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

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

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

oraz

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

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

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

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

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


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

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

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

Znajdowanie okręgu opisanego na trójkącie — język problemowy 5.2

Sformułowanie problemu 5.2.1

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

Dyskusja zadania 5.2.2

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

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

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

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

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

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

gdzie

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

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

Omówienie rozwiązania 5.2.3

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

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

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

Wewnątrz klasy **Punkty** zdefiniujemy także właściwe operacje

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

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

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

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

Atrybutami okręgu będzie jego środek i promień

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

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

Trójkąty będą zdefiniowane przez trzy punkty reprezentujące wierzchołki

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

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

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

```
pref geometria block;
```

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

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

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

Rozwiązanie 5.2.4

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



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

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

    begin

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

Sortowanie stogowe — łączenie instrukcji w modułach prefiksowanych 5.3

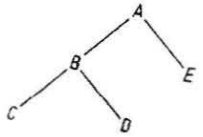
Sformułowanie problemu 5.3.1

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

Dyskusja zadania 5.3.2

Z sortowaniem tablic mieliśmy już do czynienia wcześniej, w p. 4.3, gdzie podaliśmy bardzo prosty algorytm sortowania. Teraz przedstawimy algorytm znacznie ciekawszy i bardziej efektywny, znany pod nazwą sortowania stogowego. W algorytmie tym podstawową strukturą danych jest stóg. Przypomnijmy, że stogiem nazywamy wyważone drzewo binarne, którego każde poddrzewo spełnia następujący warunek: wartość elementu związanego z korzeniem jest większa niż wartość elementów znajdujących się w poddrzewach.

Przykład stogu pokazano na rys. 5.1.



RYS. 5.1 Przykład stogu

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

Omówienie rozwiązania 5.3.3

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

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

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

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

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

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

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

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

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

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


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

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

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

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

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

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

Cały algorytm sortowania przyjmie teraz postać procedury

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

Rozwiązanie 5.3.4

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

Dalsze wiadomości o prefiksowaniu 5.4

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

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

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

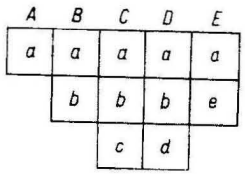
$$C_1 \supseteq C_2 \supseteq \ldots \supseteq C_k$$

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

Rozważmy następującą hierarchię klas:

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

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



RYS. 5.2 Struktura obiektów przykładowej hierarchii typów danych

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

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

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

```
x in A
```

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

```
x is A
```

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

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

```
x qua B.atr
```

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

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

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

```
this A.atr
```

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

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

oraz

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

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

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

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

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

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

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

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

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

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

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

Podprogramy wirtualne mogą zatem realizować następujące zadania:

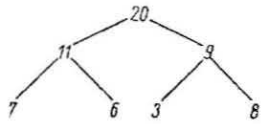
— stopniowe precyzowanie procedur i funkcji w kolejnych modułach ciągu prefiksowego;

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

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

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

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



RYS. 5.3 Przykładowa hierarchia klas

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

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

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

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

(3) muszą należeć do jednego ciągu podprogramów wirtualnych o tej samej nazwie, spełniających warunki (1) i (2), zadeklarowanych w następujących po sobie modułach spójnego fragmentu ciągu prefiksowego.

Brak deklaracji podprogramu wirtualnego, ograniczenie dostępu specyfikacją hidden (por.p. 5.7.3) lub niespełnienie warunków (1) i (2) powoduje przerwanie ciągu podprogramów wirtualnych.

Rozważmy na przykład liniową hierarchię klas

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

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

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

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

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

Sformułowanie problemu 5.5.1

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

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

gdzie ∪ oznacza sumę zbiorów obiektów.

