Część 2

Programuj z klasą
lub
programowanie obiektowe

ROZDZIAŁ 13

Klasy i obiekty $\mathcal{L}_7$

W tej części zajmiemy się klasami i obiektami. Przedstawimy też dziedziczenie klas.

1. Klasa

TODO [

- klasa Zespolone, funkcje na zewnątrz i wewnątrz klasy, notaca pre i infix-owa
- klasa geometria planarna, klasy Punkt, Linia, Okrąg, funkcje =metody tych klas
- Klasa narzędzie definiowania struktury matematycznej - liczby wymierne
- klasa definicja pewnej struktury dokumentów w biurze -rachunek
- klasa opisująca coś z życia
- parametry klasy
- dyskusja przykładu - swap ??
- procedura Gsort - dwie propozycje,
- funkcja jako wynik ?? o co mi tu chodziło?

]

W poprzednich rozdziałach nauczyliśmy się rozszerzać język wprowadzając deklaracje- definicje funkcji czyli działań oraz deklaracje – definicje procedur służące wprowdzaniu nowych instrukcji atomowych, inaczej zwanych, instrukcjami procedur. W tej części poznamy pojęcie klasy. Każda deklaracja klasy C umieszczona w naszym programie poszerza zbiór zastanych typów o nowy nowy typ C.

Składnia deklaracji klasy jest podobna do deklaracji procedury

unit K : **class**$(args)$;
  D
begin
  I
end K

Jak widać zamiast słowa procedure występuje słowo class. Struktura modułu klasy jest podobna do struktury modułu procedury.

```
unit K: class (parametry);
  <lokalne wielkości>
begin
  <instrukcje>
end K
```

```
unit P: procedure(parametry);
  <lokalne wielkości>
begin
  <instrukcje>
end P
```

definicją odpowiedniej struktury algebraicznej S_K. W dalszym ciągu zobaczymy jak bardzo skomplikowane mogą być struktury opisywane przez klasy.

Jak widać zamiast słowa procedure występuje słowo class. Deklaracja modułu klasy K, z jednej strony jest przepisem na tworzenie obiektów klasy K, z drugiej strony, opisuje strukturę algebraiczną obiektów tej klasy. zamiast instrukcji procedury call, w użyciu będzie słowo new. Wyrażenie new K(params) jest wyrażeniem obiektowym. Jego wartością jest pewien obiekt klasy K. rys. obiekt Na rysunku dostrzeżesz jak wykonanie instrukcji przypisania x:= new K(params) skutkuje utworzeniem obiektu o (klasy K) i przypisaniem go jako wartości zmiennej x.

Niezmiennik systemu Loglan. Następująca formuła jest aksjomatem języka Loglan

$$typeof(x) = K \Rightarrow \{x := \mathbf{new}\, K(params)\}(x\, \mathbf{in}\, K)$$

Klasy stanowią budulec z którego tworzymy programy obiektowe. W danym programie klasa jest niezmienna.
deklaracja klasy to dodanie nowego typu, tzn. nowej struktury algebraicznej i jej teorii! Klasa wyznacza nowy typ. Czyli strukturę algebraiczną. Taka struktura może być jednak bardziej skomplikowana od struktur badanych przez matematyków. Co prawda, klasa complex definiuje ciało liczb zespolonych. Ale moduł klasy pozwala zdefiniować bardzo skomplikowane struktury algebraiczne.
Klasa C umożliwia deklarowanie zmiennych typu C:
var z1,z2: C

Niezmiennik systemu Loglan. W trakcie obliczeń programu wartość każdej zmiennej jest albo wartością pewnego typu prostego albo obiektem pewnej klasy (lub tablicy) albo jest specjalną stałą none.

Typ wyznaczony przez deklarację klasy C jest to zbiór wszystkich obiektów jakie spełniają relację in pomiędzy obiektami i klasą, wraz z operacjami na obiektach tego typu i relacjami pomiędzy obiektami tego typu.
Przykład
Zbiór $|complex|$ obiektów klasy complex z działaniami add i mult opisanymi w tej klasie.

$$\mathcal{C} = \langle |complex|, add, mult, = \rangle$$

stanowi algebrę, którą oznaczyliśmy $\mathcal{C}$.
No tak, ale czym są obiekty klasy complex?

2. Obiekty

Obiekty klas są jednostkami dynamicznymi – tzn. powstają w trakcie wykonywania programu. Po jego zakończeniu znikają. Obiekty są wartościami zmiennych zadeklarowanych jako zmienne odpowiedniego typu opisanego klasą. Obiekt może zmieniać się w trakcie obliczeń. Jest jednostką dynamiczną. Każda klasa opisuje nie tylko strukturę obiektu, ale tkże operacje jakie można na obiektach tej klasy wykonywać. Zob. rysunek 1.
Operacją dualną do operacji utworzenia obiektu jest operacja kill – usuwania wskazanego obiektu. Więcej informacji o tej operacji znajdziesz w podrozdziale 5, na stronie 244.

2.1. Liczby wymierne. Poniższy przykład powinien Ci dać odpowiednie intuicje idące w dwu kierunkach:

- Co można umieścić w deklaracji klasy,
- Jak zaimplemntować nową strukturę algebraiczną, czyli jak opisać nowy typ danych.

Chcemy więc rozszerzyć zestaw typów o nowy typ Ułamki. Struktura lagebraiczna ułamków to zbiór i określone na nim operacje dodawania, mnożenia dzielenia oraz dwue relacje równości i mniejszości.

$$Fractions = \{U, +, *, /, =<\}$$

gdzie U jest podzbbiorem iloczynu kartezjańskiego $\mathbb{Z} \times \mathbb{Z}$. Elementami zbioru U są wszystkie pary liczb całkowitych takie, że

drugi element pary jest liczbą różną od zera.

U zbiór obiektów typu fraction, jest produktem $U - \mathbb{Z} \times \mathbb{Z}$	unit fraction: class (L.M: integer); end fraction;
$L, M \in \mathbb{Z}$ $U = \{\langle L, M\rangle : M \neq 0\}$	unit fraction: class (L.M: integer); begin if M=0 then raise notfractionError fi end fraction;

Niech $u, \hat{u} \in U$
$u + \hat{u} \stackrel{df}{=}$
$\frac{u.L * \hat{u}.M + u.M * \hat{u}.L}{u.M * \hat{u}.M}$
	unit fraction: class (L.M: integer);
	unit add: function(u:fraction):fraction;
	begin
	result := new fraction(L*u.M+u.L*M, M*u.M)
	end add;
	begin
	if M=0 then raise notfractionError fi
	end fraction;

Ukrywanie atrybutów L (a take M) to parafunkcji
−odczytaj wartość L
−zmień wartość L
lepiej je schowaj.
	unit fraction: class (L.M: integer);
	private L,M; (* L,M are hidden *)
	unit add: function(u:fraction):fraction;
	begin
	result := new fraction(L*u.M+u.L*M, M*u.M)
	end add;
	begin
	if M=0 then raise notfractionError fi
	end fraction;

Powyższe zestawienie pozwala dostrzec wiele odpowiedniości pomiędzy programowaniem klas a definiowaniem nowych struktur algebraicznych, co jest częste w algebrze ogólnej i podstawach matematyki.
Warto podkreślić modyfikator dostępu 'private'. Ma on dla programistów olbrzymie znaczenie praktyczne. Jest on w pewnym sensie dualny do definiowania operacji czyli funkcji.

2.2. Klasa complex – liczb zespolonych. Klasa może definiować strukturę algebraiczną. Dla przykładu napiszemy deklarację klasy Complex.

Specyfikacja. Naszym zadaniem jest stworzenie takiej klasy C, która definiuje zbiór liczb zespolonych wraz z operacjami na nich

$$\mathfrak{C} = \langle \mathbb{C}, add, mult, eq \rangle$$

przy czym spełnione mają być następujące wymagania

(1) $\mathbb{C} = \mathbb{R} \times \mathbb{R}$,
(2) add: $\mathbb{C} \times \mathbb{C} \to \mathbb{C}$,
(3) $mult$: $\mathbb{C} \times \mathbb{C} \to \mathbb{C}$,
(4) eq: $\mathbb{C} \times \mathbb{C} \to \mathbb{B}_0$,
(5) $\forall_{z,z'}\, add(z, z') = \langle z.re + z'.re,\, z.im + z'.im \rangle$
(6) $\forall_{z,z'}\, mult(z, z') = \langle z.re * z'.re - z.im * z'.im,\, z.re * z'.im + z'.re * z.im \rangle$
(7) $eq(z, z') \equiv (z.re = z'.re \wedge z.im = z'.im)$

Implementacja. Bez większych niespodzianek napiszemy klasę complex realizującą taką specyfikację zadania.

Zaczynamy. Deklaracja klasy complex.

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

Ad 1) umożliwia deklaracje zmiennych

```
var z,y,t:complex;
```

i polecenia stworzenia obiektu

```
y:= new complex(2.8,-9.0);
z:= new complex(3.1,5.6);
```

Wpisujemy deklarację funkcji add.

```
unit complex: class (re, im: real);
  unit add:function(z: complex):
                    complex;
  end add;
end complex
```

Ad 2) umożliwia dodawanie

```
t:= z.add(y);
```

choć wynik jest niepoprawny!

Ad 3) Wpisujemy deklarację funkcji mult.

```
unit complex: class (re, im: real);
  unit add: function(z,z1: complex): complex;
  end add;
  unit mult: function(z,z1:complex): complex;
  end mult;
end complex
```

Ad 4) Wpisujemy deklarację funkcji eq.

```
unit complex: class (re, im: real);
  unit add:
  function(z,z1: complex): complex;
  end add;
  unit mult:
  function(z,z1: complex): complex;
  end mult;
  unit eq:
  function(z,z1: complex): Boolean;
  end eq;
end complex
```

To jest sygnatura struktury algebraicznej complex. Wprowadzono nazwę elementów tej strukturym w tym przypadku complex. Wymienione zostały operacje: add, mult, eq, typy tych operacji są zgodne z specyfikacją.

Ad 5) Dodajemy treść funkcji add.

```
unit complex: class (re, im: real);
  unit add: function(z :complex): complex;
  begin
    result := new complex(z.re+re, z.im+im)
  end add;
  unit mult:function(z :complex): complex;
  end mult;
  unit eq:function(z :complex): Boolean;
  end eq;
end complex
```

Od teraz wynik dodawania liczb zespolonych będzie obliczany poprawnie.

Ad 6 i 7) Dodajemy treść funkcji mult i eq.

```
unit complex: class (re, im: real);
  unit add: function(z : complex): complex;
  begin
    result := new complex(z.re+ re, z.im+ im)
  end add;
  unit mult: function(z:complex): complex;
  begin
    result := new complex(z.re* re- im*z.im,
                          re*z.im+im*z.re)
  end mult;
  unit eq: function(z:complex): Boolean;
  begin
    result := (re=z.re and im=z.im)
  end eq;
end complex
```

Weryfikacja. Udało się zrealizować wszystkie postulaty wyliczone wcześniej. Nietrudno sprawdzić, że spełnione są wszystkie wymagania wyliczone w specyfikacji.

Zastosowanie – przykład. Poniżej przytoczymy program liczący w dziedzinie liczb zespolonych, tj. na obiektach klasy complex.

```
program ExampleComplex;
  unit complex...
  var z,u,q,r: complex;
begin
  q:=new complex(2.3, 3.2);
  z:=new complex(4.6, 13.2);
  r:= q.add(z.mult(u));
  (* podczas wykonywania tej instrukcji zostanie zgłoszony błąd *)
  (* ponieważ domyślną wartością zmiennej u jest pusty obiekt none *)
  u:= q.add(z);
  r:= q.add(z.mult(u));
  z.im:=z.im+z.re; (*czy ta instrukcja ma pozostać? *)
end
```

Ale pojawił się problem. Każdy obiekt z typu $complex$ udostępnia operacje odczytywania wartości atrybutów $z.im$ oraz $z.re$, a także operacje zmiany wartości atrybutu realizowane przez polecenia przypisania $z.re := 7.2$ itp. Jeżeli chcemy zablokować tego typu polecenia to należy napisać deklarację zamykającą dostęp do atrybutów z zewnątrz obiektu – close re, im; – jako drugą linijkę w klasie complex..[1]

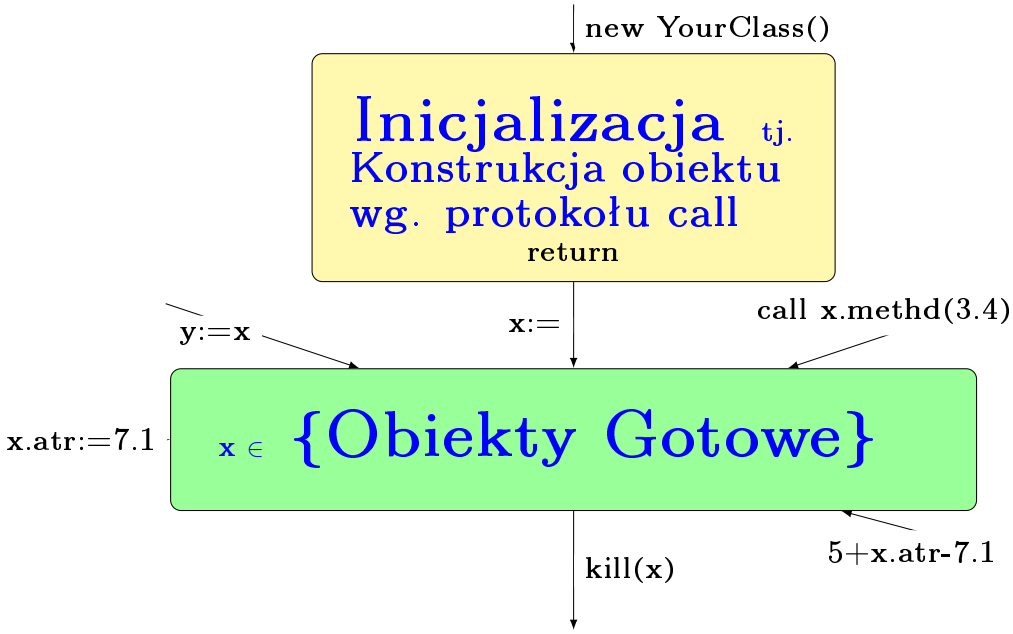
3. Scenariusz obiektu

Obiekty:

- powstają - w efekcie obliczenia wartości wyrażenia new C(params), i stają się wartośćia zmiennej odpowiedniego typu, por. x:=new C(params)
- są współdzielone przez różne zmienne – w efekcie wykonania instrukcji przypisania y:=x,
- są dostępne dla operacji:
  - odczytu wartości atrybutu,
  - zapisu (modyfikacji) wartości atrybutu,
  - usługi (serwisu),
- stają się niedostępne (stają się "śmieciami") gdy nie wskazuje na nie żadna zmienna,
- są usuwane np. przez polecenie `kill(x)`

Obiekt jako argument operacji.
Obiekt jako posiadacz stanu pamięci.
Na rysunku 1 przedstawiamy scenariusz obiektu w jego ogólnej postaci. W uzupełnieniu do rysunku przedstawiamy program ilustrujacy sytuacje w jakich może znaleźć się obiekt.
Oczywiście rysunek jest tylko ilustracją scenariusza, ... Za-

[1] W Javie i innych językach programowania stosujemy modyfikator deklaracji protect.



Rysunek 1. Scenariusz obiektu klasy i zewnętrznych operacji realizowanych na obiekcie

łóżmy więc, że mamy poniższy program

```
program demo;
  unit YCls: class(a,b:real);
    var atr: integer;
    unit methd: function(x:integer): integer;
    begin
      result := x+10;
    end methd;
  begin
  end YCls;
  var x,y:YCls, l:real;
begin
  x:= new YCls(2.6, 8.8);
  y:=x;
  l:=5+x.atr-7.1;
  l:=x.methd(9);
  x.atr:=7.1;
  x :=none;
  kill(y);
end
```

Czasami funkcja lub procedura zadeklarowane w klasie C mają zwrócić obiekt w którym funkcja jest zadeklarowana. Można to zrobić posługując się wyrażeniem this C.

```
unit Kls: class;
  unit toja: function: Kls;
  begin
    result := this KLs
  end toja;
end Kls;
```

Przykład y
zmiana promienia okręgu
wstawienie elementu do kolejki

3.1. Macierze kwadratowe. W rozdziale 6 opisaliśmy obiekty tablicowe. Operacje na takich obiektach mogą spowodować przerwanie obliczeń i zgłoszenie błędu reference to none por. ?? sprawdzanie zgodności tablic przed mnożeniem.
Możemy znacznie zwiększyć bezpieczeństwo wprowadzając klasę QuadMat.

Specyfikacja. Po rozważaniach z rodziału 6 czytelnik potrafi napisać samodzielnie specyfikację tej struktury, zob. ćwiczenie 13.1.

Implementacja.

```
unit QuadMat: class(n: integer);
  var A: arrayof arrayof real, i: integer;
  unit add: function(B: QuadMat):QuadMat;
  unit mult: function(B: QuadMat): QuadMat;
  unit init: procedure(C: arrayof arrayof real);
    (* wstaw liczby z tablicy C do tablicy A sprawdzając zakresy *)
    var i,j,l,u: integer;
  begin
    u:=upper(C); l:=lower(C);
    if n ≠ u-l+1 then raise QuadMatErr fi;
    for i := 1 to n do
      u:=upper(C[i]); l:=lower(C[i]);
      if n ≠ u-l+1 then raise QuadMatErr fi;
      for j := 1 to n do A[i,j] := C[i,j] od
    od;
  end init;
begin
  array A dim (1:n);
  for i := 1 to n do array A(i) dim (1:n); od;
  (* gotowe, kwadratowa macierz zer! *)
end QuadMat
signal QuadMatErr;
```

Zauważ. Deklaracji klasy QuadMat towarzyszy deklaracja sygnału QuadMatErr. Podnosząc ten sygnał zgłaszamy błąd podczas inicjalizacji tablicy.

3.2. Zastosowanie. Przykład programu wykorzystującego klasę QuadMat.

```
program ExQuadMat;
  unit QuadMat class ...
  signal QuadMatErr;
  var A,B,C: QuadMat;
  handlers
    when QuadMatErr: ...
  end handlers;
begin
  A := new QuadMat(3);
  B := new QuadMat(4);
  C := A.mult(B);
  (* signal QuadMatErr will be raised *)
  ...
end
```

```
program ExQuadMat1;
  unit QuadMat class ...
  signal QuadMatErr;
  var A,B,C: QuadMat;
  handlers
    when QuadMatErr: ...
  end handlers;
begin
  A := new QuadMat(3);
  B := new QuadMat(3);
  C := A.mult(B);
  ...
  ...
end
```

Może napisać bardziej rozbudowany przykład

4. Klasy mogą być zagnieżdżane

Rozpatrzmy przykład struktury algebraicznej, na którą składają się punkty, linie i okręgi.

5. Uwagi o odśmiecaniu i defragmentacji

Przykład 13.1. Obliczenie z pętla for na obiektach klasy complex wytworzy śmieci. wstaw!

Co to jest

Definicja 13.1. Obiekt jest śmieciem jeśli nie jest (i wobec tego nie będzie) dostępny. Tzn. jeśli nie jest wartością żadnej zmiennej.

⌜Uwaga.Czytelnik może zapytać, a co z obiektami, które są dostępne, ale są niepotrzebne? Czy to też są śmieci? koniec uwagi.⌟

Definicja 13.2. Odśmiecanie – proces usuwania obiektów

Odśmiecanie jest kosztownym procesem. W fazie "sweep" trzeba przejrzeć całą pamięć. Następna faza też jest kosztowna. Czy nie można by zmniejszyc potrzeby odśmiecania przez częstsze kompresowanie wolnej pamięci? Wystarczyłoby by porzucając wykorzystany obiekt pomocniczy protokół ... wpisywał ten obiekt na listę wolnej pamięci. Jak to zrobić?

Odśmiecanie nie jest koniecznym elementem systemu zarządzania obiektami! Jeśli pamiętasz nazwę obiektu, który za chwilę stanie się ąmieciem, to zrób kill! zanim będzie zapóźno. Być może w ten sposób opóźnisz odśmiecanie.
Jeśli wiesz, że obiekt x nie będzie juz potrzebny to usuń go – kill(x). Więcej o operacji kill przeczytasz w rozdziale ?

Klasa GaussC – liczb zespolonych całkowitych

Karl Gauss rozważał zbiór liczb zespolonych całkowitych.Por. W. Sierpiński Teoria Liczb, [Sie50], rozdz. XVI, str. 383

```
unit GaussC: class(re, im: integer);
    unit add: function(z: GaussC): GaussC;
    begin
      result:= new GaussC(re+z.re, im+z.im)
    end add;
    unit sub: function(z: GaussC): GaussC;
    begin
      result:= new GaussC(re-z.re, im-z.im)
    end sub;
    unit mult: function(z: GaussC): GaussC;
    begin
      result:= new GaussC(re*z.re-im*z.im, im*z.re+z.im*re)
    end mult;
    unit divi: function(z: GaussC): GaussC;
      var rz, ur: integer;
    begin
      rz := (re*z.re - im*z.im) div(z.re*z.re - z.im*z.im) ;
      ur := (im*z.re + re*z.im)
      result:= new GaussC(rz, ur)
    end divi;
    unit remi: function(y,z: gaussC): GaussC;
    begin
      result:= y.sub(z.mult(y.divi(z)))
    end remi;
    unit equal: function(y,z: gaussC): Boolean;
    begin
      result:= (z.re=y.re and z.im=y.im)
    end equal;
    unit norm: function : integer;
    begin
      result:=re*re+im*im
    end norm;
end GaussC
```

Rozważmy zbiór wszystkich możliwych obiektów klasy GaussC.

Jest to zbiór nieskończony. Oznaczmy go przez $|GaussC|$. Struktura na którą składa się zbiór $|GaussC|$ obiektów tej klasy z działaniami add i $mult$ opisanymi w tej klasie

$$\mathfrak{C}_I = \langle |GaussC|, add, mult \rangle$$

jest pierścieniem.
Popatrzmy na program wykorzystujący klasę GaussC.

```
program EuGaussC;
  unit GaussC: class ...
  var z1, z2, z3, zero : GaussC;
begin
  zero := new GaussC(0,0);
  z1:= new GaussC(...); (* jakiś z1 *)
  z2 := new GaussC(...); (* jakiś z2 *)
  z3 := z1.remi(z2);
  while z3.norm =/= 0 do
    z1 := z2 ;
    z2 := z3 ;
    z3 := z1.remi(z2);
  od ;
end
```

Co można powiedzieć o wyniku obliczenia w przypadku gdy program kończy obliczenie? Czy ten program ma obliczenia nieskończone?

6. Mnożenie macierzy liczb zespolonych

Osiem różnych wersji
Zadanie: oblicz iloczyn dwu macierzy kwadratowych liczb zespolonych mozna rozwiązać na przynajmniej osiem sposobów:

- Algorytm możenia macierzy – możesz uzyc algorytm zwyczajny lub algorytm Winograda.
- Mnożenie liczb zespolonych – możesz użyć zwyczajnego mnożenia

  $$(a+bi)\cdot(c+di) \stackrel{df}{=} (ac-bd)+(ad+bc)i$$

  lub algorytmu sprytnego

  $$\left\{\begin{array}{l} t1 \leftarrow a*c; \\ t2 \leftarrow b*d; \\ t3 \leftarrow (a+b)*(c+d); \\ Re \leftarrow t1-t2; \\ Im \leftarrow t3-t1-t2 \end{array}\right\}$$

- Sposób przedstawienia danych – możesz tablice A i B zapisać jako typu arrayof arrayof complex lub jako pary macierzy liczb rzeczywistych $A = A_{re}+A_{im}\cdot i$ i $B = B_{re}+B_{im}\cdot i$.

Należy: a) zaprogramować każdy z ośmiu algorytmów, b) udowodnić jego poprawność, c) wyznaczyć funkcję kosztu i na

podstawie eksperymentów, obliczyć współczynniki w odpowiednim wielomianie.
Zadanie to nadaje sie do przeprowadzenia w grupie studentów. Na koniec słuchacze mogą podczas mini-sympozjum przedstawić wyniki swoich prac i porównać je.

7. Klasa geometria Cyrkla i Linijki

Widzieliśmy wcześniej, że wprowadzając deklarację klasy, tym przypadku klasy complex - poszerzyliśmy język o nowy typ danych wraz z odpowiednimi operacjami na obiektach tej klasy.
W poniższym przykładzie zobaczymy, że klasy mogą być zawarte w klasie pojemniejszej. W tym przykładzie wykorzystamy zagnieżdżanie klas w sytuacji gdy struktura ma więcej typów (niektórzy używaja słowa sort).

```
unit GeoPlan: class;
  unit Point: class(x,y:real);...
  unit Line: class(A,B,C: real);...
  unit Circle: class(S:Point, r:real)...
end GeoPlan;
```

Więcej szczegółów znajdziesz poniżej.

```
unit Geoplan : class;
(* this is a problem oriented language. it offers various facilities
for problem solving in the field of analtical planar geometry.

    this class has the following structure:

    geoplan <—— class
/ |  / |  / |  point circle line <—— classes
/ |  | / |  / |  | / |  / | | | | |  <–| / | | | | |  | equals dist | | meets
|  | error | | error  opera- | | / tions
intersects | | | | parallelto <–|
    *)
```

Klasa Point ma dwa atrybuty (tj. pola) – współrzędne punktu, oraz dwie metody equals i dist.

```
unit Point : class(x,y : real);
  unit equals : function (q : Point) : boolean;
  begin
    if q = none then raise NonePointError
    else result:= q.x=x and q.y=y ;
  end equals;
  unit dist : function (p : Point) : real;
  (* distance between this point and point p *)
  begin
    if p = none then raise NonePointError
    else result:= sqrt((x-p.x)*(x-p.x)+(y-p.y)*(y-p.y))
    fi
  end dist;
end point;
```

Klasa Circle opisuje obiekty okręgi.

```
unit Circle : class (p : point, r : real);
  (* the circle is represented by its center p and the radius r *)
  unit intersects : function (c : Circle) : line;
    (* if both circles intersect at 2 points, the line joining them
    is returned. if circles intersect at one point, it is the line tangent
    to both of them. otherwise perpendicular bisection
    of their centres is returned *)
    var r1,r2 : real;
  begin
    if c=/= none
    then
      r1:= r*r-p.x*p.x-p.y*p.y;
      r2:= c.r*c.r-c.p.x*c.p.x-c.p.y*c.p.y;
      result := new line (p.x-c.p.x,p.y-c.p.y,(r1-r2)/2);
    fi
  end intersects;
begin
  if p=none    then writeln ("wrong centre")
  fi
end Circle;
```

Klasa Linia – obiekt tej klasy jest wyznaczony przez rôwnanie $Ax + By + C = 0$. Metodami oferowanymi przez tę klasę są parallelto i meets.

```
unit Line : class (a,b,c : real);
  (* line is represented by coefficients of its equation ax+by+c=0 *)
  unit meets : function (l : line) : point;
(* if two lines intersect function meets returns the point
of intersection, otherwise returns none *)
    var t : real;
  begin
    if l =/= none and not parallelto (l)
then t := 1/(l.a*b-l.b*a); result := new point (-t*(b*l.c-c*l.b), t*(a*l.c-c*l.a));
else raise LinieRownolegle fi
  end meets;
  unit parallelto : function (l : line) : boolean;
  begin
if l=/= none
then
if a*l.b-b*l.a = 0.0
then
result:=true; writeln(źle");
else
result:=false; writeln("dobrze");
fi
else
call error
fi
  end parallelto;
  var d : real;
begin (* normalization of coefficients *)
  d := sqrt(a*a+b*b);
  if d= 0.0 then writeln( źle zero"); call error
else a := a/d; b := b/d; c := c/d;
fi
end line;
```

Język programowania zorientowany do rozwiązywania problemôw geometrii cyrkla i linijki.

Zadanie Okrąg opisany na trôjkącie

Zadanie Inwersja względem okręgu

Zadanie

Teza Algorytm geometrii cyrkla i linijki, który się nie zapętla nie potrzebuje instrukcji while. odszukaj gdzie to jest napisane

8. Kolejki

W tym podrozdziale spotkamy niejawną (lub " rekurencyjną") definicję klasy. W poprzednich przykładach definicja klasy wykorzystywała składniki w pełni zdefiniowane wcześniej. W tym przykładzie w definicji klasy kolejka występuje składnik typu kolejka. Ta sytuacja sugeruje obliczenia rekurencyjne. Ale zwrôć uwagę na rôżnice. Koszt utworzenia nowego obiektu **new**

Kolejka jest stały, zob. deklarację klasy Kolejka w podrozdziale 8.2 Implementacja.
Każdy spotkał się ze strukturą kolejek. Jaka to algebra? Jakie działania wykonujemy na kolejkach i jakie mają one własności? Z góry zastrzegamy, że opisanie czym jest kolejka nie jest łatwą sprawą. Więcej informacji znajdziesz w odpowiednim rozdziale w następnej III części tej książki.
Po pierwsze, elementami kolejek mogą być bardzo różne obiekty. Z kolejkami spotykamy się w codziennym życiu, kolejki występują też ukryte w wielu różnych systemach. Np. firma ochroniarska rozpatruje zgłoszenia o nieprawidłowościach po kolei, w miarę jak napływają. W systemie operacyjnym, który zarządza Twoim komputerem też występują kolejki. Po drugie, chociaż operacje systemu algebraicznego kolejek mają pewne wspólne cechy, to sposobów realizacji kolejek i sposobów wykonywania działań na kolejkach jest nieograniczona liczba.

8.1. Specyfikacja kolejek. Spróbujemy jednak naszkicować wymagania charakteryzujące kolejki. W systemie kolejek jego uniwersum składa się z dwu rozłącznych zbiorów: zbiór E elementów i zbiór Q kolejek.
Mamy do czynienia z następującymi operacjami: wstaw element do kolejki, weź pierwszy element z kolejki, usuń pierwszy element z kolejki oraz stałą – pusta kolejka. Ponadto mamy następujące predykaty: równość elementów, równość kolejek, bycie kolejką pustą. Podsumujmy ten etap – sygnatura systemu kolejek to układ

$$\mathfrak{Q} = \langle E \cup Q, i, f, d, p; =_E, =_Q em \rangle$$

przy czym wyliczone tu funktory i predykaty mają dziedziny

$$\begin{aligned}
&i\colon E \times Q \to Q\\
&f\colon (Q \setminus p) \to Q\\
&d\colon (Q \setminus p) \to Q\\
&p \in Q\\
&em\colon Q \to Booolean\\
&=_E\colon E \times E \to Boolean\\
&=_Q\colon Q \times Q \to Boolean
\end{aligned}$$

Postulaty (lub aksjomaty) kolejek

$k1.\ \forall_{e\in E}\,\forall_{q\in Q}\,\neg em(i(e,q))$

$k2.\ \forall_{e\in E}\,\forall_{q\in Q}\,\big(\neg em(q) \Rightarrow \big(f(i(e,q)) = f(q)\big)\big)$

$k3.\ \forall_{e\in E}\,\forall_{q\in Q}\,\big(em(q) \Rightarrow \big(f(i(e,q)) = e\big)\big)$

$k4.\ \forall_{e\in E}\,\forall_{q\in Q}\,\big(\neg em(q) \Rightarrow \big(d(i(e,q)) = i(e,d(q))\big)\big)$

$k5.\ \forall_{e\in E}\,\forall_{q\in Q}\,\big(em(q) \Rightarrow \big(d(i(e,q)) = q\big)\big)$

$k6.\ \forall_{q}\ em(q) \Leftrightarrow (q =_Q p)$

$k7, k8, k9.$ zwrotność, przechodniość i symetria relacji $=_E$

$k10, k11, k12.$ zwrotność, przechodniość i symetria relacji $=_Q$

Poprzestaniemy, na razie, na tych kilku postulatach.
Kilka uwag jest na miejscu.
1°) Uważamy, że nie należy podejmować próby stworzenia klasy bez określenia jakie własności ma mieć struktura algebraiczna definiowana przez zbiór obiektów tej klasy wraz z działaniami zadeklarowanymi jako metody tej klasy. Tzn. posiadanie specyfikacji jest niezbędnym warunkiem rozpoczęcia pracy nad utworzeniem klasy.
2°) Specyfikacje mogą być różnej jakości. Nie idzie tu o liczbę wierszy jakie zajmuje specyfikacja. W dalszej części zobaczymy, że specyfikacja klasy może zawierać sprzeczność. Wiele specyfikacji nie ma własności pełność. Innymi słowami zanim zaczniemy programować klasę powinniśmy znać wymagania. Wymagania te nie mogą zawierać sprzeczności – albowiem nie istnieje żadna implementacja wymagań sprzecznych. Wymagania powinny być kompletne – bowiem istnieje ryzyko, że stworzymy klasę nie odpowiadającą nowym wymaganiom.
3°) Zwróć uwagę, że w przypadku specyfikacji klasy Complex mieliśmy do czynienia ze zbiorem jawnych definicji działań (funkcji). Podana specyfikacja Kolejek nie jest zbiorem jawnych definicji. Stąd wynika większa swoboda realizacji tych postulatów.

8.2. Implementacja. Struktura danych kolejki może być realizowana na nieskończenie wiele sposobów. Niektóre z nich mogą być zaskakujace. Wybraliśmy przykład ilustrujący możliwość zadeklarowania klasy Kolejka w taki sposób, że wartością pewnego atrybutu nxt tej klasy jest obiekt własnie klasy Kolejka.

```
unit Element: class ;
  unit eqe: function(e1,e2): Boolean; ... end eqe;
end Element;
unit Kolejka: class();
  var front: Element, nxt,last:Kolejka ;
begin
end Kolejka;
put: function(e: Element, s: Kolejka): Kolejka;
  var r: Kolejka
begin
  r:=new Kolejka(e); r.front:=e;
  if em(s) then r.nxt:=none; result,r.last:=r;
  else s.last.nxt:=r; result:= s fi
end put ;
first: function(s: Kolejka): Element;
begin
  if not em(s) then result := s.front;
  else raise EmptyKolError fi
end first ;
get: function(s: Kolejka): Kolejka;
begin
  if not em(s) then result:= s.nxt
  else raise EmptyKolError fi
end get ;
em: function(s: Kolejka): Boolean;
begin
  result := (s.front = none)
end em ;
eq: function(q1,q2: kolejka): boolean;
  var s1,s2:integer,e:Element, bb:boolean;
begin
  bb := (not em(s1) and not em(s2));
  while bb do
    e1:=first(s1); s1:=get(s1);
    e2:=first(s2); s2:=get(s2);
    bb := (not em(s1) and not em(s2));
  od;
  result:= bb
end eq;
signal EmptyKolError, KolejkaNone;
```

8.3. Weryfikacja. Nalezy udowodnić, że powyższa implementacja spełnia wymagania specyfikacji. A więc powinniśmy wykazać, że każda z dwunastu {k1 –k12} jest prawdziwa przy interpretacji opisanej w podrozdziale 8.2. Mamy zbadać czy te dwie klasy Element i Kolejka wraz z deklaracjami funkcji put, first, get, em, eq spełnają postulaty wymienione w specyfikacji.

Pierwsza, łatwiejsza część zadania to upewnienie się, że struktura danych zdefiniowana przez dklarację klasy KolejkiL ma sygnaturę zgodną z sygnaturą stanowiącą pierwszą część specyfikacji 8.1. Przyjmiemy, że zbiór Q to zbiór wszystkich obiektów klasy Kolejka, zbiór E to zbiór wszystkich obiektów klasy Element. Możemy napisać tabelkę

i	f	d	p	em	$=_Q$	$=_E$
put	first	get	new Kolejka	em	eq	eqe

Przyjmujemy , że funkcja put jest realizacją operacji i, itd. Pewien problem wynika z tego, że wyraźnie widać iż klasa Element nie jest w pełni określona – mamy tylko jej szkic. Oznacza to tyle, że przyjmujemy założenie, że w konkretnym przypadku programista napisze konkretną deklarację klasy Element i znajdzie się w niej deklaracja funkcji eqe porównującej dany obiekt klasy Element z innym obiektem tej klasy i że zapewnione będą własności: symetrii, przechodniości i zwrotnosci funkcji eqe.
Przyjmując takie założenie sprawdzamuy po kolei postulaty specyfikacji:

Ad k1) Czy dla dowolnej pary obiektów e klasy Element i q klasy kolejka zachodzi $\neg em(put(e,q))$?
Niech o pewien obiekt klasy Element będzie wartością zmiennej e i niech obiekt o' klasy Kolejki będzie wartością zmiennej q. Łatwo widać, że obiekt r zwracany jako obliczona wartość wyrażenia $put(e,q)$ spełnia warunek $r.front = e$. W konsekwencji obliczona wartość wyrażenia Boolowskiego $em(put(e,q))$ jest rôwna false.
Pierwszy postulat k1 jest więc formułą prawdziwą w modelu KolejkiL.

Ad k2) Czy postulat k2. $\neg em(q) \Rightarrow (f(i(e,q)) = f(q))$ jest prawdziwy? Oczywiście tak jest. Jeśli spełniony jest warunek $\neg em(q)$ to oznacza to tyle wartością wyrażenia $q.front$ jest jakiś obiekt o klasy Element. Łatwo sprawdzić, że wartość wyrażenia $first(put(e,q)$ to ten sam obiekt o.

Ad k3)
Ad k4)
Ad k5)
Ad k6)

Możemy podsumować powyższe rozważania zapisując następujące twierdzenie.
Niech moduł Element będzie dowolnie określoną klasą, z tym, że zawarta w tej klasie funkcja boolowska eqe spełnia postulaty k8,k9,k10.

Twierdzenie 13.1. Klasy *Element* i *Kolejka* wraz z funkcjami *put,... eq* stanowią model zbioru postulatów k1 –k12.

8.4. Przykład zastosowania. Po co wprowadzamy sygnał EmptyErr? Jak zareagować? Gdzie wstawić handlery?

9. Gdy trzeba zadeklarować funkcję f jako wynik innej funkcji h

Rozważmy jeszcze raz algorytm bisekcji. Załóżmy, że deklaracji funkcjonału Bisec towarzyszy deklaracja sygnału ZnakiFunkcjiRowne.

```
unit Bisec: functional(p,k:real; function g(x: real):real): real;
...
end Bisec;

signal ZnakiFunkcjiRowne;
```

Napiszemy taka klasę Gfun.

```
unit Gfun: class(function f(x:real):real )
   unit Nf : function(a,b:real): real;
   begin
      result := Bisec(a,b,f)
   end Nf
end Gfun
```

Jeśli teraz stworzę dwa obiekty klasy Gfun.

```
o1:= new Gfun(sin);
o2:= new Gfun(cos);
```

to mogę uznać, że napisy o1.Nfun i o2.Nfun są nazwami dwu różnych funkcji.

Twierdzenie 13.2. Jeśli klasa Gfun ma dostęp do deklaracji funkcjonału Bisec, to obiekty tej klasy oferują funkcje

Nazewnik funkcyjny o2.Nfun(a,b) zwraca wartość będącą miejscem zerowym funkcji cos leżącym pomiędzy liczbami a i
textsfb lub sygnalizuje blad ...gdy znaki funkcji na końcach przedziały a,b są jednakowe.
W podobny sposób możemy opisać obliczanie funkcji pierwotnej wykorzystując funkcjonał calka ...

