Biblioteka
Inżynierii Oprogramowania

Redaktor serii BARBARA OSUCHOWSKA

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

ISBN 83-204-1295-1

Tytuł dotowany przez
Ministra Edukacji Narodowej

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.

Spis treści

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

$$\mathsf{T}(\mathsf{E}) = (D;\ \mathsf{f}_1, \mathsf{f}_2, \ldots, \mathsf{r}_1, \mathsf{r}_2, \ldots)$$

gdzie T jest nazwą typu, E, jeśli występuje, jest typem parametrycznym, D — zbiorem wartości zwanym też *nośnikiem* typu, $\mathsf{f}_1, \mathsf{f}_2, \ldots$ — funkcjami oraz $\mathsf{r}_1, \mathsf{r}_2, \ldots$ — 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

$$\mathsf{stosy}(\mathsf{E}) = (\mathit{stosy};\ \mathsf{włóż}, \mathsf{zdejmij}, \mathsf{wierzchołek}, \mathsf{pusty})$$

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 odstepu.

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 nastepująco:

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 I1 fi;
(2) if b then I1 else I2 fi;
(3) case w
        when w1,...,wn:I1;
        ...
        when z1,...,zk:Im;
        otherwise I
        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 I, I1,...,Im są ciągami instrukcji.

Wykonanie instrukcji **if** polega na sprawdzeniu warunku b i w zależności od jego wartości na wykonaniu ciągu instrukcji I1 (gdy wartością b jest **true**) lub, w wypadku drugiego wariantu instrukcji **if**, wykonaniu ciągu instrukcji I2 (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 I1;

...

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

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

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 $n \times n$ 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;
```

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

Sformułowanie problemu 2.3.1

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.

Dyskusja zadania 2.3.2

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

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

Wartość niewiadomej x_n wyznaczamy z ostatniego równania

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

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

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

Omówienie rozwiązania 2.3.3

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;
```

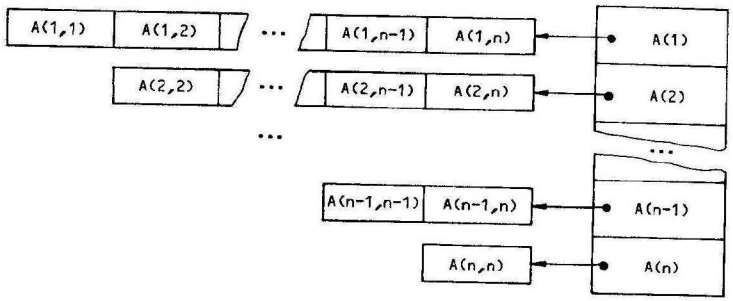
Wartością zmiennej A może być wskaźnik do tablicy, której elementami są wskaźniki do obiektów tablic liczb rzeczywistych. Jedną instrukcją tworzącą obiekt tablicy można utworzyć jedynie tablicę jednowymiarową

```
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 $\boldsymbol{b}$. Obliczony wektor $\boldsymbol{x}$ 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).
```

3.2.2 *Dyskusja zadania*

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.

3.2.3 *Omówienie rozwiązania*

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;
```


Binarne drzewa wyważone — rekurencyjne definiowanie typów danych 3.3

Sformułowanie problemu 3.3.1

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.

5 Prefiksowanie — operacja rozszerzania modułów

5.1 Wprowadzenie

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