Dyskusja zadania 5.5.2

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

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



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

Powyższe definicje zapewniają następujące zawierania zbiorów obiektów:

człowiek ⊇ dorosły ∪ kobieta ∪ mężczyzna ∪ dziecko
dorosły ⊇ kobieta ∪ mężczyzna

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

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

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

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

Omówienie rozwiązania 5.5.3

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

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

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

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

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

jest równoważny warunkowi

```
this człowiek in dorosły.
```

Warunek instrukcji (1) może więc przyjąć następującą postać:

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

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

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

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

```
var baza: array_of człowiek.
```

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

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


Zauważmy, iż badając warunek

```
baza(i) qua dorosły.liczba_dzieci>0
```

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

Rozwiązanie 5.5.4

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

Zbiory rozmyte — użycie podprogramów wirtualnych 5.6

Sformułowanie problemu 5.6.1

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

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

gdzie

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

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

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

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

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

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

Dyskusja zadania 5.6.2

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

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

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

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

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

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

Omówienie rozwiązania 5.6.3

Zbiór rozmyty zdefiniujemy jako klasę:

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

Przykładowa operacja sumowania zbiorów rozmytych jest definiowana następująco:

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

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

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

Rozważmy pewne zastosowanie tak zdefiniowanej klasy.

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

Przeanalizujmy działanie tego fragmentu programu. W wyniku prefiksowania bloku klasą wszystkie wielkości zadeklarowane w klasie prefiksującej stają się bezpośrednio dostępne w bloku. Wykonanie takiego bloku polega na utworzeniu jego obiektu i wykonaniu jego listy instrukcji. Lista ta jest połączeniem listy instrukcji klasy prefiksującej (w naszym przypadku jest to lista pusta) i listy instrukcji bloku. Instrukcje klasy prefiksującej są wykonywane w środowisku tej klasy. Dostępne są wielkości zadeklarowane w klasie i jej otoczeniu syntaktycznym. Instrukcje bloku wykonują się w połączonym środowisku bloku i klasy. W środowisku tym są określone wszystkie wielkości zdefiniowane w bloku i jego środowisku oraz te spośród dostępnych w klasie (zadeklarowanych w niej samej lub jej środowisku), które nie zostały zasłonięte ponowną deklaracją w bloku.

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

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

Po wykonaniu instrukcji zostaną zatem wypisane liczby:

```
0.8 0.8 0.2 0.8.
```

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

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

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

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

w wyniku jego działania zostałyby wypisane liczby

0.8 0.8 0.16 0.9664.

Rozwiązanie 5.6.4

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

Tablice rozproszone — ochrona atrybutów przy prefiksowaniu 5.7

Sformułowanie problemu 5.7.1

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

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

gdzie

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

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

Dyskusja zadania 5.7.2

Implementacja tablic rozproszonych sprowadza się do rozwiązania następujących dwóch zadań:

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

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

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

Omówienie rozwiązania 5.7.3

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

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

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

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

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

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

gdzie

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

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

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

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

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

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


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

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

taken wstaw, początek, kolejka;

Ogólnie, specyfikacja

taken a1,...,an;

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

hidden a1,...,ak;

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

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

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

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

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

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

Rozwiązanie 5.7.4

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



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

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

Sformułowanie problemu 5.8.1

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

Dyskusja zadania 5.8.2

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

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

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

posortuj E według malejących kosztów tak, by
  $E = (e_1, \ldots, e_n)$, gdzie $w(e_1) >= w(e_2) >= \cdots >= w(e_n)$;
$S := \emptyset$;
dla i od 1 do n powtarzaj:
  jeśli $S \cup \{e_i\} \in J$ to $S := S \cup \{e_i\}$.

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

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

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

Omówienie rozwiązania 5.8.3

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

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

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

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

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

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

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

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

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


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

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

Graf będzie dany za pomocą tablicy:

```
var graf: array_of krawędź;
```

Tablicę tę utworzymy następująco:

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

Wynikiem funkcji w jest atrybut krawędzi, reprezentujący jej koszt

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

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

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

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

gdzie

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

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

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

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

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

```
var las: Find_Union;
unit virtual warunek: function: boolean;
    var k,l: integer;
    begin
    k:= las.Find(E(i) qua krawędź.i1);
    l:= las.Find(E(i) qua krawędź.i2);
    if k=/=l then (* wierzchołki krawędzi
                  leżą w różnych zbiorach *)
      result:=true; l:=las.Union(k,l) fi
    end warunek;
```