10. Gsort

Zadanie sortowania częściej odnosi się do obiektów rozmatych klas niż do sortowania liczb.. Przedstawimy dwie propozycje

- procedura Gsort z typem formalnym,
- klasa z metodą Gsort

W pierwszym przypadku

```
unit Gsort: procedure(type T,A: arrayof T, function less(x,y:T): Boolean);
...
end Gsort;
```

Inna propozycja

```
unit SoTab: class;
...
end SoTab;
```

11. Komputer $\mathcal{K}_7$

Komputer $\mathcal{K}_7$ tworzy obiekty, umożliwia dostęp do ich atrybutów, usuwa obiekty (kill). Protokół new tworzenia obiektu klasy jest bardzo podobny do protokołu call wykonywania instrukcji procedury. Ponadto komputer $\mathcal{K}_7$ potrafi wykonać następujące instrukcje (polecenia) atomowe:

- y:= x
  pre- wartością zmiennej x jest pewien obiekt typu T, zmienne x oraz y są tego samego typu T,
  post- wartością zmiennej y jest obiekt będący wartościa zmiennej x,

$$\{y := x\}(x = y)$$

- y:=copy(x)
  to polecenie skutkuje utworzeniem kopii obiektu x i przypisaniem nowego obiektu jako wartość zmiennej y. Zwróć uwagę, $\{y := x\}(y \neq x)$, ale dla każdego atrybutu at zachodzi równość $\{y := x\}(y \neq x \wedge x.at = y.at)$
- z:= x.at
  Jeśli zmienna x jest typu T i jej wartością jest obiekt typu T tzn. $x \neq$ none i ponadto zmienna z oraz atrybut at klasy T są tego samego typu U, to wartość wyrażenia $x.atr$ jest przypisana zmiennej z,
- x.atr:= τ

- call x.metoda(args)
- x:= none
- kill(x)

11.1. Nieformalny opis struktury do zarządzania obiektami. Obiekty są wartościami zmiennych (odpowiednich typów). Komputer $\mathcal{K}_7$ zapewnia przy tym zachowanie następującej własności Jeśli zmienna z jest zadeklarowana jako typu T (np. unit T; class ...) to jej wartością jest obiekt klasy T lub wartością jest **none**.

11.2. lokalizacja potrzebnej zmiennej ... Komputer $\mathcal{K}_7$ umie znaleźć zmienną, która jest potrzebna do obliczenia wartości wyrażenia lub do wykonania instrukcji przypisania. Problem pojawia się gdy szukana zmienna nie jest zmienną lokalna.

Ćwiczenia

13.1. Napisz specyfikację algebry macierzy kwadratowych liczb typu real.

13.2. Udowodnij, że struktura $\mathfrak{C}_I$ jest pierścieniem.

13.3. Udowodnij, że struktury tej nie da się wzbogacić żadną relacją (funkcją boolowską) tak by zachowane były aksjomaty pierścienia i ponadto własności $z < x \Rightarrow z + y < x + y$.

13.4. Opisz efekt wykonania każdej instrukcji w programie demo 3.

13.5. Napisz specyfikację dla funkcjonału całka obliczającego wartość całki oznaczonej $\int_a^b f(x)dx$ z żądaną dokładnością ϵ.

13.6. Napisz deklarację funkcjonału całka obliczającego wartośc całki oznaczonej $\int_a^b f(x)dx$

ROZDZIAŁ 14

Dziedziczenie albo rozszerzanie klas $\mathcal{L}_8$.

Co to takiego?
Do składania modułów stosowane są dwa narzędzie: zagnieżdżanie i rozszerzanie, zwane także dziedziczeniem, ang. inheritance).
Z zagnieżdżaniem modułów spotkaliśmy się wcześniej: moduł A może być zawarty w module B. Zapisujemy to w taki sposób $a\mathbf{decl}\,B$
Dziedziczenie to inna operacja. Gdy dane są moduły A i B to wynikiem operacji dziedziczenia jest moduł wynikający z połączenia treści tych modułów.... Zagnieżdżanie umożliwia modułowi zagnieżdżonemu dostęp do wszystkich zasobów zadeklarowanych w module obejmującym. Jeśli moduł M obejmuje moduły M1, M2, ... w nim zawarte to dzieli się swoimi zasobami z modułami Mi (i=1,2, ...) a każdy zasób zadeklarowany w module Mi jest jego zasobem prywatnym, niedostępnym z zewnątrz tego modułu.
A właściwie powinniśmy mówić: poprawny program jest grafem modułów w relacjami

$$P = \langle M; \mathbf{decl}, \mathbf{inh}, \mathbf{call} \rangle$$

Dziedziczenie klas zwane też rozszerzaniem (ang.extending) jest narzędziem potężnym, słabo rozumianym i rzadko w pełni wykorzystywanym. Deklaracja klasy wprowadza nowy typ danych, jest jego definicją.

1. Reguła konkatenacji

Dziedziczenie czyli rozszerzanie definicji klasy pozwala tworzyć hierarchie typów. Dwie deklaracje klas

$$\mathbf{unit}\ A:\ \mathbf{class}(\mathbf{args}_A);\ D_A\ \mathbf{begin}\ I_A\ \mathbf{end}\ A$$

oraz

$$\mathbf{unit}\ B:\ \mathbf{class}(\mathbf{args}_B);\ D_B\ \mathbf{begin}\ I_B\ \mathbf{end}\ B$$

mogą znaleźć się w relacji dziedziczenia, w taki oto sposób

$$\underbrace{\mathbf{unit}\ B : A\ \mathbf{class}}_{\substack{klasa\ B \\ \quad rozszerza\ A}}\ (\mathbf{args}_B);\ D_B\ \mathbf{begin}\ I_B\ \mathbf{end}\ B$$

W takiej sytuacji mówimy, że klasa B dziedziczy z klasy A lub klasa B rozszerza klasę A. W największym skrócie i z wieloma

zastrzeżeniami, należy rozumieć, że klasa B jest faktycznie zadeklarowana w taki sposób

$$\textbf{unit } B: \ \textbf{class}(\textbf{args}_A, \textbf{args}_B);\ D_A;\ D_B \ \textbf{begin}\ I_A;\ I_B \ \textbf{end}\ B$$

Mówimy, że relacja rozszerzania klas jest opisana przez regułę konkatenacji klas. Deklaracja klasy B jest wynikiem konkatenacji deklaracji klasy A i deklaracji klasy B. Przypominamy, podana powyżej wersja reguły konkatenacji klas jest znacznie uproszczona w stosunku do jej pełnej wersji. Inheritance is a complementary way of composition of modules. When nesting assures sharing, the inheritance is to assure that an object (or activation record of ...) will be equipped with its private copy of the resources defined in an inherited module. Two are needed to dance the waltz. Similarly, two modules are needed for the inheritance operation on modules: one module, say A„ which is inherited. And another module, say B, which inherits. One can say also that module B is an extension of the module A

Subtypes. This notion of inheritance permits to talk of subtypes of a type.

1.1. Reguła konkatenacji, odsłona 1. Objaśnienie reguły konkatenacji zaczniemy od przyjrzenia się przykładowi.

Przykład 14.1. Rozważmy przykład klasy rachunek i różne odmiany tej klasy. Każdy rachunek zawiera informacje o sumie do zapłacenia, datę wystawienia rachunku, notkę informującą czy opłata już została wniesiona, czy nie oraz inne informacje specyficzne dla rodzaju świadczonej usługi.
Przyglądając się rachunkom za energię elektryczną, za gaz i rachunkom telefonicznym dostrzegamy od razu, że te trzy rodzaje rachunków mają część wspólną. Możemy stworzyć klasę rachunek

```
unit rachunek: class(kto: Osoba, dzien: date);
var kwota: integer, paid: Boolean;
end rachunek;
```

Obiekty tej klasy mają nastepującą strukture.

Pole	Typ
kto	Osoba
day	date
kwota	integer
paid	Boolean

Ale nie zamierzamy tworzyć tych obiektów. Klasa rachunek jest początkiem hierarchii klas pochodzacych od klasy rachunek. Naszym zamiarem jest utworzyć trzy klasy

```
unit rachunekZaEnergie: rachunek class(kWh: integer);
begin
kwota := kWh * cena_kWh
(* zauważ cena_kWh jest wielkością nielokalną *)
end rachunekZaEnergie;
```

```
unit rachunekZaGaz: rachunek class(m3: integer);
begin
kwota := m3 * cena_m3
(* zauważ cena_m3 jest wielkością nielokalną *)
end rachunekZaGaz;
```

```
unit rachunekZaTelefon: rachunek class(impulsy, inne: integer);
begin
kwota := impulsy * cena_impulsu+ inne
(* zauważ cena_impulsu jest wielkością nielokalną *)
end rachunekZaTelefo;
```

Obiekty klas RachunekZaEnergie, RachunekZaGaz i RachunekZaTelefon będą mieć następujące struktury

Pole	Typ
kto	Osoba
dzien	date
kwota	integer
paid	Boolean
kWh	integer

rachunek za energię

Pole	Typ
kto	Osoba
dzien	date
kwota	integer
paid	Boolean
m3	integer

rachunek za gaz

Pole	Typ
kto	Osoba
dzien	date
kwota	integer
paid	Boolean
impulsy	integer
inne	integer

rachunek za telefon

W obiektach tych widzimy dwie warstwy:

- cztery pola: kto,day,kwota, paid to warstwa klasy rachunek,
- kolejne pola są wyznaczone, odpowiednio, przez deklarację podklasy.

Wykorzystując powyższy przykład formułujemy następującą regułę konkatenacji. Załóżmy, że dane są dwie klasy A i B

unit A: class(pf_A);
  <deklaracje$_A$>
begin
  <instrukcje$_A$>
end A
	unit B: class(pf_B);
	  <deklaracje$_B$>
	begin
	  <instrukcje$_B$>
	end B

jeśli nazwa A pojawi się przed słowem class w B, to klasa B dziedziczy z klasy A: parametry formalne, deklaracje lokalne i instrukcje inicjalizujące.	

unit B: A class(pf_B);
  <deklaracje$_B$>
begin
  <instrukcje$_B$>
end B
	unit B: A class(pf_A, pf_B);
	  <deklaracje$_A$>;
	  <deklaracje$_B$>
	begin
	  <instrukcje$_A$>;
	  <instrukcje$_B$>
	end B

Powyższa tabelka prezentuje jeden z prostszych przypadków dziedziczenia. Można powiedzieć, że dziedziczenie pozwala na wyciągnięcie wspólnej części deklaracji klas A oraz B "przed nawias". W przykładzie z klasami prezentującymi rozmaite rachunki widać to wyraźnie.
Reguła konkatenacji jest jednak bardziej skomplikowana i stwarza więcej mozliwości.

1.2. Co wiadomo o wartości zmiennej x? Jeśli klasa B dziedziczy z klasy A to mówimy także: klasa B jest podklasą klasy A.

$$B \,\mathbf{inh}\, A.$$

Relacja bycia podklasą jest przechodnia tzn. jeśli klasa C jest podklasą klasy B i klasa B jest podklasą klasy A to klasa C jest podklasą klasy A.
W ten sposób możemy tworzyć hierarchie klas. Hierarchia podklas danej klasy tworzy drzewo. Relacja inh jest statyczna, tj. nie zmienia się podczas wykonywania programu[1]. Podobnie jak relacja decl zawierania jednego modułu programuw innym.

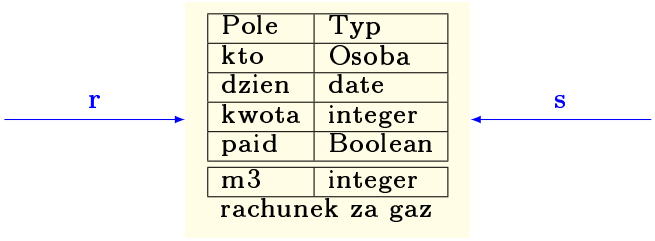
Jeśli, w danym miejscu programu obowiązuje deklaracja

var x: T; – gdzie T jest nazwą klasy.

to mówimy, że typem (tego wystąpienia) zmiennej x jest T. Jest niezmiennikiem (czyli aksjomatem) języka programowania obiektowego, że wartością zmiennej x może być tylko obiekt jakiejś podklasy U klasy T lub obiekt pusty (specjalna wartość) none [2].
W Loglanie i Simuli67 występują dwa predykaty dwuargumentowe is oraz in. Niech T będzie nazwą pewnej klasy (czyli typu

[1]W języku Java słowo extends opisuje tę relację.

[2]W wielu językach w tej roli występuje napis null



Rysunek 1. zmienna r wskazuje warstwę Rachunek, zmienna s obie warstwy

opisanego przez klasę T. Niech ω będzie wyrażeniem obiektowym, np zmienną, wyrazeniem generującym obiekt lub nazewnikiem funkcyjnym pewnej funkcji f zwracającej obiekt jako wynik. Napisy postaci $(\omega \mathbf{is} T)$ oraz $(\omega \mathbf{in} T)$ są elementarnymi wyrażeniami boloowskimi
Niech v będzie stanem pamięci. Wartością wyrazenia boolowskiego $(\omega \mathbf{is} T)$ w stanie v jest true gdy wartość wyrażenia ω w stym stanie v jest obiektem typu T, w pozostałych przypadkach wartością wyrażenia $(\omega \mathbf{is} T)$ jest false. Relacja is zachodzi pomiędzy wartością zmiennej x i nazwą T klasy wtedy i tylko wtedy gdy obiekt – wartość zmiennej x jest obiektem typu T i nie jest obiektem jakiegokolwiek podtypu typu T.
Relacja in zachodzi pomiędzy wartością zmiennej x i nazwą T klasy wtedy i tylko wtedy gdy obiekt – wartość zmiennej x jest obiektem typu T lub jest obiektem jakiegokolwiek podtypu U typu T.
Aksjomaty tych relacji są następujące

$$
\begin{aligned}
Ax_{is} \quad & \big((\mathbf{new}\, T(args))\, \mathbf{is}\, T\big) \\
Ax_{in'} \quad & \big((x\, \mathbf{is}\, T) \Rightarrow (x\, \mathbf{in}\, T)\big) \\
Ax_{in} \quad & \big((x\, \mathbf{in}\, T) \wedge (T\, \mathbf{inh}\, U) \Rightarrow (x\ \mathbf{in} U)\big)
\end{aligned}
$$

Stosując aksjomat instrukcji przypisania otrzymujemy następujący fakt

$$\boxed{\big((x\, \mathbf{is}\, T) \wedge (type(y) = U) \wedge (T\, inh^*\, U) \Rightarrow \{y := x\}(y\, \mathbf{is}\, T)\big)}$$

Wynika stąd, że aktualnym typem zmiennej ...
Przykład. Dane są zmienne r: Rachunek i s: RachunekZaGaz. Wspólną wartoścla tych zmiennych może być w pewnym momencie obiekt typu RachunekZaGaz. Wyrażenie s.m3 jest poprawne, a wyrazenie r.m3 nie jest poprawne i kompilator zgłosi błąd.

tu Rysunek !
Zmiana kwalifikacji - zmiana typu aktualnego wyrażenia obiektowego qua
Jeśli zmienna x jest typu U i U inh T

1.3. Jakie działania udostępnia obiekt x? Podobnie jak w przypadku obiektów tablicowych można obiektowi wskazywanemu przez zmienną x nadać dodatkową nazwę y. Należy w tym celu wykonać instrukcję przypisania

$$y := x$$

Uwaga! Typ zmiennej y musi być zgodny z typem zmiennej x i obiektu o wskazywanego przez zmienna x. W najprostszym przypadku wystarczy by type te były równe.

Pola obiektu x mogą być odczytywane z zewnątrz. Np. x.kwota. Jest to wyrażenie całkowito-liczbowe. Pole obiektu x może otrzymać nową wartość

$$x.kwota := \omega$$

Metody obiektu x mogą być wykonywane zdalnie, jeśli klasa T zawiera lokalną deklarację funkcji f to wyrażenie $x.f(...)$ spowoduje obliczenie An object is offering its resources. Which ones?

1.4. Reguła konkatenacji, odsłona 2 (inner).

Przykład 14.2. (c.d.) Co się stanie gdy dodamy instrukcje drukowania rachunku? Można to zrobić w klasie rachunek i w klasach dziedziczących z tej klasy.

```
unit rachunek: class(who: person, dzien: date);
  var kwota: integer, paid: Boolean;
begin
  writeln("rachunek z dnia: ", day);
  writeln(" dla P. ", who);
  inner; (* ta pseudo instrukcja oddziela prolog od epilogu. *)
  writlen("kwota do zapłaty: ", kwota)
end rachunek;
```

Teraz możemy przygotować rachunki za prąd

```
unit RachunekZaPrad: rachunek class(kWh: integer);
begin
  kwota := kWh * price_kWh;
  writeln("zużyto kWh", kWh, "w cenie:", price_kWh);
end RachunekZaPrad;
```

Pseudoinstrukcja inner w klasie bazowej rachunek zostaje zastąpiona instrukcjami klasy dziedziczącej rachuneZaEnergie i w ten sposób osiągamy spójną postać rachunku za prąd.

unit A: **class**(pfA); *declA* **begin** ; I; **inner**; (* a slot to put instructions*) J **end** A	**unit** B: **class**(pfB); *declB*; **begin** K **end** B
if you prefix the word class in the declaration of B by name A	
unit B: A **class**(pfB); **begin** K **end** B	
then class B is concatenated to class A	
	unit B: **class**(pfA, pfB); *declA; declB* **begin** I; K; J **end** B

Pseudoinstrukcja inner może być pojmowana jako ramka w której można umieszczać instrukcje klas dziedziczących z danej klasy.

Uwaga 1. Ta pseudo instrukcja moze zastapić dowolną instrukcję atomowa w klasie bazowej. Nie musi ona , tak jak w przykładzie dzielić instrukcje klasy bazowej na prolog i epilog. Możesz ją umieścić wewnątrz instrukcji warunkowej lub wewnątrz instrukcji powtarzania. Zobacz interesujący przykład zastosowania inner w algorytmie heapsort.

Uwaga 2. Jeśli nie napiszesz tej pseudoinstrukcji sam, to domyslnie instrukcja inner umieszczona jest jako ostatnia instrukcja klasy.

Jeszcze jedno, pseudoinstrukcja inner może wystąpić tylko raz w klasie bazowej i kazdej klasie pochodnej (podklasie).

1.5. Reguła konkatenacji, odsłona 3 (Widoczność i statyczne dowiązania. Popatrzmy na następujący przykład

unit A: **class**; **var** x: T1, z: T3 **begin** I(x,z); **inner**; J(x) **end** A;	**unit** B: A **class**; **var** x: T2, y: T4 **begin** K(x,z,y) **end** B

W obecności powyższych deklaracji następujące polecenie

p:=**new** B

zostanie wykonane tak jak by deklaracja klasy B miała postać:

```
unit B;
  var x_A: T1, z_A: T3; var x_B: T2, y_B: T4
begin
  I(x/x_A, z/z_A);
  K(x/x_B, z/z_A, y/y_B);
  J(x/x_A)
end B
```

Struktura obiektu p jest więc taka

	x_A:	T1
	z_A:	T3
	x_B:	T2
p →	y_B:	T4
		I(x/x_A, z/z_A); K(x/x_B, z/z_A, y/y_B); J(x/x_A)

No dobrze, utworzyliśmy obiekt o klasy B i przypisaliśmy go jako wartość zmiennej p. Zostały wykonane instrukcje I(); K(); J() inicjalizujące odpowiednio pola tego obiektu.

1.6. Co udostępnia obiekt p? Pamiętamy ... W tym konkretnym przypadku wyrażenie p.z zwróci wartość pola z_B typu T3. A co będzie wartością wyrażenia q.x? Musimy znać typ zmiennej p. Przypuśćmy, że zadeklarowano dwie zmienne

var p:a, q:B;

Przypuśćmy, że wykonano instrukcje

q := new B;
p := q;

Uwaga! Typy i wartości wyrażeń q.x oraz p.x mogą być całkowicie odmienne!
Wartością wyrażenia q.x jest wartość pola x_B, typem tego wyrażenia jest T2. Natomiast, wartością wyrażenia p.x jest wartość pola x_A a typem tego wyrażenia jest T1.
Jeśli x jest nazwą procedury (lub funkcji) zadeklarowanej dwukrotnie w klasie A i w klasie B to rzecz ma się podobnie tzn. wyrażenia

p.functn(...) i q.functn...

mają odmienne znaczenia.

1.7. Co wiadomo o wartości zmiennej x? c.d. Z powyższych rozważań wynika, że statyczny typ zadeklarowanej zmiennej p nie musi być równy dynamicznemu typowi wyrażenia p. Zadeklarowanym typem zmiennej p była nazwa klasy A, podczas wykonywania programu zmiennej ten przypisano obiekt typu B. Nie narusza to aksjomatów języka. Ponieważ przypisując obiekt typu B zmiennej p działamy zgodnie z aksjomatem Ax_{in}. Przypuścmy, że oprócz zmiennej p jeszcze zmienna r jest takze

zadeklarowana jako typu B.
Po wykonaniu polecenia p:=q polecenie r:=p nie jest poprawne. Natomiast instrukcja r:= p **qua** B jest w tej sytuacji poprawne, ponieważ bieżącą wartośćia zmiennej p jest obiekt ypu B. rys.

1.8. Reguła konkatenacji, odsłona 4 (Sposoby ochrony). Closing the access to an attribute. Hiding an attribute. Filtering the inherited attributes. In any class you can put a restriction specification. There are three forms of such specification. One can forbid a remote access to the variables of a class. It is done by means of close specification. It is possible to hide an attribute to any module that inherits from a current module. It is done by means of hidden specification. If you wish you may inherit only the attributes from a list. taken In any case the absence of a restriction specification means no restriction at all.

1.9. Reguła konkatenacji, odsłona 5 (Metody wirtualne). Zdarza się, że w penej hierarchii klas, każda podklasa defiuneje pewną operację na swôj sposôb. Pomyśl o klasie Expression i operacji wyznaczania pochodnej. klasa A jest

Przykład 14.3. W zadaniu symbolicznego rôżniczkowania należy zdefiniować klasę Expression i metodę obliczającą nowe wyrażenie tj. pochodną wyrażenia ω ze względu na zmienną x.

```
unit Expr: class;
  unit virtual deriv: funtion(z: variable): Expr;
  end deriv
begin
end Expr
```

i wiele podklas tej klasy, np. klasa opisująca sumy

```
unit Sum: Expr class(L,R: Expr);
  unit virtual deriv: funtion(z: variable): Expr;
  begin
    result:= new Sum(L.deriv(z),R.deriv(z))
  end deriv
begin
end Sum
```

klasa iloczyn

```
unit Mult: Expr class(L,R: Expr);
  unit virtual deriv: funtion(z: variable): Expr;
  begin
    result:= new Sum(new Mult(L.deriv(z),R),
                 new Mult(L,R.deriv(z)))
  end deriv
begin
end Mult
```

i klasy Sin oraz Cos

```
unit Sin: Expr class(a: Expr);
  unit virtual deriv: funtion(z: variable): Expr;
  begin
     result:= new Mult(new Cos(a),a.deriv(z))
  end deriv
begin
end Sin
```

Klasę Cos potrafisz napisać sama. Warto też stworzyć deklaracle klas Zmienna i Stała. ♦

Po tym przykładzie spróbujemy teraz opisać znaczenie słowa virtual w bardziej abstrakcyjny, ogólny sposób.

```
unit A: class;
  unit virtual p:
    procedure(x,y:T);
    ...
  end p;
begin
  I(p);
  inner;
  J(p)
end A
```

```
unit B: A class;
  unit virtual p:
    procedure(u,v: T1);
  (* UWAGA! typy T i T1
   muszą być zgodne *)
  end p;
begin
  K(p)
end B
```

Inaczej niż widzieliśmy to wcześniej, wirtualna metoda zadeklarowana w klasie dziedzicącej (tj. w podklasie) zastąpi metodę p zadeklarowaną w klasie bazowej.

```
unit B: A class;
  unit virtual pB: procedure(u, v: T1)
  end pB;
begin
  I(p/pB);
  K(p/pB);
  J(p/pB)
end B
```

Uwaga! Mechanizm wirtualnych funkcji i procedur wymaga by słowo virtual pojawiło się i w klasie A i w klasie B. Jeżeli jednego zabraknie to instrukcje w treści klasy B przyjmą zwykłą postać

```
begin
  I(p/pA);
  K(p/pB);
  J(p/pA)
end B
```

The mechanism of virtual procedures and functions offers many interesting possibilities. On the other hand it is considered as slowing the execution.

2. Rozszerzanie innych modułów

Rozszerzanie może być przydatne w projektowaniu i tworzeniu innych modułów, nie tylko klas. Przypomnijmy, w programie mogą pojawić się: bloki, funkcje, procedury, klasy, współprogramy, procesy i moduły obsługi sygnałów (handlery). Rozszerzanie jest operacją tworząca na podstawie dwu modułów A i B nowy moduł A moduł B. Pierwszym argumentem tej operacji jest moduł klasy (lub współprogramu lub procesu). O pierwszym argumencie rozszerzana mówimy krótko prefix. Drugim argumentem może być dowolny moduł inny niz moduł obsługi sygnałów.

Wynik rozszerzania modułu

prefiks A moduł	unit A: class	unit A: coroutine	unit A: process
pref A block	blok	niedozwolone	niedozwolone
unit p: A procedure	procedura	niedozwolone	niedozwolone
unit f: A function	funkcja	niedozwolone	niedozwolone
unit B: A class	klasa	współprogram	proces
unit B: A coroutine	współprogram	współprogram	proces
unit Pr: A process	proces	proces	proces

Rozszerzane tj. dziedziczone mogą być klasy, współprogramy lub procesy. Te trzy przypadki wyliczamy w kolumnach: drugiej, trzeciej i czwartej.
W kolejnych wierszach tablicy opisujemy rodzaj modułu wynikowego wymieniamy moduły, które mogą rozszerzać dany moduł A – prefiks.

Obietnice

Wykorzystując dziedziczenie i zagnieżdżanie modułów programiści mogą osiągać wiele celów:

- wydzielanie wspólnych części rozmaitych struktur danych,
- konstrukcja hierarchii struktur,
- wyciąganie wspólnych części różnych algorytmów,
- implementacja różnych języków problemowo zorientowanych,
- realizacja działań algebry zbiorów na typach danych,
- tworzenie abstrakcyjnych algorytmów, Jitem i wiele innych...

2.1. nnn.

2.2. hierarchie. Przykładem mogą służyć klasy rachunek i podklasy tej klasy.
Inne hierarchie to hierarchia klas reprezentujących obiekty wyrażeń, ...

2.3. faktoryzacja algorytmów. Przykład to algorytmy insert, member i delete w strukturze BST drzew binarnych poszukiwań.

2.4. języki problemowo zorientowane. Struktura geometrii cyrkla i linijki
class Geoplan

2.5. operacje algebry zbiorów. Iloczyn kartezjański - klasa
Suma teoriomnogościowa: klasa rerezentująca sumę dwu zbiorów obiektów ...

2.6. algorytmy abstrakcyjne. W literaturze występują nazwy: algorytmy generyczne, algorytmy uogólnione, ... np. Gsort

3. Protokoły call i new

W tym podrozdziale omówimy bardziej szczegółowo protokoły call oraz new. Pierwszy z tych protokołów wyznacza sposób współpracy modułu (zleceniodawcy) wydającego polecenie wykonania procedury lub obliczenia wartości funkcji z modułem (zleceniobiorcą) realizujacym takie polecenie. Moze to być instrukcja procedury lub nazewnik funkcyjny. Natomiast protokół new pomimo podobieństw ma nieco inne zadanie.
Działanie protokołu rozpoczyna się przekazaniem parametrów aktualnych instrukcji procedury lub odpowiednio argumentów polecenia new utwórz nowy obiekt.
Co może się znaleźć na liście parametrów formalnych modułu?

- zmienna
- nazwa funkcji,
- nazwa procedury,
- nazwa typu,
  opisanego w pewnej deklaracji klasy lub współprogramu lub procesu

Jakie wyrażenia mogą się znaleźć na liście parametrów aktualnych instrukcji procedury call (wyrażenia generującego obiekt new)?
Parametrami formalnymi mogą być:

- zmienna z opisem input,
- zmienna z opisem out,
- zmienna z opisem inout,
- function,
- procedure,
- type,

Przykład 14.4. Zadanie polega na udostępnieniu funkcji Tk obliczającej wartość trójmianu kwadratowego.

```
unit Tk: function(a,b,c,x:real):real;
begin
  result := (a*x+b)*x+c
end Tk;
```

Jeśli chcemy wykorzystać tę funkcję to należy wykonać np. takie polecenie

```
y:=Tk(2.3+exp(2,2.2), cos(3.67), -88.1, 5.0)
```

4. Statyczna struktura programu

W tym podrozdziale opisujemy warunki, jakie musi spełniać tekst by uznać, że jest to poprawnie napisany program.
Przy pierwszym czytaniu możesz pominąć ten podrozdział.
Każdy program P jest zbiorem modułów.

$$P = \{M\}$$

Na zbiorze modułów określone są dwie funkcje $decl$ oraz inh.
Funkcja

(75) $$decl\colon M \to M$$

Zbiór M razem z funkcją $decl$ jest drzewem. Korzeniem tego drzewa jest gówny blok, czyli moduł zawierający wszystkie moduły programu. Wynikiem funkcji $decl(m)$ jest moduł m' bezpośrednio zawierający dany moduł. Funkcja $decl$ nie jest określona dla głownego bloku programu, tzn. na korzeniu tego drzewa.
Większość modułów ma przypisaną im nazwę.

(76) $$name(m) \in \{Identyfier\}$$

Zauważ, nie każdy moduł posiada nazwę. Moduły bloku oraz moduły obsługi sygnału nie posiadają nazwy.
Jest to pierwsza przyczyna dla której nie możemy utożsamiac modułu z nazwą podaną w jego deklaracji. Po drugie, jak to już mogłaś zauważyć w programie może wystąpić, w różnych jego miejscach, wiele różnych modułów o tej samej nazwie.

Problem. W jaki sposób mamy identyfikować moduły?

Na ogół nie ma potrzeby zajmowania się tym problemem. Ale ...

Druga funkcja określona na zbiorze modułów programu to inh. Najpierw zanotujmy, że w zbiorze modułów znajduje się predefiniowany moduł klasy o nazwie DI – dynamic instance. Klasa ta jest ukryta przed programistą. Jej atrybuty to m.in SL odnośnik do modułu obejmującego ... i inne.
Funkcja inh jest określona na modułach klas lub modułach rozszerzających jakąś klasę. Niech $Classes$ będzie nazwą tego zbioru modułów. $Classes \subset M$.

(77) $$inh\colon Classes \to Classes$$

I tu mamy kolejny problem. Ponieważ moduł klasy nie musi być jednoznacznie określony przez nazwę klasy podaną w jej

deklaracji. Pojęcie wystąpienia identyfikatora ...
Wewnatrz danego modułu m znaczenie danej nazwy id jest takie samo dla rôznych wystąpień tej nazwy id w module m. Pozwala to nam wprowadzić kolejne funkcje

(78) $$type\colon Identifiers \times M \to Typ$$

oraz

(79) $$bind\colon Identifiers \times M \to Typ$$

Zauważ, inh jest funkcją określoną na modułach, a w tekście programu mamy tylko nazwę klasy rozszerzanej. Wiemy, że nie wystarczy to do identyfikacji modułu. Inaczej mówiąc nazwa Cl występująca w różnych miejscach programu jako prefiks może mieć (i często ma) różne znaczenia.
Ponadto
bind
typ

Warunki. Jakie warunki ma spełniać tekst by uznać, że jest to poprawnie napisany program?

I) Wymagania I
   (1) wyrażenia muszą mieć poprawną strukturę,
   (2) instrukcje przypisania mają byc parami <zmienna>, <wyrażenie>,
   (3) instrukcje warunkowe mają mieć postać
   if γ then K else M fi
   gdzie napis γ ma być wyrażeniem boolowskim, a napisy K,M mają być instrukcjami,
   (4)

II) Wymagania zgodności typów (?)
   (1) każde wystąpienie nazwy(tj. identyfikatora) ma mieć określony typ,
   (2) w obrębie danego modułu M: kazde wystąpienie nazwy id ma mieć ten sam typ
   (3)

5. Dynamiczna struktura

W trakcie wykonywania programu P mamy do czynienia z grafem jednostek dynamicznych utworzonych na podstawie modułów.
Funkcje
SL
DL
ext
Relacje
is
in

Ćwiczenia

14.1. Napisz hierarchię co najmniej 6 podklas danej klasy bazowej C. Dwie z klas pochodnych nie mają być w relacji $X \,\mathbf{inh}\, Y$ ani $Y \,\mathbf{inh}\, X$.

14.2. Napisz jak najwięcej klas pochodnych klasy $Expr$. Kazda z nich ma zawierać funkcję wirtualna $deriv$.

14.3. Utwórz obiekt reprezentujący wyrażenie $sin(x+1/x)/log(cos(x)$ i oblicz pochodną tego wyrażenia ze względu na zmienną x.

14.4. Oblicz pochodną tego samego wyrazenia NIE posługując się wirtualną funkcją deriv.

Część 3

Teorie algorytmiczne niektórych struktur danych

ROZDZIAŁ 15

Wprowadzenie

Motywacje

Utarł się pogląd, że dowodzenie poprawności programu jest nierealne, ponieważ jest trudne i wogóle ...
Naszym zdaniem nie tylko można przedstawiać argumenty na rzecz tez w rodzaju: obliczenie programu P będzie skończone lub jeśli dane spełniają warunek α to program K będzie mieć obliczenie skończone i wyniki spełnią warunek β.
Nie wystarczy intuicja, nie wystarcze przekonanie, że jakoś to będzie.
Pokażemy, że powszechnie akceptowane intuicyjne definicje struktur danych stosy, kolejki obejmują patologiczne struktury, oprócz tych prawidłowych.

Dlaczego taki tytuł?

Czy taki tytuł tej części jest jakoś uzasadniony? Dwa są powody by wyodrębnić algorytmiczną teorię zbiorów od pozostałych rozdziałów:

(1) Zastosowania. Od samego początku istnienia komputerów zastosowania w biznesie, administracji etc., zdominowały inne ich zastosowania np. naukowe. W tych zastosowaniach operacjami dominującymi były działania na skończonych zbiorach danych np. danych o stanie magazynu, danych potrzebnych dostworzenia listy płac. Z biegiem lat programy operowały na coraz większych zbiorach danych. Równolegle, technologia komputerowa umożliwiała przechowywanie coraz większych zbiorów danych.
A dzisiaj? W każdym telefonie komputer zarządza zbiorem kontaktów, by wymienić tylko jeden przykład działań na zbiorach.
Big data to modne dziś hasło. Idzie o to, że w sieci gromadzi się bardzo duże zbiory danych i że zawierają one cenne dla właściciela informacje.

(2) Dorobek. Ostatnie 70 lat przyniosły tak wiele wyników, że warto je zgrupować pod oddzielnym hasłem. Mamy tu na myśli struktury danych opisane i analizowane w algorytmice. Ale nie tylko ...

Czy zastanowiłeś się jaki wspólny tytuł nadać trzeciemu tomowi monografii Knutha "Wyszukiwanie i sortowanie"? A to właśnie algorytmiczna teoria zbiorów. W czasie jaki upłynął od ukazania się tej książki dokonał sie olbrzymi postęp i dziś lepiej dostrzegamy słuszność ...

Co tu znajdę?

Dwa pierwsze rozdziały poświęcone są stosom i kolejkom. Czyli skończonym ciągom elementów z operacjami wstawiania i usuwania elementu wykonywanymi na krańcu ciągu. Podamy aksjomatyczne definicje stosów i kolejek. Omówimy kilka oczywistych i parę nieoczywistych implementacji tych definicji. Weryfikacja ...

Kolejne rozdziały zajmują się teoriami zbiorów w których elementy nie są uszeregowane tj. nie są ciągami.
Kontenery – zbiory skończone bez wybranej relacji porządkującej elementy
Kolejki priorytetowe – zbiory skończone z wskazaną relacja porządku w zbiorze elementów.
Drzewa BST
Kopce

Struktury nieskończone definiowane przez klasy

Dwie uwagi:

- Każda klasa K definiuje nieskończony zbiór obiektów. Można utożsamiać klasę K ze zbiorem obiektów klasy K. A więc algorytmiczna teoria zbiorów dotyczy nie tylko zbiorów skończonych, które komputer "ma w garści". W pewnym momencie powinniśmy się zastanowić jakie operacje można wykonywac na tych nieskończonych zbiorach i jakie te operacje mają własności. Zadanie to szkicujemy w nastepnym punkcie.
  Ponadto klasa definiuje operacje. A więc, klasa jest definicją struktury algebraicznej.
- WYZWANIE. Warto zbadać relacje pomiędzy pojęciem klasy w programowaniu obiektowym i słabo zbadane pojęcie definiowalności struktury algebraicznej. W historii matematyki zapamietano słowa Leopolda Kroneckera (1823-1891), który zasłynął - między innymi - zdaniem: "dobry Bóg stworzył liczby naturalne, reszta jest dziełem człowieka".

  Popatrzmy na następującą kolekcję klas
  unit natural: class ...
  unit integer: class
  unit pair: class
  unit properpair: pair class

unit fraction: properpair class

Jesteśmy przekonani, że niedługo zostaną odkryte nowe problemy i ...

ROZDZIAŁ 16

Stosy

Ten rozdział poświęcamy strukturze stosów. Dwa są powody by obszerniej zająć się tematem stosów:

- ponieważ struktura ta ma wiele zastosowań, i
- ponieważ na przykładzie struktury stosów przedstawimy problemy wiążące się z specyfikowaniem klas i z przeprowadzaniem dowodów.

1. Specyfikacja stosów

Czym są stosy? Jak Twoim zdaniem powinna brzmieć odpowiedź na to pytanie?

Na ogół nie zawracamy sobie głowy tym problemem. W dawnej Polsce w encyklopedii o tytule "Nowe Ateny" pod hasłem koń czytamy koń jaki jest każdy widzi. I słusznie. W tamtych czasach, jeśli ktoś umiał czytać, to na pewno widział konia.

Większość z nas spotkała się ze stosem. Najkrótsza definicja stosu to LIFO – skrót angielskieo zwrotu Last In First Out. Ale czy to jest definicja?

Spróbuj jednak napisać definicję stosu. A przynajmniej spróbuj napisać wymagania W na to by klasa K mogła być uznana za klasę definiująca stosy liczb całkowitych.

Aha, nie jest to oczywiste. Programujący w Javie napiszą taki interfejs

```
interface Stosl {
  void push(int e)
  void pop()
  int top()
}
```

lub coś podobnego i powiedzą Stos ma trzy metody: push, pop i top. Porównaj []. Świetnie. Czy jednak następująca deklaracja klasy jest poprawną implementacją tego interfejsu?

```
class Stos implements StosI {
  private int[8] tabl;
  private int i,j;
  void push(int e) {tabl[j++] = e }
  void pop() {i++ }
  int top() {return tabl[i]}
}
```

Czy powyższa klasa Stos implementuje interfejs StosI? Kompilator Javy zaakceptuje zdanie "class Stos implements (interface) StosI". Podstawą jest obserwacja, że klasa Stos zawiera deklaracje trzech metod push, pop i top, a ponadto typy argumentów i wyników są zgodne dla każdej z tych metod. Czy jednak jest to stos? Hmm! Większość programistów wykrzyknie, "przecież ta klasa implementuje kolejki FIFO"! Czegoś zabrakło w naszej specyfikacji – interfejsie.
Zgadzamy się co do tego, że każda implementacja stosów liczb całkowitych, musi udostępniać trzy działania:

$push$: $int \times Stos \to Stos$
pop: $Stos \to Stos$
top: $Stos \to int$

Ale takie wyliczenie to za mało. Powinniśmy jeszcze scharakteryzować efekty działania tych operacji. Tak jak to zrobiliśmy w rozdziale 3 opisując własności działań na typach prostych. Pamietasz?
Co więc należy podać jako własności (t.j. aksjomaty) stosów? Zauważ następujący związek operacji $push$ i top.

$$top(push(e, s)) = e$$

Powyższą formułę czytamy: operacja top zastosowana do wyniku operacji $push(e, s)$ zwraca element e. Możemy ją poprzedzić kwantyfikatorem ogólnym. Mamy więc pierwszy aksjomat stosów

$$\forall_s \forall_e top(push(e, s)) = e$$

?Można tę formułę napisać w ortografii programowania obiektowego w taki sposób:

$$\forall_s \forall_e (s.push(e)).top = e$$

Dodajmy jeszcze podobną formułę

$$\forall_s \forall_e pop(push(e, s)) = s$$

oraz kolejne postulaty

$$empty(newStos)$$

$$\forall_s \forall_e \neg empty(push(e, s))$$

Tutaj $empty$ jest funkcją boolowską (czyli predykatem). Wartość wyrażenia $empty(s)$ jest true wtedy i tylko wtedy, gdy argument s jest stosem pustym.

newStos – zwraca stos pusty. Zauważ, że z własności klas wynika, że $newStos \neq newStos$. Mamy już cztery aksjomaty. Czy to wystarczy?

