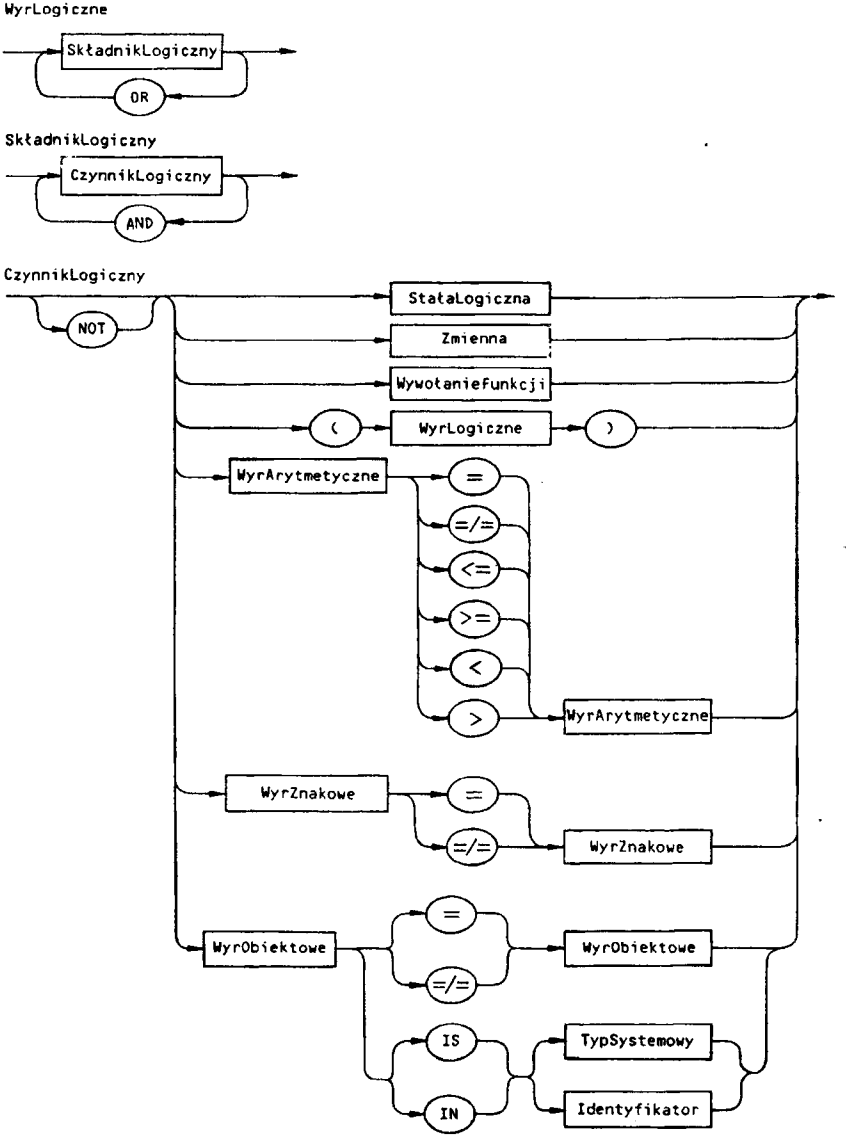



Składnik

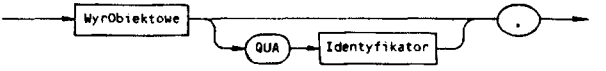



Czynnik

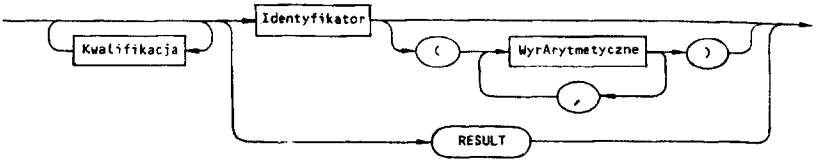



WyrLogiczne
SkładnikLogiczny
OR
SkładnikLogiczny
CzynnikLogiczny
AND
CzynnikLogiczny
NOT
StałaLogiczna
Zmienna
WywołanieFunkcji
(
WyrLogiczne
)
WyrArytmetyczne
=
=/=
<=
>=
<
>
WyrArytmetyczne
WyrZnakowe
=
=/=
WyrZnakowe
WyrObiektowe
=
=/=
WyrObiektowe
IS
IN
TypSystemowy
Identyfikator