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

Rozwiązanie 5.8.4

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

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

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



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

Podsumowanie 5.9

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Współprogramy 6

Wprowadzenie 6.1

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

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

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

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

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

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

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

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

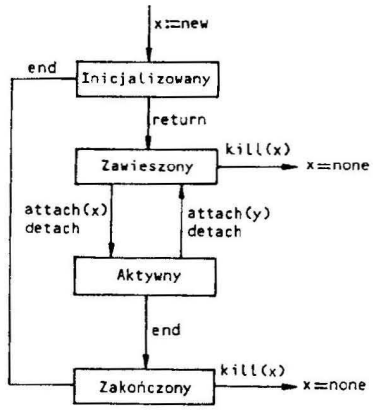
Widzimy zatem, że obiekt współprogramu, który zakończył fazę inicjalizacji znajduje się na przemian w stanach zawieszenia i aktywności. Jednocześnie, ponieważ uaktywnienie współprogramu wiąże się z zawieszeniem obecnie aktywnego, jest spełniony warunek, że w danej chwili dokładnie jeden obiekt współprogramu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

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

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

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

Aby opisać działanie współprogramów pomiędzy ich utworzeniem



RYS. 6.1 Schemat zmian stanów współprogramów

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

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

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

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

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

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

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

Symulacja licznika — podstawowe własności współprogramów 6.2

Sformułowanie problemu 6.2.1

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

Dyskusja zadania 6.2.2

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

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

Omówienie rozwiązania 6.2.3

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

> Jeśli cyfrą wskazywaną przez koło jest 9 — zmień swoją cyfrę na 0, przekaż impuls następnemu kołu; po powrocie sterowania przekaż je obiektowi, który nadesłał impuls.
>
> W przeciwnym razie — zwiększ cyfrę o 1 i zwróć sterowanie obiektowi, który je przekazał.

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

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

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

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

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

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

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

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

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

Inne koła tworzymy według identycznego wzorca.

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

Algorytm działania koła jest zgodny z wcześniej omówionym.

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

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

Rozwiązanie 6.2.4

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

Generowanie permutacji — zawieszanie działania algorytmów 6.3

Sformułowanie problemu 6.3.1

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

Dyskusja zadania 6.3.2

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

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

Omówienie rozwiązania 6.3.3

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

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



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

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

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

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

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

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

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

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

Rozwiązanie 6.3.4

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

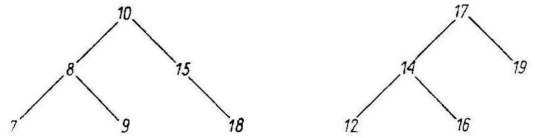
Scalanie ciągów liczb — korzystanie z częściowych wyników działania algorytmu 6.4

Sformułowanie problemu 6.4.1

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

Dyskusja zadania 6.4.2

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



RYS. 6.2 Przykłady drzew binarnych poszukiwań

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

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

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

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

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

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

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

Omówienie rozwiązania 6.4.3

Element drzewa binarnego zdefiniujemy jako typ klasowy.

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

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

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

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

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

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



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

Rozwiązanie 6.4.4

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

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

Wieże z Hanoi — współprogramy a procedury rekurencyjne 6.5

Sformułowanie problemu 6.5.1

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

— jednorazowo można przenosić tylko jeden krążek;
— żaden krążek nie może zostać położony na krążku o mniejszej średnicy.

Dyskusja zadania 6.5.2

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

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

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

Omówienie rozwiązania 6.5.3

Przyjrzyjmy się najpierw schematowi działania następującej procedury rekurencyjnej:

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

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

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

Utworzymy trójwymiarową tablicę obiektów współprogramu **Hanoi**:

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

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

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

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

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

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

Opisane przez nas rozwiązanie można zastosować także we wszystkich sytuacjach, w których występuje wielokrotne wywoływanie procedury rekurencyjnej z tymi samymi parametrami. Wystarczy wówczas z każdym takim wywołaniem związać odpowiedni obiekt współprogramu i zamienić wywołanie na wznowienie tego obiektu.