Odejdźmy od założenia, że elementami stosu są liczby naturalne. Przyjmijmy, że dana jest pewna klasa Elem i to właśnie obiekty tej klasy są elementami stosów. Operacja $push$ wkłada element e do stosu s[1] a jej wynikiem jest stos $push(e, s)$. Inna operacja pop zdejmuje element ze stosu i zwraca stos $pop(s)$. Operacja top zwraca element $top(s)$. Dwie ostatnie operacje nie są zdefiniowane, gdy argument s jest stosem pustym, tj. gdy zachodzi $empty(s)$.

Możemy to podsumować w ten sposób: struktura algebraiczna stosów ma swoje uniwersum, na które składa się unia dwu zbiorów: zbioru elementów E i zbioru stosów S. Ponadto mamy trzy operacje: $push, pop, top$ oraz dwa predykaty $empty$ i równość $=$.

Tablica 15.1. Specyfikacja $S1$ struktury stosów

Sygnatura czyli interfejs	Komentarze
Uniwersum $= E \cup S$	
E	zbiór elementów
S	zbiór stosów
Operacje	
$push : E \times S \longrightarrow S$	włóż element e do stosu s
$pop : S \longrightarrow S$	zdejm wierzchołek stosu
wynik jest określony	wttw gdy $\neg empty(s)$
$top : S \longrightarrow E$	zwróć wierzhołek stosu
wynik jest określony	wttw gdy $\neg empty(s)$
$newStack :\longrightarrow S$	stos pusty
Relacje	
$empty : S \longrightarrow \{true, false\}$	czy stos jest pusty?
$=: E \times E \cup S \times S \longrightarrow \{true, false\}$	relacja równości
Aksjomaty czyli niezmienniki stosów	**t.j. własności operacji**
s1) $\forall_{e \in E} \forall_{s \in S}\ \neg empty(push(e, s))$	wynik operacji push jest stosem niepustym
s2) $\forall_{e \in E} \forall_{s \in S}\ e = top(push(e, s))$	element ostatnio włożony na stos jest na wierzchołku stosu
s3) $\forall_{e \in E} \forall_{s \in S}\ s = pop(push(e, s))$	po wykonaniu operacji push, operacja pop odtwarza stos
s4) $empty(newStack)$	nowy stos jest pusty

Zbiór S jest pojmowany zazwyczaj jako zbiór wszystkich obiektów jakie można utworzyć na podstawie klasy S, podobnie pojmowany jest zbiór E.

[1] Zauważ, czasami wystarczy do stosu wpisać odnośnik do elementu e

Tablica 15.2. Dwa modele specyfikacji S_1

Model zaprogramowany	Model mathematyczny
unit Elem: **class**; ... **end** Elem;	$E = \{a, b, c, ...\}$
class **Stos** {	S = **set of all finite sequences over alphabet E,**
private class Linkage {	**the empty sequence** λ **included.**
Linkage next;	$newstack = \lambda$
Elem el;	$push(e, \{e_1, e_2, ..., e_n\}) = \{e, e_1, e_2, ..., e_n\}$
Linkage(Elem e, Linkage n){el=e; next=n;}	
} // end Linkage	
public Linkage topv;	
public Stos(){topv=null;}	
public static final Stos **push**(Elem e, Stos s) {	$top(\{e_1, ..., e_n\}) = e_1$
Stos n = new Stos();	$top(\lambda)$ **is undefined**
n.topv = new Linkage(e, s.topv);	
return n; } // end push	$pop(\{e_1, e_2, e_3, ..., e_n\}) = \{e_2, e_3, ..., e_n\}$
public static final Elem **top**(Stos s) throws Undef {	$pop(\{e_1\}) = \lambda$
if (s.topv=null) throw new Undef();	$pop(\lambda)$ **is undefined**
return s.topv.el;	
} // end top	
public static final Stos **pop**(Stos s) throws Undef {	$empty(s) \equiv s = \lambda$
if (s.topv==null) throw new Undef();	**equality** = **is meant as identity**
Stos n =new Stos();	
n.topv=s.topv.next; return n;	
} //end pop	
public static final Boolean **empty**(Stos s) {	
return (s.topv==null);	
} // end empty	
public static final Boolean **equal**(Stos s1,Stos s2) {	
Boolean aux=true;	**stacks are equal iff**
Boolean aux1=Stos.empty(s1);	**they have the same elements**
Boolean aux2=Stos.empty(s2);	**on the same positions.**
while (!aux1&&!aux2&&aux) {	
aux = (Stos.top(s1) == Stos.top(s2));	
s1 = Stos.pop(s1); aux1 = Stos.empty(s1);	
s2 = Stos.pop(s2); aux2 = Stos.empty(s2);	
}	
return (aux1 && aux2 && aux);	
} // end equal	
} // end Stos	
class Undef extends Exception { ... }	

Tabela 15.2 zawiera dwie implementacje specyfikacji S1 z tablicy 15.1. W lewej kolumnie umieściliśmy klasę Stos implementującą specyfikację S1. Prawa kolumna zawiera matematyczny model tej specyfikacji. Model matematyczny nazywać będziemy modelem standardowym stosów. Dla dowolnego zbioru E można skonstruować model standardowy bazując na zbiorze E. Wszytkie takie modele są podobne. Nie muszą jednak być izomorficzne. Wystarczy rozważyć dwa modele standardowe, jeden zbudowany nad zbiorem E_1 i drugi nad zbiorem E_2, przy czym zbiory te są rôżnej mocy, $card(E_1) \neq card(E_2)$.

Wielu autorów przyjmuje formuły zawarte w tabeli 15.1 jako specyfikację stosów np. [EM85], [AVN+07]. Jednak ten zbiór formuł nie mówi całej prawdy o stosach. Wynika to z następującego lematu.

Lemat 16.1. Formuła

$$(\forall s \in S)\ \neg empty(s) \Rightarrow\ \ s = push(top(s), pop(s))$$

jest niezalezna od aksjomatów s1 - s4.
Formuła ta mówi: dla każdego niepustego stosu s, wynik operacji $push$ włozenia elementu $top(s)$ do stosu $pop(s)$ jest stosem s.

Dowód. Rozwazmy strukturę I_2, opisaną w Tablicy 15.3. Nietrudno sprawdzić, że jest to model aksjomatów s1 - s4, tj. wszytkie cztery formuły są prawdziwe w strukturze I_2. Zobaczmy, że formuła wymienion w lemacie nie jest prawdziwa w tej strukturze. Rozpatrzmy stos $s = \{e_1, e_2, e_3, ..., e_n\}$ taki, że $e_1 \neq e_2$. Oczywiście $top(s) = e_1$ and $pop(s) = \{e_3, ..., e_n\}$. Ale $push(top(s), pop(s) = \{e_1, e_1, e_3, ..., e_n\} \neq s$. □

Tablica 15.3 Model I_2

$E = \{a, b, c\}$
S = zbiór wszystkich skończonych ciągów znaków ze zbioru E, włączając pusty ciąg λ.
$push(e, \{e_1, e_2, ..., e_n\}) = \{e, e, e_1, e_2, ..., e_n\}$
$top(\{e_1, ..., e_n\}) = e_1$
$top(\lambda)$ niekreślony
$pop(\{e_1, e_2, e_3, ..., e_n\}) = \{e_3, ..., e_n\}$
$pop(\{e_1\}) = pop(\{e_1, e_2\}) = \lambda$, $pop(\lambda)$ nieokreślony

Upoważnia nas to do przedstawienia pełniejszej specyfikacji stosów, zob. Tablicza 15.4.

Tablica 15.4 Specyfikacja stosów S2

Sygnatura	taka sama jak w $S1$
Aksjomaty	
aksjomaty s1 - s4 oraz	
s5) $(\forall s \in S)\ \neg empty(s) \Rightarrow$ $s = push(top(s), pop(s))$	dla każdego niepustego stosu s wynikiem operacji $push$ na elemencie $top(s)$ i stosie $pop(s)$ jest stos s

Ktoś może pomyslec, im więcej formuł (dodamy do specyfikacji) tym lepiej. To się jednak może skończyć źle. W wyniku mozna otrzymać specyfikację sprzeczną.
Spójrzmy na następujący przykład S3 w tabeli 1.

Tablica 15.5 Specyfikacja stosów S3

Sygnatura
podobnie jak w specyfikacji S1, powiększona o dwie stałe a, b typu E
Aksjomaty
aksjomaty s1 - s5 oraz sQ) $\neg empty(s) \Rightarrow push(e, pop(s)) = pop(push(e, s))$ oraz aksjomat s2E) $a \neq b$

Twierdzenie 16.2. Zbiór formuł $\{s1 - s5, sQ, s2E\}$ jest zbiorem sprzecznym.

Dowód. Aksjomat s2E) stwierdza, że zbiór elementów E ma conajmniej dwa elementy a i b różne. Załóżmy, że $s \in S$ jest stosem niepustym. Mamy wtedy:

(1)	$s_1 \stackrel{df}{=} push(a, s)$	z definicji
(2)	$s_2 \stackrel{df}{=} push(b, s)$	z definicji
(3)	$s = pop(s_1)$	z (1) na mocy s3)
(4)	$s_2 = push(b, pop(s_1))$	z (2) i (3), s jest niepusty
(5)	$s_2 = pop(push(b, s_1))$	z (4) na mocy sQ)
(6)	$s_2 = s_1$	z (5) na mocy s3)
(7)	$b = top(s_2) = top(s_1) = a$	z (6) na mocy s2)

Sprzeczność! A więc specyfikacja S3 jest sprzeczna. □

Wniosek 16.3. Specyfikacja S3 nie ma żadnej implementacji.

Skąd wiadomo, że tak jest? Gdyby istniała jakaś implementacja tego zbioru aksjomatów, to musiałaby równocześnie spełniać dwie formuły $a = b$ i $\neg(a = b)$.
Stwierdzenie, że specyfikacja pozwala wyprowadzić zarówno pewną formułę α jak i jej negację (a więc, że jest sprzeczna) to sygnał alarmowy.W żadnym przypadku nie należy podejmować się zlecenia na wytworzenie oprogramowania, które ma spełniać wymagania sprzeczne. A jeśli nie jesteśmy pewni, że specyfikacja, która przecież jest załącznikiem do umowy o dzieło, jest wolna od sprzeczności, to wpiszmy do umowy odpowiedni kodycyl stanowiący o wysokości odszkodowania dla zleceniobiorcy za utracony czas i inne szkody. Ma to na celu nakłonienie zleceniodawcy do refleksji. Najlepiej jednak by zleceniodawca powierzył opracowanie specyfikacji fachowcowi
Wracamy do specyfikacji S2. Po dokładniejszej analizie zauważamy, że można dodac do tej specyfikacji nieskończony zbiór dodatkowych formuł. Wszytkie te formuły mają strukturę zgodną

ze schematem indukcji (strukturalnej) dla stosów. W ten sposób dochodzimy do kolejnej specyfikacji S4, por. Tablica 15.6.

Tablica 15.6 Specyfikacja stosów S4

Sygnatura
taka sama jak w S1
Aksjomaty
aksjomaty s1 - s5 oraz
wszystkie formuły o następującym schemacie IS
$\alpha(s/s_0) \wedge \{(\forall_{s\in S}(\forall_{e\in E}(\alpha(s) \Rightarrow \alpha(s/push(e,s)))\} \Rightarrow \forall_{s\in S}\alpha(s)$
gdzie α jest dowolną formułą pierwszego rzędu i $s_0 = newStack$

Schemat indukcji powiada: jeśli formuła $\alpha(x)$ jest prawdziwa dla stosu pustego $\alpha(x/s_0)$ i jeśli dla dkażdego stosu s i dla każdego elementu e, $\alpha(x/s)$ implikuje $\alpha(x/push(e,s))$ to dla każdego stosu s zachodzi formuła $\alpha(x/s)$.
Zauważ, schemat indukcji nie gwarantuje nieobecności stosów patologicznych. Może ktoś powiedzieć: rozpatrywać będziemy tylkoo stosy standardowe tj. takie, które powstały ze stosu pustego w wyniku skończonej liczby ooperacji $push$. Ale jak wyrazić tę własność w języku formuł? Może więc zastąpić to wymaganie innym? rozważać będziemy tylko programowalne modele specyfikacjji S4. Okazuje się, że przyjęcie tego exra wymagania nie eleminuje stosów patologicznych. W pracach [MS96, MSST00] udowodniono, że istnieją programowalne, patologiczne modele specifikacji S4. W tych modelach istnieją stosy takie, że można powtarzać operację pop i po żadnej skończonej liczbie powtórzeń nie uzyskamy pustego stosu.

Twierdzenie 16.4. Istnieje programowalny model specyfikacji S4 taki, że dla pewnego stosu s_1 program

$$\text{while } \neg\ empty(s_1) \text{ do } s_1 := pop(s_1) \text{ done}$$

nie kończy obliczeń.

Taki stos $s1$, nazywamy nieosiągalnym. Praca [MSST00] przynosi dwa kolejne fakty

Twierdzenie 16.5. Niech E będzie zbiorem skończonym. Niech $\mathfrak{S}$ oznacza strukturę stosów nad zbiorem E. Zbiór formuł pierwszego rzędu prawdziwych w strukturze $\mathfrak{S}$ jest rozstrzygalny.

Wydaje się, że jest to dobra wiadomość. Dobrze jest mieć procedurę rozstrzygania o prawdziwości formuł. Jednak, okazuje się, że jest to zła wiadomość. Wynika to z kolejnego twierdzenia.

Twierdzenie 16.6. Każda rozstrzygalna teoria pierwszego rzędu $\mathcal{T}$ posiada model programowalny i nieosiągalny, a więc patologiczny.

Wydaje się, że jesteśmy w impasie. Że nie można skonstruować aksjomatyzacji dla struktur danych o nieskończonym zbiorze. Okazuje się, że logika algorytmiczna przychodzi tu z pomocą (zanotujmy niespełnioną obietnicę [Dil90]). Rozważmy mianowicie następująca specyfikację S5. Schemat indukcji zostaje tu zastąpiony przez pojedynczą formułę algorytmiczną, por. Tabela 15.7.

Tablica 15.7 Specyfikacja S5

Sygnatura
taka sama jak w S1
Aksjomaty
aksjomaty s1 - s5 i
s6) $\forall_{s \in S}$ {**while** $\neg empty(s)$ **do** $s := pop(s)$ **done**} $empty(s)$
ten program nie zapętla się, tzn. każdy stos jest skończony

Zwróć uwagę na następujące twierdzenie, por. [MS87] str. 165.

Twierdzenie 16.7. (o reprezentacji) Każdy model specyfikacji S5 jest izomorficzny z pewnym standardowym modelem stosów.

Twierdzenie to mówi, że specyfikacja S5 uchwyciła wszystkie własności struktury algebraicznej stosy. Dowolny model zbioru aksjomatów S5 jest izomorficzny ze biorem skończonych ciągów elementów ze zbioru E, a operacje push, pop i top są określone tak jak w tablicy 15.2.
Zauważyłeś, że jeden z aksjomatów stwierdza, że dla każdego stosu s program wymieniony w aksjomatach zawsze kończy obliczenie. Ta własnośc okaże się bardzo przydatna w dowodach poprawności innych algorytmów.
Widzieliśmy różne specyfikacje stosów. Porównajmy je, zob. poniższą tablice 15.8.

Tablica 15.8 Porównanie specyfikacji S1 - S5

Specyfikacja	Uwagi
S_1	informacja o stosach niekompletna, np. formuła s5 jest niezależna od $\{s1, s2, s3, s4\}$ S_1 ma zaskakujące modele por. implementacja I_2
S_2	jeśli $card(E) = k, k \in N$, to teoria pierwszego rzędu S_2 jest rozstrzygalna, niekompletna informacja, dopuszcza modele patologiczne
S_3	specyfikacja sprzeczna, por. twierdzenie 16.2 nie istnieje żaden model
S_4	rozstrzygalna, niekompletna informacja dopuszcza implementacje patologiczne
S_5	informacja kompletna, każdy model jest izomorficzny z pewnym modelem standardowym algorytmiczna teoria jest nierozstrzygalna.

Uwaga 16.8. Zbiór formuł pierwszego rzędu prawdziwych w strukturze danych stosów nad skończonym zbiorem E elementów jest rozstrzygalny. Równocześnie specyfikacje S2 i S4 mają modele niestandardowe. W modelach tych polecenie s:=pop(s) może byc powtarzane dowolnie wiele razy i nie doprowadza do stosu pustego empty(s).

2. Implementacje stosów

Powszechnie znane są dwie implementacje stosów: stos jako tablica i wskaźnik, oraz stos jako lista. Przedstawimy jeszcze jedną, mniej znaną implementację stosów – stosy jako liczby.

2.1. Stosy jako tablice.

2.2. Stosy jako listy. Ta implementacja – lepiej mówić ten model, została opisana w tabeli 15.2. Klasa Stosy jaką tam zamieściliśmy jest definicją struktury algebraicznej $\mathfrak{S}_l$.

$$\mathfrak{S}_l \stackrel{df}{=} \langle \{|Elem| \cup |Stos|\}, push, pop, top, empty, equal \rangle$$

Zbiór $|Elem|$ jest zbiorem obiektów o, które spełniają relację $o\ is\ Elem$. Podobnie, zbiór $|Stos|$ jest zbiorem obiektów o, które spełniają relację $o\ is\ Stos$. Oba zbiory nie są zbiorami istnejącymi w trakcie wykonywania jakiegokolwiek programu. Myślimy o nich jako o abstrakcji, zbiorach potencjanych obiektów. Deklaracje funkcji stanowią definicje

2.3. Stosy jako liczby naturalne. Niech zbiór E elementów będzie skończony. W szczególnym przypadku gdy liczba elementów równa jest 10, możemy utożsamiać stosy z liczbami naturalnymi l, takimi, że $l \geq 10$. Stos pusty jest liczbą 10. Niech stos s będzie reprezentowany przez liczbę n. Wtedy wynik operacji $push(e, s)$ będzie reprezentowany przez liczbę $n * 10 + e$. W tym przypadku rozważamy następująca klasę.

```
unit Stosy10: class;
  const k=10;
 signal Emptystack;
 signal WrongElem;
 unit Elem: class(l: integer);
 begin
  if l<0 orif l>k-1 then raise WrongElem fi
 end Elem;
```

```
 unit Stos: class(l: integer);
  unit push: function(e:Elem, s:Stos):Stos;
  begin
    result:=new Stos((s.l-k)*k+e.l+k+1);
  end push;
  unit pop: function(s:Stos):Stos;
  begin
    if not empty(s) then
     result:= new Stos( ((s.l-k-1)div k)+k )
    else
     raise EmptyStack
    fi
  end pop;
  unit top: function(s:Stos):Elem;
  begin
    if not empty(s) then
     result:= new Elem( (s.l-k-1)mod k )
    else
     raise EmptyStack
    fi
  end top;
  unit empty: function(s:Stos):Boolean;
  begin
    result:=s.l=k
  end empty;
  unit equal: function(s1,s2: Stos):Boolean;
  begin
   result:= s1.l=s2.l
  end equal;
 begin
  if l<k then l := k fi
 end Stos
end Stosy10;
```

Klasa $Stosy10$ jest definicją struktury algebraicznej $\mathfrak{S}_{10}$. Na uniwersum tej struktury składają sie dwa zbiory $|Elem|$ oraz $|Stos|$ opisane przez klasy $Elem$ i $Stos$. Działania tej struktury są zdefiniowane przez metody (tj. funkcje) $push, pop, top, empty, equal$.

$$\mathfrak{S}_{10} \stackrel{df}{=} \langle \{|Elem| \cup |Stos|\}, push, pop, top, empty, equal \rangle$$

Zbiór $|Elem|$ zawiera dziesięć obiektów jakie można utworzyć obliczając wartość wyrażenia new Elem(i), i=0 1 ... 9. Zbiór $|Stos|$ jest nieskończonym zbiorem obiektów klasy $Stos$. Działanie $push$: $|Elem| \times |Stos| \rightarrow |Stos|$ zdefiniowaliśmy podając deklarację funkcji $push$. Można łatwo sprawdzić każdy z aksjomatów stosów. Czy dla każdych $e \in Elem$ i $s \in Stos$ jest prawdą, że

$$top(push(e, s) = e$$

Wynik operacji $push(e, s)$ jest obiektem typu $Stos$ dla którego wartość atrybutu l jest równa $(s.l - k) * k + e.l + k + 1$. Zastosujmy do tej liczby działanie top. A więc obliczmy wartość wyrażenia $((s.l - k) * k + e.l + k + 1 - k - 1)$**mod** k. Widać że jest to $e.l$. I to się zgadza.
I tak po kilku krokach sprawdzimy że każdy z aksjomatów stosów jest prawdziwy w implementacji opisanej przez klasę Stosy10. Wskazówka. Możesz się zastanawiać w jaki sposób sprawdzić że dla każdego stosu s program **while** not empty(s) **do** s:= pop(s) **od** zakończy obliczenia tj. nie zapętli sie? Przypomnij sobie odpowiedni aksjomat liczb całkowitych.

2.4. Implementacja specyfikacji S4. W tym miejscu podajemy dwie implementacje specyfikacji S4 por.1 zrealizowane jako klasa Stosy13 i klasa Stosy14. W obu klasach znajdujemy klasę Nat. klasa Nat zawarta w klasie Stosy13 różni się od klasy Nat zawartej w klasie Stosy14.
Poniżej znajdziesz specyfikację SNat.

A w tej tabeli są dwie różne implementacje specyfikacji SNat.

To jest implementacja I4.

```
unit Stosy13: class;
  const k=10;
  signal Emptystack, WrongElem, NegatvNat;
  unit Nat: class(l: integer);
    add: function(n: Nat): Nat;
    begin
      result:= new Nat(l+n.l)
    end add;
    unit zero: function : Nat;
    begin    result:= new Nat(0)
    end zero;
  begin
    if l < 0 then raise NegatvNat fi;
  end Nat;
  unit Elem: class(l: Nat);
  begin    if l.l<0 orif l.l>k-1
    then raise WrongElem fi
  end Elem;
  unit Stos: class(l: Nat);
    push: function(e:Elem, s:Stos): Stos;
    begin
      result:=new Stos(
       (s.l.l-k)*k+e.l.l+k+1);
    end push;
    unit pop: function(s:Stos):Stos;
    begin
      if not empty(s) then
       result:= new Stos(
      ((s.l.l-k-1)div k)+k )
      else
       raise EmptyStack
      fi
    end pop;
    unit top: function(s:Stos): Elem;
    begin
      if not empty(s) then
       result:= new Elem( (s.l.l-k-1) mod k )
      else
       raise EmptyStack
      fi
    end top;
    empty: function(s:Stos):Boolean;
    begin       result:=s.l.l=k
    end empty;
    equal: function(s1,s2: Stos): Boolean;
    begin
      result:= s1.l.l=s2.l.l
    end equal;
  begin
    if l.l<k then l.l := k fi
  end Stos
end Stosy13;
```

```
unit Stosy14: class;
  var k: Nat; signal Emptystack,
    WrongElem, NegativeNat;
  unit Nat: class(i,l,m: integer);
    unit add: function(n: Nat): Nat;
    begin result:= new Nat(
      i+n.i,l*n.m+n.l*m , m*n.m)
    end add;
    unit zero: function : Nat;
    begin result:= new Nat(0, 0, 1)
    end zero;
  begin
    if l < 0 then
      raise NegativeNat fi;
  end Nat;
  unit Elem: class(l: Nat);
  begin    if l.l<0 orif l.l>k-1 then
      raise WrongElem fi
  end Elem;
  unit Stos: class(l: Nat);
    push: function(e:Elem, s:Stos): Stos;
    begin
      result:=new Stos(
       (s.l.l-k)*k+e.l.l+k+1);
    end push;
    unit pop: function(s:Stos):Stos;
    begin
      if not empty(s) then
        result:= new Stos( ( (s.l.l-k-1)div k)+k )
      else  raise EmptyStack    fi
    end pop;
    unit top: function(s:Stos):Elem;
    begin
      if not empty(s) then
       result:= new Elem(
      (s.l.l-k-1)mod k )
      else  raise EmptyStack
      fi
    end top;
    unit empty: function(s:Stos): Boolean;
    begin
      result:=s.l.l=k
    end empty;
    equal: function(s1,s2: Stos): Boolean;
    begin
      result:= s1.l.l=s2.l.l
    end equal;
  begin
    if l.l<k then l.l := k fi
  end Stos
begin
  k:= new Nat(10,0,1);
end Stosy14;
```

Obie klasy Stosy13 i Stosy14 spełniają wszystkie wymagania wyliczone w specyfikacji S4. Czytelnik zechce to sprawdzić por.

ćwiczenie 16.3 oraz ćwiczenie 16.4.

Implementacja I4 rôzni się od poprzedniej inną definicją klasy Nat.
Zachowanie się tych klas jest istotnie rôżne. Czytelnik zechce przeprowadzič samodzielnie eksperymenty.

3. Abstrakcyjna klasa stosy z konkretną klasą MElem

W przykładzie (?) opisaliśmy abstrakcyjną klasę Stosy z abstrakcyjną klasą Elem. W zastosowaniach powinniśmy w jakiś sposôb ukonkretnič o jaką klasę naszych elementôw nam chodzi.
Znane są dwa podejścia
Typ formalny czy dziedziczenie ? W tym miejscu porôwnamy dwa sposoby wprowadzania typu element do struktury stosôw:
- element jako typ formalny – poprawnie, ale ...
- element jako typ abstrakcyjny, niekompletny, do dalszego sprecyzowania poprzez dziedziczenie, wada – dopuszcza rôzne rozszerzenia.
Ale można się przed tym zabezpieczyc zamykając dziedziczenie zob, typ człowiek w podręczniku Loglanu.

Przykład 16.1. Typ El jako parametr formalny.

```
unit Stosy: class(type El, function equal(e1, e2:El):Boolean);
   unit Stos: class; ... end Stos;
   push: function(e:El, s: Stos): Stos;
   end push;
   pop
   end pop;
   ...
end Stosy;
```

W takim podejściu zanim użyjesz generatora new Stosy lub bloku prefiksowanego
Stosy(...) block .. end;
musisz utworzyć typ ELA – parametr aktualny i funkcję eq – funkcję charakterystyczna relacji rôwności, a więc funkcja eq ma miec odpowiednie własności. Teraz
Stosy(ELA,eq) block ... end
powinno zadziałač poprawnie – algorytm wewnątrz tego bloku realizowany w środowisku Stosy(ELA,eq). Tzn. elementy maja być typu ELA i do porôwnywana elementôw wykorzystuje się funkcję eq.

A więc na użytkownika klasy Stosy nakłada się obowiązki ...
Inaczej mozna tak

Przykład 16.2. Typ El jako atrybut abstrakcyjnej klasy Stosy.

```
unit Stosy: class;
  unit El: class;
    unit virtual eq: function(e1, e2: El):Boolean; ... end eq
  end El;
  unit Stos: class; ... end Stos;
  push: function(e:El, s: Stos): Stos;
  end push;
  pop
  end pop;
  ...
end Stosy;
```

W takim podejściu trzeba rozszerzyc deklaracje

```
unit MojeStosy: Stosy class;
  MojeEl: El class;
    unit virtual eq: function(e1,e2: MojeEl): Boolean; ... end eq;
  MojeEl;
end MojeStosy;
```

Tu też musze pamiętac czego oczekuje się od funkcji eq.
Wada: jak zapanować nad tym co będzie wstawiane do stosu?

Drobna modyfikacja powyższego podejścia polega na zabronieniu rozszerzenia typu MojeEl w sposób dowolny.

Przykład 16.3. Typ zabezpieczony przed ...?

```
unit Stosy: class;
  unit El: class;
    unit virtual eq: function(e1, e2: El):Boolean; ... end eq
  end El;
  unit Stos: class; ... end Stos;
  push: function(e:El, s: Stos): Stos;
  end push;
  pop
  end pop;
  ...
end Stosy;
```

W takim podejściu trzeba rozszerzyć deklaracje

```
unit MojeStosy: Stosy class;
  MojeEl: El class;
    unit virtual eq: function(e1,e2: MojeEl): Boolean; ... end eq;
  begin
    if not (this El is MojeEl) then raise ExceptionEB fi;
  MojeEl;
end MojeStosy;
```

Teraz do stosu wchodza tylko obiekty typu MojeEl.

4. Przykład dowodu programu na stosię

Wymyśl coś!

Ćwiczenia

16.1. Uzupełnij dowôd twierdzenia ... Sprawdź pozostałe aksjomaty.

16.2. Załôżmy, że zbiôr E ma trzy elementy $E = \{e_1, e_2, e_3\}$. Opisz zbiôr $|Stos|$. Czy jest to drzewo? Skończone?

16.3. Sprawdź, że implementacja specyfikcji S4 opisana w deklaracji klasy Stosy14 jest poprawna, wszystkie wymagania są spełnione.

16.4. Program
utwôrz stos s
wstaw do tego stosu 3 elementy 9,7,4.
wykonaj następujący program
while not empty(s) do
e:= top(s)
write(e)
s:= pop(s)
od
Co się stanie gdy wykonasz ten program z klasą Stosy12 , a co gdy wykonasz go z klasą Stosy

16.5. Udowodnij, że schemat indukcji strukturalnej dla stosôw jest twierdzeniem algorytmicznej teorii stosôw.

ROZDZIAŁ 17

Kolejki FIFO

Struktura kolejek występuje w tak wielu zastosowaniach komputerów i w rozmaitych elementach systemów operacyjnych, baz danych, protokołach komunikacyjnych, etc, że musimy się jej przyjrzeć bliżej.
Struktura kolejek jest bliska strukturze stosów. Widzieliśmy wcześniej 1, że łatwo zaimplementować interfejs stosów realizując w istocie system kolejek. Dzieje się tak i w drugą stronę: po podaniu interfejsu I kolejek można podać klasę będącą modelem struktury stosów, która będzie implementować ten interfejs.
Czym jest więc struktura kolejek?

Definicja 17.1. Każda struktura algebraiczna spełniająca następujące warunki

(U) Uniwersum struktury jest sumą dwu rozłącznych zbiorów E i $Q, E \cap Q = \emptyset$,

(S) Sygnatura struktury zawiera operacje f, g, h i relacje $e, =_E, =_Q$ takie, że
$f\colon E \times Q \to Q$
$g\colon Q \to Q$
$h\colon Q \to E$
oraz
$em\colon Q \to \{true, false\}$
$=_E\colon E \times E \to \{true, false\}$
$=_Q\colon Q \times Q \to \{true, false\}$

(A) Aksjomaty. Struktura zapewnia prawdziwość następujących formuł
s1) $\forall_{e\in E}\forall_{s\in Q}\ \neg em(f(e,s))$
s2) $\forall_{e\in E}\forall_{s\in Q}\ em(s) \Rightarrow h(f(e,s)) =_E e$
s3) $\forall_{e\in E}\forall_{s\in Q}\ em(s) \Rightarrow g(f(e,s)) =_S s$
s4) $\forall_{e\in E}\forall_{s\in Q}\ \neg em(s) \Rightarrow h(s) =_E h(f(e,s))$
s5) $\forall_{e\in E}\forall_{s\in Q}\ \neg em(s) \Rightarrow f(e, g(s)) =_Q g(f(e,s))$
s6) $\forall_{s\in Q}\{$**while** $\neg em(s)$ **do** $s := g(s)$ **od**$\}\ em(s)$

jest strukturą kolejek (FIFO).

Zazwyczaj zbiór E nazywamy zbiorem elementów a zbiór Q zbiorem kolejek. Operacje f, g, h nazywane są zazwyczaj: $put, get, first$, po polsku: wstaw, usuń, pierwszy. Funkcja boolowska em sprawdza czy kolejka jest pusta. Można łatwo sprawdzić, że dwie

znane implementacje kolejek spełniają tę definicję.

Przykład 17.1. Rozważmy następującą klasę

```
unit KolejkiT: class;
  unit Element: class ;
  end Element;
  unit Kolejka: class;
    var T: arrayof Element, i,j: integer;
  begin
    array T dim (1:50);
  end Kolejka;
   signal EmptyError, FullError;
  unit put: function(e: Element, s: Kolejka): Kolejka;
  begin
    if j<>50 then s.T(s.j) := e; s.j := s.j+1; result:=this Kolejka
    else raise FullError fi;
  end put ;
  unit first: function(s: Kolejka): Element;
  begin
    if not em(s) then result := s.T(s.i);
    else raise EmptyError fi
  end first ;
  unit get: function(s: Kolejka): Kolejka;
  begin
       if not em(s) then s.i := s.i+1; result:=this Kolejka
       else raise EmptyError fi
  end get ;
  unit em: function(s: Kolejka): Boolean;
  begin
  result := s.i=s.j
  end em ;
end KolejkiT;
```

Bardziej "obiektowo" wygląda ta sama idea kolejki zapisana w ten sposób.

Przykład 17.2. Zamiast klasy KolejkiT można rozważać klasę KolejkiT2. W tym przypadku mówimy o metodach put, get, first i em klasy Kolejki. Zauważ, że obliczanie wartości wyrażeń obiektowych wymaga przekazywania mniejszej liczby parametrów.

```
unit KolejkiT2: class;
  unit Element: class ;
    unit virtual equal: function(e: Element): Boolean;  end equal
  end Element;
  unit Kolejka: class;
    var T: arrayof Element, i,j: integer;
    unit put: procedure(e: Element );
    begin
      if j<>50 then T(j) := e; j := j+1
      else raise FullError fi;
    end put ;
    unit first: function: Element;
    begin
      if not em(s) then result := T(i);
      else raise EmptyError fi
    end first ;
    unit get: procedure;
    begin
      if not em(s) then i := i+1;
      else raise EmptyError fi
    end get ;
    unit em: function : Boolean;
    begin
      result := (i=j)
    end em ;
  begin
    array T dim (1:50);
  end Kolejka;
  signal EmptyError, FullError;
end KolejkiT2;
```

W tym przypadku można zmienić ortografię wymagań. Np. warunek

$$\neg em(s) \Rightarrow first(s) =_E first(put(e, s))$$

zapiszemy tak

$$\neg s.em \Rightarrow s.first.equal(s.put(e).first)$$

Nietrudno zauważyć, że tak zmienione aksjomaty kolejek są spełnione przez klasę KolejkiT2.

Z taką klasą KolejkiT2 można pokusić się o jej zastosowanie.

Przykład 17.3. Ten blok wykorzystuje klasę KolejkiT2 (i równocześnie rozwija klasę Element) w celu ...

```
KolejkiT2 block
  Elem: Element class;
    unit virtual equal: function; ... end equal;
  end Elem;
  k1, k2: Kolejka, e:Elem;
begin
  k1:=new Kolejka; k2:= new Kolejka;
  e:=new Elem(...);
  k1.put(e);
  k2.get;
end
```

uzupełnić

Zauważmy, że z pary obiektów klasy Kolejki możemy stworzyć stos. (Podobnie z pary obiektów klasy Stos możemy zbudować kolejkę.) Te związki ukazują więź łączącą pojęcia stosu i kolejki.

Przykład 17.4. Kolejki jako listy

Te modele teorii kolejek LIFO są algorytmicznie nieodróżnialne tzn. dla dowolnej formuły algorytmicznej α, formuła ta jest spełniona w implementacji (tj. modelu) KolejkiT wtedy i tylko wtedy gdy jest spełniona w modelu KolejkiL. Sformułujemy twierdzenie wzmacniające tę obserwację. Niech E będzie dowolnym zbiorem wyposażonym w relację równości $=_E$ spełniająca aksjomaty równości: zwrotność, przechodniość i antysymetrię. Rozważmy zbiór $Fseq(E)$ skończonych ciągów elementów ze zbioru E. (Ciąg pusty $\emptyset \in Fseq(E)$.
Para $\langle E, Fseq(E)\rangle$ rozpatrywana z następującymi operacjami jest modelem teorii kolejek.

Model standardowy kolejek
Niech E będzie zbiorem. Rozpatrzmy zbiór $Fseq(E)$ skończonych ciągów elementów ze zbioru E. Operacje określamy w następujący sposób:

Niesprzeczność.

Twierdzenie 17.1. Algorytmiczna teoria kolejek FIFO ma model(e), jest więc niesprzeczna.

Tw. o reprezentacji
Aksjomatyzacja pierwszego rzędu ma modele patologiczne.

Ćwiczenia

ROZDZIAŁ 18

Zbiory skończone czyli kontenery

W wielu programach operacjami dominującymi sa operacje na zbiorach skończonych. Komputer w naturalny sposób operuje na liczbach. Jednak większość oprogramowania wymaga przechowywania i odszukiwania informacji.
Jenostka informacji może zawierac nie tylko liczby, ale także napisy, wartości Boolowskie, znaki, obrazy, dźwięki i inne.
Informacja taka jest oczywiście zawarta w pewnym zbiorze skończonym. W zależności od zadania jakie program ma rozwiązywać zbiór skończony będzie implementowany w sposób zapewniający efektywność ale i bezpieczeństwo.
W latach 70 0 80 ubiegłego wieku programiści używali słowa dictionary lub po polsku słownik, zob. [AHU74]. Ostatnio przyjęło się mówić o kontenerach (ang. container) jako pojemnikach mieszczących skończone zbiory obiektów. Zastanówmy się jak ma wyglądać specyfikacja pojęcia kontener. Kontenery czyli pojemniki to pojęcie abstrakcyjne. W rzeczywistości mamy wiele różnych struktur algebraicznych, które realizują pojęcie pojemników.

1. Specyfikacja kontenerów

Po doświadczeniach ze strukturą stosów spróbujemy podać definicję aksjomatyczną struktury kontenerów.

Definicja 18.1. *Kontenerem* nazywamy strukturę algebraiczną $\mathcal{K}$, której uniwersum składa się z dwu rozłącznych zbiorów E i S, $E \cap S = \emptyset$. W strukturze tej mamy następujące operacje i, d, a oraz predykaty r, p, c, q.

$$\mathcal{K} = \langle E \cup S; i, d, a, r, p, c, q \rangle$$

Sygnatura tej struktury jest wyliczona poniżej

i: $E \times S \to S$
d: $E \times S \to S$
a: $S \to E$

Relacje

r: $E \times E \to B_0$
p: $S \to B_0$
c: $E \times S \to B_0$
q: $S \times S \to B_0$

Aksjomaty struktury kontenerów są następujące

k1) $\forall_{e\in E}\,\forall_{s\in S}\,(c(e,i(e,s))\wedge\forall_{e'\in E\wedge e'\neq e}(c(e',s)\Leftrightarrow c(e',i(e,s)))$

k2) $\forall_{e\in E}\,\forall_{s\in S}\,(\neg c(e,d(e,s))\wedge\forall_{e'\in E\wedge e'\neq e}(c(e',s)\Leftrightarrow c(e',d(e,s)))$

k3) $\forall_{s\in S}(p(s)\Leftrightarrow\forall_{e\in E}\,\neg c(e,s))$

k4) $\big(\neg p(s)\Rightarrow c(a(s),s)\big)$

k5) $\forall_{e\in E}\,\forall_{s\in S}\;\; c(e,s)\Leftrightarrow$
$\left.\begin{array}{l}\textbf{block}\\ \quad\textbf{var}\ bool:Boolean, s1:S\\ \textbf{begin}\\ \quad s1:=s;\ bool:=false;\\ \quad\textbf{while}\ \neg bool\wedge\neg p(s1)\ \textbf{do}\\ \qquad e1:=a(s1);\\ \qquad bool:=r(e1,e);\\ \qquad s1:=d(e1,s1)\\ \quad\textbf{od}\\ \textbf{end}\end{array}\right\} bool$

k6) $\forall_{s\in S}\,\{\textbf{while}\ \neg p(s)\ \textbf{do}\ s:=d(a(s),s)\ \textbf{od}\}\ p(s)$

k7) $q(s,s')\Leftrightarrow\forall_{e\in E}(c(e,s)\Leftrightarrow c(e,s'))$

k8) $\forall_{e\in E}\ r(e,e)$

k9) $\forall_{e,e'\in E}\ r(e,e')\Leftrightarrow r(e',e)$

k10) $\forall_{e,e',e''\in E}\ (r(e,e')\wedge r(e',e''))\Rightarrow r(e,e'')$

Tak jak zwykle nasuwają się dwa pytania

(1) Czy istnieją jakieś kontenery? Czyli, czy zbiór formuł k1) - k10) jest niesprzeczny?