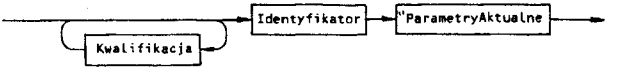
Kwalifikacja



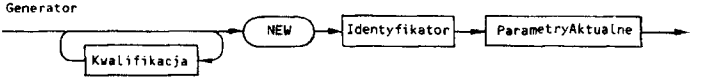

Zmienna



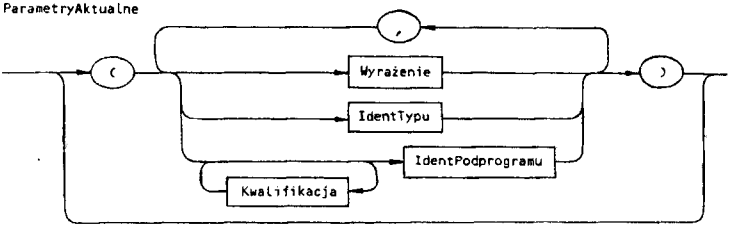

WywołanieFunkcji



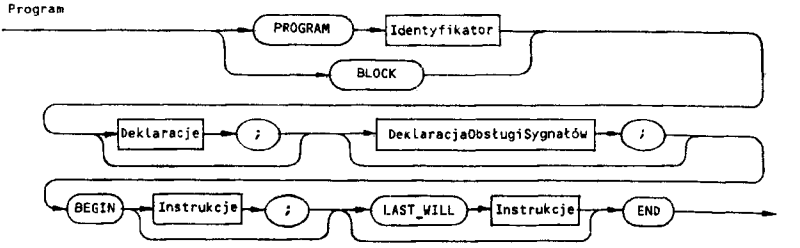

Generator



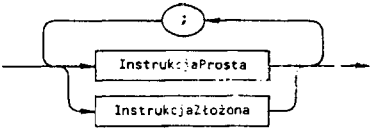

ParametryAktualne



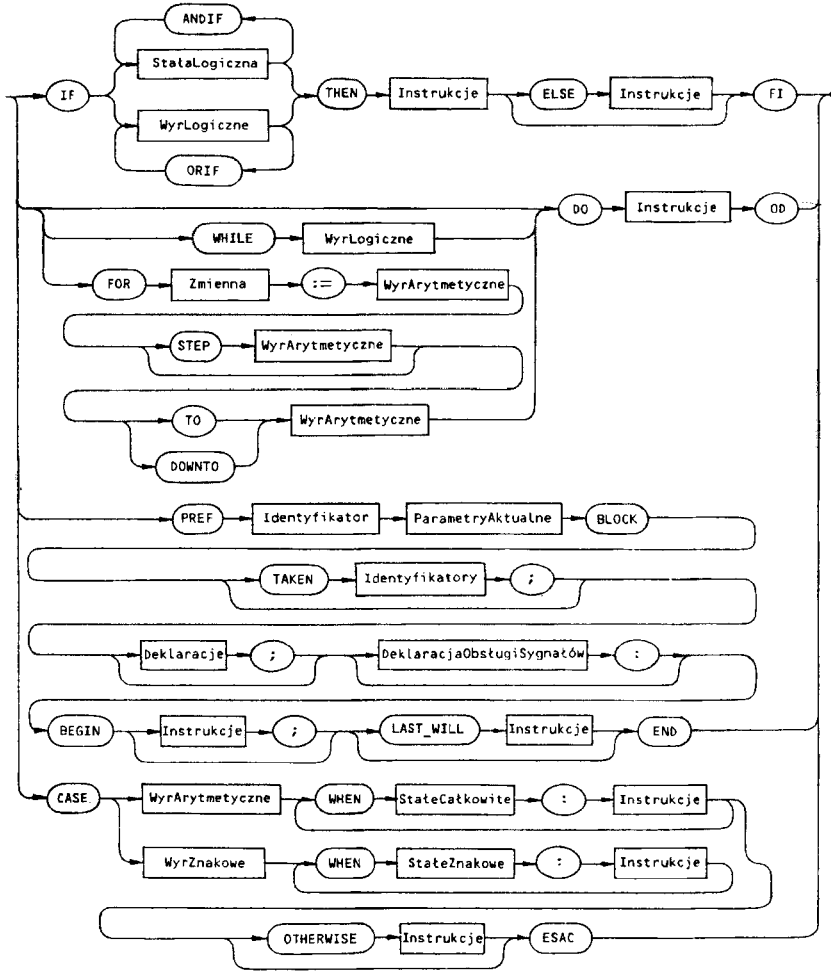

Program



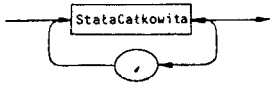

Instrukcje



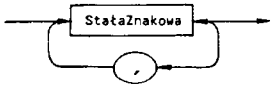
InstrukcjaZłożona

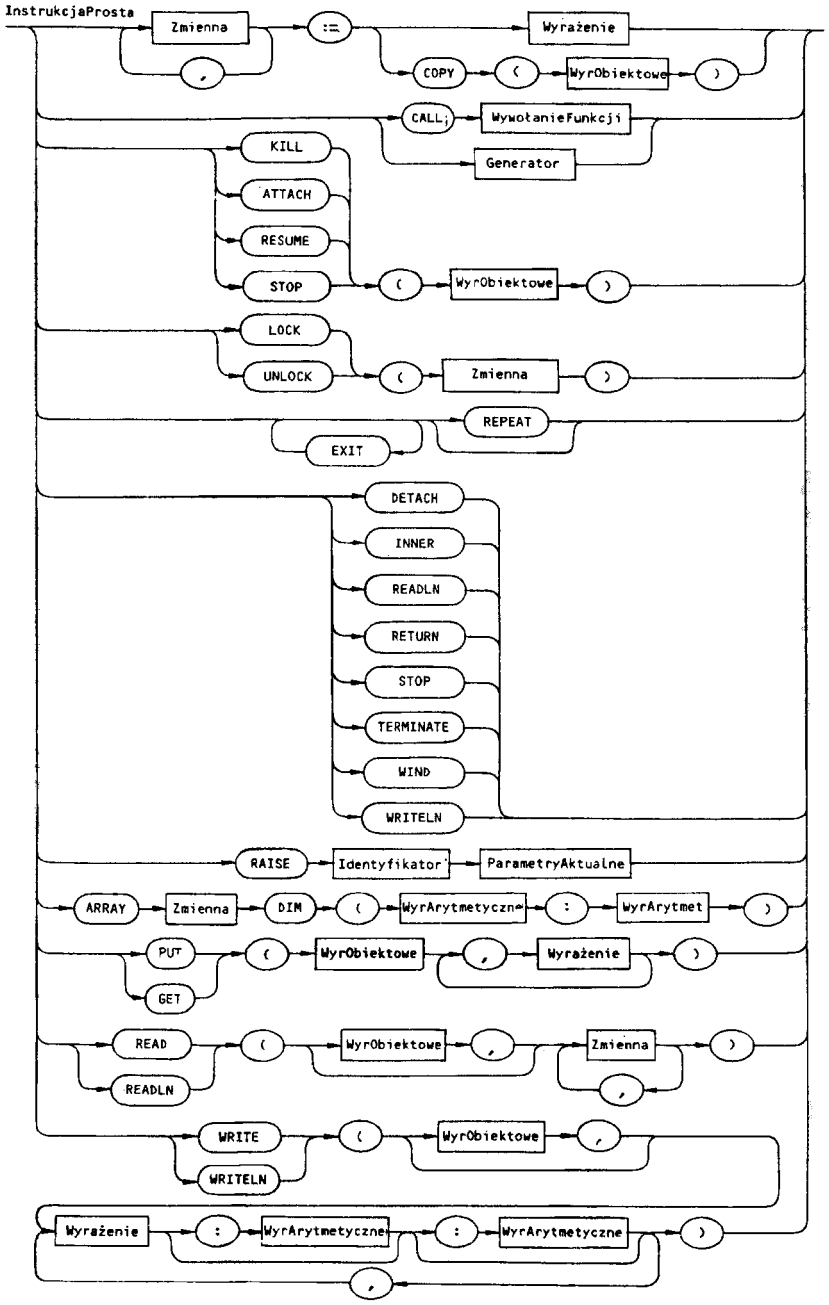



StałeCałkowite




StałeZnakowe




InstrukcjaProsta
Zmienna
,
:=
Wyrażenie
COPY
(
WyrObiektowe
)
CALL;
WywołanieFunkcji
Generator
KILL
ATTACH
RESUME
STOP
(
WyrObiektowe
)
LOCK
UNLOCK
(
Zmienna
)
EXIT
REPEAT
DETACH
INNER
READLN
RETURN
STOP
TERMINATE
WIND
WRITELN
RAISE
Identyfikator
ParametryAktualne
ARRAY
Zmienna
DIM
(
WyrArytmetyczne
:
WyrArytmet
)
PUT
GET
(
WyrObiektowe
,
Wyrażenie
)
READ
READLN
(
WyrObiektowe
,
Zmienna
,
)
WRITE
WRITELN
(
WyrObiektowe
,
Wyrażenie
:
WyrArytmetyczne
:
WyrArytmetyczne
)
,