Rozwiązanie 6.5.4

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

Podsumowanie 6.6

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

programu może być aktywny co najwyżej jeden współprogram.

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

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

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

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

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

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

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

Obsługa sytuacji wyjątkowych 7

Wprowadzenie 7.1

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

— **memerror** — oznacza brak pamięci do utworzenia nowego obiektu;

— **numerror** — oznacza błąd numeryczny (np. przekroczenie zakresu liczb, dzielenie przez zero itp.);

— **logerror** — oznacza dowolny błąd wykrywany w trakcie wykonania programu związany z użyciem instrukcji niezgodnym z przeznaczeniem (np. przekazanie sterowania w sposób niezgodny z regułami Loglanu, próba usunięcia obiektu aktywnego itp.);

— **conerror** — oznacza, iż został przekroczony zakres indeksów tablicy lub, że zakresy tablicy są niepoprawne;

— **syserror** — oznacza dowolny rodzaj błędu sygnalizowanego przez system operacyjny (np. błąd wejścia/wyjścia, błąd parzystości itp.).

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

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

Sformułowanie problemu 7.2.1

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

Dyskusja zadania 7.2.2

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

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

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

Omówienie rozwiązania 7.2.3

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

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

```
signal zero_na_przekątnej.
```

Ogólna postać tej klauzuli jest następująca:

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

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

Sygnały standardowe nie muszą być deklarowane.

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

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

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


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

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

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

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

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

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

Definiowana klasa przyjmie następującą postać:

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

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

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

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

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

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

(1) pref układ_równań block

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

(2) pref układ_równań block

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


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

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

Rozwiązanie 7.2.4

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

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

Sformułowanie problemu 7.3.1

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

Dyskusja zadania 7.3.2

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

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

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

Mając procedurę p, można z łatwością zapisać algorytm, który rozwiązuje nasze zadanie. Należy w tym celu sprawdzić dla każdego k (lower(A)<=k<=upper(A)), czy procedura p znajdzie szukany ciąg indeksów startując z s = 0. Do rozwiązania pozostaje jeszcze problem wypisania ciągu indeksów w razie ich znalezienia. Korzystając tylko z procedur rekurencyjnych musielibyśmy zapamiętywać znalezione indeksy, usuwając niektóre z nich w razie wycofywania się z ewentualnych nieudanych wyborów. Tymczasem, aktualnie rozpatrywany ciąg indeksów jest dany jako ciąg wartości parametru k w kolejnych rekurencyjnych wołaniach procedury p. Jest więc on zawarty w łańcuchu dynamicznym. Sugeruje to, by znalezienie właściwego ciągu indeksów potraktować jako sytuację wyjątkową, której wystąpienie spowoduje zakończenie wykonywania obiektów łańcucha dynamicznego. W tej sytuacji można zdefiniować instrukcje końcowe, które zostałyby wykonane po zasygnalizowaniu sytuacji wyjątkowej. W naszym zadaniu taką instrukcją końcową będzie po prostu wypisanie wartości k. Zauważmy, że ze względu na sposób „zwijania" łańcucha dynamicznego (począwszy od ostatnio wykonywanego obiektu do obiektu, w którym jest obsługiwany wyjątek), indeksy zostaną wypisane w porządku malejącym.

Omówienie rozwiązania 7.3.3

Procedurę p można uściślić w następujący sposób:

```
unit p: procedure(s,k: integer);
    var i: integer;
    begin
    if s+A(k)>v then return fi;
    if s+A(k)=v then raise znaleziony fi;
    s:=s+A(k);
    for i:=k to upper(A) do call p(s,i) od;
    ...
    end p;
```

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

```
last_will: I;
```

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

Instrukcje końcowe procedury p zdefiniujemy za pomocą klauzuli

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

Procedura rozwiązująca postawiony problem przyjmie taką oto postać:

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