(2) Czy definicja kontenera wyklucza niepożądane struktury? Czy modelami zbioru aksjomatów są tylko takie struktury jakie akceptuje nasza intuicja?

Zacznijmy od następującego twierdzenia

Twierdzenie 18.1. Zbiór aksjomatów k1) – k10) jest niesprzeczny.

Dowód. Dowód polega na pokazaniu modelu dla tego zbioru. Niech E będzie dowolnym zbiorem, niech $=_E$ oznacza relację identyczności w zbiorze E.

Rozpatrzmy zbiór $Fin(E)$ wszystkich skończonych podzbiorów zbioru E. Przyjmiemy następującą interpretację symboli sygnatury kontenera:

$i(e,s)$	$d(e,s)$	$a(s)$	$p(s)$	$c(e,s)$	$q(s,s')$
$s\cup\{e\}$	$s\setminus\{e\}$	$a(s)\in s$	$s=\emptyset$	$e\in s$	$s=_S s'$

Łatwo widać, że wszystkie aksjomaty kontenerów są w tej interpretacji prawdziwe. Wyjaśnienia wymaga jedynie interpretacja operatora a, funkcja ta nazywana bywa selektorem. Istnienie funkcji $a\colon (Fin(E)\setminus\emptyset)\to E$ wynika z następującego twierdzenia teorii zbiorów, które tradycyjnie nazywane jest Aksjomatem

wyboru dla zbiorów skończonych
Tw. ACF Dla każdego zbioru X istnieje funkcja f przyporządkowująca każdemu niepustemu, skończonemu podzbiorowi zbioru X, jeden element $x \in X$. por. [Mos45]
Dowód twierdzenia ACF jest efektywny, nie wymaga pewnika wyboru.
Podsumowując, stwierdzamy, że zbiór formuł k1) - k10) jest niesprzeczny. □

Struktura algebraiczna opisana powyżej nazywana będzie modelem standardowym.

Wspomnieliśmy wcześniej, że znanych jest wiele różnych modeli teorii kontenerów.
Tablice
Jeśli wiadomo, że zbiór E jest pewnej niedużej mocy, powiedzmy $card(E) < 1000$ to każdy podzbiór zbioru E można zapisać w tablicy Boolowskiej o 1000 elementach. Operacje wstawiania, usuwania i wybierania elementu będą szybkie, ich koszt nie przekroczy $c * 1000$, gdzie c jest pewną stałą. Podobnie będzie ze sprawdzaniem czy element e należy do zbioru s i czy zbiór s jest pusty. Ten sposób przedstawiania zbiorów występuje w języku Pascal.

Kłopot występuje gdy liczność zbioru E jest dużo większa lub gdy równocześnie chcemy przechowywac wiele podzbiorów zbioru E. Wtedy koszt operacji staje się zbyt wielki.
Jeżeli w działaniach na podzbiorach zbioru E nie występuje operacja $mb(e, s)$ sprawdzania przynależności elementu e do zbioru s i ponadto wiadomo, że operacja usuwania elementu $del(e, s)$ odnosić się będzie zawsze do tego elementu e, który do zbioru s został wstawiony najdawniej, to najlepiej przedstawić kontener w postaci kolejki ??.
Stosy warto przyjąć jako implementację kontenera gdy operacja $mb(e, s)$ ogranicza się do sprawdzenia czy element e jest równy wierchołkowi stosu i gdy wiadomo, że operacja usuwania elementu zawsze odnosić się będzie do elemntu ostatnio wpisanego do stosu.

Hash tables

Liczba różnych modeli pojęcia kontenera jest nieograniczona. Skąd wiadomo, że nie istnieje model zbioru formuł k1 – k10) patologiczny, nieodpowiadający naszej intuicji? Na to pytanie odpowiedzi udziela następujące

Twierdzenie 18.2. Każda struktura algebraiczna $\mathcal{K} = \langle E \cup S; i, d, a, r, p, c, q \rangle$ będąca modelem kontenera jest izomorficzna z modelem standardowym nad zbiorem E.

Dowód. Dla dowodu należy pokazać, że istnieje taka funkcja $f\colon S \to Fin(E)$, która jest różnowartościowa i na, i taka, że spełnione są następujące równości
$f(i(e,s)) = f(s) \cup \{e\}$
$f(d(e,s)) = f(s) \setminus \{e\}$
$a(f(s)) = a(s)$
$mb(e,s) \Leftrightarrow e \in f(s)$
...
Definiujemy funkcję f w ten sposób
$f(s) \stackrel{df}{=} \{e \in E : mb(e,s)\}$
Funkcja f jest różnowartościowa ponieważ ...
Funkcja f odwzorowuje zbiór S na zbiór $Fin(E)$ ponieważ ...
Funkcję a: definiujemy ...
Łatwo sprawdzic, że zachodzą równości ... □

Istnieje wiele struktur spełniających tę definicję. Programistów interesują struktury, które zaimplementowano jako klasy. W niemal każdym podręczniku Algorytmów i struktur danych znajdziesz mnóstwo przykładów struktury kontener.

Programy wykorzystujące strukturę kontenera będą działać szybciej jeśli koszt operacji wymienionych w powyższej definicji jest mały. Powstało wiele implementacji struktury kontener. Na ogół, implementacje te wykorzystują strukturę drzewa i wtedy koszt pojedynczej operacji jest proporcjonalny do logarytmu liczby elementów zawartych w kontenerze.
naszkicuj klasę kontener

2. Definicja instrucji forall

Programiści odczuwają potrzebę instrukcji podobnej do polecenia for, ale pozwalającej na powtórzenie pewnych instrukcji dla każdego elementu z pewnego skończonego zbioru S. Przypomnijmy, instrukcja for obejmuje pewną sekwencję S instrukcji, wskazuje pewną zmienną liczbowa i jako zmienną kontrolującą powtarzanie ciągu instrukcji S i co najważniejsze zapewnia, że nie będzie nieograniczonej liczby powtórzeń. Czyli programista wykorzystując taką instrukcję, w sposób rozumnu, nie musi przeprowadzać dowodu własności stop!
W tym rozdziale przedstawimy pewien sposób realizacji polecenia foreach. Klasa Container może zawierać taką klasę.

```
unit foreach: class;
        var e: Elem;
begin
        if ¬ empty() then
            e := first();
            while ¬ at_last() do
                INNER;
                e := next()
            od
        fi
end foreach;
```

Zastosowanie
Poniższa instrukcja bloku prefiksowanego powtórzy <moje instrukcje> dla każdego elementu e znajdującego się w bieżącej instancji containera. Jest wskazane by intrukcje te posługiwały się zmienną nielokalną e.
Jest również wskazane by instrukcje te nie zmieniały tej zmiennej - dokładniej by nie wykorzystywały poleceń first() ani next(). Co z instrukcjami ins oraz del? Mogą one zmienić strukturę kontenera i spowodować pominięcie lub powtórzenie iterowanej instrukcji.

```
pref foreach block
begin
<moje instrukcje>
end
```

Ta instrukcja jest równoważna następującej instrukcji.

```
block
   var e: Elem;
begin
   if not empty() then
      e := first();
   while not at_last() do
      <moje instrukcje>;
      e := next()
   od
   fi
end (* forall *) ;
```

Jeśli chcemy udowodnić, że instrukcja <moje instrukcje> zostanie powtórzona dla każdego elementu znajdujacego się w kontenerze - w chwili rozpoczęcia instrukcji foreach
to musimy wykazać, że spełnione są pewne dodatkowe warunki: jakie?

Warto zbadać inne warianty:
- unit forall: class(inout a:Elem, c: Container); ...

lub
- unit forall: class(a: Elem); ...

Ćwiczenia

18.1. Napisz klasę implementująca strukturę kontenerów w tablicach i oszacuj koszt każdej operacji.

18.2. Napisz klasę implementującą strukturę kontenerów w kolejkach i oszacuj koszt każdej operacji.

18.3. Napisz klasę implementująca strukturę kontenerów w stosach i oszacuj koszt każdej operacji.

ROZDZIAŁ 19

Zarządzanie stertą obiektów

Ten rozdział jest poświęcony analizie struktury danych, w której należy zarządzać skończonym zbiorem obiektów pewnego typu.

Obiekty tej struktury mogą posiadać wielokrotne odnośniki, tzn. jeden obiekt może być wartością wielu zmiennych. Dzieje się tak gdy wykonane zostaną polecenia

x:= new T(...); . . . ; y:= x;

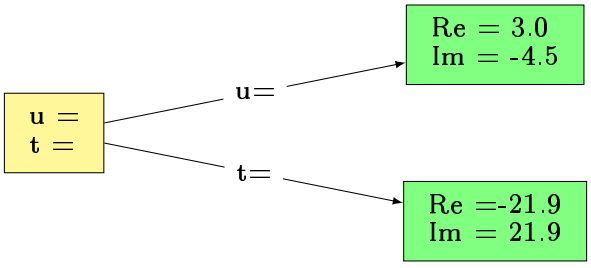
Struktura taka to system zarządzania obiektami klas tzw. heap (nazwa ta jest myląca ponieważ w literaturze z tą samą nazwą można spotkać się gdy mowa o strukturze kopców, zob. rozdział 22).

W większości języków programowania obiektowego wykonywanie programu może doprowadzić do groźnego i bardzo trudnego do wykrycia błędu.
Błąd taki powstaje gdy jeden obiekt jest wartością dwu, lub więcej, zmiennych. W sytuacjach podobnych do tej z rysunku 2 wykonanie instrukcji delete(u) prowadzi do sytuacji pobnej do tej z rysunku 3.

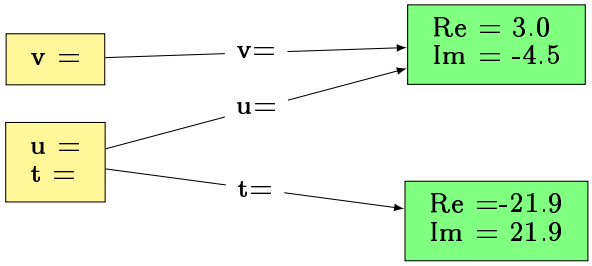
Co się dalej może zdarzyć?
Błędna interpretacja danych, jeśli na miejsce usuniętego obiektu pojawi się inny obiekt z.

Inny błąd – "opóźnionej destrukcji" występuje, gdy zostanie wykonane polecenie delete(v).

Jak powstają obiekty śmieci?

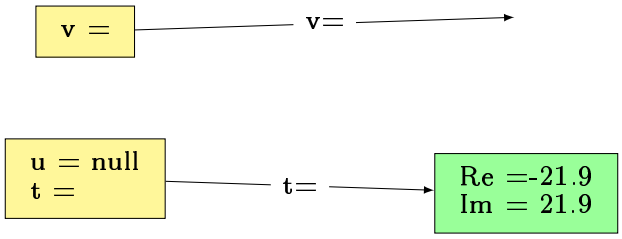


Rysunek 1. Zmienne t i u wskazują na dwa obiekty.



Rysunek 2. Zmienne u i v wskazują na ten sam obiekt. Wartości wyrażeń u.Im oraz v.Im są równe. Wykonanie polecenia v.Re ← 50 spowoduje, że od tej pory wartość wyrażenia u.Re będzie równa 50. W takim przypadku mówi się o aliasingu zmiennych u i v.

Po wykonaniu polecenia delete(u).



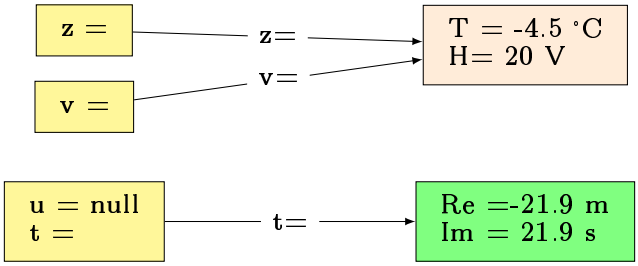
Rysunek 3. Od tej pory obiekt wskazywany przez zmienną u nie istnieje (u=null). A zmiennej v nie odpowiada żaden obiekt! Próba wyznaczenia wartości u.Re lub u.Im spowoduje zgłoszenie błędu. Wartości wyrażeń v.Re i v.Im są fałszywe. Obliczenie jest kontynuowane, bez ostrzeżenia o błędzie!!
Na tym polega błąd DANGLING REFERENCE. Groźne jest to, że program nie sygnalizuje błędu. Programista może stracić miesiące na zrozumienie co zaszło i wykrycie takiego błędu. Koszty?

Po wykonaniu polecenia u←null ————————————————————-
Po wykonaniu instrukcji v←null

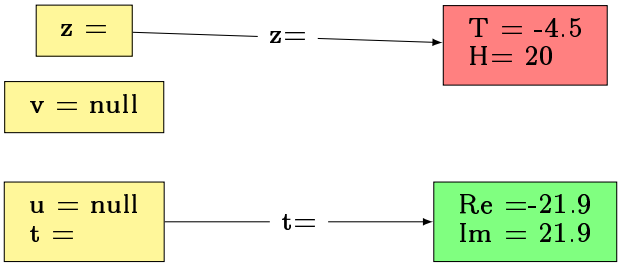
Inną ważną cechą systemu zarządzania obiektami jest konieczność usuwania obiektów niepotrzebnych (niedostępnych) i ponownego wykorzystania zwolnionego miejsca w pamięci. Obiekt staje się niedostępny gdy nie jest wartością żadnej zmiennej programu. Ilustruje to następujący przykład

x:=new T(1,,2); . . . ; x:= new T(3,4);

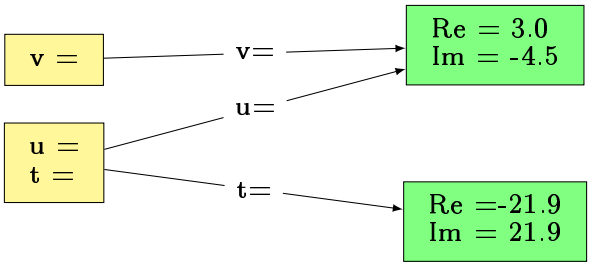
obrazek Opiszemy taki system podając jego specyfikację tzn.



Rysunek 4. Widać to wyraźnie z rysunku: wartośc v.Re jest nie tylko fałszywa numerycznie, błędem jest te¿ traktowanie jej jako zmierzonej w jednostkach m.



Rysunek 5. W efekcie usunięto poprawny i potrzebny obiekt z, zamiast wcześniej usuniętego obiektu u.

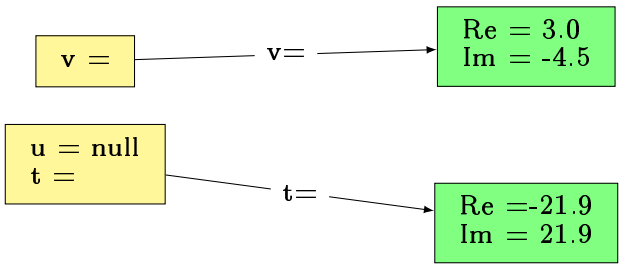


Rysunek 6. bb

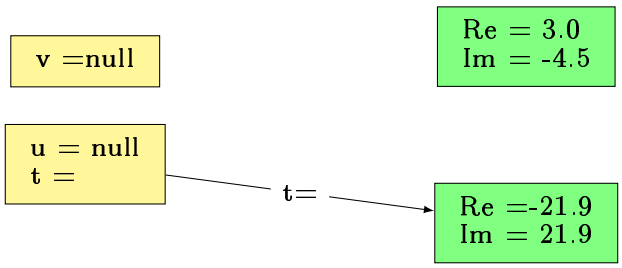
aksjomatyczny opis oraz implementację wymyśloną przez Antoniego Kreczmara.

Zwracamy uwagę na możliwośc zastosowania takiego systemu w wielu programach, nie tylko w maszynie wirtualnej (ang.running systemie). Tu podać nieformalny opis ...

Napisz, powoli rozwijając specyfikację



Rysunek 7. ccc



Rysunek 8. Na tym obrazku pojawił się śmieć! Wskaż go!.

(1) aksjomaty Hani Oktaby i ich model, nie zapewniają wszystkich pożądanych cech systemu zarzadzania Heapem HM
(2) wzmianka o kolejnym podejściu Oktaby
(3) specyfikacja z instrukcjami kill oraz gc układ aksjomatów AS+AZ
(4) model nowej specyfikacji opisany w Loglanie
dwie tablice tablica M arrayof protoObject i
tablica H array of reference
unit reference: class(serialNo: integer, wskaznik: protoObject) end reference;
unit protoObject: class(serialNo: integer) end protoObject;
deklaracje: new, member, kill, gc, ...

VLP – maszyna wirtualna Loglanu zarządza obiektami klas i tablic, jakie powstają podczas wykonywania programu. System $\mathcal{AK}$ por.[CK84] obiektów rôżni się od kontenerów omawianych wcześniej.

1. Specyfikacja

Poprawić

Poniższa specyfikacja nie mówi nic o wartościowaniach zmiennych w obiektach. Napisać nową specyfikację.

Koniec uwagi

Najpierw w r. 1979 Antoni Kreczmar obmyślił i zrealizował system zarządzania pamięcią przeznaczoną na przechowywanie obiektów zob.[CK84]. Specyfikacja czyli aksjomatyczny opis systemu zarzadzania zbiorem obiektów powstała później. W r. 1982 Hanna Oktaba przedstawiła kolejno algorytmiczną teorię referencji i algorytmiczną teorię zarządzania obiektami[MS87] str.328-344.

Definicja 19.1. System obiektów Loglanu jest to struktura algebraiczna

$$\mathcal{AK} = \langle Fr \cup S \cup \{none\}; r, a, i, d, m, k, e_S \rangle$$

taka, że jej uniwersum jest unią rozłącznych zbiorów Fr i S oraz {none}.
Struktura $\mathcal{AK}$ ma następujące operacje.

Sygnatura	Komentarz
$r\colon S \to Fr$	rezerwuje nową, wolną ramkę $r(s)$
$a\colon S \to Fr$	w stanie s wybierz ramkę
$i\colon Fr \times S \to S$	dołącz ramkę do stanu
$d\colon Fr \times S \to S$	usuń ramkę
$m\colon Fr \times S \to \{true, false\}$	czy należy ?
$k\colon Fr \times S \to S$	zabij
$e \in S$	stan początkowy (pusty)

W alfabecie algorytmicznej teorii ATHM oprócz wymienionych powyżej funktorów i predykatów pojawiaja się zmienne typu Fr. Zazwyczaj będziemy je oznaczać $f, f', ...$ i zmienne typu S - dla stanów, oznaczane przez $s, s', ...$

Stała none $\notin \{Fr \cup S\}$. Wartością zmiennej f typu Fr jest element zbioru Fr lub stała none.

Aksjomaty specyficzne teorii $\mathcal{ATHM}$ podane są poniżej

HM$_1$) $\forall_{s \in S}\ \neg m(r(s), s)$
Dla każdego stanu $s \in S$, operacja $r(s)$ zwraca nowa ramkę nie należąca do zbioru s

HM$_2$) $\forall_{f \in Fr}\ \neg m(f, e)$
stan początkowy e jest pustym zbiorem ramek

HM$_3$) $\forall_{s \in S} \left\{ \begin{array}{l} \text{while } s \neq e \text{ do} \\ \quad s := d(a(s), s) \\ \text{od} \end{array} \right\} (s = e)$
Dla każdego stanu s, powyższy program while kończy obliczenie, a więc
każdy stan jest skończonym zbiorem ramek.

HM$_4$) $\forall_{s\in S}\ s \neq e \Rightarrow m(a(s), s)$
Dla każdego niepustego stanu s, funkcja amb zwraca jakąś ramkę należącą do tego stanu s.

HM$_5$) $\forall_{f\in Fr}\forall_{s\in S}\{s' := i(f,s)\}(m(f,s')\wedge\forall_{f'\in Fr}(f' \neq f \Rightarrow m(f',s) \Leftrightarrow m(f',s'))$
operacja ins dodaje ramkę f do stanu s

HM$_6$) $\forall_{f\in Fr}\forall_{s\in S}\{s' := d(f,s)\}(\neg m(f,s') \wedge \forall_{f'\in F}(f' \neq f \Rightarrow m(f',s) \Leftrightarrow m(f',s'))$
operacja del usuwa ramkę f fze stanu s.

HM$_7$) $m(f,s) \Leftrightarrow$
```
begin
  s1 := s; bool := false;
  while s1 ≠ e ∧ ¬bool
  do
    f1 := a(s1);
    if f = f1 then bool := true fi;
    s1 := d(f1, s1);
  od
end
```
$\Big\}\ bool$

Powyższa formuła definiuje relację member w terminach operacji a, d i e. To nie jest implementacja tej relacji.
A. Kreczmar podał sposób obliczania odpowiedzi o koszcie stałym.

HM$_8$) Operacja kill jest scharakteryzowana przez aksjomaty o następującym schemacie.
Wskaźnik p może być dowolną liczbą naturalną większą od zera $p > 0$. Niech $1 \leq i \leq p$.

$$\underbrace{((f_1 = ... = f_p) \wedge m(f_1, s))}_{precondition} \Rightarrow \underbrace{[s' := k(f_i, s)]}_{\text{statement}} \underbrace{(f_1 = ... = f_p = \textbf{none})}_{postcondition}$$

Każda formuła tej postaci jest aksjomatem. Stwierdza ona, że operacja $kill$ w jednym ruchu anuluje wszystkie odnośniki do obiektu wskazywanego przez tę zmienną f_i.
I rzeczywiście w systemie Kreczmara koszt operacji $kill$ jest stały.

HM$_9$) $\forall_{s\in S}\,\forall_{f\in Fr}\, m(f,s) \Rightarrow f \neq none$

HM$_{10}$) $\forall_{s,s'\in S}\, s =_S s' \Leftrightarrow \forall_{f\in F}(m(f,s) \Leftrightarrow m(f,s'))$

1.1. Własności specyfikacji HM. One can investigate the properties of the specification itself. We are able to state an important metatheorem about the system of axioms in HM. The following theorem was not formulated in [CK84]. H. Oktaba proved a theorem on consistency for a similar set of axioms [Okt82], basically it was the set {HM$_1$ - HM$_7$ }.

Twierdzenie 19.1. (on consistency of the set {HM1-8})
The system of axioms HM$_1$ – HM$_8$ has a model.

For a sketch of the proof see the Appendix A. The model constructed in the proof will be called the standard model.

H. Oktaba proved another important fact:

Twierdzenie 19.2. (representation theorem)
Every two models of the axioms HM_1 – HM_7 are isomorphic, up to implementation of operations *amb* and *res*, to the standard model.

The meaning of the theorem is: the operations ins, mb, are precisely described in the set of axioms, when the description of properties of operations res and amb was left understated. This was done intentionally, for there are various possibilities how to treat the released memory and how to choose where(address) to allocate a new object.. Everybody agrees that the operations *res* and *amb* may be implemented in several versions. Many different implementation of *res* (respectively *amb*) operation allows to prove that the requirements mentioned in the axioms HM_1 and HM_4 are satisfied. Therefore, it is not easy to prove the representation theorem for the axioms HM_1 – HM_{10}. One must decide on his choice of the *res* operation – how it will be implemented in one system, while other systems may prefer different solution. podaj przykłady

1.2. Odmiany układu aksjomatów. Czy uproszczenia jakie przyjęliśmy formułując powyższe aksjomaty są istotne? Zauważmy:

- Można rozważyć nieco zmienioną operację res rezerwuj. Operacja res ma teraz parametr *appetite* definiujący rozmiar potrzebny dla utworzenia obiektu. Modyfikujemy w odpowiedni sposób sygnaturę $res : S \times N \to Fr$ prowadzi to do nowego (niesprzecznego) układu aksjomatów.
- Kolejne rozszerzenie naszego systemu HM może być zdefiniowane gdy opisujemy wewnętrzna strukturę obiektu. (Ta struktura jest wyznaczona przez deklarację odpowiedniej klasy). Takie rozszerzenie teorii HM jest także teorią niesprzeczną.

Do tej pory nie było potrzeby wprowadzania operacji odśmiecania. W modelu naszej abstrakcyjnej teorii zbiór ramek Fr jest izomorficzny ze zbiorem liczb naturalnych. Bardziej realistyczna wersja teorii powinna wprowadzić postulat, zbiór Fr jest skończony. W tym przypadku powstaje potrzeba odśmiecania (ang. garbage collection).

W jaki sposób wyrazić własność zbiór Fr ramek jest skończony? Nietrudno odgadnąć, że poniższa formuła jest dobrym kandydatem:
HM_{11}) $\quad \exists_{s_0 \in St} \forall_{f \in Fr} mb(f, s_0)$

Co czytamy tak: zbiór ramek wypełnia pewien stan, a więc Fr jest zbiorem skończonym.
Taki zbiór aksjomatów $HM_1 - HM_{11}$ jest sprzeczny.

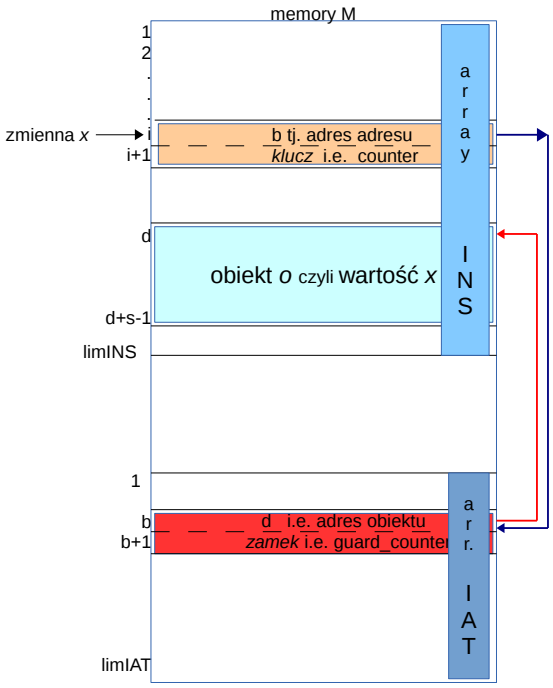
19.1. (1) Wykaż to.
(2) Napraw to, tzn. podaj niesprzeczny układ aksjomatów.
Wskazówka. Wprowadź predykat $full$ tj. pełny.

Na system składają się zbiór Fr (ramek), zbiór St (stanów), wyróżniony element none oraz działania:in, del, kill i relacje mb, em,

2. Model AK

W tym podrozdziale przedstawimy szkic systemu zarządzania pamięcią obiektów klas.

Pamięć jest zorganizowana w następujący sposób Przyjęto kilka



Rysunek 9. Schemat organizacji pamięci obiektów

założeń ... Zmienne typów obiektowych nie wskazują bezpośrednio na obiekt. Wartością takiej zmiennej jest adres pośredni czyli para wielkości $\langle adreswtabiltyIATadresw, klucz\rangle$. Każdy dostęp do obiektu wskazywanego przez zmienną x rozpoczyna się od sprawdzenia czy obiekt istnieje.
Jeżeli obiekt został już usunięty bądź zmiennej x nie przypisano jeszcze żadnego obiektu to zostanie podniesiony alarm "reference to none".
W przeciwnym przypadku akcja member zwróci bezpośredni adres obiektu i program będzie kontynuowany: operacje odczytu pola, zapisu nowej wartości pola lub zdalnego wywołania metody obiektu zostaną wykonane.

```
unit AK: class;
  unit protoObject: class(size:integer);
  end protoobject;
  unit indirAdres: class(adresAd: adres, klucz:integer);
  end indirAdres;
  var Ins: arrayof protoobject, IAT: arrayof indirAdres;
  unit new: function(apetite:integer): indirAdres;
      var c : boolean, d : address;
  begin
      c := false;
      if Head = 0 {FIFO is empty}
      then
       if limIAT - LastItem < 3 {no space for IAT entry}
       then
         c := true; call compactor
      fi;
      if c then raise ErrBrakujePamieci {end of computation} fi;
      LastItem := LastItem + 2; b := LastItem;
      IAT[b].guard_counter := 0; {initialize new IAT entry}
      else {take from FIFO}
        b := Head; Head := IAT[b].d
        fi;
        if limINS - LastUsed < s + 1 {Free Space too small}
        then
          if search(s, d) {frame found}
          then
         IAT[b].d := d; counter := IAT[b]. guard_counter; return
    fi;
    if c then {end of computation} else call compactor fi;
    if LastItem - LastUsed < s + 1 then {end of computation} fi;
      fi;
      d := LastUsed + 1; INS[d].size := s; LastUsed := LastUsed + s;
      result:= new indirAdres(d, IAT[b].guard_counter)
  end new;
  unit mb: function(b,key: integer):integer
      (*funkcja mb zwraca adres obiektu lub zgłasza Error-ref2none *)
  begin
      if key=IAT[b].lock
      then result:=IAT[b].d
      else raise ref2none
      fi
  end mb;
  unit kill: procedure(b:address, key:integer);
      var d: address;
  begin
      if key ≠ IAT[b].lock then return fi ;
      IAT[b].lock := IAT[b].lock+1;
      d := IAT[b]d; (* put this IAT[b] into FIFO *)
      IAT[Tail].d := b; IAT[b].d:=0; Tail:=b;
      if d+INS[d].size=LastUsed+1 then LastUsed:=LastUsed+INS[d].size
      else call insert2Freed(INS[d].size, d) fi
  end kill;
begin
  array IAT dim (1:limIAT);
  array INS dim (1:limINS);
end AK;
```

Twierdzenie 19.3. System AK jest modelem algorytmicznej teorii ATHM (algorithmic theory of heap management)

Dowôd. Dowôd jest modyfikacją dowodu z pracy [CK84]. □

3. Porównania i zastosowania

System AK został obmyślony i zrealizowany dla języka Loglan w r. 1979. Jest on częścią running systemu (maszyny wirtualnej) języka Loglan.

3.1. Ilustracja zastosowania. Zastosowanie systemu AK w programie może wyglądać np. tak

```
pref AK block
   unit T: protoObject class(attr:real); ... end T;
   var x,y,z: T;
begin
   ...
   x:= new() ;
   ...
   x.attr:=7; (* musimy zacząć od sprawdzenia mb() czy x≠none? *)
   y:=x; z:=y;
   ...
   kill(y);
   if x=none then write("x=none") fi;
   if z=none then write("z=none") fi;
end block
```

Ten przykład celowo nie jest dopracowany do końca. Spróbuj sama napisać taki przykład w ulubionym języku programowania.

3.2. Porównanie. W tablicy 1 porównujemy sposoby usuwania obiektów jakie są używane w różnych językach programowania. Wyróżniamy trzy grupy: pierwsza grupa składa się z języka Loglan. Języki drugiej grupy zezwalają na programowaną dealokację obiektów (C++, Pascal, etc.). Trzecia grupa zawiera języki w których instrukcja dealokacji obiektu jest zabroniona. W językach tej grupy program musi polegać na sprawnym odśmiecaniu nieużytecznych obiektów.
Przy porównaniach wzięto pod uwagę następujące aspekty (pięć): Pre-tj. warunek wstępny - we wszystkich przypadkach jest on jednakowy, Kod - instrukcje prowadzące do dealokacji obiektu, Post-warunki końcowe - zauważ różnice, Koszt - czas potrzebny dla dokonania dealokacji obiektu, Ryzyko - nieudanej dealokacji lub wystąpienia błędu.

Some explanations concerning cost of deallocation operation seem necessary: The cost of delete() in C++, dispose() in Pascal, and free() in Ada are known [Str13, JW74, Bar96]. The cost of

kill() is calculated in [CK84], and it will be explained below, see Appendix A. The cost of any garbage collector gc() is known as $O(m)$, where m is the size of heap i.e. object memory. Note, any garbage collecting algorithm must visit each object in the heap.

Czy warto instalować system AK w ...? Oceńmy możliwe profity z zainstalowania systemu Kreczmara w jakimś popularnym języku programowania.

Javie? Java jest dumna z przyjęcia zakazu używania instrukcji dealokacji, por. [GM95]. W wielkim skrôcie Java môwi: nie ma instrukcji dealokacji – nie ma ryzyzka wiszących referencji, a polecenie odśmiecacza gc() zajmie się problemem wycieku pamięci. To jest prawda, ale jaki z tym wiąże się koszt?
Rozważ następujący fragment kodu

```
class Complex { }
/* Deklaracja klasy complex z metodami: add i mult */
Complex x,t,z;
/* zadeklarowano trzy zmienne typu complex */
 z=new Complex(3.0, -4.5);
 t=new complex(-21.9, 21.9);
 x=t.add(z.mult(new complex(3.2, 4.3)));
 z=x.mult(z.add(x));
/* utworzono siedem obiektôw, cztery z nich to śmieci */
```

Nic nadzwyczajnego. The garbage collector $gc()$ will dispose them. Now, imagine that the program creates a lot of garbage. This happens in scientific or engineering projects. Imagine the four assignments seen above, are inserted inside a for- statement and executed say 100000 times. How many garbage objects are created?
Using kill() instruction would allow to save a lot of work.
Moreover, an object programming language that prohibits the instruction $delete()$, becomes prone to memory leakage errors.

C++. We are convinced that it is worthwhile to equip C++ with a system similar to this of Kreczmar.

- For it eliminates the risk of dangling references. Each attempt to acces a killed object will raise an exception reference to none and a proper diagnostic of the error.
- It also protects against the errors of destruction and contradiction.
- Together with the operations kill and member, the operation gc() - garbage collector – may be offered to programmers use.

Niektôrzy programiści argumentują, że proces testowania (debugowania) programu pozwala na identyfikację zagrożeń i ich

eliminację. A to nie jest prawdą.

System AK Kreczmara gwarantuje że ewentualne wiszące referencje zostaną wykryte i ostrzeżenie zostanie wysłane do użytkownika programu.	Testowanie może zmniejszyć ryzyko błędu, jednak nie eliminuje zagrożenia jakim są niewykryte błędy wiszących referencji.

Pozostawiamy czytelnikowi zadanie wyrobienia sobie własnego poglądu na sprawę dealokacji obiektów.
For those who object: "the cost of new system of heap management is too high", we propose to use a compile-time switch that turns off the checks at indicated lines of code. Attention! the programmer may switch off checks at his/her responsibility. In other words: when you turn off the check against dangling reference error in lines, say 260-280 of your code, you should be able to prove that the error will not appear.

Ćwiczenia

19.2. Czy obiekt niepotrzebny i obiekt niedostępny to to samo pojęcie? Podaj przykłady.

19.3. Porównaj specyfikację struktury kontenerów i sterty obiektów.

19.4. Co się stanie gdy w specyfikacji pominiemy aksjomat $\mathcal{ATHM}8$)?

Tablica 1. Trzy modele dealokacji obiektów.

	Model D (Loglan'82)	Model A (np. C++, Pascal)	Model B (np. Java, Python)
Pre-	Obiekt o jest wartością zmiennych $x_1 = x_2 = \ ... \ = x_n, \quad 1 \leq i \leq n.$		
Kod	kill(x_i)	delete(x_i); $x_i = null$	Instrukcja $gc()$ spowoduje usunięcie obiektu o. Pod warunkiem, że wcześniej wykonano $x_1 = null;$ $x_2 = null;$ ... $x_n = null;$
Post-	Wszystkie zmienne przyjmują wartość none. Obiekt o jest usunięty.	Obiekt o został usunięty. Zmienna x_i ma wartość null. Pozostałe zmienne nadal wskazuja na usunięty obiekt – to jest groźny błąd – wiszących referencji.	Obiekt o został usunięty – pod warunkiem, że wszystkim wskazaniom na usuwany obiekt została wcześniej przypisana wartość null.
Koszt	$O(1)$	$O(1)$	$O(n+m)$ m jest globalnym rozmiarem pamięci przeznaczonej na obiekty.
Ryzyko	Brak(!) Ponieważ każda próba dostępu do usuniętego obiektu (zapisu, odczytu, wywołania metody) spowoduje podniesienie alarmu `reference to none`.	Jeśli n>1 to powstaje błąd wiszących referencji. Wysokie prawdopodobieństwo wystąpienia błędu sprzecznych informacji i/lub błędu destrukcji.	Spore szanse, że programista zapomni usunąć któryś wskaźnik do obiektu o, a wtedy obiekt nie zostanie usunięty.

ROZDZIAŁ 20

Kolejki priorytetowe

Pojęcie kolejki priorytetowe jest zbliżone do pojęcia kontenera. Mówimy o kolejkach priorytetowych gdy zbiór E elementów jest wyposażony w dwie relacje: $=_E$ – relację równości oraz relację $<$ porządku w zbiorze E.
Kolejki priorytetowe mają niewiele wspólnego z kolejkami w zwykłym sensie tj. kolejkami FIFO. Nie są znane implementacje kolejek priorytetowych inne niż w drzewach.

1. Pojęcie kolejki priorytetowej

Definicja 20.1. *Kolejką priorytetową* nazywamy strukturę algebraiczną $\mathcal{KP}$, której uniwersum składa się z dwu rozłącznych zbiorów E i S, $E \cap S = \emptyset$.

$$\mathcal{KP} = \langle E \cup S; i, d, m, r, \leq, p, c, q \rangle$$

W strukturze tej mamy następujące operacje i, d, m oraz predykaty $r, <, p, c, q$. Sygnatura operacji jest wyliczona poniżej

$i\colon E \times S \to S$
$d\colon E \times S \to S$
$m\colon S \to E$

Relacje

$r\colon E \times E \to B_0$
$\leq\colon E \times E \to B_0$
$p\colon S \to B_0$
$c\colon E \times S \to B_0$
$q\colon S \times S \to B_0$

Aksjomaty struktury kolejek priorytetowych

k1) $\forall_{e\in E}\, \forall_{s\in S}\, (c(e, i(e,s)) \wedge \forall_{e'\in E \wedge e' \neq e}(c(e', s) \Leftrightarrow c(e', i(e,s)))$
k2) $\forall_{e\in E}\, \forall_{s\in S}\, (\neg c(e, d(e,s)) \wedge \forall_{e'\in E \wedge e' \neq e}(c(e', s) \Leftrightarrow c(e', d(e,s)))$
k3) $\forall_{s\in S}(p(s) \Leftrightarrow \forall_{e\in E}\, \neg c(e,s))$
k4) $\forall_{s\in S}(\neg p(s) \Rightarrow c(m(s), s))$
k5) $\forall_{e\in E}\, \forall_{s\in S}\, (c(e,s) \Rightarrow \neg(e \leq m(s)))$

k6) $\forall_{e\in E}\forall_{s\in S}\ c(e,s) \Leftrightarrow$
$$\left\{\begin{array}{l}\textbf{block}\\ \quad\textbf{var}\ bool : Boolean, s1 : S\\ \textbf{begin}\\ \quad s1 := s;\ bool := false;\\ \quad\textbf{while}\ \neg bool \wedge \neg p(s1)\ \textbf{do}\\ \quad\quad e1 := m(s1);\\ \quad\quad bool := r(e1,e);\\ \quad\quad s1 := d(e1,s1)\\ \quad\textbf{od}\\ \textbf{end}\end{array}\right\} bool$$

k7) $\forall_{s\in S}\{\textbf{while}\ \neg p(s)\ \textbf{do}\ s := d(m(s),s)\ \textbf{od}\}\ p(s)$

k8) $\forall_{s,s'\in S}\, q(s,s') \Leftrightarrow \forall_{e\in E}(c(e,s) \Leftrightarrow c(e,s'))$

k9) $\forall_{e\in E}\ r(e,e)$

k10) $\forall_{e,e'\in E}\ r(e,e') \Leftrightarrow r(e',e)$

k11) $\forall_{e,e',e''\in E}\ (r(e,e') \wedge r(e',e'')) \Rightarrow r(e,e'')$

k12) $\forall_{e\in E}\ e \leq e$

k13) $\forall_{e,e'\in E}\ e \leq e' \wedge e' \leq e \Rightarrow r(e,e')$

k14) $\forall_{e,e',e''\in E}\ (e \leq e' \wedge e' \leq e'') \Rightarrow e \leq e''$

Koniec definicji kolejki priorytetowej.