# 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: real): 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


## A

accept 166
array dim 37
Atrybut 43, 45, 76, 77, 79
–, ochrona 45, 91, 104
attach 104, 135

## B

Blok 26
– prefiksowy 67

## C

call 22
case 23
Ciąg prefiksowy 75
close 46
copy 22, 41

## D

Deklaracja moduł, 19
detach 104, 135
disable 166
do 24

## E

enable 166
eof 30
eoln 30
exit...exit repeat 24, 25

## F

for 24
Funkcja 26, 78

## G

get 30

## H

hidden 91

## I

Identyfikator 14
if 23
in 76
inner 63
inout 27
input 27
Instrukcja iteracji 24
– kontynuacji po obsłudze wyjątku 122
– przekazywania sterowania 104
– przypisania 22
– warunkowa 23
Instrukcje końcowe 122, 129
is 76

## J

Jednostki leksykalne 14

## K

kill 22, 29, 41, 104, 136
Klasa 40
–, atrybuty 64
–, deklaracja 41, 64
–, tworzenie obiektu 43, 44, 64
Komentarz 14

## L

last will 129

lock 136

## Ł

Łańcuch dynamiczny 121
– – współprogramu 105

## M

main 26, 105
Maska procedur 166
Moduł 20
–, deklaracja 21, 63
–, egzemplarz 21
–, instrukcja 64, 21
–, nagłówek 21
– obsługi sygnałów 124
– prefiksowany 63
– prefiksujący 63
– rozszerzany 62
– rozszerzony 63

## N

new 21, 43, 103, 135
none 21, 41

## O

Obce wołanie procedury 165
Obiekt 20
–, kopiowanie 22
– pusty 21
–, tworzenie 21
–, usuwanie 22
Odległy dostęp 43, 76, 104, 165
Ogranicznik 14
open 29
otherwise 23
output 27

## P

Parametr 52
– aktualny 27, 52, 53, 68, 124
– będący podprogramem 52, 163
– – typem 52
– formalny 26, 52, 53, 63, 124
–, przekazywanie 52, 103
Plik 28
– binarny 29
–, instrukcja otwarcia 29
–, – zamknięcia 29
– tekstowy 30
Podklasa 75
Podprogram 26
Podprogram wirtualny 78, 79, 163
Podtyp 74
Prefiks 63
Prefiksowanie 62
Procedura 26, 78
– odblokowana 166
– zablokowana 166
Proces, deklaracja 135
–, tworzenie obiektu 135
– współbieżny 135
put 30

## Q

qua 76

## R

raise 124
readln 30
read 30
reset 29
result 27
resume 136
return 27, 44, 103, 122, 166
rewrite 29
Rozszerzenie 62

## S

Semafor 136
signal 124
Słowo kluczowe 14
Stała 20
stop 136
Sygnał 121
–, deklaracja 124
–, wysłanie 124
Sygnały standardowe 122
Sytuacja wyjątkowa 121

## Ś

Środowisko syntaktyczne modułu 63

## T

Tablica dynamiczna 33
–, tworzenie obiektu 37
–, – tablic 33
taken 91
terminate 122
this 50, 77
this coroutine 106
this process 136

ts 136
Typ aktualny 53
- boolean 15
- character 18
- coroutine 106
- danych 10
- -, nosnik 10
- formalny 53, 162
- - własny nagłówka 164
- - zewnętrzny 164
- integer 16
- klasowy 40
- obiektowy 21
- pierwotny 15
- plikowy 28
- process 136
- real 16
- statycznie określony 163
- string 18
- tablicowy 33
type 52

## U

unlock 136

## W

wait 136
while 24
wind 122
write 30
writeln 30
Współprogram 103, 135
- aktywny 104
-, deklaracja 103
-, tworzenie obiektu 103
- zawieszony 104

## Z

Zmienna 20
- wskaźnikowa 21
Znaki podstawowe 14
- specjalne 14

# 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 доставлыают 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 — Logłan
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