2. Niesprzeczność, tw. o reprezentacji, modele

Ta specyfikacja kolejek priorytetowych jest niesprzeczna.

Twierdzenie 20.1. Zbiór formuł k1) – k14) posiada model.

Dowód. Dowód tego twierdzenia przebiega podobnie do dowodu twierdzenia 18.1. Niech E będzie dowolnym zbiorem, uporządkowanym przez relację $\leq_E$, niech $=_E$ oznacza relację identyczności w zbiorze E.

Rozpatrzmy zbiór $Fin(E)$ wszystkich skończonych podzbiorów zbioru E. Przyjmiemy następującą interpretację symboli sygnatury kontenera:

$i(e,s)$	$d(e,s)$	$m(s)$	$r(e,e')$	$e \leq e'$	$p(s)$	$c(e,s)$	$q(s,s')$
$s \cup \{e\}$	$s \setminus \{e\}$	$min(s)$	$e =_E e'$	$e \leq_E e'$	$s = \emptyset$	$e \in s$	$s =_S s'$

Przy takiej interpretacji symboli każda formuła ze zbioru k1) – k14) jest prawdziwa. □

Strukturę algebraiczną $E \cup Fin(E)$ z działaniami opisanymi w powyższej tabelce będziemy nazywać standardowym modelem kolejek priorytetowych wyznaczonym przez zbiór E.

Twierdzenie 20.2. Każda kolejka priorytetowa $\mathcal{B}$ jest izomorficzna z standardowym modelem kolejek priorytetowych wyznaczonym przez zbiór elementów struktury $\mathcal{B}$.

Dowód. Dowód przebiega w sposób podobny do dowodu twierdzenia 18.2. □

3. Zastosowania

Oba znane algorytmy budowania drzewa rozpinającego grafu: algorytm Kruskala i algorytm Prima wykorzystuja strukturę kolejki priorytetowej, algorytm Chartresa znajdowania kolejnej liczby pierwszej, algorytm Dijkstry znajdowania najkrôtszj ścieżki w grafie, zarządzanie zdarzeniami w klasie Simulation,

Ćwiczenia

20.1. Przeprowadź dowôd twierdzenia 20.1.

20.2. Przeprowadź dowôd twierdzenia 20.2.

ROZDZIAŁ 21

Drzewa BST

Przyjmujemy, że dany jest zbiór E uporządkowany przez relację $<$. W praktyce programistycznej mamy najczęściej do czynienia ze zbiorem rekordów, takim, że klucze przypisane rekordom są uporządowane przez pewną relację zwrotną, przechodnią i antysymetryczną $less$.

Drzewa binarnych poszukiwań są znane od połowy XX wieku. Stosowanie drzew BST umożliwia szybkie wykonanie następujących operacji.

Każda implementacja drzewa BST pozwala zaimplementować kolejkę priorytetową.

PLAN

(1) definicja (algebry) struktury drzew BST,
(2) aksjomaty
(3) model standardowy - S-wyrażenia
(4) model klasa Node,
(5) własności struktury BST
(6) implementacja kolejek PQ

1. Struktura drzew BST?

Niech E oznacza zbiór uporządkowany przez relację $\leq_E$, zwrotną przechodnią i antysymetryczną. Równość $=_E$ elementów zbioru E ...

Definicja 21.1. *Strukturą drzew binarnych poszukiwań* nazywamy system algebraiczny

$$\mathcal{BST} = \langle E \cup BST \cup \{none\}; v, l, r, n, ul, ur; m, \leq, = \rangle$$

gdzie

v jest jednoargumentową operacją typu $(BST \to E)$
l, r są jednoargumentowymi operacjami typu $(BST \to \{BST \cup \{none\}\})$
n jest jednoargumentową operacją typu $(E \to BST)$
ul, ur są dwuargumentowymi operacjami typu $(BST \times BST \to BST)$
m jest funkcją charakterystyczną relacji $m\colon E \times BST \to B_0$

Ponadto, dla dowolnego $e \in E$ i dowolnych drzew $n_1, n_2 \in BST$ prawdziwe są następujące własności (Aksjomaty):

bst1) $v(n(e)) = e \wedge l(n(e)) = none \wedge r(n(e)) = none$

bst2) $S : \left\{ \begin{array}{l} \textbf{begin} \\ \quad n1 := n; continue := true; \\ \quad \textbf{while } (\mathrm{n1} \neq \underline{none}) \wedge continue \\ \quad \textbf{do} \\ \quad\quad \textbf{if } \mathrm{e} = \mathrm{n1.val} \textbf{ then } continue := false \\ \quad\quad \textbf{else} \\ \quad\quad \textbf{if } e < n1.val \textbf{ then } n1 := n1.l \\ \quad\quad \textbf{else } n1 := n1.r \textbf{ fi} \\ \quad\quad \textbf{fi}; \\ \quad \textbf{done} \\ \textbf{end} \end{array} \right\} true$

bst3) $m(e,n) \Leftrightarrow M : \left\{ \begin{array}{l} \textbf{begin} \\ \quad n1 := n; result := \underline{false}; \\ \quad \textbf{while} \neg result \wedge \ n1 \neq \underline{none} \\ \quad \textbf{do} \\ \quad\quad \textbf{if } e = n1.v \textbf{ then } result := \underline{true} \\ \quad\quad \textbf{else} \\ \quad\quad\quad \textbf{if } e < n1.v \textbf{ then } n1 := n1.l \\ \quad\quad\quad \textbf{else } n1 := n1.r \textbf{ fi} \\ \quad\quad \textbf{fi} \\ \quad \textbf{done} \\ \textbf{end} \end{array} \right\} result$

bst4) $m(e, n.l) \Rightarrow e < v(n)$

bst5) $m(e, n.r) \Rightarrow v(n) < e$

bst6) $(n = n' \equiv (n = none = n') \vee (n.v = n'.v \wedge n.l = n'.l \wedge n.r = n'.r))$

bst7) $\big(n.r = n'' \wedge n.v = e \wedge \left(\left\{ \begin{array}{l} K : \textbf{begin} \\ \quad n2 := n'; \\ \quad \textbf{while } n2.r \neq none \\ \quad \textbf{do} \\ \quad\quad n2 := n2.r \\ \quad \textbf{done}; \\ \quad bol := n2.v < n.v \\ \textbf{end} \end{array} \right\} bol \vee n' = none \right) \big)$
$\Rightarrow \{n3 := ul(n', n)\}(n3.r = n'' \wedge n3.v = e \wedge n3.l = n')$

bst8) $\big(n.l = n'' \wedge n.v = e \wedge (\left\{ \begin{array}{l} L : \textbf{begin} \\ \quad n2 := n'; \\ \quad \textbf{while } n2.l \neq none \\ \quad \textbf{do} \\ \quad\quad n2 := n2.l \\ \quad \textbf{done}; \\ \quad bol := n.v < n2.v \\ \textbf{end} \end{array} \right\} bol \vee n' = none)\big)$
$\Rightarrow \{n3 := ur(n', n)\}(n3.r = n' \wedge n3.v = e \wedge n3.l = n'')$

bst9) aksjomaty liniowego porządku $\leq$ i równości $=$

bst10) aksjomaty o Exception - do wymyslenia

Będziemy także badać zbiór konsekwencji przyjętego układu aksjomatów.

Definicja 21.2. Algorytmiczną teorią drzew binarnych poszukiwań nazywać będziemy zbiór konsekwencji wyznaczony przez zbiór formuł $\mathcal{B}st = \{bst1, \ldots, bst10\}$.

$$\mathcal{ATBST} = \mathcal{C}(\mathcal{B}st).$$

Lemat 21.1. M oznacza program występujący w aksjomacie bst2. Własność stopu tego programu jest twierdzeniem teorii $\mathcal{ATBST}$.

$$\mathcal{ATBST} \vdash \mathbf{M}\,\mathrm{true}$$

Dowód. Łatwo zauważyć, że program M ma obliczenie skończone wtedy i tylko wtedy gdy obliczenie skończone ma program S. Z twierdzenia o pełności wynika istnienie dowodu dla formuły M$true$. Można jednak łatwo wykazać istnienie dowodu na innej drodze. Zauważmy, że dla każdej liczby naturalnej $i \in N$ poniższa implikacja jest tautologią.

$$\left\{ \begin{array}{l} n1 := n; continue := true; \\ \left(\begin{array}{l} \textbf{if } (\mathrm{n1} \neq \underline{none}) \wedge continue \\ \quad \textbf{then} \\ \quad\quad \textbf{if } \mathrm{e} = \mathrm{n1.val} \textbf{ then } continue := false \\ \quad\quad \textbf{else} \\ \quad\quad \textbf{if } e < n1.val \textbf{ then } n1 := n1.l \\ \quad\quad \textbf{else } n1 := n1.r \textbf{ fi} \\ \quad\quad \textbf{fi}; \\ \quad \textbf{fi} \end{array} \right)^i \end{array} \right\} true \Rightarrow$$

$$\left\{ \begin{array}{l} n1 := n; result := false; \\ \left(\begin{array}{l} \textbf{if } (\mathrm{n1} \neq \underline{none}) \wedge \neg result \\ \quad \textbf{then} \\ \quad\quad \textbf{if } \mathrm{e} = \mathrm{n1.val} \textbf{ then } result := true \\ \quad\quad \textbf{else} \\ \quad\quad \textbf{if } e < n1.val \textbf{ then } n1 := n1.l \\ \quad\quad \textbf{else } n1 := n1.r \textbf{ fi} \\ \quad\quad \textbf{fi}; \\ \quad \textbf{fi} \end{array} \right)^i \end{array} \right\} true \tag{80}$$

Dowód tego można przeprowadzić przez indukcję. Dla $i = 0$ mamy oczywiscie

$$\vdash \{n1 := n;\ continue := true\} true \Rightarrow \{n1 := n;\ result := false\} true$$

Załóżmy, że istnieje dowód dla implikacji 80... czytelniej proszę Zapiszmy jej schemat

$$\vdash \{N;\ (if\ \gamma\ then\ M\ fi)^i\} true \Rightarrow \{K;\ (if\ \delta\ then\ L\ fi)^i\} true$$

Wykorzystując wyłącznie rachunek zdań sprawdzamy że tautologią jest też formuła postaci

$$\vdash \{N;\ (if\ \gamma\ then\ M\ fi)^{i+1}\} true \Rightarrow \{K;\ (if\ \delta\ then\ L\ fi)^{i+1}\} true$$

Stąd wnioskujemy, że dla każdego $i \in N$ jest tautologią formuła

$$\vdash \{N;\,(if\ \gamma\ then\ M\ fi)^{i+1}\}true \Rightarrow \{K;\,(while\ \delta\ do\ L\ od)\}true$$

czytelniej W dowodzie wykorzystujemy aksjomat Ax21 logiki algorytmicznej 1.3.1. Wiedząc, że dla każdego $i \in N$ formuła o powyższej postaci jest tautologią możemy zastosować regułę R3 1.3.1 i otrzymujemy że implikacja ?? jest tautologią. Dowód kończy zastosowanie reguły odrywania R1. □

2. Modele. Tw. o niesprzeczności

Opisana powyżej teoria ATBST ma wiele modeli. Poniżej przytoczymy dwa takie modele. Sformułujemy i udowodnimy twierdzenia: o niesprzeczności zbioru aksjomatów bst1) – bst10) oraz twierdzenie o reprezentacji.

Model standardowy. Przykładem struktury drzew BST jest zbiór S-wyrażeń z odpowiednio określonymi operacjami.

Przykład 21.1. Niech E będzie zbiorem uporządkowanym przez relację $<$. Zbiór S-wyrażeń nad zbiorem E jest to najmniejszy zbiór wyrażeń S taki, że

e) dla każdego $e \in E$ napis $(()e())$ nalezy do zbioru S,
c) jeśli napisy t_1 i t_2 należą do zbioru S, $e \in E$ i ponadto największy element zbioru E występujący w napisie t_1 jest mniejszy od e oraz najmniejszy element zbioru E występujący w napisie t_2 jest większy od e, to napis $(t_1 e t_2)$ należy do zbioru S

W zbiorze S-wyrażeń określamy następujące działania

$$\begin{aligned}
new(e) &= (()e()) \quad \text{dla dowolnego } e \in E \\
val(t_1\, e\, t_2) &= e \\
left(t_1\, e\, t_2) &= t_1 \\
right(t_1\, e\, t_2) &= t_2 \\
upl(t,(t_1\, e\, t_2)) &= \begin{cases} (t\, e\, t_2) & \text{wttw } (t\, e\, t_2) \text{ jest elementem zbioru } S \\ \text{nieokreślony} & \text{w pozostałych przypadkach} \end{cases} \\
upr(t,(t_1\, e\, t_2)) &= \begin{cases} (t_1\, e\, t) & \text{wttw } (t_1\, e\, t) \text{ jest elementem zbioru } S \\ \text{nieokreślony} & \text{w pozostałych przypadkach} \end{cases}
\end{aligned}$$

jest strukturą drzew BST.

Zauważ, że każde wyrażenie ze zbioru BST mozna narysować jako drzewo.

Przykład 21.2. rysunek

Twierdzenie 21.2. (o niesprzeczności)
Zbiór aksjomatów bst1) – bst8) posiada model.

Dowód. Rozważmy następująca interpretację

n	l	r	v	ul	ur
new	left	right	val	upl	upr

□

Strukturę opisaną powyżej nazywać będziemy modelem standardowym teorii BST.

Twierdzenie 21.3. (o reprezentacji)
Każdy model teorii BST jest izomorficzny z pewnym modelem standardowym.

Model obiektowy. W programach, struktura drzew binarnych poszukiwań jest implementowana na podstawie następującej lub podobnej deklaracji klasy.

```
unit Node: class(e: E);
  var l,r: Node
end Node
```

gdzie E jest nazwą pewnej klasy opisującej typ obiektów klasy E. Zakładamy, że deklaracja tej klasy zawiera dwie deklaracje funkcji definiujące: relację $\underset{E}{\leq}$ porządku w zbiorze obiektów typu E, oraz relację równości $\underset{E}{=}$ w tym samym zbiorze. rys. obiektu Node Z tą niepozorną deklaracją klasy wiąże się następująca struktura danych

$$\langle E \cup Node, v, l, r, new, ul, ur, isnone, \underset{E}{\leq}, \underset{E}{=} \rangle$$

gdzie ...
Własności klasy Node są zbyt liberalne na nasz gust. Pozwalają na stworzenie dowolnego grafu. Zaradzimy temu modyfikując powyższą deklarację.
Zmodyfikowana klasa Node, modyfikator private (tj. hidden) nie pozwoli na dowolne manipulacje na atrybutach klasy node. Oferowane metody ul i ur zapewniają, że wynik modyfikacji atrybutu będzie znowu drzewem BST.

Algorytm 21.3.

```
unit Node: class(e: E);
  (* private *)   hidden e,l,r;
  signal SigA;    (* naruszono bst4 lub bst5 *)
  var l,r: Node
  unit cl : function: Node;
  begin
    result := l
  end cl;
  unit ul: procedure(n1: Node);
  begin
    if max(n1) < e then l:=n1 else raise SigA fi
  end ul;
  unit cr : function: Node;
  begin
    result := r
  end cr;
  unit ur: procedure(n1: Node);
  begin
    if min(n1) > e then r:=n1 else raise SigA fi
  end ur;
  unit max : function(n: Node): E;
    var n2: Node
  begin
    n2:=n;
    while n2.r ≠ none do n2 :=n2.r od;
    result := n2.e
  end max;
  unit min : function(n: Node): E;
    var n2: Node
  begin
    n2:=n;
    while n2.l ≠ none do n2 :=n2.l od;
    result := n2.e
  end min
  handlers
    when SigA: writeln(" naruszenie aksjomatu bst4 lub bst5")
  end handlers
end Node
```

21.1. Czytelnik zechce sprawdzić, że wszystkie formuły bst1 – bst10) są prawdziwe przy takiej interpretacji.

3. Rozszerzenie struktury BST

Strukturę BST wzbogacimy o kilka definicji, uzasadniając ich poprawność.

Definicja 21.4. Operacja min zwraca najmniejszą wartość zapisaną w drzewie n.

$$min(n) \stackrel{df}{=} \left\{ \begin{array}{l} \textbf{if } n \neq none \\ \textbf{then} \\ \quad n1 := n \ ; \\ \quad \textbf{while } n1.l \neq none \textbf{ do } n1 := n1.l \textbf{ od} \\ \textbf{else} \text{ raise SigRefToNone} \\ \textbf{fi} \end{array} \right\} n1.v$$

Lemat 21.4. Dla dowolnego n takiego, że $n \neq none$, wartość $min(n)$ jest określona.

Dowód. Dla dowodu wystarczy zaobserwować, że w dowolnym modelu teorii ATBST, każde obliczenie poniższego programu

$$\textbf{while } n1 \neq none \textbf{ do } n1 := n1.l \textbf{ od}$$

jest udane i skończone. Jest to konsekwencja twierdzenia o reprezentacji 21.3. Fakt ten można też udowodnić formalnie wyprowadzając formułę stopu z aksjomatu bst1. □

Lemat 21.4 pozwala nam bezpiecznie dodać deklarację funkcji min do pozostałych deklaracji klasy $Node$. Kolej na definicję operacji member (programiści czasami nazywają ją contains). Aksjomat bst3 jest definicją funkcji boolowskiej sprawdzającej czy element $e \in E$ należy do drzewa n.
Lemat 21.1 upewnia nas, że można w bezpieczny sposób posługiwać się tą definicją, czytaj operować funkcją member zadeklarowaną w klasie . . .

Kolejny lemat pokazuje związek pojęć min i member.

Lemat 21.5. Następująca formuła jest twierdzeniem teorii $\mathcal{ATBST}$

$$\mathcal{ATBST} \vdash \forall_{n \in N} \forall_{e \in E} (member(e, n) \Rightarrow min(n) < e$$

i wobec tego formuła ta jest prawdziwa w każdym modelu algorytmicznej teorii $\mathcal{ATBST}$ drzew binarnych poszukiwań.

Opiszemy teraz dwie operacje porzednika $pred$ i następnika $succ$.
Operacja $succ(n, r)$ dla zadanego węzła n drzewa o korzeniu r wyznacza węzeł n' taki, że przypisana mu wartość $n'.v$ jest najmniejszą z wartości większych od $n.v$, zapisanych w drzewie r.

$$succ(n, r) \stackrel{df}{=} n' \text{ takie, że } m(n'.v, r) \wedge n'.v = Min\{e : m(e, r) \wedge n.v < e\}$$

Podobnie definiujemy operację $pred$

$$pred(n, r) \stackrel{df}{=} n' \text{ takie, że } m(n'.v, r) \wedge n'.v = Max\{e : m(e, r) \wedge e < n.v\}$$

A oto algorytm $SUCC$ wyznaczania następnika. Zauważ, że wykorzystujemy funkcję $father$ zwracająca ojca danego węzła w

drzewie, $father(root) = none$. Funkcję tę można z łatwością zaprogramować lub zapisać jej wykres w węzłach drzewa dodając pole $father$ obok pól $left, right$ i odpowiednio zmieniając algorytmy.

Algorytm 21.5.

(81)
$$SUCC: \left\{ \begin{array}{l} \textbf{if } n.right \neq none \\ \textbf{then} \\ \quad result := Min(n.right) \\ \textbf{else} \\ \quad y := father(n);\ x := n; \\ \quad \textbf{while } y \neq none \wedge x = y.right \textbf{ do} \\ \quad\quad x := y;\ y := father(y) \\ \quad \textbf{done}; \\ \quad result := y \\ \textbf{fi} \end{array} \right\}$$

Można sprawdzić, że algorytm ten poprawnie oblicza wartość funkcji $succ(n, r)$.

Lemat 21.6. Jeśli n jest węzłem drzewa r, to po wykonaniu algorytmu $SUCC$ zmienna result ma wartość równą $succ(n, r)$

$$m(n.v, r) \Rightarrow \{SUCC\}(result = succ(n, r))$$

Dowód. Dowód nie jest trudny. Należy rozpatrzyć dwa przypadki:
a) gdy węzeł n ma prawe poddrzewo $n.right$, wtedy następnikiem jest węzeł zawierający najmniejszy element tego poddrzewa.
b) w przeciwnym wypadku następnikiem węzła n jest taki węzeł y, przodek węzła n, który jest najbliższy węzłowi n i taki, że y ma lewego syna. Tzn.

$$y = father^j(n) \text{ gdzie } j = \mu i(father^i(n) = y \wedge y.l \neq none)$$

zachodzi przy tym równość $n.v = max\{e : m(e, y.l) \wedge e > n.v\})$. □

Działanie wstawiania elementu e do drzewa o korzeniu n opisuje następujący algorytm $INSERT$.

Algorytm 21.6.

$$insert(e,n) \stackrel{df}{=}$$

$$(82) \quad M: \left\{ \begin{array}{l} n1 := n;\ bol := false;\ res := n1; \\ \textbf{while } \neg(n1 = none \vee bol) \textbf{do} \\ \quad n2 := n1; \\ \quad \textbf{if } e = n1.v \textbf{ then } bol := true \\ \quad \textbf{else} \\ \quad\quad \textbf{if } e < n1.v \textbf{ then } n1 := n1.l \textbf{ else } n1 := n1.r \textbf{ fi} \\ \quad \textbf{fi} \\ \textbf{done}; \\ \textbf{if } \neg bol \textbf{ then} \\ \quad aux := \textbf{new } Node(e); \\ \quad \textbf{if } n2 = none \textbf{ then } n1 := aux \textbf{ else} \\ \quad\quad \textbf{if } e < n2.v \textbf{ then } n2.l := aux \textbf{ else } n2.r := aux \textbf{ fi} \\ \textbf{fi} \end{array} \right\} n1$$

Lemat 21.7. Niech M będzie programem z poprzedniej definicji 21.6.
Poniższa formuła jest prawdziwa w każdym modelu $\mathfrak{M}$ teorii $\mathcal{ATBST}$ drzew binarnych poszukiwań.

$$\mathcal{ATBST} \models \forall_{n \in BST} \forall_{e \in E} (\{M\}\, m(e,n1) \wedge (\forall_{e' \neq e} m(e',n) \Leftrightarrow M\, m(e',n1)))$$

Dowód. Każde dwa modele o tym samym zbiorze elementów E są izomorficzne. Wystarczy więc, jeśli sprawdzimy, że formuła ta jest prawdziwa w jakimkolwiek modelu teorii $\mathcal{ATBST}$ (nie korzystając przy tym z żadnych założeń o zbiorze elementów E).
Możemy jednak postąpić inaczej. Udowodnimy, że formuła ta jest twierdzeniem teorii $\mathcal{ATBST}'$, jaka powstaje gdy do teorii $\mathcal{ATBST}$ dodajemy nowy symbol operacji $insert$ i definiujący ją aksjomat tj. definicje 21.6.
Zauważmy, że $Strue \Rightarrow Mtrue$ jest tautologią.
Z kolei albo

$$((m(e,n) \wedge Mbol)$$

lub

$$(\neg m(e,n) \wedge M(n1 = none \wedge pred(n2).v < e < succ(n2) \wedge father(n1) = n2))$$

Czyli program M ustala czy element e należy do drzewa n i nadaje zmiennej bol odpowiednią wartość, ponadto, w przypadku gdy, po wykonaniu programu M zmienna bol ma wartość false, obliczona przez program M wartość zmiennej $n1$ wskazuje gdzie należy wstawić nowy węzeł $n(e)$ jako odpowiedniego syna węzła $n2$.
Rozpatrzmy formuły postaci ... □

Algorytm 21.7.

$delete(e, n) \stackrel{df}{=}$

Δ {

Λ {
$n1 := n;\ bol := false;\ res := n1;$
while $\neg(n1 = none \vee bol)$**do**
  $n2 := n1;$
  if $e = n1.v$ **then** $bol := true$
  else
    if $e < n1.v$ **then** $n1 := n1.l$ **else** $n1 := n1.r$ **fi**
  fi
done;
}

Ψ {
if bol
then
  if $e < n2.v$ **then** $tolft := true$ **else** $tolft := false$ **fi**;
  if $n1.l = none \wedge n1.r = none$
  then
    if $tolft$ **then** $n2.l := none$ **else** $n2.r := none$ **fi**
  else
    if $n1.l = none$
    then
      if $n1 = n$
      then $res := n1.r$
      else
        if $tolft$ **then** $n2.l := n1.r$
        else $n2.r := n1.r$ **fi**
      fi
    else
      if $n1.r = none$
      then $(*\, n1\ has\ the\ left\ son\ only\, *)$
        if $n1 = n$
        then $(*found\ in\ the\ root*)$
          $res := n1.l$
        else
          if $tolft$
          then $n2.l := n1.l$
          else $n2.r := n1.l$
          fi
          $(*n2\ ommited\ n1\ to\ son*)$
        fi
      else $\{n1\ has\ two\ sons\}$
        $n4 := n1.r;$
        while $n4.l \neq none$
        do
          $n5 := n4;$
          $n4 := n4.l$
        done $(*\ n4.v = min(n1.r)\ *)$
        $n5.l := n4.r;$
        $n1.v := n4.v\ (*\ i.e.\ n1.v \leftarrow min(n4.r)\ *)$
      fi$(*\ n1.r = none2\ *)$
    fi$(*\ n1.l = none\ *)$
  fi$(*\ n1.l = none \wedge n1.r = none\ *)$
fi$(*bol*)$
}

} res

Lemat 21.8. Dla każdego elementu e i dla każdego drzewa BST n, program Δ występujący w definicji działania *delete* kończy obliczenie.

Dowód. Dla dowodu wystarczy zauważyć, że wykonywanie (jedynej) instrukcji Λ zawierającej instrukcję while, która występuje w tym programie kończy się po skończonej liczbie kroków.
Można to wyprowadzic formalnie z aksjomatu bst2 (por. dowód lematu 21.1) □

Lemat 21.9. Dla każdego elementu e i dla każdego drzewa BST n po wykonaniu instrukcji $\{n1 := n; bol := false; res := n1; \Lambda\}$ zachodzi dokładnie jedna z poniższych formuł

i) $\neg bol$ gdy element e nie występuje w drzewie n,
lub
ii) $bol \wedge n1.v = e \wedge (n2.left = n1 \vee n2.right = n1)$

Dowód. Dowód pozostawiamy czytelnikowi □

Następny lemat stwierdza poprawnośc algorytmu Ψ względem warunku początkowego α : $bol \wedge n1.v = e \wedge (n2.left = n1 \vee n2.right = n1)$ i warunku końcowego β : $\neg m(e,n) \wedge ...$

Lemat 21.10.

$$\mathcal{ATBST} \vdash \alpha \Rightarrow \Delta\beta$$

Dowód. □

Łącząc powyższe spostrzeżenia stwierdzamy, że aksjomat k2) kolejek priorytetowych jest twierdzeniem algorytmicznej teorii drzew BST $\mathcal{ATBST}_{ext}$ wzbogaconej o zestaw definicji *declar* jeśli zinterpretować nazwy ...

Lemat 21.11.

$$\mathcal{ATBST}_{ext} \vdash \forall_{e \in E} \forall_{s \in S} (\neg c(e, d(e,s)) \wedge \forall_{e' \in E \wedge e' \neq e} (c(e', s) \Leftrightarrow c(e', d(e,s))))$$

Udowodnić lemat stanowiący, że program S zawsze kończy obliczenie tj. bst2.

4. Implementacja kolejek priorytetowych

W tej sekcji udowodnimy, poprawność implementacji kolejek priorytetowych, jaka opisana jest w poniższej klasie PQS.
klasa Poniżej prezentujemy moduł klasy PriorityQueues, który implementuje strukturę kolejek priorytetowych na bazie klasy Node realizującej strukturę drzew binarnych poszukiwań. Dyskusja przeprowadzona w poprzednim podrozdziale pozwala nam zapewnić użytkownika o poprawności tej implementacji. Udowodniliśmy bowiem, ze kazdy aksjomat kolejek priorytetowych jest prawdziwy w strukturze obiektów typu Node.

```
unit PriorityQueues : class (type E; function less(e, e' : E) : Boolean);
(* the class PriorityQueues implements a family of abstract data types.
A data type is determined by its class of elements and an ordering relation less.
User! make sure that less fulfills the axioms of order *)
  unit node : class (v : E);
    variable l, r : node;
  end node;

  unit min : function (n : node) : E;
  begin
    while n.l ≠ none do n := n.l od;
    result := n.v
  end min;

  unit member : function (e : E, n : node) : Boolean;
    variable n1 : node, bool : Boolean;
  begin
      n1 := n;
      bool := false;
      while n1 ≠ none ∧ ¬ bool do
          if n1.v=e then
            bool := true
          else
            if less(e, n1.v) then
              n1 := n1.l
            else
              n1 := n1.r
            fi
          fi
      od;
      result := bool
  end member;


  unit empty : function (n : node) : Boolean :
  begin
    result := (n = none)
  end empty;
```

```
unit insert function (e : E, n : node) : node;
   variable n1, n2, n3 : node, bool : Boolean;
begin
  n1 := n; n3 := n; bool := false;
  while ¬ n1 = none ∧¬ bool
  do
  | n2 := n1;
  | if e = n1.v then bool := true
  | else
  |    if less(e, n1.v) then n1 := n1.l else n1 := n1.r fi
  | fi
  od;
  if ¬ bool
  then
     aux:=new node(e);
     if n3 = none then n3 := aux
     else if less(e, n2.v) then n2.l := aux else n2.r := aux fi
     fi
  fi;
  result := n3
end insert;



unit delete : function(e : E, n : node) : node;
variable n1, n2, n3, n4, n5 : node, bool1, leftson : Boolean;
begin
n1 := n; n3 := n; bool1 := false;
(* search e *)
while ¬ n1 = none ¬ bool1
do
n2 := n1;
if e = n1.v
then
bool1 := true
else

if less(e, n1.v)
then
n1 := n1.l
else
n1 := n1.r
fi
fi
od

if bool1
then e is found in n1, n2 is the father of n1
```

```
leftson := less(e, n2.v)
leftson n1 is the left son of n2;
if n1.l = none n1.r = none
then n1 is a leaf
if leftson then n2.l := none
else n2.r := none fi;
else n1 is not a leaf
if n1.l = none
then n1 has the right son only
if n1 = n
then
n3 := n1.r
else
if leftson
then
n2.l := n1.r
else
n2.r := n1.r
fi (* since now n2 is the father of the only son of n1 *)
fi n = n1?
else n1 has the left son only
if n1.r = none
then n1 has the left son only
if n1 = n
then found in the root
n3 := n1.l
else
if leftson
then
n2.l := n1.r
else
n2.r := n1.r
fi (* since now n2 is the father of the only son of n1 *)
fi
else n1 has two sons
n4 := n1.r;
while n4.l none
do
n5 := n4;
n4 := n4.l
od; (* n4 is the least element in the left subtree of n1 *)
n5.l := n4.r; (* i.e. we omitted the n4 node *)
n1.v := n4.v (* the value found in the n4 is put in n1 node *)
fi n1.r = none?
fi n1.l = none?
fi n1.l = none n1.r = none?
fi bool1?
result := n3
```

```
end delete

end PriorityQueues
```

Taki moduł klasy PriorityQueues implementuje strukturę kolejki priorytetowej wyznaczonej przez pewien typ T i zadaną w nim relację porządku *less*. Jeśli więc mamy klasę o strukturze podanej poniżej i towarzyszącą jej funkcję

```
unit T: class ...
   virtual unit equal: function(x:T): Boolean;

   end equal;
end T;
unit less: function(x:T): Boolean;
...
end less;
```

Przy czym funkcja less zapewnia prawdziwość następujących warunków:

s) $\forall x, y \ in \ T(less(x, y) \vee less(y, x))$
z) $\forall x \ in \ Tless(x, x)$
p) $\forall x, y, z \ in \ T((less(x, y) \wedge less(y, z)) \Rightarrow less(x, z))$
a) $\forall x, y \ in \ T((less(x, y) \wedge less(y, x) \Rightarrow x.equal(y))$

a funkcja *equal* spełnia trochę inny zestaw warunków.

21.2. Napisz te warunki.

I masz ponadto blok, w którym zapisano pewien algorytm abstrakcyjny, posługujący się zmiennymi typu T i zmiennymi typu *node* oraz działaniami *insert*, *member*, *delete*, *min* etc.
Na przykład blok

$APr:$
```
block
   var e,e1:E, n,n1,n2:node
begin
   e:= new T(...); ???
   n := new node(e);
   n1 := insert(e1,n);
...
end
```

To łącząc te cztery moduły: T, less, PriorityQueues i APr w jedną instrukcję bloku prefiksowanego klasą otrzymujemy pożyteczny(?) program

```
pref PriorityQueues(T,less) block
   var e,e1:T, n,n1,n2:node
begin
   e:= new T(...);
   n := new node(e);
   n1 := insert(e1,n);
...
end
```

Powyższa instrukcja bloku

A module which implements PriorityQueues can be used many times for different abstract programs. In order to do so one should prefix the block containing the abstract program with the following line

pref PriorityQueues (an actual type E, an actual procedure of comparison) block
...an abstract program, see.section.5.1
end of prefixed block, see.[9].

Adding just one line which fixes the name of the implementing module and its actual parameters causes that the operations of data structure of priority queues are realized in accordance with the meaning given by the implementing module PriorityQueues.

ROZDZIAŁ 22

Kopce

Struktura danych – kopce, ma wiele zastosowań. Omówimy parę ważnych przykładów: algorytm sortowania przez kopcowanie *heapsort* i zastosowanie kopców do efektywnej implementacji struktury kolejek priorytetowych gwarantującej pesymistyczny czas operacji $O(log\ n)$.

1. Definicja

Czym jest kopiec? Student udzieli szybkiej odpowiedzi - kopiec to drzewo binarne, doskonałe, w którym każda ścieżka jest uporządkowana niemalejąco. Bardzo dobra odpowiedź. A jak to przetłumaczyć na język programistów? Wielu progamistów powie kopiec jest to tablica. Niektórzy programiści powiedzą, kopiec to drzewo o pewnych własnościach... Jak z tych propozyji wybrać tę właściwą? Poszukujemy definicji podobnej do definicji stosów, kolejek, drzew BST. Trzeba określić z jakimi operacjami mamy do czynienia i jakie mają one właściwości.

Definicja 22.1. Struktura algebraiczna $\mathcal{K}$

$$\mathcal{K} = \langle E \cup N \cup K \cup \{none\}; r, o, l, p, f, v; \leq, =, m \rangle$$

przyjmijmy, dla wygody, że $Nn \stackrel{df}{=} N \cup \{none\}$

jest strukturą kopców wtedy i tylko wtedy gdy wyliczone powyżej funkcje mają liczbę i typy argumentów podane w poniższym zestawieniu

$$\begin{array}{lll} r\colon K \to N, & o\colon K \to N, & v\colon N \to E, \\ p\colon N \to Nn, & l\colon N \to Nn, & f\colon N \to Nn, \\ \leq\colon E \times E \to B_0, & m\colon N \times K \to B_0, & =\colon E \times E \to B_0, \end{array}$$

Ponadto prawdziwe są własności k1) – k11), zob. poniżej. popraw k3, dodaj uporządowanie ścieżki

k1) $\forall_{k \in K}\, \{n := o(k);$ **while** $n \neq r(k)$ **do** $n := f(n)$ **od**$\}(n = r(k))$

k2) $\forall_{k\in K}\forall_{n\in N}\left(m(n,k) \Rightarrow \left\{\begin{array}{l} n1 := n;\ n2 := o(k);\ bl := true; \\ \textbf{while}\ n1 \neq r(k) \wedge n2 \neq r(k)\ \textbf{do} \\ \quad n1 := f(n1); \\ \quad \textbf{if}\ n1 = none\ \textbf{then}\ bl := false; exit \\ \quad\quad \textbf{else}\ n1 := f(n1)\ \textbf{fi}; \\ \quad n2 := f(n2)\ ; \\ \quad \textbf{if}\ n1 \neq r(k) \wedge n2 = r(k) \\ \quad\quad \textbf{then}\ bl := false\ \textbf{fi} \\ \textbf{od} \end{array}\right\} bl\right)$

k3) $\forall_{k\in K}\,(n = o(k)) \Rightarrow (l(n) = none \wedge p(n) = none \wedge (nie\ istnieje\ na\ prawo))$

k4) $\forall_{n,n'\in N}\, f(n') = n \Rightarrow (l(n) = n' \vee p(n) = n')$

k5) $\forall_{n,n'\in N}\, l(n') = n \Rightarrow f(n) = n'$

k6) $\forall_{n,n'\in N}\, p(n') = n \Rightarrow f(n) = n'$

k7) $\forall_{k\in K}\forall_{n,n'\in N}\,(n = f(n') \wedge n \neq f(o(k))) \Rightarrow (l(n) \neq none \wedge p(n) \neq none)$
$\forall_{k\in K}\forall_{n,n'\in N}\,(n \neq o(k)) \Rightarrow (l(f(n)) \neq none \wedge p(f(n)) \neq none)$

k8) $\forall_{k\in K}\forall_{n\in N}\,(n \neq r(k) \wedge m(f(n),k)) \Rightarrow (v(f(n)) \leq v(n))$

k9) **aksjomaty porządku $\leq$, zwrotność**

k10) **przechodniość**

k11) **antysymetria**

k1y) $sub(n,n') \stackrel{df}{=} \{n1 := n;\ while\ n1 \neq n'\ do\ n1 := f(n1)\ od\}(n1 = n')$

k1z) $onleft(n,n') \stackrel{df}{=} \exists h(sub(n, h.left) \wedge sub(n', h.right))$

Pytanie. Czy te aksjomaty posiadają model? a może są sprzeczne?
Odpowiedzi na to pytanie udziela następujące

Twierdzenie 22.1. Powyższy układ aksjomatów posiada model, a więc jest niesprzeczny.

Dowód. Powszechnie znaną realizacją kopca jest tablica. O strukturze E elementów nie musimy niczego zakładać. Dla ustalenia uwagi, przyjmijmy więc, że elementami są liczby naturalne z ich naturalnym porządkiem. Kopcem k będzie więc tablica A liczb naturalnych. Węzłami będą zmienne – elementy tablicy $A[1], \ldots, A[n]$. Interpretujemy nazwy działań w następujący sposób

$r(k)$	$o(k)$	$l(A[i])$	$p(A[i])$	$f(A[i])$	$v(A[i])$	$\leq$	$m(A[i], A)$
$A[1]$	$A[n]$	$A[2*i]$	$A[2*i+1]$	$A[i \div 2]$	$val(A[i])$	$\leq$	$1 \leq i \leq n$

Dokładniej, korzeniem kopca k jest zmienna $A[1]$, wartością funkcji $o(k)$ jest zmienna $A[n]$. Wartości funkcji l, p, f są opisane w powyższej tabelce, z następującym zastrzeżeniem, jeśli wartość wyrażenia $2*i, 2*i+1, i \div 2$ wykracza poza zakres $1 \ldots n$, to przyjmujemy, że wartość funkcji jest none. Wartością wyrażenia $v(A[i])$ jest liczba przypisana zmiennej $A[i]$. Po kolei sprawdzamy prawdziwość każdego aksjomatu w tej interpretacji i przekonujemy się, że wszystkie własności k1) – k12) są prawdziwe.
A więc istnieje model teorii $\mathcal{ATK}$ algorytmicznej teorii kopców i wobec tego, teoria ta jest niesprzeczna. □

Niech tablica A będzie odpowiednio duża tj. $lower(A) \lneq l \leq u \lneq upper(A)$. Warto zadać sobie pytanie: czy fragment tej tablicy $A[l], \ldots, A[u]$ jest kopcem?

Kolejne pytanie jakie staje przed nami, brzmi: czy każda struktura kopców jest izomorficzna ze strukturą kopców w tablicach? Na to pytanie udziela odpowiedzi następujące

Twierdzenie 22.2. (o reprezentacji) Każda struktura kopców $\mathcal{K}$ jest izomorficzna ze strukturą $\mathcal{KT}$ kopców zapisanych w tablicach o tym samym zbiorze elementów E.

Uwaga. Powinniśmy odróżniać strukturę kopców od zbioru kopców.
Powyższe aksjomaty opisują obiekty, które kwalifikują się do nazwy kopiec. Ale na kopcach wykonywać można operacje, które zwracają inne kopce.
Trzeba to podkreślić i uzupełnić. Koniec uwagi

2. Heapsort - sortowanie w kopcu

W tym podrozddziale pokażemy jak można wykorzystać dziedziczenie by "wyciągnąć wspólną część algorytmu przed nawias". Efekt ten osiągamy przez dziedziczenie.

Antoni Kreczmar zaprogramował w Loglanie algorytm heapsort wykorzystując możliwość dziedziczenia klasy przez procedurę.

2.1. Konstruowanie algorytmu. Niech A będzie tablicą liczb całkowitych $A[1], A[2], \ldots, A[n]$. Niech $p = n$ i $l = p$ **div** 2.
Zacznijmy od analizy następującego kodu

Kopcuj:

```
i :=l; j := 2*i; x :=A(i);
while j<=p do
   if j<p andif 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;
done;
A(i) := x;
```

Niech dane będą: warunek wstępny
α: tablica A na miejscach $l+1, \ldots, p$ reprezentuje las kopców,
oraz warunek końcowy
β: tablica A na miejscach $l, l+1, \ldots, p$ reprezentuje las kopców.

Lemat 22.3. Program Kopcuj jest poprawny ze względu na warunek początkowy α i warunek końcowy β

$$\alpha \Rightarrow \{\textbf{Kopcuj}\}\, \beta.$$

Dowód. Gdy $j > p$ to $A[l]$ jest liściem, $A[l]$ jest też korzeniem. A więc fragment tablicy $A[l], A[l+1], \ldots, A[p]$ jest lasem kopców. Rozważmy przypadek gdy $j \leq p$. Zacznijmy od obserwacji, że po wykonaniu instrukcji
if j<p andif A[j] < A[j+1] then j := j+1 fi
zachodzi $A[j] = max\{A[2i], A[2i+1]\}$. W przypadku gdy $x > A[j]$ to z założenia indukcyjnego x jest większe od wszystkich węzłów poddrzewa $A[j]$, a więc i od wszystkich węzłow poddrzewa $A[j-1]$. Nie ma nic więcej do roboty i opuszczamy (exit) petlę. Jeśli tak nie jest to kładąc A[i]:=A[j]; i:= j; j:=2*i; sprowadzamy nasz problem do problemu przesiewania w poddrzewie drzewa początkowego. Po pewnej liczbie powtórzeń algorytm zatrzyma się ponieważ kolejne drzewo będzie liściem. □

Rozpatrzmy teraz następujący kod

BS:
```
l:= (upper(A) div 2)+1;
do
   l:= l-1;
   if l=lower(A) then exit fi;
   Kopcuj
od
```

Założyliśmy, że $lower(A) = 1$.

Lemat 22.4. Program BS buduje kopiec z elementów tablicy A.

Dowód. Po wykonaniu instrukcji l:=(upper(A)div 2)+1 fragment tablicy $A : A[l], A[l+1], \ldots, A[upper(A)]$ jest lasem drzew. W każdej iteracji pętli do ... od powiększamy fragment zawierający las kopców. Powtarzanie zakończy się gdy $l = lower(A)$. Wtedy las kopców jest jednym kopcem o korzeniu $A[1]$. □

Teraz rozważmy kod

ST:
```
p:= upper(A); l:=lower(A);
do
   x:=A[l]; A[l]:=A[p]; A[p]:= x;
   p:=p-1;
   if p=lower(A) then exit fi;
   Kopcuj
od
```

Udowodnimy następujący

Lemat 22.5. Jeśli tablica A jest kopcem, to program ST sortuje elementy tablicy A w porządku niemalejącym.

Dowód. W pierwszym kroku iteracji zamieniamy miejscami $A[l]$ i $A[p]$, ponadto zmniejszamy wartość p. Ostatnim elementem tablicy A jest więc element największy. Gdy $l = p$ to tablica A jest uporządkowana niemalejąco i można zakończyć działanie programu ST. W przeciwnym przypadku może być tak, że fragment $A[l], \ldots, A[p]$ tablicy A nie jest kopcem. Ale po wykonaniu polecenia Kopcuj fragment ten będzie kopcem. Kolejny co do wielkości element trafi na miejsce p w tablicy A. Proces ten kończy się i z chwilą jego zakończenia tablica A jest uporządkowana niemalejąco. □

Co dalej?
Jak skonstruować procedurę heapsort?
Można zaproponować trzy rozwiązania:

- Treścią procedury mogą stać się programy BS i ST, w których powtarza się program Kopcuj.
- Treść procedury może polegać na wywołaniu po kolei procedur BudujKopiec i Sortuj, każda z tych procedur wywołuje procedurę Kopcuj.
- Na treść procedury heapsort składają się klasa Przesiej, procedura BudujKopiec rozszerzająca klasę Przesiej i procedura Sortuj też rozszerzająca klasę Przesiej.

Zestawmy te trzy możliwości obok siebie

```
unit heapsortA: procedure
    <deklaracje>
begin
    BS;
    ST
end heapsortA
```

```
unit heapsortB: procedure
    unit Kopcuj: procedure
    end Kopcuj
    unit BudKop: procedure
    ... call Kopcuj ...
    end BudKop
    unit Sortuj: procedure
    ... call Kopcuj ...
    end Sortuj
begin
    call BudKop;
    call Sortuj;
end heapsortB
```

```
unit heapsortC: procedure
    unit Przesiej:
      class ...
    unit BudKop:
      Przesiej procedure
    unit Sortuj:
      Przesiej procedure
begin
    call BudKop;
    call Sortuj;
end heapsortC
```

Każda z tych propozycji pozwoli skonstruować poprawną procedurę heapsort.

Która droga prowadzi do najlepszego rozwiązania? Procedura heapsortB ma największy koszt. Podczas wykonywania instrukcji BudKop i Sortuj instrukcja *call* Kopcuj będzie wykonywana wielokrotnie. Wielokrotnie trzeba będzie tworzyć rekord aktywacji tej procedury. Procedura heapsortC pozwala uniknąć tego narzutu. Zobaczymy jak. Ale najbardziej wydajna jest

chyba procedura heapsortA. Tyle, że wymaga to od nas dwukrotnego napisania kodu Kopcuj, wewnatrz programu BS i wewnątrz programu ST. Nie ma w tym nic złego tak długo jak długo nie trzeba wprowadzać zmian w kodzie Kopcuj – musimy tylko pamiętać, że trzeba to zrobić w dwu miejscach!

Dokończmy konstrukcję procedury heapsortC. Tworzymy klasę Przesiewanie.

```
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 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;
        done;
        A(i) := x;
    done
end przesiewanie;
```

i dwie procedury dziedziczące klasę Przesiewanie: budujKopiec i sortuj.

```
unit budujKopiec: Przesiewanie procedure;
begin
        koniec := (l=lower(A)); l :=l-1;
end budujKopiec;
```

Nie trzeba długo dowodzić, że prawdziwą treść procedury budujKopiec stanowi treść klasy Przesiewanie w której pseudoinstrukcję inner zastąpiono dwoma instrukcjami z deklaracji procedury budujKopiec. Czyli należy pamiętać, że dzieki regule konkatenacji powinniśmy przyjąć, że instrukcja procedury budujKopiec zostanie wykonana tak jak gdyby procedura ta została zadeklarowana w sposób przytoczony poniżej. Dzięki zastosowaniu dziedziczenia skróciliśmy tekst i co ważniejsze uniknęliśmy wywołania procedury budującej kopiec.

```
unit budujKopiec: (* Przesiewanie *) procedure;
    var koniec: boolean
begin
    do
        koniec := (l=lower(A)); l :=l-1;
        if koniec then exit fi;
        i :=l; j := 2*i; x :=A(i);
        while j<=p do
            if j<p and 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;
        done;
        A(i) := x;
    done
end budujKopiec;
```

Wcześniej przekonaliśmy się, że ten algorytm przekształca tablicę A w kopiec. Podobnie można sprawdzić, że instrukcja call sortuj uporządkuje tablicę A w ciąg niemalejacy.

```
unit sortuj: Przesiewanie procedure;
begin
    d:= lower(A); koniec:=(p=d);
    x:=A(d); A(d) := A(p); A(p) := x;
    p := p-1;
end sortuj;
```

Po przyjęciu tych trzech deklaracji łatwo uzupełnić treść procedury
heapsortC tak jak to widać poniżej.

```
unit heapsortC: procedure(A: arrayof integer);
    var i, j, l, p, x: integer;
    unit Przesiewanie: class ...
    unit budujKopiec: Przesiewanie procedure ...
    unit sortuj: Przesiewanie procedure ...
begin
    l := upper(A) div 2 +1;
    p := upper(A);
    call budujKopiec;
    call sortuj
end heapsortC
```

2.2. Kompletny algorytm. Poniżej przytaczamy kompletny tekst procedury heapsort (w wersji C). Z poprzednich rozważań wynika, że algorytm ten poprawnie porządkuje zadaną tablicę A.

```
unit heapsort: procedure(A: arrayof integer);

    var i,j,l,d,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 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;
            done;
            A(i) := x;
        done
    end Przesiewanie;

    unit budujKopiec: Przesiewanie procedure;
    begin
        koniec := (l=lower(A)); l :=l-1;
    end budujKopiec;

    unit sortuj: Przesiewanie procedure;
    begin
        d:= lower(A); koniec:=(p=d);
        x:=A(d); A(d) := A(p); A(p) := x;
        p := p-1;
    end sortuj;

  (* tu zaczynają się instrukcje procedury heapsort *)
begin
    l:= upper(A) div 2 +1;
    p:= upper(A);
    call budujKopiec;
    call sortuj;
end heapsort;
```

3. Czy to jest kolejka priorytetowa?

Tu pokażemy jak sprawdzić czy zadana klasa K spełnia specyfikację S i dowieść swojej racji.

3.1. Deklaracja klasy PQS. Poniżej znajdziesz pełny tekst klasy PQS napisanej wiele lat temu przez W.M. Bartol i D. Szczepańską. Jest to część większego programu symulacji pracy oddziału bankowego [RBSW79] [RBSW07]. Spróbuj przekonać kolegę, szefa, że jest to poprawna imlementacja struktury kolejki priorytetowe..

```
unit PQS : class; (* priority queues system *)
   unit PQ: class;
      var last,root:node;
      unit min: function: elem;
      begin
         if root=/= none then result:=root.el else raise ReftoNone fi;
      end min;
      ---------------------------------------------------
      unit insert: procedure(r:elem);
         var x,z: node;
      begin
         x:= r.lab;
         if last=none then
            root:=x; root.left, last:=root
         else
            if last.ns=0 then
               last.ns:=1; z:=last.left; last.left:=x;
               x.up:=last; x.left:=z; z.right:=x;
            else
               last.ns:=2; z:=last.right; last.right:=x;
               x.right:=z; x.up:=last; z.left:=x;
               last.left.right:=x; x.left:=last.left; last:=z;
            fi
         fi;
         call correctUp(r)
      end insert;
      ---------------------------------------------------
      unit delete: procedure(r: elem);
         var x,y,z:node;
      begin
         x:=r.lab; z:=last.left;
         if last.ns =0 then
            y:= z.up;
            if y=none then root:=none else y.right:= last fi;
            last.left:=y; last:=y;
         else
            y:= z.left; y.right:= last; last.left:= y;
         fi;
```

```
        z.el.lab:=x; x.el:= z.el; last.ns:= last.ns-1;
        r.lab:=z; z.el:=r;
        (* the following three instructions were added during our verification *)
        z.left.right :=none; z.ns:=0; z.left, z.right, z.up := none;
        if x.less(x.up) then
            call correctUp(x.el)
        else
            call correctDown(x.el)
        fi;
    end delete;
    ---------------------------------------------------------
    unit correctUp: procedure(r: elem);
        var x,z: node, t: elem, fin, log: Boolean;
    begin
        z := r.lab;
        x:= z.up;
        do
            if x=none then log:=true else log:=x.less(z) fi;
            if log then exit fi;
            t:=z.el; z.el:=x.el; x.el:=t;
            x.el.lab:=x; z.el.lab:=z; z:=x; x:z.up;
        od
    end correctUp;
-----------------------------------------------------------
unit correctDown: procedure(r: elem);
    var x,z: node, t: elem, fin, log: Boolean;
begin
    z := r.lab;
    while not fin
    do
        if z.ns = 0 then fin :=true
        else
            if z.ns=1 then x := z.left
            else
                if z.left.less(z.right) then x:= z.left else x:=z.right fi
            fi;
            if z.less(x)
            then
                fin := true
            else
                t:= x.el; x.el :=z.el; z.el:=t;
                z.el.lab :=z; x.el.lab:=x
            fi;
        fi;
        z:=x;
    od
end correctDown;
end PQ;
```

```
unit node: class (el:elem);
    var left,right,up: node, ns:integer;
    unit less: function(x:node): boolean;
    begin
        if x= none then
            result:=false
        else
            result:=el.less(x.el)
        fi;
    end less;
end node;
```

```
unit elem: class(prior:real);
    var lab: node;
    unit virtual less: function(x:elem):boolean;
    begin
        if x=none then
            result:= false
        else
            result:= prior ≤ x.prior
        fi;
    end less;
begin
    lab:= new node(this elem);
end elem;
end PQS (* priority queues system *);
```

3.2. Introduction. This paper presents a proof that the class PQS is a correct implementation of priority queues data structure. The class (see Appendix) is written in an object-oriented programming language. The specification (see Table 1) consists of a few algorithmic formulas. Formally, the proof of correctnes of the class w.r.t. the specification resembles a proof that a given algebraic structure $\mathbb{A}$ is a *model* of some axiomatic theory $\mathcal{T}$. Literature knows many examples of the proofs of correctnes of algorithms with respect to their pre- and post-conditions. However, the union of proofs of a class' methods (i.e. algorithms) does not result in the proof of the class correctness.

Each class can be considered as a description of an algebraic structure (a data structure). The universe of the structure is the set of all potential objects of the class, its functions and relations are defined by class methods.

Dealing with a class we are confronted with several properties, sometimes called invariants of the class (B. Meyer in Eiffel [Mey87]), that must be valid for all objects of the class. Moreover, an external characterization of the class requires some properties that express corellations between different methods.

All these properties will be called a specification of a class.

Our message has several layers:

- first, we offer a discussion of the methods of software's verification.
- second, we propose to think what a software's verification is,
- finally, we need to know what a class specification looks like?

Two views and two practices are colliding in production of software. One view is that programs should be accompanied by solid arguments demonstrating the correctness and the completeness of software. The practice associated with this view consist in proving properties of software. At present, we observe that there are more and more cases when the industry demands the proofs of correctness of programs. It is especially true of these cases where security must be guaranteed. Another view is followed by the majority of the individual programmers and the big software companies. They are of the opinion that testing of programs is the sufficient activity before the software is delivered to clients. Hence we have two practices that seem to exclude one another: testing or proving. In the presented paper we argue[1] that the two approaches may be synthesized to a completely different scheme of practice. We propose to replace the term testing by another word experimenting. Testing limits itself to the execution of program and the comparison of its effects with the predicted, supposedly correct results. For the majority of people testing is the practice of searching bugs in software. Experimenting has larger horizons, it covers not only searching of errors (aka counter-examples) but also gathering the positive evidence. Sometimes during experiments we begin to believe that objects obey certain rules and later we try to find more evidence confirming our beliefs. During experiments one is going to execute program with different data, to collect results and to present them in graphical mode, in tables, in data bases, etc. It is suggested that the experiments were done with some plan. Next, one should analyze the gathered experience and search some regularities. This process should lead to the formulation of lemmas and propositions. After this is done, there is the time of proving the hypotheses. Obviously, the process sketched above may need to be iterated for different reasons, e.g. when our program is modified. We consider the method given below

$$experiment \rightarrow observe \rightarrow formulate\ hypotheses \rightarrow prove.$$

[1] following to some extent the ideas of Georg Hegel who used to say that from a pair: thesis and antithesis we should make a synthesis

as a proper approach to the production of the high integrity software [MSS08], [Bar03], [?]Key:BB:RH:PS, [Ame01].

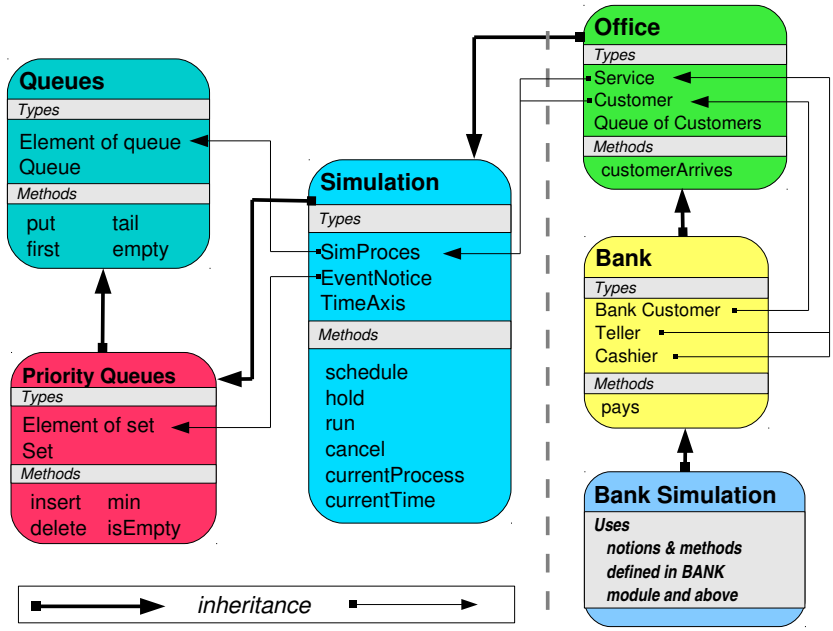
How to define the goals of software's verification? Assuming that a specification of a class is given in the form of a set of formulas, the verifier agent should prove that every object of the analyzed class will satisfy every formula of the set, whenever any method of the class has been completed.

We are proposing the methodology which consists of algorithmic logic AL and an environment SpecVer – a plugin into the Eclipse IDE. A SpecVer project can be developed from its initial phase of specifying algorithms and classes, through implementing them in an object programming language, to the phase of verification of implementation against its specification.

We present an almost complete case study which consists of a specification $ATPQ$ of priority queues, a class `PQS`, and the verification of the thesis that the class `PQS` correctly implements the specification $ATPQ$.

Class PQS may be used in several applications. For example, it is a part of a bigger program Simulation of a Bank Department. The structure of the simulation program is shown in Figure 1. The program contains 6 external classes, 13 innner classes and 20 methods. The relation between inner classes and the containing them external classes are shown on the diagram. The relation of inheritance is also shown on the diagram. The arrows lead from one class to its direct superclass. The thick arrows start at external classes. The thin arrows lead from an inner class K to inner class I inherited by the class K. We conceive the five external classes as implementations of data structures: Class `BankDepartment` extends the structure of `Office`. Class `Office` is based on the class `Simulation`, it uses also the data structure of `FIFOQueues`. The class `Simulation` relies on the class `PriorityQueues`. These relations are shown on the diagram by thick arrows. (As no one class may inherit two classes, we decided to make the class `FIFOQueues` the base class of the class `PriorityQueues`.) The diagram shows also the inheritance relation between inner classes. This allows us to introduce more subtle relations between classes. For example, any object of class `Customer` is queueable since the class `Customer` inherits from the class `Simprocess` which in turn inherits from the class `ElemFIFO`. An object of class `Customer` may be activated and made passive several times since the class `SimProcess` is a coroutine. The value of unique variable `ExperimPlan` is a set of `EventNotices`. During the execution of our simulation experiment the variable `ExperimPlan` has various sets of `EventNotices` as its value. An object of the class `EventNotice` is a pair: $\langle s.t \rangle$, where s is a `SimProcess` object and t is a time. Objects of class `EventNotice` are inserted and/or deleted from the set `ExperimPlan`. `EventNotices` are ordered by a

relation *less*. The class `Simulation` is to guarantee that the active object of the experiment, in our case it will be either a customer or a teller object, will be active iff its activation time is the minimum of all times of `EventNotices` in `ExperimPlan` set. Encapsulating two inner classes `SimProcess` and `EventNotice` and the variable `ExperimPlan` of type `PQ` makes the process of coding of the class `Simulation` simpler.



Rysunek 1. Moduły programu symulującego bank

The term high integrity software was introduced in [Bar03]. For us the high integrity programming means the activity which involves specification of software modules, implementation the modules (i.e. classes and methods) and verification of the modules against their specification. We shall use the structure shown in Figure 1 to illustrate what has to be done.
For each external class C three documents should be produced:

- A specification S of the class C. Specification is a set of formulas which express the properties of methods and invariants of objects of the class. Each specification should enjoy two properties:
  - Consistency
    There is no implementation of an inconsistent

specification. An inconsistent specification has no sense.
  - Completeness
    An incomplete specification allows various models. Not all of them are desired. A complete specification brings enough information on properties of objects of class to distinguish between a desired and an improper implementation, and therefore can be used as a criterion of acceptance of a class. It is sufficient to produce the proofs or verification reports.
- The file containing the class C itself. Usually this file contains all the methods and inner classes of the class C.
- The verification report. This file should contain arguments that soothe our conscience and convince the user of our class. The arguments used may be more formal - having form of a mathematical proof or may recall a dialogue. Rarely a formal proof is needed. The verification report should be rather an evidence of analysis and it should serve to convince its reader that the conclusions of the report are sound. A verification report is of good quality if it is an intersubjective experience. Surely, a formal proof of a correctnes has this quality, but frequently it is not readable by a human being. A balance between two extremities is needed.

In earlier papers we have presented the work on specifications of classes [MSS08]. Specifications are subjects of studies and analyses. In other words, one should visualize a process of development and amelioration of a specification.

This paper illustrates the process of verification of a class C against a specification S. In the most cases implementation of a class precedes the process of verification. Sometimes, the verification can be done simultaneously with the production of the needed class. In another paper we shall exemplify the (rare) case when a class, the Simulation class, is systematically elaborated together with the proof of its correctness. For this we need only the specification of the base class `PQS` and the target class `Simulation`.

We propose to make an experiment. The reader will look at the appendix and try to give arguments that the class PQS correctly implements a priority queues system. Those who forgot what a priority queue is, may find its axiomatic definition in Table 1 below. We ask the reader how much of time he/she needs to convince someone that the class `PQS` implements the abstract data type of priority queues.
We did the following:

(1) Experiments - we executed methods of the class by hand and have drawn some pictures.
(2) Observations - we analyzed the pictures and searched for some regularities.
(3) Conclusions - we formulated several hypotheses (lemmas) and propositions.
(4) Proving - we proved the lemmas.
(5) Finally - putting together the facts, we proved the correctness theorem.

3.3. Priority Queues Specification. Before implementing a data structure one should write down its specification. The first part of the specification – the signature – enlists the sorts, the operations and the relations. The second part enlists the properties, also called the axioms, Table 1 contains the specification of the abstract data type of priority queues [MS87, p.154]. Remark that besides the formulas of first-order logic we use algorithmic formulas [MS87]. An example of algorithmic formula is the axiom (a2). Its structure is as follow:

$$\langle program\ P\rangle\langle formula\ \alpha\rangle.$$

The meaning of the whole formula is after execution of program P the formula α is valid". In our example the formula (a2) takes value true if and only if the program halts. Hence, including such formula among the axioms of our specification, means "the program P always terminates". The semantic meaning of the axiom (a2) is the set q is finite. The axiom (a8) is in fact an explicit, algorithmic definition of the relation $member$.

Algorithmic formulas allow to express the semantic properties of programs such as termination, correctness, etc. Almost 40 years ago we proved that the calculus is sound and complete [MS87]. It means that we have choice between showing that some semantic property of a program is valid or proving the formula that expresses the property. We gave several examples of specification of abstract data types as algorithmic theories. $ATPQ$ is one of such examples.

Table 1. Specification $ATPQ$ of priority queues.

Signature	Comments
Sorts	$Universe = E \cup PQ$
E	set of elements
PQ	set of priority queues
Operations	let $e \in E$ and $q \in PQ$
$insert : E \times PQ \longrightarrow PQ$	put e into q
$delete : E \times PQ \longrightarrow PQ$	delete e from q
$min : PQ \longrightarrow E$	find the minimum element
$empty : PQ \longrightarrow \{true, false\}$	is a priority queue q empty?
$member : E \times PQ \longrightarrow \{true, false\}$	does $e \in q$?
$\leq : E \times E \longrightarrow \{true, false\}$	the ordering relation

Axioms

(a1) *The set E ofelements is linearly ordered by the relation* $\leq$.

(a2) [**while** *not* $empty(q)$ **do** $q := delete(min(q), q)$ **done**] *true*

This axiom says for all q program halts, i.e. the priority queue q is finite

(a3) $[q1 := insert(e, q)]\{member(e, q1) \wedge (\forall_{e1 \neq e}\ member(e1, q1) \Leftrightarrow member(e1, q))\}$

(a4) $[q1 := delete(e, q)]\{\neg member(e, q1) \wedge (\forall_{e1 \neq e}\ member(e1, q1) \Leftrightarrow member(e1, q))\}$

(a5) $empty(q) \Rightarrow (\forall_{e \in E}\ \neg member(e, q))$

(a6) $\neg empty(q) \Rightarrow (\forall_{e \in E}\ member(e, q) \Rightarrow min(q) \leq e))$

The operation min finds the least element of the set q.

(a7) $[e := min(q)]true \Leftrightarrow \neg empty(q)$

Axiom (a7) says the result of expression $min(q)$ is defined iff $\neg empty(q)$

(a8) $member(e, q) \Leftrightarrow$ **begin**

```
s1 := q; result := false;
while not empty(s1) and not result do
   if e =min(s1) then result:=true fi;
   s1 := delete(min(s1),s1)
done
end result
```

$ATPQ$ is an acronym of Algorithmic Theory of Priority Queues. The theorems of the theory are the formulas provable by the calculus of algorithmic logic [MS87] from the axioms of $ATPQ$.

Uwaga 22.6. In the literature, the frequent choice when speaking on the abstract data type of priority queues, is the operation deletemin, instead of the operation delete. Our choice was different and deliberate. With the present set of operations we are able to construct the specification which enjoys the property of being almost complete.

It was shown in [MS87] that the algorithmic theory of priority queues has a metalogical property known as the representation meta-theorem: Every model of $ATPQ$ is isomorphic to the standard model of priority queues. [2]

[2] Algorithmic theories (e.g. [Sal80], [MS87]) were studied since 70's of XX century.

Let us add a few words on the $ATPQ$ specification. An important fact states that any implementation of the axioms of $ATPQ$ where a concrete set E was given, is isomorphic to the structure of finite subsets of the set E with the operations

$$insert(e, s) = \text{increase the set } s \text{ by adding element } e \text{ to it,}$$

and

$$delete(e, s) = \text{supprime the element } e \text{ from the set } s.$$

The consequences of the theorem are of general methodological nature:

- the specification of $ATPQ$ is complete. If a new formula is added then either it is a logical consequence of the axioms or it leads to contradiction, or it expresses the properties of the elements only.
- the specification can be used as the criterion of correctness of a proposed implementation,
- the proofs of properties of programs that use this data structure may be based on axioms of priority queues listed in Table 1. No other properties of priority queues are ever needed.

3.4. Experiments. Our work on verification of the class PQS began by experimenting. The experiments consisted in executing (by hand) operations insert and delete, drawing pictures, and observing the changes in the constellation of objects of type node. Figure 2 shows a few snapshots of our experimentation. In fact, we did twice as much drawings. The reader may wish to continue our experiments on his own, e.g. by inserting the element $e6$ or deleting the element $e2$ after insertion of the element $e5$ was done. It is wortwhile to observe that the snapshot after execution of $delete(e2)$ will show the picture which is essentially the same as after insertion $e1, e2, e3, e4$ with following difference: Instead of $e2$ we find $e5$, and the pair of objects $\langle$ $e2$, its companion node object $\rangle$ is an isolated (not connected) part of the graf. During the experiments we neglected the ordering relation between elements.

Our first impression, when looking at the drawings of Figure 2, is a complete chaos. Slowly we commence to distinguish some parts and we begin to perceive some regularities. First, we remark that in each of 6 pictures there is exactly one object of type `PQ` which has two fields: *root*, *last*. Next, we remark that objects of type `Elem` and of type `Node` come in pairs, each pair is connected by *.el* and *.lab* arrows. Our next observation is that the arrows *.up* form lists of objects of type `Node` (no cycles). The meaning of arrows *.left* and *.right* is less evident that it may be suggested by their names. On the diagram some of them are solid and some of them are dotted. This comes as the result of analysis, the arrows *.left* and *.right* pay two roles. We see

that the solid arrows $.left$ and $.right$ go against to the arrows $.up$. While the observation that there is no cycle in paths composed of $.up$ arrows may lead to the statement that on each diagram we have a tree of node objects, the present observation states that the tree is a binary one.
Our observations need to be properly formulated and proved. This will be done in the next section. Before that, one may execute further experiments, e.g. by executing the command $delete(e2)$.

3.5. Observations and Lemmas. We shall study the class `PQS`, see the Appendix. We can assume that a usage of `PQS` consists in a finite sequence of creation of `newElem()` objects and calls: `call q.insert(e), call q.delete(e')`.
Following the intuition gained from Figure 2 we shall introduce the notion of observable states. The initial state s_0 is the graph consisting of exactly one object o of type `PQ`, $s_0 = \{new$ `PQ`$\}$ and no edges.

Definicja 22.2. (of the observable states) The set $\mathcal{S}$ of observable states is the least set which contains the initial state s_0 and which is closed with respect to the operations insert and delete and creation of `new Elem()` objects.

Each state consists of a set of objects and the edges connecting them. The examples of states are presented in Figure 2. The class `PQS` may be viewed as a definition of the relational structure PQS. The universe U of the structure consists of the objects of the inner classes of the class `PQS`. The attributes of objects of U define functions between objects. The set of objects of type `Node` will be denoted Node. analogously for the set of objects of type `Elem` and of type `PQ`:

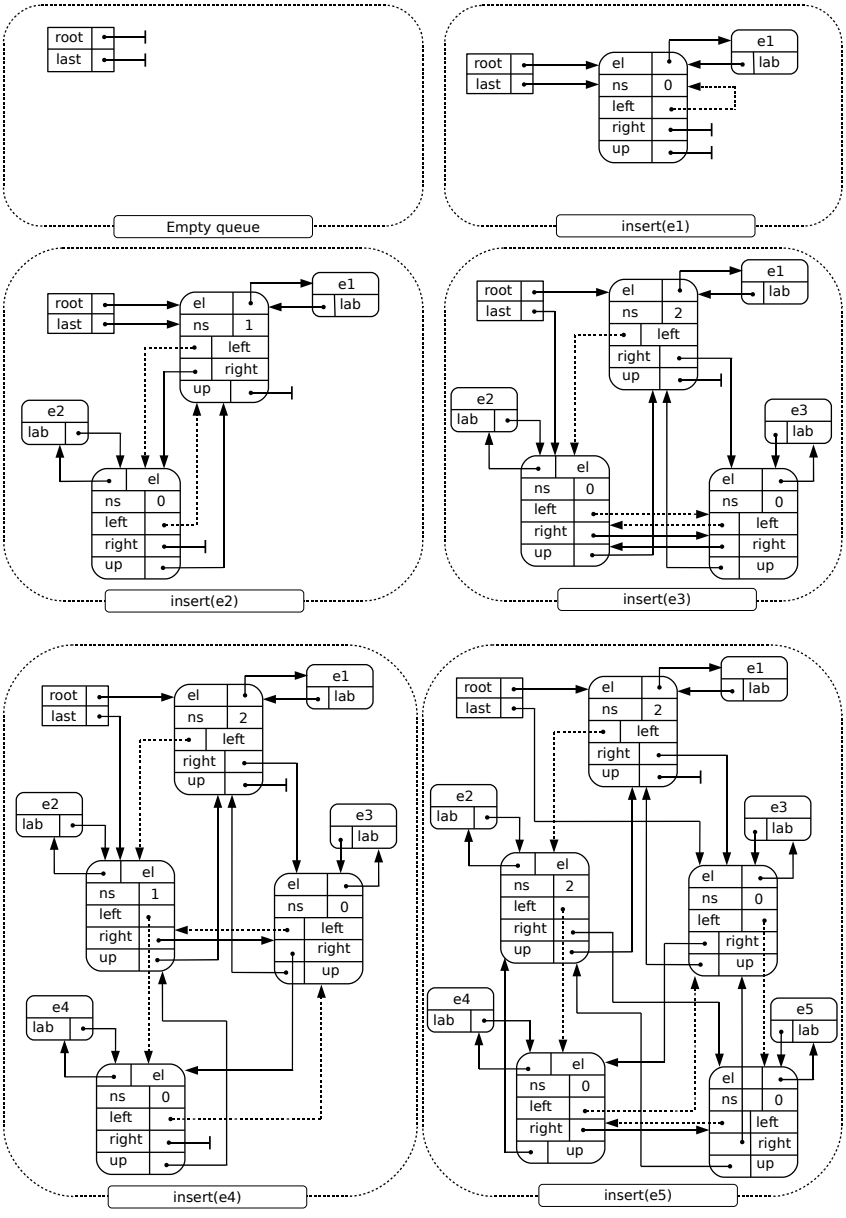
$$\begin{aligned} \text{Node} &= \{n:\ n \text{ instanceof } \texttt{Node}\},\\ \text{Elem} &= \{e:\ e \text{ instanceof } \texttt{Elem}\},\\ \text{PQ} &= \{q:\ q \text{ instanceof } \texttt{PQ}\}. \end{aligned}$$

Definicja 22.3. The class `PQS` determines an algebraic structure

$$\text{PQS} = \langle U, .up, .left, .right, .el, .lab. \rangle,$$

where U is a subset of the union Node$\cup$Elem$\cup$PQ which satisfies the condition

$$(\forall_{n\in \text{Node}})\ (\forall_{e\in \text{Elem}})\ n.el = e \Leftrightarrow e.lab = n.$$



Rysunek 2. Wstawianie elementów e1 - e5

The functions of the structure PQS are defined as follows

$$\begin{aligned}
.up &\stackrel{df}{=} \{\langle n, n'\rangle : \ n, n' \in \textbf{Node}, \ n' = n.up\},\\
.left &\stackrel{df}{=} \{\langle n, n'\rangle : \ n, n' \in \textbf{Node}, \ n' = n.left\},\\
.right &\stackrel{df}{=} \{\langle n, n'\rangle : \ n, n' \in \textbf{Node}, \ n' = n.right\},\\
.el &\stackrel{df}{=} \{\langle n, e\rangle : \ n \in \textbf{Node}, e \in \textbf{Elem}, \ e = n.el\},\\
.lab &\stackrel{df}{=} \{\langle e, n\rangle : \ e \in \textbf{Elem}, n \in \textbf{Node}, \ n = e.lab\}.
\end{aligned}$$

Our first observation is that if we abstract from (i.e. we forget about) the arrows *.left* and *.right* then for each state s its graph shows a tree.

Definicja 22.4. Let s be an observable state, consider the objects of type `Node` in this state. Let T_s be the set of these object of type `Node` that access the object *.root* object by a path composed from *.up* arrows only.

$T_s = \{o \in \textbf{Node} \cap s : \textit{there is a path composed from}$
.up arrows only, leading from object o to object .root$\}$

Lemat 22.7. In any observable state s the pair $\langle T_s, .up\rangle$ is a tree.

Definicja 22.5. (of a son) We say that a node n is a <u>son</u> of a node f in the tree T_s if and only if $n.up = f$. A node n is said to be a <u>leaf</u> of the tree T_s if and only if it has no sons.

Our next observation is

Lemat 22.8. For every $o \in T_s$, $o.ns$ = number of sons of o.

Definicja 22.6. We say that an arrow *.left* from the node n is <u>solid</u> iff $n.ns > 0$. We say an arrow *.right* from the node n is <u>solid</u> iff $n.ns = 2$. Otherwise the arrows are said <u>weak</u>, or dotted.

Next, we observe that solid arrows lead against *.up* arrows.

Lemat 22.9. For every state s, the tree T_s with solid arrows only, forms a binary tree.

Proof: For every two nodes n and f of the tree T_s the following properties hold:

- if a solid .left arrow leads from node f to node n then
  $n.up = f$
  $(f.left = n \wedge f.ns > 0 \ \ ) \Rightarrow n.up = f$,
- if a solid .right arrow leads from node f to node n then
  $n.up = f$
  $(f.right = n \wedge f.ns = 2 \ \ \wedge) \Rightarrow n.up = f$,
- $n.up = f \Leftrightarrow (f.left = n \vee f.right = n)$.

Hence T_s is a <u>binary</u> tree. Our next observation can be stated as follows:

Lemat 22.10. There exists at most one node n in T_s such that $n.ns = 1$. If it is the case then $last = n$.

Now, we observe that the leaves of tree T_s are on two levels only.

Lemat 22.11. For every state s, there exists a natural number $k(T_s)$ such that every leaf of the tree T_s is on the level $k(T_s)$ or $k(T_s) - 1$.

The number $k(T_s)$ is equal the length of the path composed from $.up$ arrows leading from the object $last.left$ to the $root$ object. It is equal 0 if $root = none$.

Now we try to guess the rôle of non-solid arrows $.left$ and $.right$. The following property holds:

Lemat 22.12. The object referenced by the variable $last$ in the tree T_s is the leftmost node on the level $k(T_s) - 1$ which has less than two sons.

Let us return to the Figure 1 and observe the following facts:

Uwaga 22.13. A) If a node n has two sons then its left brother has also two sons.
B) If a node n has one son then it is its left son.
C) If a node n has one son then its brother from the left has two sons and its brother from the right is a leaf.

Lemat 22.14. The value of the variable $last$ is a head of a list of leaves linked together via $.right$ (weak) arrows.

Lemat 22.15. The value of the variable $last$ is a head of a cyclic list of leaves linked by (weak) $.left$ arrow.

We see that all leaves on the level $k(T_s)$ are grouped to the left.

Lemat 22.16. Tree T_s is a perfect binary tree i.e. all the levels are completely filled with an eventual exception on the deepest level, in this case all the leaves are grouped to the left.

The following four lemmas have similar form $(\alpha \Rightarrow I\beta)$, where I is either instruction $insert(e)$ or instruction $delete(e)$, α is a precondition and β is a postcondition of the instruction I.

Lemat 22.17. Let $I : insert(e)$ and
$\alpha_1 : \{last = o \wedge o.left = n1 \wedge o.right = k \wedge o.ns = 0 \wedge e.lab = n \wedge n.el = e\}$,
$\beta_1 : \{last = o \wedge o.ns = 1 \wedge o.left = n \wedge o.right = k \wedge n.up = o \wedge n.left = n1\}$.
The instruction I is correct with respect to the precondition α_1 and postcondition β_1 in the structure PQS

$$\text{PQS} \models (\alpha_1 \Rightarrow I\beta_1).$$

Dowód. How to read this lemma? It states that in any state s, if the precondition α_1 is satisfied by s, then the execution of

instruction $insert(e)$ will succesfully lead to certain state s' and the postcondition β_1 will be satisfied by s'. What is the meaning of the precondition α_1 of the instruction insert? We can draw it, see Fig. 3.

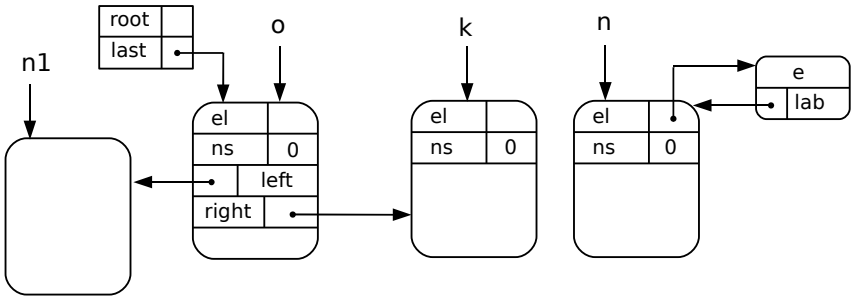


Fig. 3 Warunek wstępny α_1.

We check that the configuration of objects drawn on Fig. 3 satisfies the precondition α_1, The equality $last = o$ is satisfied since both variables point to the same object. The variable $last.left$ points to the object pointed by $n1$. $last.right$ point to the object pointed by k. The objects e and n are linked together, $e.lab = n$ and $n.el = e$.
Next, we can follow step by step the execution of the command $q.insert(e)$ with the text of method $insert$ in hand. We start observing that the precondition α implies $last.ns = 0$ hence the instructions executed by $insert$ are:
x:=e.lab; last.ns := 1; z := last.left; last.left := x; x.up := last; x.left := z; z.right := x;last:=z;
Now we can draw modifications to the picture following the instructions.

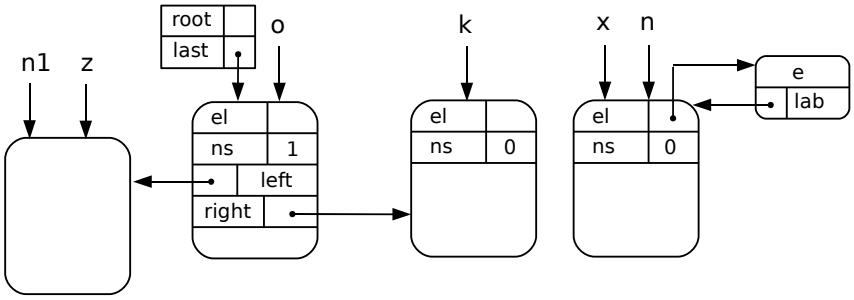


Fig. 4 Migawka po wykonaniu 3 instrukcji z treści instrukcji insert

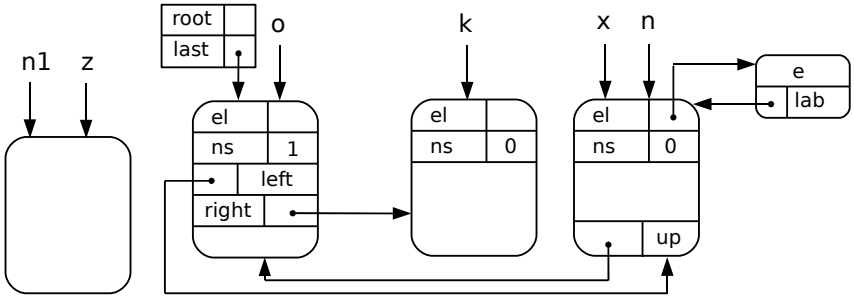


Fig. 5 Migawka po wykonaniu 5 instrukcji z treści instrukcji insert

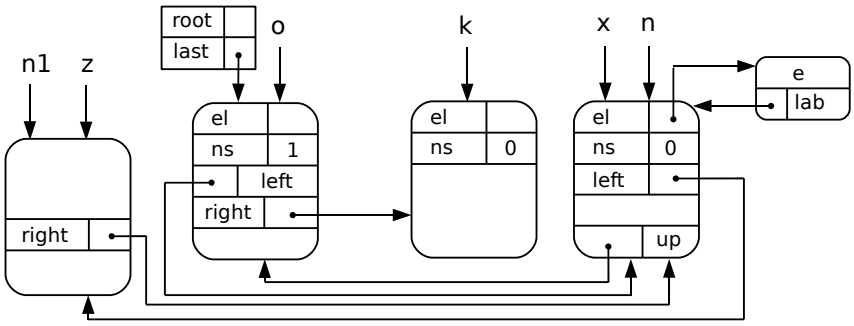


Fig. 6 Migawka po zakończeniu instrukcji insert .

The state shown on Fig. 6 can be described by the following formula $\gamma : \{last = o \wedge o.ns = 1 \wedge o.left = n = n1.right = e.lab \wedge o.right = k \wedge n.up = o \wedge n.left = n1 \wedge n.el = e\}$. It is easy to observe that the formula γ implies the formula β_1. Note that if the initial state s satisfies the precondition α_1 then at the end of the execution of the instruction $insert(e)$ in PQS we obtain the state s' satisfying the postcondition β_1.

Lemat 22.18. Let $I : insert(e)$ and
α_2:{last=o∧o.left=n1∧o.right=n2∧o.ns=1∧e.lab=n},
β_2: {last=n2∧o.ns=2∧o.left=n1∧o.right=n∧n.up=o∧n.left=n1 ∧n1.right=n}.
The instruction $I : insert(e)$ is correct with respect to the precondition α_2 and the postcondition β_2.

$$\text{PQS} \models (\alpha_2 \Rightarrow I\beta_2).$$

Dowód. One can easily verify that if $last.ns = 1$ then the actions executed by the method $insert$ are those presented in Table 2 below.

Table 2. Proof of lemma 3.

Precondition: α_2:(last=o∧o.left=n1∧o.right=n2∧o.ns=1 ∧e.lab = n)	
Instruction	*Effect*
x := e.lab	x=n, since precondition says e.lab=n
last.ns:=2	o.ns=2, since last=o
z := last.right	z=n2, because last.right=n2
last.right:=x	o.right=n, because last=o and x=n
x.right := z	n.right=n2, because x=n
x.up:= last	n.up=o, because last=o and x=n
z.left:=x	n2.left=n, because z=n2
last.left.right:=x	n1.right=n, because last.left=n1
x.left:=last.left	n.left =n1, because last.left=n1 and x=n
last :=z	last=n2, because z=n2
Postcondition: β':(o.ns=2∧o.right=n∧n.right=n2∧n.up=o∧ n2.left=n∧n1.right=n∧ n.left=n1∧last=n2∧o.left=n1∧e.lab=n)	

The postcondition β' collects the facts enlisted in the column Effect extended by the formulas e.lab=n and o.left=n1 for they remain satisfied after execution of *insert*. In this way we obtained a postcondition which is even stronger than the formula β_2.

The proofs of lemmas 22.17 and 22.18 exemplify two different ways of argumenting that a semantical property is valid. The first one, informal, consists in drawing the pictures. It may be related to drawing Venn's diagrams in the algebra of sets. Like diagrams of Venn it does not replace the proving but it is helpful. The second one is nearer to the goal of mechanization of proving.
In the following two lemmas we analyze properties of algorithm *delete*.

Lemat 22.19. Let the formulas α_3, β_3, and the instruction D be defined as follows:
$\alpha_3 : \{last = o \wedge o.left = n1 \wedge o.right = n2 \wedge o.ns = 0 \wedge e.lab = n \wedge n1.left = n3\}$,
$\beta_3 : \{last = n1.up \wedge last.right = o \wedge last.ns = 1 \wedge o.left = last \wedge o.right = n2 \wedge n1.ns = 0 \wedge n1.up = n1.left = n1.right = none \wedge n3.right = none \wedge n1.el = e\}$,
$D : delete(e)$.
The instruction $delete(e)$ is correct with respect to conditions α_3 and β_3, i.e.

$$\text{PQS} \models (\alpha_3 \Rightarrow D\beta_3).$$

Now we consider another case of applying the instruction *delete*.

Lemat 22.20. Let the formulas α_4, β_4 be defined as follows:
$\alpha_4 : \{last = o \wedge o.left = n1 \wedge o.right = n2 \wedge o.ns = 1 \wedge e.lab = n \wedge n1.left = n3\}$,
$\beta_4 : \{last = o \wedge o.left = n3 \wedge o.ns = 0 \wedge o.right = n2 \wedge n1.el = e \wedge n1.up =$

$n1.left = n1.right = none \wedge n3.right = none\}$.
The instruction $delete(e)$ is correct with respect to conditions α_4 and β_4, i.e.

$$\text{PQS} \models (\alpha_4 \Rightarrow D\beta_4).$$

The instructions call correctUp() and call correctDown(), that end the execution of procedures insert and delete, serve to guarantee that for each path in the tree T of nodes the elements associated to the nodes of the path form a decreasing sequence with the minimum in the root.

Lemat 22.21. (on procedure correctUp)
Procedure instruction call correctUp(r) is correct w.r.t. the precondition γ_1 and the postcondition γ_2 given below
$\gamma_1 : r$ **in** $elem \wedge r.less(r.lab.up.el) \wedge last.left.el = r$
(The first condition r in $elem$ is checked by compiler.) The second condition says the newly added element is less or equal than the element associated with the father of r. The third condition says: the companion node n of the element r is pointed by the pointer $last.left$.
γ_2 :for every node n on the path beginning at the element r, the following condition holds $n.up.less(n) \vee n.up = none$.

Lemat 22.22. (on procedure correctDown)
Procedure correctDown is correct w.r.t. the precondition γ_3 and the postcondition γ_4 given below
$\gamma_3 : r$ **in** $elem \wedge \neg r.less(r.lab.up.el$
γ_4 :for every node n on the path beginning at the element r, the following condition holds $n.up.less(n) \vee n.up = none$.

Lemat 22.23. For every two nodes x, y in the tree T_s, $x.up = y \Rightarrow y.less(x)$.

Dowõd. This property is invariant with respect to the operations insert and delete. At the very beginning the tree T is empty and the property holds. Assume that the property holds for a certain tree T. Consider another tree T' which is the result of operation insert or delete. After the insertion of an element the procedure correctUp is called and the tree is going to be repaired to keep the property. The same remark may be repeated in the case when the tree T' is the result of the operation delete on tree T. □

This sequence of observations leads to the following:

Lemat 22.24. In each observable state s the tree T_s is a heap.

Before proving the corectness of the implementation we should extend the class `PQ` adding two methods *empty* and *member*.

Boolean empty() { return root=none}
Boolean member(Elem e, PQ q) { the body of this method is

given on the righthand side of the equivalence a8 of Table 1}.

Now we are ready to verify the implementation PQS of priority queues against the specification $ATPQ$ given in Table 1.
Let us start with the structure consisting of elements and heaps $\{T_s : s \in \mathcal{S}\}$ and operations insert and delete, min, and the relations empty and member. Consider the quotient structure $\mathcal{PQS}$ in which we identify the heaps that have the same sets of elements. We claim that it is a priority queues structure, a standard model of the theory of $ATPQ$. It suffices to verify that the axioms of Table 1 are formulas valid in the structure $\mathcal{PQS}$.
a2) The program while not q.empty() do q.delete(q.min()) done always terminates since each operation delete removes one element from a heap.
a3) This follows from the lemmas 22.17 and 22.18.
a4) This follows from the lemmas 22.19 and 22.20.
The verification of the remaining axioms of $ATPQ$ is left to the reader. We can conclude:

Twierdzenie 22.25. The structure $\mathcal{PQS}$ of elements and PQ objects implemented by the class PQS is a priority queue.

Remarks on cost:
The pessimistic cost of an operation *insert* or *delete* is $O(logn)$, where n is the number of elements in the priority queue. This property is very important in the application of priority queue as plan of experiment in the class Simulation that inherits (extends) the class PriorityQueue. Imagine, in an simulation experiment of a pandemia of influenza where the objects of SimProcess class count in hundreds of thousands and the number of EventNotice objects goes in milions, any implementation with the cost worse than $O(logn)$ would be impractical.

3.6. Final remarks. We have demonstrated the work on verification of a given class K against a specification S. Aswering the question is the class K a correct implementation of the specification S is a task completely different than proving correctness of an algorithm with respect to a given pre- and post-conditions. This paper shows that the formal counterpart of the task is asking whether a given class implements a set of axioms. In this context it is natural to conceive the class K as an algorithmic definition of some algebraic structure $\mathbb{A}$ and to study the question is the structure $\mathbb{A}$ a model of the specification S. We are stressing that high integrity programming requires many skills and a lot of invention. The analysis of this case study shows the wide repertoire of questions that may appear during the work on a software project. The incomplete list contains the following kinds of subgoals:

- specification of algorithms,
- specification of classes (this work is strongly related to the goal of specification of a data structure),

- construction of a method (i.e. procedure or function),
- construction of a class,
- verification of a conjecture given class K correctly implements some specification S.

In a future paper we shall demonstrate that, in a favorable circumstances, one can construct a class together with a proof of its correctness.

From previous sections we conclude that EOP (experiment, observe, prove) method can be successfully applied to software analysis. We do not claim it is a trivial task but it is at least realizable. As a matter of fact most programmers would agree that formal methods are generally better than sophisticated testing and simultaneously most of them are not using such methods at all. It seems that essential to the problem is lack of proper tools and experience i.e., tools which can integrate specification, implementation and verification tasks into single, consistent process. Experience can be gained only through everyday practice. Our goal is to develop integrated programming environment supporting every phase of software construction using formal methods. EOP idea presented in this paper is a part of bigger scheme called temporarily SpecVer programming. We assume that whole process of software production needs preparation of:

- specification documents: formal texts along with some math analysis (is it astonishing that some specifications are incorrect, or mor precisely they may be inconsistent or incomplete?),
- implementation code,
- verification reports: again formal texts with some proofs about implementation's quality.

As you can remark, EOP method could be used both for specification and verification tasks as soon as we can perform observations analogous to these made about Fig. 2. This again direct us towards programming tools. We need some object debugger showing program memory from object perspective, some formal support tools helping in logical theorems construction and perhaps checking if formulas are properly written and much, much more ... Some work has already been started, but a lot of it still should be done. For now we can present initial implementation of Eclipse plugin [?]bibSpecVerEl supporting creation of specification documents.

Ćwiczenia. Dla sformułowania kilku następnych zadań przyjmujemy, że A jest tablicą liczb całkowitych o n elementach.

Zastanów się nad następującymi zadaniami.

22.1. Sformułuj warunki niezbędne na to byśmy mogli fragment tablicy A złożony z kolejnych elementów $A[l], \ldots A[p], lower(A) \leq l \leq p \leq upper(A)$ traktować jako las kopców.

22.2. Załóżmy, że fragment $A[l], \ldots, A[u]$ tablicy A jest lasem kopców. Rozważmy fragment $A[l-1], A[l], \ldots, A[u]$. Co można o nim powiedzieć? Co trzeba poprawić w tym fragmencie tablicy?

22.3. Przyjmijmy to samo założenie co poprzednio. Rozpatrujemy teraz fragment $A[l], \ldots, A[u], A[u+1]$. Czy można o nim powiedzieć, że jest kopcem? Co trzeba poprawić?

22.4. Napisz kompletny program, który wczytuje zadany ciąg liczb i sortuje go.

22.5. Napisz program zawierający procedury heapsortB i heapsortC. Tworzy (dość dużą) tablicę liczb, np. wybierając liczby pseudolosowe i porządkuje ją dwukrotnie, raz przy pomocy heapsortB i drugi raz przy pomocy heapsortC. Porównaj czasy działania tych algorytmów.

22.6. Niech T będzie typem opisanym przez deklarację pewnej klasy T. Czy potrafisz zmienić procedurę heapsort tak by porządkowała tablice elementów typu T?

22.7. Porównaj czasy wykonania procedury heapsort opisanej powyżej (w Loglanie) i podobnej procedury zapisanej w języku C++ lub Java.Przeprowadź odpowiednie eksperymenty i stwórz funkcje kosztu dla obu wersji algorytmu. Czy są to funkcje tego samego rzędu? Objaśnij Twoją odpowiedź.

22.8. Spróbuj zapisać algorytm heapsort w języku C++ lub Java. W językach C++ i Java, konstruktor klasy rozszerzającej wywołuje konstruktor klasy rozszerzanej. Sprawdź czy efekt osiągnięty w ten sposób jest podobny do wariantu B czy C?

ROZDZIAŁ 23

Zbiory nieskończone

W tym miejscu warto zrobić parę uwag. Do tej pory rozważaliśmy rozmaite struktury danych pozwalające przechowywać zbiory skończone.

W programowaniu z klasami i dziedziczeniem pojawia się możliwość definiowania klas jako zbiorów nieskończonych i operacji na obiektach.

Warto podjąć tę tematykę.

Możemy opisać klasę definiującą ciało liczb wymiernych.

Dla podkreślenia ... zrobimy to krok po kroku.

Dany jest typ pierwotny integer.

Produkt kartezjański

```
unit Para: class(L,M: integer);
end Para;
```

Podzbiór wcześniej zdefiniowanego zbioru. W tym przypadku wymagane jest spełnienie warunku $M \neq 0$.

```
unit Ulamek: Para class;
begin
if M=0 then raise ErrorMianownika fi
end Ulamek;
```

Wybieramy reprezentanta klasy równoważności.

```
unit UlamekWlasciwy: Poprawna class;
   var c: integer;
begin
   c:= GCD(L,M);
   L := L div c; M :+ M div c
end UlamekWlasciwy;
```

Określamy działania na elementach zbioru – tu definiujemy ciało ułamków.

```
unit Fraction: UlamekWlasciwy class;
   unit add: function()
   unit mult: function(u: Fraction): Fraction;
end Fraction;
```

Część 4

Programowanie z agentami

ROZDZIAŁ 24

$\mathcal{L}_9$ Współprogramy

1. Moduły i obiekty współprogramów

W tym rozdziale omówimy moduły współprogramów i współpracę z obiektami współprogramów.
Współprogram (ang. coroutine) jest modułem programu podobnym do klasy. Różnice można sprowadzić do krótkiego zdania: słowo class zastąp słowem coroutine i zezwól programiście stosować polecenie attach(c). Podczas obliczenia mogą pojawiać się obiekty współprogramów. Po utworzeniu obiekt taki jest pasywny tak jak obiekty klas. Możliwe jest jednak wznowienie wykonywania instrukcji współprogramu. Niech c będzie zmienną wskazująca na obiekt jakiegoś współprogramu. Wykonanie instrukcji attach(c) powoduje zawieszenie wykonywanie bieżących instrukcji (w Aktywnym obiekcie współprogramu) i wznowienie wykonywania instrukcji w obiekcie c. Ważne jest, by pamiętać, że instrukcja attach(c) może pojawiać się nie tylko w treści współprogramu, ale i w modułach funkcji i procedur zadeklarowanych we współprogramie. Wynika z tego następująca zasada: instrukcje występujące w treści współprogramu dzielą się na dwie części – te wykonywane aż do polecenia return włącznie z nim, to tzw. konstruktor, pozostałe instrukcje do wykonywanie których można powrócić gdy inny obiekt współprogram wyda polecenie attach(c) to są instrukcje wątku współprogramu, niektórzy mówią instrukcje włókna współprogramu.

Składnia. Rozszerzamy język wprowadzając deklaracje współprogramów, wyrażenia obiektowe typu współprogram i instrukcje atomowe właściwe dla obiektów współprogramów.
Moduły współprogramów.

Definicja 24.1. Współprogram jest to moduł o następującej strukturze

```
unit M: coroutine(args);
   𝔻
begin
   𝕀₁ (* instrukcje konstruktora *)
   return;
   𝕀₂ (* włókno lub wątek współprogramu *)
end M
```

gdzie M jest nazwą współprogramu,
args – listą parametrów formalnych,
$\mathbb{D}$ – listą deklaracji lokalnych,
$\mathbb{I}_1$, $\mathbb{I}_1$ – są ciągami instrukcji.

□

Moduł współprogramu może dziedziczyć klasę, w ten sam sposób jak to opisaliśmy w rozdziale o dziedziczeniu.

Wyrażenia obiektowe typu współprogram. Wartością wyrażenia może być obiekt współprogramu.

Definicja 24.2. Wyrażeniami obiektowymi typu współprogram są :

- odpowiednio zadeklarowana zmienna,
- wyrażenie new(), generujące obiekt współprogramu,

- nazewnik funkcyjny zwracający obiekt współprogramu.

□

Instrukcje atomowe.

Definicja 24.3. Instrukcja wznawiania współprogramu ma postać następującą

attach(x) .

Zmienne typu współprogram deklarujemy w ten sam sposób co zmienne typu klasa.
Np. **var** x,y: MyCoroutine, u,v: MyClass;
Jeśli chcesz to możesz tworzyć tablice współprogramów, zob. ??

□

Semantyka. Semantyka obiektów współprogramów ma dwie strony:
bierną – gdy obiekt współprogramu jest przedmiotem działań oraz
czynną gdy obiekt współprogramu wykonuje swoje kolejne instrukcje (gdy sterowanie zostało przekazane do takiego obiektu).
Zechciej porównać – sterowanie nie może zostać przekazane do obiektu klasy. Z drugiej strony instrukcja procedury wytwarza anonimowy rekord aktywacji procedury i przekazuje sterowanie do tego rekordu. Obiekt współprogramu może posiadać nazwę, tak jak obiekt klasy, ponadto inny współprogram może mu przekazać sterowanie. Do tego służy instrukcja attach(c). Program główny jest współprogramem. Program główny może wytworzyć obiekty współprogramów i może przekazać sterowanie do jednego z nich. Zobacz podrozdział 2 Producent i konsument, poniżej.

Wartością wyrażenia obiektowego, np. new M(1,2), jest nowy obiekt współprogramu. Tworzy się go w ten sam sposób jak obiekt klasy. Każda operacja wykonywana na obiekcie klasy może też być wykonana na obiekcie współprogramu.

Ponadto Pasywny obiekt *c* współprogramu może przejść do stanu Aktywny i wznowić wykonywanie swoich instrukcji.
Niech y będzie aktywnym obiektem, wykonanie instrukcji attach(c) polega na zapamiętaniu gdzie należy wznowić działanie gdy (inny) obiekt współprogramu wykona polecenie attach(y) (kiedyś, w przyszłości) i na wznowieniu wykonywania instrukcji współprogramu c, w miejscu w którym zawiesił on wykonywanie instrukcji. Za pierwszy razem polecenie attach(c) wznowi instrukcję następująca po poleceniu return. Każde kolejne polecenie attach(c) wznowi wykonywanie instrukcji od instrukcji następującej po ostatnio wykonanym poleceniu attach(x) w treści obiektu c. Obrazowo, wznawiamy działanie w miejscu w którym przerwaliśmy.

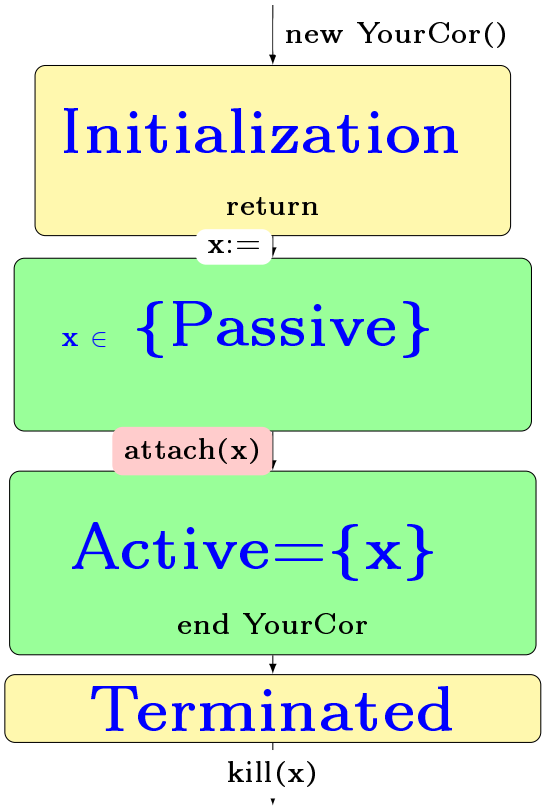
Poniżej przedstawiamy diagram stanów obiektu współprogramu Z rysunku 1 widać, że po wyczerpaniu wszystkich instrukcji do wykonania w obiekcie współprogramu, przechodzi on w stan Terminated. Polecenie attach(c) jest w takiej sytuacji błędem i zostanie to zgłoszone w trakcie wykonywania programu wraz z diagnostyka błędu.

Do wyjaśnienia: gdzie zostaną wznowione działania gdy obiekt współprogramu zakończył działanie?

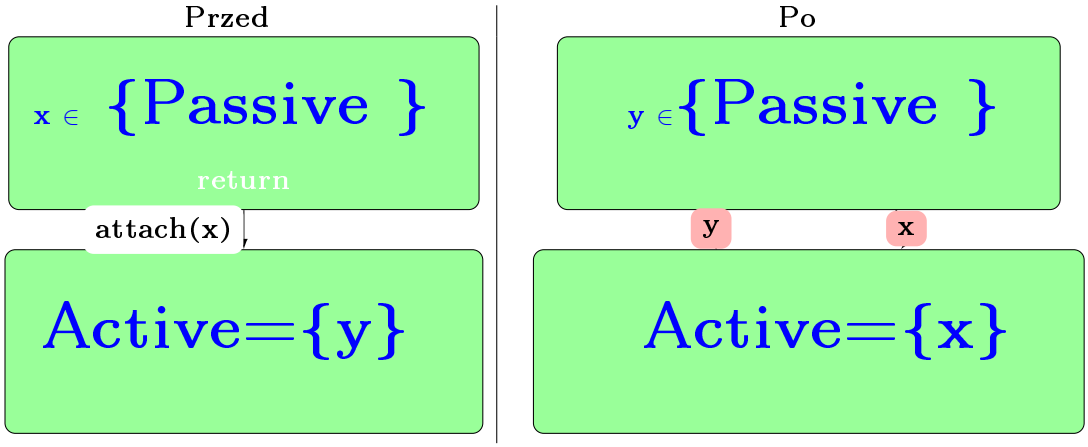
Na rysunku 2 widać, stany systemu współprogramów przed i po wykonaniu instrukcji attach(x). Zbiory stanów Pasywny i Aktywny zamieniły się obiektami. Obiekt aktywny y przeszedł w stan Pasywny a obiekt x jest teraz Aktywny. Instrukcja ta jest wykonywana przez obiekt aktywny. Zwracamy uwagę na to, że instrukcja attach(x) powodując uaktywnienie obiektu x równocześnie przerywa wykonywanie ciągu instrukcji w aktywnym dotąd obiekcie y. To czego na rysunku nie widać, to fakt, że obiekt x wznawia obliczenia w tym miejscu w którym zostały one przerwane. Zobacz Przykład ...

Porównaj z diagramem stanów obiektu klasy. Łatwo zauważyć, że obiekt współprogramu może znaleźć się w nowym stanie Aktywny.

Pojęcie współprogramu (ang. coroutine) pojawiło się w latach 60 XX wieku. Wielu do dziś przyjmuje jako definicję zdanie sformułowane przez D. Knutha w [Knu77], subroutines are special case of coroutines.



Rysunek 1. Diagram stanów obiektu współprogramu



Rysunek 2. Zmiana stanów systemu współprogramów

Trzeba jednak pamiętać o kontekście w jakim zdanie to zostało umieszczone. Dla Donalda Knutha środowiskiem w którym należy pisać programy jest assembler (wtedy MIX, dziś MMIX). W assemblerze pojęcie podprogramu ma inny sens niż pojęcie procedury w językach pochodzących od Algolu60. Podprogram jest fragmentem programu do którego możemy wykonywać skoki ze śladem. Podprogram ma jedno wejście tj. początek i w zasadzie jedno wyjście. Po ponownym wejściu do podprogramu nie wiemy nic o wartościach zmiennych lokalnych, jakie zostały obliczone poprzednio. Podprogram nie ma stanu jaki by mógł przetrwać od jednego do ponownego uruchomienia tego podprogramu. Korutyna (coroutine) pozwala wznowić obliczenia w miejscu z którego ją opuszczono. W języku programowania obiektowego współprogram jest specjalnym rodzajem klasy. Umożliwia to tworzenie wielu obiektów danego współprogramu C tak jak się tworzy obiekty klas. Po utworzeniu przez polecenie new obiekt taki pozostaje pasywny i można z nim postępować jak z obiektem klasy.
Możemy taki obiekt pasywny uaktywnić.

Współprogramy (ang. coroutines) - Pojęcie współprogramu ma dwie odmienne definicje(!).
Obie definicje zgodnie stwierdzają, że współprogram cechuje się posiadaniem ciągu instrukcji do wykonania i ponadto możliwością zawieszania wykonywania jednego współprogramu A i przenoszenia wykonywania do innego współprogramu B. W szczególności można wznowić pracę zawieszonego współprogramu A, a wykonywanie będzie podjęte w miejscu, w którym zostało zawieszone. Tym co różni obie definicje jest zdolność współpracy z rekurencyjnymi procedurami. (Nb. W językach programowania funkcyjnego koncepcja współprogramu istnieje pod postacią kontynuacji - pojęcia wprowadzonego niemal równocześnie z współprogramami.)

Obiekt współprogramu jest quasi-wątkiem. Tak jak wątek, obiekt współprogramu ma ciąg instrukcji do wykonania. W odróżnieniu od wątków obiekty współprogramów nie działają równolegle. Jest niezmiennikiem systemu współprogramów to, że w każdej chwili, dokładnie jeden obiekt współprogramu wykonuje swoje instrukcje:

$$card\{coroutine : \ coroutine \ is \ Active \ \} = 1.$$

W literaturze znaleźć można termin włókno (ang. fiber) dla odróżnienia od wątku (ang. thread).

Współprogramy jako specjalny rodzaj klas	Współprogramy jako bogatszy rodzaj podprogramów
W językach programowania: Simula67 , Loglan 82, BETA można tworzyć moduły coroutine tj. współprogramów. Składnia współprogramu różni się od składni klasy tym, że zamiast słowa class piszemy coroutine, i co ważniejsze, wewnątrz takiego modułu wolno używać instrukcji atomowych attach oraz detach. Instrukcje takie mogą się też pojawiać wewnątrz metod zadeklarowanych w współprogramie. Stwarza to nowe i interesujące możliwości współpracy współprogramów i procedur(funkcji) rekurencyjnych.	Šubroutines are special cases of coroutines."$_{D.Knuth}$. W assemblerze od dawna występuje pojęcie podprogramu - nie należy go mylić z pojęciem procedury. Podprogram istnieje w kodzie programu i ma co najwyżej jedną instancję. Nie jest możliwe rekurencyjne wykonywanie podprogramów.
Przykład: Łączenie drzew BST **var** T : *arrayof Traverser*; **unit** *Traverser* : **coroutine**(n : *node*); **var** *kolejny* : *integer*; **unit** *traverse* : **procedure**(m : *node*); **begin** **if** $m \neq$ *none* **then** **call** *traverse*($m.left$); $kolejny := m.val$; **detach**; (∗ instrukcja detach wznawia współprogram, który ostatnio uaktywnil ten (this) obiekt Traverser wykonujac attach(.) ∗) **call** *traverse*($m.right$); **fi** **end** *traverse*; **begin** **return**; **call** *traverse*(n) **end** *Traverser*; (*Stworz drzewa BST, w liczbie np.k drzew.* *Dla kazdego drzewa d* *stworz* $T[i]$:= **new** *Traverser*(d) (*Kolejne uaktywnienie tego wspolprogramu* **attach**($T[i]$) *spowodujewykrycie kolejnego co do wielkosci elementu z drzewa d*)	Przykład **var** q := **new** *kolejka* **coroutine** *produkuj* **loop** **while** *q nie jest pelna* *stworz troche nowych przedmiotow* *wstaw przedmioty do q* **yield to** *konsumuj* **coroutine** *konsumuj* **loop** **while** *q jest niepusta* *zdejmuj troche przedmiotow z q* *skonsumuj te przedmioty* **yield to** *produkuj*
Uwagi. 1Mamy tu do czynienia z dynamicznym systemem współprogramów. Wyrażenia generujące postaci "'new"' Traverser(...) umożliwiają stworzenie wielu obiektów typu współprogram Traverser. 2. Obiekt współprogramu Traverser wywołuje metodę traverse(). Łań-	Uwagi. 1. Ten system współprogramów jest statyczny: zawiera dwa współprogramy. Deklaracja coroutine automatycznie tworzy obiekty typu produkuj i konsumuj. 2. W tym syste-

- dokładnie "jeden" współprogram wykonuje swoje instrukcje, tzn. jest aktywny,
- współprogram aktywny może przejść w stan pasywny wskazując przy tym na inny wątek, który ma być uaktywniony,
- współprogram x uaktywniony w efekcie wykonania instrukcji $attach(x)$ (w dotychczas aktywnym współprogramie y) kontynuuje wykonywanie instrukcji od odpowiedniego punktu wejścia, dokładniej: pierwsze uruchomienie instrukcji wątku współprogramu spowoduje wykonanie pierwszej instrukcji wątku, każda następna instrukcja $attach(x)$ wznawiająca wykonywanie wątku współprogramu x rozpoczyna wykonywanie instrukcji od punktu wejścia wyznaczonego przez ostatnio wykonaną w nim instrukcję $attach(...)$.

Zasada działania współprogramów. System współprogramów można nazwać systemem "quasi-współbieżnym". Nazwa ta jest uzasadniona dwojako: liczne przykłady programów współbieżnych np. producent-konsument, czytelnicy-pisarze itd. zapisane przy pomocy współprogramów okazuja się wystarczająco adekwatne do zastosowań. Inny argument wspierający użycie tej nazwy to fakt, że od bardzo dawna stosuje się współprogramy do symulacji systemów, np. w Simuli67, Loglanie'82 i in. Odpowiednia klasa Simulation dostarcza klasę wewnętrzną simproces - obiekty klas pochodnych od klasy simproces symulują rzeczywiste procesy np. pacjentów w systemie symulacji epidemii choroby, pojazdy w systemie symulacji ruchu w mieście itp.

Wielu autorów uważa, iż "współprogramy to podprogramy wykonywane w taki sposób, że sterowanie może zostać przekazywane pomiędzy nimi wielokrotnie, przy czym wywołanie danego współprogramu powoduje wykonywanie instrukcji od miejsca ostatniego przerwania wykonania (ostatniego punktu wyjścia), a nie od początku". Nie jest to całkiem ścisłe. Podprogramy (funkcja, metoda) tworzą rekordy aktywacji. Po opuszczeniu takiego rekordu jest on automatycznie usuwany i nie ma możliwości wznowienia go. Współprogramy wymagają więc wątków i są realizowane jako obiekty odpowiednich klas, a nie jako podprogramy czy procedury. Cytowany wyżej pogląd mocno zawęża koncepcję współprogramów. Co więcej, nie można zapominać, że instrukcjami wątku współprogramu mogą być instrukcje wywołania jego prywatnych metod(procedur). Metody te mogą zawierać instrukcje $attach(...)$ przekazujące sterowanie z jednego do innego współprogramu. Dokładniej, instrukcja attach przekazując [[sterowanie]] z jednego do drugiego współprogramu przenosi je z łańcucha dynamicznego jednego współprogramu do łańcucha dynamicznego innego współprogramu.

Łańcuch dynamiczny współprogramu zawiera obiekt i wątek współprogramu i ponadto, jeśli wykonano instrukcję procedury, to do łańcucha dynamicznego dołączony jest rekord aktywacji procedury. Zakończenie wykonywania instrukcji procedury(metody) powoduje skrôcenie łańcucha dynamicznego. Instrukcja $attach(x)$ wykonana w rekordzie aktywacji procedury powoduje przejście do punktu wejścia w łańcuchu dynamicznym współprogramu x. Punktem wejścia (powrotu) dla dotychczas aktywnego współprogramu jest instrukcja w rekordzie aktywacji procedury występująca bezpośrednio za instrukcją $attach(x)$. Widać stąd, że liczba punktów wejścia (powrotu) danego współprogramu może być zmienna w czasie i może nie być niczym ograniczona!

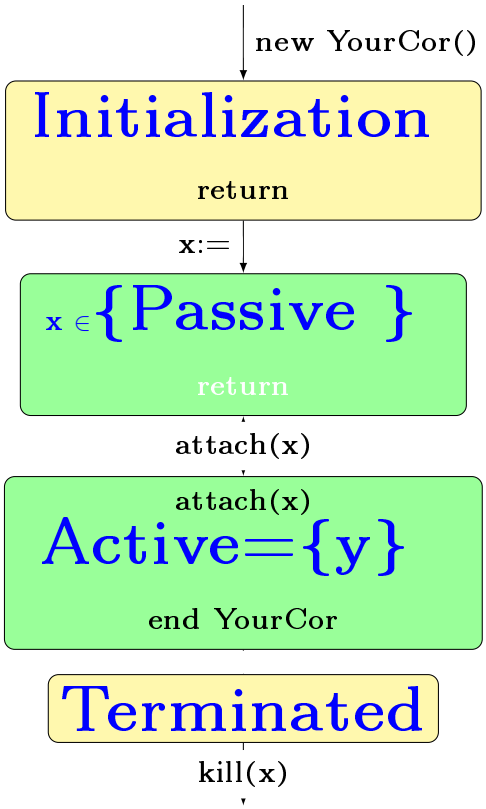
Podsumowując, współprogramy to więcej niż obiekty zwyczajnych klas, a mniej niż obiekty aktywne wątków (ang. threads).

Schemat zmian stanów obiektu współprogramu. Poniżej przedstawiamy diagram stanów obiektu współprogramu W poniższym zestawieniu widać stany systemu współprogramów przed i po wykonaniu instrukcji attach(x). Instrukcja ta jest wykonywana przez obiekt aktywny. Zwracamy uwagę na to, że instrukcja attach(x) powodując uaktywnienie obiektu x równocześnie przerywa wykonywanie ciągu instrukcji w aktywnym dotąd obiekcie y. To czego na rysunku nie widać, to fakt, że obiekt x wznawia obliczenia w tym miejscu w którym zostały one przerwane. Zobacz Przykład ...

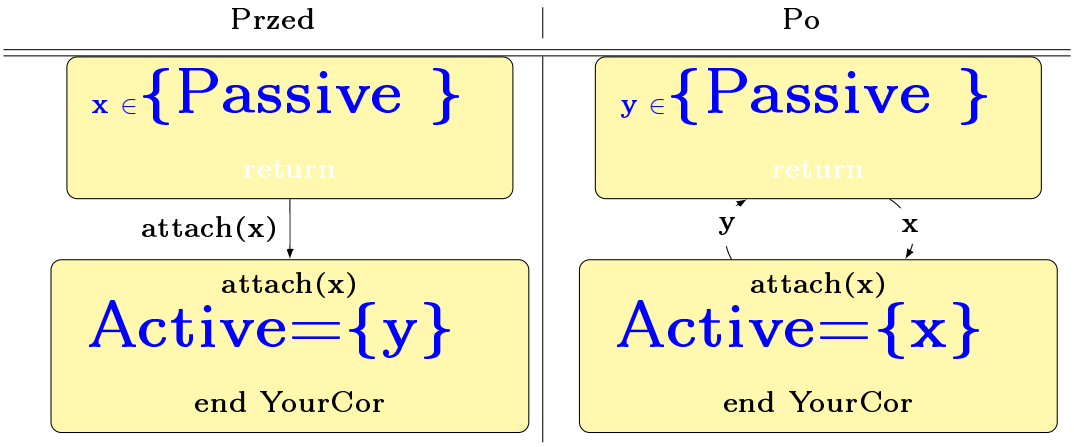
Współprogramy w języku Loglan '82.

- instrukcją przenoszenia sterowania z aktywnego współprogramu do drugiego współprogramu x jest $attach(x)$,
- moduł współprogramu jest specyficzną klasą (stosuje się słowo 'coroutine' zamiast 'class'),
- instrukcja $attach(x)$ może występować nie tylko w wątku współprogramu, lecz także w (prywatnych) metodach współprogramu(!),
- "punktami wejścia" do współprogramu są: pierwsza instrukcja wątku współprogramu oraz każda instrukcja następna po instrukcji $attach(x)$,
- można też używać bezparametrowej instrukcji $detach$, odpowiada instrukcji 'attach(ten współprogram, który ostatnio mnie wezwał)'.

Instrukcje przenoszenia sterowania między współprogramami w różnych językach programowania. Instrukcje przenoszące sterowanie z jednego do drugiego współprogramu to



Rysunek 3. Diagram stanów obiektu współprogramu



Rysunek 4. Zmiana stanów systemu współprogramów

- attach(x) oraz detach – w językach [[Simula 67]] i [[Loglan 82]],
- yield(x) – w nowszych językach, np. w [[Python]]ie,
- cede - w [[Perl]]u, w bibliotece Coro
- instrukcja skoku - w [[assembler]]ze oraz w [[Fortran]]ie, gdzie podprogram nie jest procedurą,
- trzeba odnotować też [[Implementacja (informatyka)|implementację]] współprogramów w [[Java|Javie]], jako wyspecjalizowanych wątków Javy.

Argument x wskazuje na współprogram pasywny w danej chwili.

Przykładowe zastosowania współprogramów.

- historycznie pierwsze współprogramy to skaner i analizator składniowy kompilatora [Con63],
- podobny schemat występuje w wielu sytuacjach np. jeden współprogram zbiera wyniki pomiarów i zapisuje je w bazie danych, a drugi współprogram opracowuje zebrane wyniki, ogólny schemat to producent-konsument,
- jeżeli jakaś metoda (funkcja lub procedura) jest wykonywana wielokrotnie z tymi samymi parametrami aktualnymi, to warto utworzyć odpowiednie obiekty współprogramu (tablicę współprogramów) dla każdego zestawu parametrów aktualnych. Następnie każdą instrukcję wywołania procedury zastępujemy odpowiednią instrukcją $attach(..)$. Zysk może okazać się znaczny, ponieważ wykonanie instrukcji $attach$ jest znacznie prostsze od tworzenia rekordu aktywacji procedury.
- Jeśli współprogramy są klasami wyposażonymi w instrukcję $attach(...)$ to można tworzyć hierarchie współprogramów wykorzystując dziedziczenie.
- główne zastosowanie współprogramów to narzędzia symulacji, takie jak klasy Simulation w Simuli 67 i w Loglanie'82.
- współprogramy umożliwiają tworzenie obiektów, które realizują obliczenia funkcjonałów, Rozważ: zamiast procedury bisekcja tworzymy współprogram z parametrem f: function(x; real):real.
  Polecenie
  f1:= new coroutine(f);
  tworzy obiekt współprogramu. Teraz polecenia:
  a:=77;b:= 98; attach(f1);
  a:=-18;b:=102; attach(f1);
  prowadzą do obliczenia - szukania miejsc zerowych .

2. Producent i konsument

Popatrzmy jak wygląda jeden z najczęściej omawianych przykładów. Program producent konsument ma następującą strukturę.

```
program PRODCONS;
  var prod:producer,cons:consumer,n:integer,mag:real,last:bool;

  unit producer: coroutine; ...
  end producer;

  unit consumer: coroutine(n:integer); ...
  end consumer;

begin (* MAIN program *)
  prod:=new producer;
  read(n);
  cons:=new consumer(n);
  attach(prod); (* R *)
  writeln;
end PRODCONS;
```

Obliczenie programu rozpoczyna się od utworzenia obiektu *prod* współprogramu producer. Następnie zostanie odczytana wartość n – będzie to rozmiar bufora. Z kolei program główny tworzy obiekt *cons* współprogramu consumer. Instrukcja *attach*(*prod*) przenosi procesor do obiektu *prod*. Po powrocie obliczeń do wykonywania programu głównego wykonamy instrukcję writeln, Miejsce powrotu dla programu głównego jest zaznaczone jako (* R *).
Dalsza analiza programu wymaga przeczytania treści współprogramu producer,

```
unit PRODUCER: coroutine;
begin
  return; (* 1 *)
  do
    I_P : { read(mag); (* markujemy algorytm produkcji – obliczenia mag *)
            (* mag is nonlocal variable, common store*)
            if mag=0
            then (* end of data *)
                 last:=true;
                 exit
            fi;
          }
    attach(cons); (* A *)
  od;
  attach(cons) (* B *)
end PRODUCER;
```

a także treści współprogramu consumer.

```
unit CONSUMER: coroutine(n:integer);
  var Buf:arrayof real; var i,j:integer;
begin
  newarray Buf dim(1:n);
  return; (* 2 *)
  do
    I_C : { for i:=1 to n
            do
              Buf(i):=mag;
              attach(prod); (* C *)
              if last then exit exit fi;
            od;
            for i:=1 to n
            do (* print Buf *)
              write(' ',Buf(i):10:2)
            od;
            writeln;
          }
  od;
  (* print the rest of Buf *)
  for j:=1 to i do write(' ',Buf(j):10:2) od;
  writeln;
  attach(main); (* D *)
end CONSUMER;
```

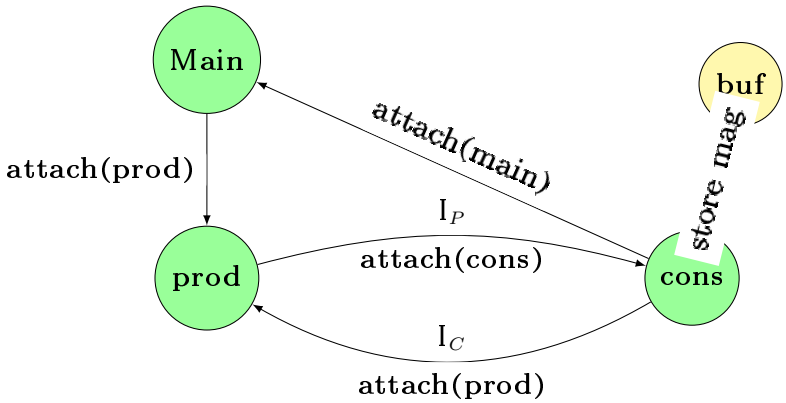
Podczas wykonywania tego programu powstaną trzy obiekty współprogramów. Na poniższym rysunku 5 są one przedstawione w kolorze zielonym. Strzałki koloru czarnego wskazują, że wartością zmiennej *prod* jest obiekt typu producer do którego prowadzi ta strzałka, a wartością zmiennej *cons* jest obiekt

typu consumer wskazany przez strzałkę zaczynająca się od cons. Natomiast czerwone strzałki pokazują efekt wykonania operacji `attach( )`. Wykonywanie programu rozpoczyna się w rekordzie aktywacji programu głownego. Pierwsza instrukcja spowoduje utworzenie obiektu współprogramu producer i przypisanie tego obiektu jako wartości zmiennej *prod*. Druga instrukcja powoduje wczytanie (z klawiatury) liczby i przypisanie jej do zmiennej n. Trzecia instrukcja spowoduje utworzenie obiektu współprogramu consumer i przypisanie go do zmiennej *cons*. W efekcie inicjalizacji (konstrukcji) obiektu *cons* powstaje tablica *Buf* o elementach numerowanych od 1 do n.
Czwarta instrukcja programu głównego przenosi procesor do obiektu *prod*, w miejsce w którym instrukcja return zakończyła inicjalizację tego obiektu. Proces produkcji przedmiotu i wstawienia do magazynu ilustrujemy przy pomocy instrukcji read(mag). W praktycznych zastosowaniach może to być np. ciąg instrukcji dokonujących pomiarów. Przyjęliśmy umowę, że wczytanie wartości mag=0 kończy pracę programu. Jeśli wczytana wartość mag $\neq$0 to wykonamy instrukcję attach(cons). Za pierwszym razem wznowimy działanie obiektu *cons* za instrukcja return. Proces konsumpcjijest tu zamarkowany przez wstawienie wczytanej wartości do bufora *Buf* (i wydrukowanie bufora co jakiś czas). poczym konsument *cons* przekazuje sterowanie do producenta *prod*, wykonując polecenie attach(prod). Producent powtara etap produkcji i ponownie przekazuje sterowanie konsumentowi. Zakładamy, że w pewnym momencie zostanie wczytana wartość mag=0. Wtedy producent ustawia wartośc zmiennej boolowskiej *last* true i przekazuje sterownie do konsumenta. Konsument w każdym cyklu sprawdzał czy wartość last jest true. (Z definicji początkowa wartośc last to false). Gdy jednak last jest true konsument drukuje częściowo wypełnioną tablicę *Buf* i przekazuje sterowanie do programu głównego wykonując polecenie attach(main). Przypominamy w tym miejscu, że program główny jest obiektem współprogramu.
Od tej pory obiekty *prod* i *cons* kooperują ze sobą, przekazując sobie sterowanie (lub jeśli wolisz przenosząc processor) za pomocą instrukcji `attach`. Uwaga. Zakończenie wykonywania programu nastapi gdy obiekt *prod* ustawi wartość zmiennej *last* true i przekaże sterowanie obiektowi *cons*. Z kolei obiekt *cons* wydrukuje tele liczb, ile znajduje sie w buforze i oddaje sterowanie programowi głównemu, dla zakończenia obliczeń.[1] Analiza tego programu prowadzi do następujących wniosków:

- Podczas wykonywania programu istnieja cztery jednostki dynamiczne: main – rekord aktywacji programu

[1]Zwracamy uwagę na fakt, że ani obiekt *prod*, ani obiekt *cons* nie wyczerpuja list swoich instrukcji do końca. Tzn. nie dochodzi do zmiany stanu obiektu *prod* na Terminated.

Rysunek 5. Trzy obiekty współprogramów: prod, cons i main



Rysunek 6. Diagram współpracy agentów prod, cons i main

głównego, obiekty współprogramów $prod \in$ producer, $cons \in$ consumer i tablica buf.

- Wprowadzenie czerwonych strzałek stworzyło wrażenie chaosu, a tak nie jest. Lepszy obraz współdziałania tych trzech obiektów współprogramów uzyskamy na rysunku 6 poniżej .

Pewnym przybliżeniem, sekwencyjnego przecież, programu prodcons może być wyrażenie regularne

$$I_{main}; (I_P; I_C)^+; R_C; R_{main}$$

w którym I_P oznacza ciąg instrukcji oznaczony, a_p instrukcję attach(prod), ...

Wniosek 24.1. Z powyższych obserwacji wynika, że program prodcons można zastąpić równoważnym programem iteracyjnym, wolnym od modułów współprogramów producer i consumer.

Jaki byłby koszt tego przekształcenia? Zapewne spory. Wydaje się, że taniej jest napisać program z wykorzystaniem współprogramów. Bierzemy tu również pod uwagę, że zastosowanie współprogramów zmniejsza ryzyko popełnia błędu w organizacji współpracy.

3. Instrukcja detach

W poprzednim przykładzie użyliśmy instrukcji attach do przełączania obliczeń pomiędzy obiektami współprogramów. W tej sekcji poznamy instrukcję detach.

Przykład.

Mamy do dyspozycji dwie drukarki, jedna o szerokości m1 i druga o szerokości m2. Program wczytuje ciąg bloków liczb. Każdy blok rozpoczyna się od nagłówka będącego liczbą m1 lub m2. Każdy blok kończy się liczbą 0. Koniec pliku wejściowego jest zgłaszany jako blok o nagłówku 0. Nasz program może wyglądać tak:

```
program READER_PRINTERS;
  const m1=10,m2=20;
  var reader:reading,printer_1,printer_2:writing;
  var n:integer,new_sequence:boolean,mag:real;
    unit writing:coroutine(n:integer); ...
    end writing;
    unit reading:coroutine ; ...
    end reading;
begin
  reader:=new reading;
  printer_1:=new writing(m1); printer_2:=new writing(m2);
  do
    read(n);
    case n
    when 0: exit
    when m1: attach(printer_1)
    when m2: attach(printer_2)
    otherwise write("wrong data"); exit
    esac
  od
end;
```

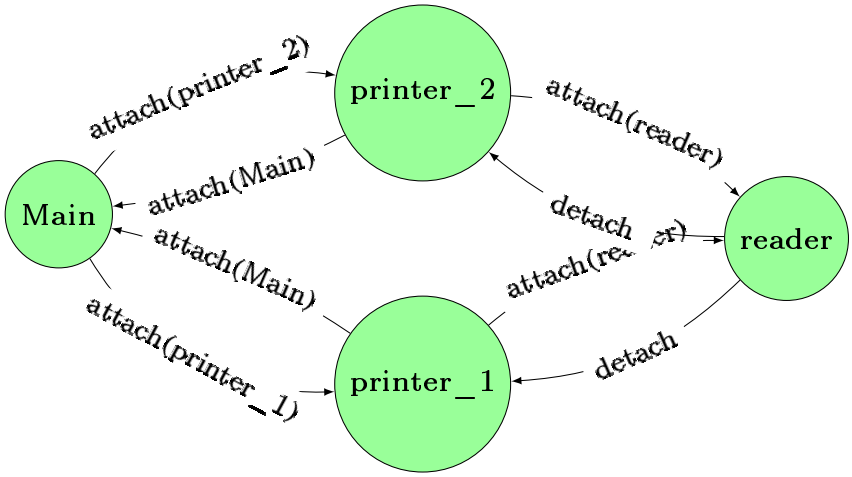
Współprogram reading ma następująca treść

```
unit READING: coroutine;
begin
  return;
  do
    read(mag);
    if mag=0 then new_sequence:=true; fi;
    detach;
    (* detach returns control to printer_1 or printer_2
     depending which one reactivated reader *)
  od
end READING;
```

W jaki sposób zapewnić, że ...?
Czy we współprogramach ...?
Współprogram writing wygląda tak:

```
unit WRITING: coroutine(n:integer);
  var Buf: arrayof real, i,j:integer;
begin
  array Buf dim (1:n); (* array generation *)
  return;
  do
    attach(reader); (* reactivates coroutine reader *)
    if new_sequence
    then
    (* a new sequence causes buffer Buf to be printed out *)
      for j:=1 to i do write(' ',Buf(j):10:2) od; writeln;
      i:=0; new_sequence:=false;
      attach(main)
    else
      i:=i+1; Buf(i):=mag;
      if i=n
      then
        for j:=1 to n do write(' ',Buf(j):10:2) od; writeln; i:=0;
      fi
    fi
  od
end WRITING;
```

Dlaczego w współprogramie reading występuje polecenie detach? Co by się działo gdybyśmy użyli polecenia attach?
W tym przykłdzie zamiast instrukcji detach mozna uzyć
if szer=m1 then attach(printer_1) else attach(printer_2) fi
No i trzeba jeszcze zadbac by wartośc szer była odpowiednio aktualizowana. Napisz odpowiednie polecenie(a).
A co zrobić gdy obiektów uaktywniających współprogram reader jest dużo? gdy liczba ta nie jest znana przed wykonaniem programu?



Rysunek 7. Diagram współpracy agentów printer_1, printer_2, reader i Main

Widac, że polecenie detach jest przydatne.

Poniższy rysunek 7 ilustruje związki pomiędzy czterema współprogramami: Main, printer_1, printer_2 i reader.

A może narysować podobny, inny diagram? Tak by było wyraźnie widać, że uaktywnieniu obiektu reader współprogramu Reading z obiektu printer_i odpowiada powrót do aktywności w tym samym obiekcie printer_i. W ten sposób dochodzimy do zrozumienia uwagi Donalda Knutha sprzed 40 lat: "podprogramy to szczególny przypadek współprogramów". Rzeczywiście, obiekt reader zachowuje się jak podprogram.
Niektórzy (Dahl, Kreczmar) mówia w takim przypadku o semicoroutinach.

4. Licznik - dynamiczna tablica obiektów współprogramu.

W tym paragrafie zorganizujemy współpracę systemu obracających się kółek, który w efekcie zachowuje się jak wiele liczników:np. licznik zuzytej energii elektrycznej, licznik przejechanych kilometrów, etc. Program licznik pokazuje jak wykorzystać współprogramy w większym programie: moduł A może zajmować się obserwacją "impulsów", zdarzeń powodujących, że moduł B wytwarza kolejną kombinację – tu przedmiotów 0,1,2, ... ,9 i przekazuje sterowanie do kolejnego
modułu C, który w jakiś sposób pożytkuje tę informację (np. o stanie licznika).

Częściej potrzebna nam będzie kolejna permutacja. Odpowiednio zmodyfikowany program PawelG zajmie się wytworzeniem wszystkich permutacji. zmodyfikuj program PawelG, zastąp instrukcję drukowania instrukcją detach i ...

5. Scalanie drzew BST – łańcuch dynamiczny

Ten paragraf poświęcony jest pojęciu łańcucha dynamicznego.
Zaczniemy od przykładu. Przykład zaczerpnięty z pracy bibojd:aw[?], por. [Sza91]. Zadanie.
Dane są drzewa BST w liczbie k. Należy utworzyć ciąg niemalejący elementów zapisanych w węzłach tych drzew.
Zadanie to można rozwiązać na wiele sposobów. Wykorzystanie współprogramów okaże się bardzo naturalnym podejściem do tego problemu.
Dane to k elementowa tablica D drzew BST (a dokładniej, tablica węzłów-korzeni tych drzew). Z każdym drzewem $D[i]$ związemy jeden współprogram $T[i]$.

Analiza zadania.
Drzewo BST może być opisane np. tak

```
unit Tree: class;
   var root: node;
   unit node: class(val: integer);
     left, right: node;
   node;
   traverse: procedure(v:node);
   begin
     if v=none then return
     else call traverse(v.left); write(v.val); call traverse(v.right);
     fi
   end traverse;
   unit insert: procedure(e:integer); ... end insert;
end Tree;
```

Zaobserwujmy

Lemat 24.2. Niech k będzie obiektem klasy Tree. Instrukcja call traverse(k) spowoduje wydrukowanie wszystkich liczb zapisanych w drzewie o korzeniu k.

A teraz zmienimy nasz program. W procedurze traverse instrukcję write zastąpimy poleceniem detach. Rozpatrzmy następujący program test.

```
program test;
  unit node: class(val:integer);
    var left,right: node;
  end node;
  var A: CA, B: CB, kolejny, integer, n: node, czyWszystkie:boolean;
unit CA: coroutine(n:                 | unit CB: coroutine;
node);                                | begin
  unit T: procedure(y:                |   return;
node);                                |    while not czyW-
  begin                               | szystkie do
   if y<>none then                    |    attach(A)
    call T(y.left);                   |    write(kolejny);
     kolejny := y.val;                |   od
detach;                               | end CB
    call T(y.right)
   fi
  end T;
begin
            czyWszyst-
kie:=false;
  return;
  call T(n);
            czyWszyst-
kie:=true;
end CA;
begin
  n:= new node;    (* tu instrukcje wypełniające drzewo n *)    A:= new C
  B:= new CB;
  attach(B)
end test
```

Lemat 24.3. Wykonanie powyższego programu spowoduje wydrukowanie wszystkich liczb zapisanych w drzewie w porządku rosnącym.

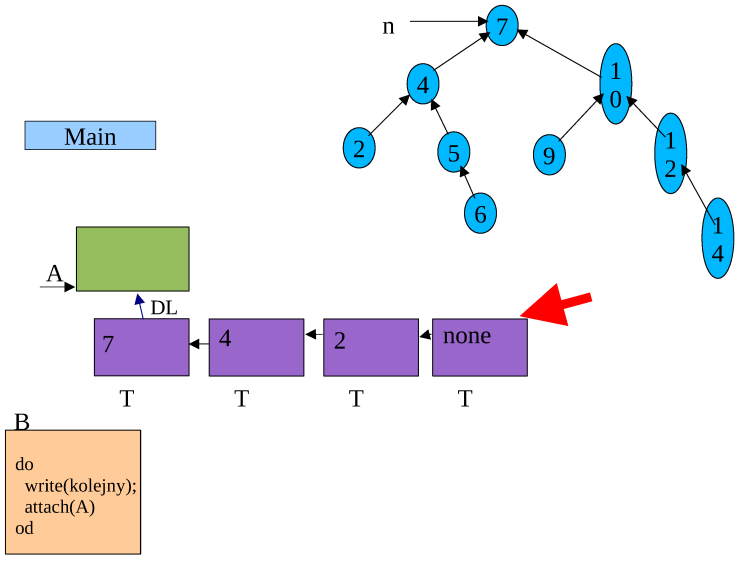
Dowôd. Dowôd wynika z poprzedniego lematu i z własności operacji detach oraz attach. Dowôd przez indukcję ze wględu na wysokość drzewa n. □

Jaki sens ma zastępowanie prostszego programu przez program bardziej skomplikowany?
Wyobraźmy sobie teraz, że mamy tyle współprogramów ile jest drzew. Program główny może "poprosić" dowolny z współprogramów by podał kolejną co do wielkośći wartosc przechowywana w drzewie . Oto szkic algorytmu scalania drzew BST.

1. niech każdy współprogram poda najmniejszy element w drzewie D[i].
2. powtarzaj ...
3. wybierz najmniejszy element e, zapamiętaj nr drewa w j,
4. dopóki choćiaż jedno drzewo ma jeszcze element do pokazania

Definicja tego pojęcia

Łańcuchem dynamicznym obiektu c pewnego współprogramu nazywamy ciąg jednostek dynamicznych taki, że (Baza) do łańcucha należy obiekt c, (krok indukcyjny) a) jeśli wykonywane jest polecenie tworzące nową jednostkę dynamiczną bloku, procedury, funkcji lub obiektu to wskaźnik systemowy DL nowej jednostki wskazuje na poprzednio aktywną jednostke dynamiczną. Jednostka ta jest w ten sposôb dołączana do łańcucha wydłużając go,
b) jeśli w ostatnim elemencie łańcucha zakończono wykonywanie ostatniej instrukcji (bloku, procedury, funkcji lub inicjalizacji klasy) to jednostka taka jest odłączana od łańcucha dynamicznego skracając go.



1

6. attach zastępuje call

W tym miejscu (po raz drugi) nawiązujemy do zdania wypowiedzianego przez Donalda Knutha. W strukturze drzew binarnych poszukiwań implementujemy trzy operacje:

m) sprawdzania czy element e nalezy do drzewa,
i) wstawianie elementu e do drzewa,
d) usuwanie eleentu e z drzewa.

Rozważmy program, który wielokrotnie wykonuje każdą z tych operacji. Powiedzmy, że operacja sprawdzania jest wykonywana k razy, operacja wstawianie l razy, operacja usuwania jest wykonywana m razy. W związku z tym program utworzy $k+l+m$ razy rekord aktywacji odpowiedniej procedury. Taki rekord istnieje tylko przez chwilę, potem, po zakończeniu procedury jest automatycznie usuwany. Kolejne wywołanie (instrukcja call) powoduje powstanie niemal identycznego rekordu aktywacji i ... jego usunięcie i tak wiele razy. Czy można tego uniknąć? Spróbujemy zastąpić instrukcje call member(e), call insert(e) i call delete(e) przez instrukcje attach aktywujące odpowiedni z trzech obiektów współprogramów. Deklarację klasy BST zawierającą funkcję member i procedury insert oraz delete zastąpimy deklaracją podobnej klasy BSTC w której zamiast procedur pojawią się współprogramy.

```
unit BSTC: class (type t; function less(x, y:t):boolean);
   var root: node, member: e, insert: i, delete: d;
   unit node: class (value: t);
      var l, r: node;
   end node;

   unit e: coroutine; ...

   unit help: e coroutine; ...

   unit i: help coroutine; ...

   unit d: help coroutine; ...

begin
   member:=new e; insert:=new i; delete:=new d;
   inner;
   kill(member); kill(insert); kill(delete)
end BSTC;
```

Zauważ, że współprogramy zadeklarowane w tej klasie tworzą hierarchię.

W programie głównym możemy zechcieć zadeklarować klasę MT i funkcję boolowska *mniejsze* porównującą obiekty klasy MT. Wtedy następująca instrukcja bloku dziedzicząca z klasy *BSTC* zastąpi instrukcję bloku dziedziczącą z klasy *BST*.

```
pref BSTC(MT, mniejsze) block
var y:MT;
...
begin

... (* call insert(y) *)
insert.x:=y;
attach(insert);

... (* elem:= contains(y) *)
member.x:=y;
attach(member);
if member.elem then ... fi;

... (* call delete(y) *)
delete.x:=y;
attach(delete);
...
end;
```

Obejrzyjmy deklaracje współprogramów. Współprogram e ma następującą treść

```
unit e: coroutine;
(*elem- output attribute*)
  var trick, elem: boolean, q, v: node, x:t;
begin
  return;
  do trick, elem:=false; (* loop for member *)
    q:=root;
    v:=none;
    while q=/=none
    do
      if less(x, q.value)
      then v:=q; q:=q.l
      else
        if less(q.value, x)
        then v:=q; q:=q.r
        else elem:=true; exit
        fi
      fi
    od;
    (* elem=true iff x belongs to S *)
    inner;
    detach;
  od
end e;
```

Pomocniczy współprogram *help* dziedziczy z e.

```
unit help: e coroutine;
begin
   inner; (* trick=true iff x does not belong to S *)
   if not trick then exit fi;
   if v=none
   then root:=q
   else
      if less(x, v.value)
      then v.l:=q
      else v.r:=q
      fi (* after insert or delete *)
   fi (* attach new node q to its father v *)
end help;
```

Współprogram *help* jest rozszerzany przez współprogramy *insert* i *delete*.

```
unit i: help coroutine;
begin
   trick:=true;
   if elem then exit fi;
   q:=new node(x) (* insert is a dummy if x belongs to S *)
end i;
```

Współprogram *d* jest trochę dłuższy.

```
unit d: help coroutine;
  private w, u, s;
  var w, u, s: node;
begin (* delete is a dummy if x does belong to S *)
  if not elem then exit fi;
  w:=q;
  if q.r=none
  then q:=q.l
  else
  if q.l=none
  then q:=q.r
  else u:=q.r;
  if u.l=none
  then u.l:=q.l; q:=u
  else
  do s:=u.l;
  if s.l=none then exit fi;
  u:=s
  od;
  s.l.:=w.l; u.l:=s.r;
  s.r:=w.r; q:=s
  fi
  fi
  fi;
  kill(w)
end d;
```

Załóżmy, że MT jest klasą i że funkcja *mniejsze* spełnia warunki wyliczone w definicji relacji porządkującej, tzn. jest zwrotna, przechodnia i antysymetryczna. Program (blok) P2 powstaje z programu P1 przez zastapinie każdej instrukcji call przez odpowiednią parę instrukcji wg następującej tabeli.

call insert(a);	≡	insert.x:=a; attach(insert);
bl:=member(b);	≡	member.x:=b; attach(member); bl:=member.elem;
call delete(c);	≡	delete.x:=c; attach(delete);

Twierdzenie 24.4. Przy spełnieniu powyższych założeń programy P1 i P2 są równoważne.

Dowód. Dowód przebiega przez indukcję ze względu na liczbę instrukcji call w programie P1. W dowodzie wykorzystamy następujące lematy.

□

7. Wieże Hanoi

Przyjrzyjmy się ponownie zadaniu przenoszenia krążków w buddyjskiej świątyni w Hanoi. W rozwiązaniu rekurencyjnym dostrzeżemy szansę na skrócenie czasu obliczeń. Podstawową operacją jest wywołanie procedury *move* czyli przenieś n krążków

z wieży o numerze f na wieżę o numerze t. Wiemy, że takie operacje trzeba wykonać 2^n razy. Tymczasem liczba różnych zestawów argumentów wynosi $n \times 3 \times 3$ i jest znacznie mniejsza od 2^n. Każde wykonanie instrukcji call move(a,b,c) wymaga utworzenia rekordu aktywacji dla procedury move i jego zamknięcia. Proponujemy by

(1) zastąpić procedurę $move$, współprogramem o nazwie wz. Zobacz tabelę poniżej.
(2) utworzyć $9n$ (a nawet tylko $6n$) obiektów tego współprogramu z wszystkimi dopuszczalnymi wartościami argumentów i zapisać je w trójwymiarowej tablicy P,

```
var P: arrayof arrayof arrayof wz;
```

(3) zastąpić każdą instrukcję call move(a, b, c) przez instrukcję attach(P(a,b,c))

Ad 1. Porównajmy procedurę $move$ i współprogram wz.

```
unit
move:procedure(n,f,t:integer);
(* n rings from stick f to
stick t *)
    var k:integer;
begin


   k:=6-(f+t);
   if n>1 then call move(n-
1,f,k); fi;
   call modyf(f,t);
     (* move only one ring
*)
   if n>1 then move(n-
1,k,t); fi;
   return

end move;
```

```
unit
wz:coroutine(n,f,t:integer);
(* n rings from stick f to
stick t *)
    var k:integer;
begin
   return;
   do
     k:=6-(f+t);
     if n>1 then attach
(p(n-1,f,k)); fi;
     call modyf(f,t);
     (* move only one ring
*)
     if n>1 then attach
(p(n-1,k,t)); fi;
      detach;
    od;
end wz;
```

Ad 3. Porównanie instrukcji call move(4, 1, 2) i instrukcji attach(P(4,1,2))

prowadzi do następującego wniosku.

każde wywołanie call move(a,b,c); spowoduje – utworzenie rekordu aktywacji move – przesłanie argumentów a, b, c – wykonanie treści procedury – usunięcie rekordu po zakończeniu	var x: wz; ... x:=new wz(a,b,c); ... każde polecenie attach(x) jest równoważne poleceniu call move(a,b,c).

Wniosek 24.5. Przyjmujemy, że dla każdego $1 \leq i \leq n$, dla każdego $1 \leq j \leq 3$ i dla każdego $1 \leq k \leq 3$ zachodzi $P[i,j,k] =$ new $wz(i,j,k)$.
Wtedy wykonanie instrukcji *call move(i,j,k)* daje dokładnie ten sam efekt, co wykonanie instrukcji *attach(P[i,j,k])*.
Natomiast instrukcja attach pozwala zaoszczędzić czas potrzebny na utworzenie rekordu aktywacji i na przekazanie parametrów.

Na żółto pomalowany jest rekord aktywacji procedury *move*.

```
SL=MAIN   |   DL=
n: integer =
f: integer =
t: integer =
k: integer
begin

 k:=6-(f+t);
 if n>1 then call move(n-1
 call modyf(f,t);
   (* move only one ring *)
 if n>1 then move(n-1,k,t)
 return
end move;
```

```
SL=MAIN   |   DL=
n: integer =
f: integer =
t: integer =
k: integer
begin
 return;
  do
   k:=6-(f+t);
   if n>1 then call move(n-1,f,k); fi;
   call modyf(f,t);
    (* move only one ring *)
   if n>1 then move(n-1,k,t); fi;
   detach
  od
end wz;
```

Rekord taki tworzony jest za każdym wywołaniem call move(n, f, t). Przenoszone sa do niego wartości parametrów n, f, t. Wykonują się instrukcje z treści procedury *move*. Potem ten rekord aktywacji jest usuwany.
Na zielono pomalowany jest obiekt współprogramu wz. Obiekt taki jest tworzony jeden raz dla każdej kombinacji argumentów n, f, t przez polecenie P[n,f,t]:= new wz(n,f,t). Każde polecenie

call move(n, f, t) **może być zastąpione przez polecenie** attach(P[n,f,t]) **z tym samym efektem na zmiennych programu głównego. Nie trzeba tworzyć rekordu aktywacji – obiekt jest gotowy. Nie trzeba przesyłac parametrów – to zostało zrobione podczas tworzenia obiektu współprogramu. Nie trzeba usuwać obiektu – pozostaje on w gotowości do następnego wykorzystania.**

A tutaj mamy cały program

```
program HanoiTowers;
(* towers of hanoi *)
(* there are three towers built of decreasing rings stringed onto sticks *)
(* at the initial state all rings are stringed onto stick no. 1. our job is *)
(* to move all rings from the stick 1 to the stick 3. the difficulty is *)
(* that we mustn't violate the following conditions *)
(*  1. we can move only one ring at one step *)
(*  2. each ring may be placed only onto a greater one *)
(* to manage with this difficult problem we have an auxilliary stick 2 *)
unit wz:coroutine(n,f,t:integer);
(* move n rings from stick f to stick t *)
var k:integer;
begin
return;
do
k:=6-(f+t);
if n>1 then attach (p(n-1,f,k)); fi;
call modyf(f,t); (* move only one ring *)
if n>1 then attach (p(n-1,k,t)); fi;
detach;
od;
end wz;

unit modyf:procedure(f,t:integer);
(* move the topmost ring from stick f to stick t *)
begin
top(t):=top(t)+1;
w(t,top(t)):=w(f,top(f));
w(f,top(f)):=0;
top(f):=top(f)-1;
call displ;
end modyf;

unit displ:procedure;
(* printing *)
var t,i,j,k,m,n:integer;
begin
t:=1;
for i:=2 to 3 do
if top(i)>top(t) then t:=i fi od;
```

```
t:=top(t);
for i:=t downto 1 do
m:=15;
for j:=1 to 3 do
for k:=1 to m do write(); od;
if w(j,i)=/=0 then for k:=1 to w(j,i) do write("*") od;
fi;
m:=15-w(j,i);
od;
writeln;
od;
for i:=1 to 15 do write(); od;
for i:=1 to 45 do write(-"); od;
writeln;
end displ;

var w:arrayof arrayof integer, (* how many rings are stringed *)
(* on each stick *)
top:arrayof integer, (* the topmost ring size on each stick *)
nb,i,j,k,timeb:integer,
p:arrayof arrayof arrayof wz; (* coroutine pointers *)

begin
array w dim(1:3);
array top dim(1:3);
writeln("program towers of hanoi");
writeln("version with coroutines");
do writeln("give the number of rings");
read(nb);
writeln(nb);
if nb>0 then exit else writeln(ńumber of rings must be greater than 0")
fi od;
timeb:=time;
top(1):=nb;
array w(1) dim(1:nb);
array w(2) dim(1:nb);
array w(3) dim(1:nb);
k:=nb;
for i:=1 to nb do w(1,i):=k;
k:=k-1;
od;
(* stick 1 is full *)
writeln("the algorithm acts as follows");
call displ;
    array P dim (1:nb);
    for i:=1 to nb
    do
      array P(i) dim(1:3);
```

```
      for j:=1 to 3
      do
        array P(i,j) dim(1:3);
        for k:=1 to 3
        do if j=/=k then P(i,j,k):=new wz(i,j,k) fi      od
        od
      od;
   (* ∀1≤i≤n ∀1≤j≤3 ∀1≤k≤3 P[i,j,k] = new wz(i,j,k) *)
   attach (P(nb,1,3));
   writeln(" execution time for",nb:4," rings =",time-timeb," sec");
end
```

8. Treegen

W tym paragrafie przedstawiamy hierarchię typów współprogramów. W dowodzie twierdzenia o poprawności programu korzystającego z tej hierarchii stosujemy indukcję ze względu na tę hierarchię, a właściwie jest to indukcja ze względu na długość wyrażenia.

Definiowanie skończonych zbiorów. Zadaniem programu treegen.log jest wydrukowanie wszystkich słów z języka opisanego przez wyrażenie regularne, niezawierające gwiazdek. Zwróć uwagę, na ogół zbiory skończone przechowywane sa w jakiejś strukturze danych. Ale nie musi tak być zawsze. Jeśli potrafisz zapisać zbiór odpowiednią formułą to możesz zyskać w wielu aspektach. Przykładem jest formuła $23 \leq i \leq 2306$. Dzięki niej możemy precyzyjnie i krótko opisać zbiór $A = \{i \in mathbbN : 23 \leq i \leq 2306\}$.

Struktura programu. Program główny dzieli się swoimi dwoma zasobami z obiektami współprogramów, jakie są tworzone w dalszym ciągu. Są to tablica znaków WORD i zmienna całkowitoliczbowa i.

```
var WORD: arrayof char, i: integer, ...
```

Zacznijmy od współprogramu konsumenta

```
(* otoczenie wspólne dla wszystkich współprogramów *)
  var WORD: arrayof char,
      i: integer;
unit konsumuj: coroutine;
begin
  return;
  do
    attach(o);
    (* print the WORD *)
    for J:=1 to I
    do
      write(WORD(J))
    od;
    writeln;
    if o.B then exit fi
  od
end konsumuj;
```

Program – konsument – instrukcja attach(o) przekazuje sterowanie do obiektu

Hierarchia współprogramów. Tworzymy kilka modułów współprogramów.

RYsunek – drzewo hierarchii REGEXP

```
unit REGEXP:coroutine;
  var B:BOOL; (* B ≡ all the words of the language were shown *)
begin
  return
  inner;
  B := true
end REGEXP;
```

```
unit ATOM: REGEXP coroutine(c:CHAR);
begin
  do
    i:=i+1; (* update the position *)
    WORD(i):=c;
    B:=TRUE;
    detach
  od
end ATOM;
```

```
unit UNiON: REGEXP coroutine(l,r:REGEXP);
  var m: INTEGER;
begin
  do
    m:=i;
    do
      attach(l);
      if l.B then exit fi;
      detach;
      i:=m
    od;
    l.B:=FALSE;
    do
      detach;
      i:=m;
      attach(r);
      if r.B then exit fi;
    od;
    r.B:=FALSE;
    B:=TRUE;
    detach;
  od;
end UNION;
```

```
unit CONCATENATION: REGEXP class(l,r:REGEXP);
  var N,m:INTEGER;
begin
  do
    m:=i;
    do
      attach(l);
      N:=i;
      do
        attach(r);
        if r.B then if l.B then exit exit else exit fi fi;
        detach; i:=N
      od;
      r.B:=FALSE;
      detach;
      i:=m
    od;
    r.B, l.B:=FALSE;
    B:=TRUE;
    detach
  od;
end CONCATENATION;
```

Twierdzenie 24.6. Niech o będzie obiektem podklasy klasy **Regexp**, *o in Regexp*.

Poniższy program *Pr* wydrukuje wszystkie słowa należące do języka regularnego $|o|$ opisanego przez wyrażenie regularne o i zatrzyma się.

```
Pr: I:=0;
do
attach(o);
if o.B then exit
else
(* print the WORD *)
for J:=1 to I
do
write(WORD(J))
od;
writeln;
fi
od
```

Dowôd. Dowôd jest rozproszony pomiędzy podklasy klasy Regexp. Ktoś mógłby powiedzieć, że mamy do czynienia z dowodem ze względu na hierarchię podklas klasy regexp. W rzeczywistości nasz dowôd przebiega przez indukcję ze względu na długość wyrażenia regularnego ω, jakie zostało zakodowane w obiekcie o. Baza) Jeśli obiekt spełnia relację o is Regexp to wykonanie instrukcji attach(o) powoduje natychmiastowy powrôt z wartościa o.B true. Takie wyrażenie opisuje pusty język. Teza twierdzenia jest w tym przypadku spełniona.
Udowodnimy nieco mocniejszy lemat:

Lemat 24.7. Przypuśćmy, że wcześniejsze aktywacje obiektu *o* współprogramu *Regexp* wypełniły tablicę *WORD* na miejscach Let i0 be the value of the variable I. Suppose that the some words of the language L(o) were generated by the earlier activations of the coroutine o.
An execution of command attach(o) has the following effect: the subsequent word of the language L(o) is concatenated to the content of the WORD(1), ... , WORD(I); i.e. the new word is placed beginning of the position WORD(I+1). The value of B attribute becomes true iff all the words of the language L(o) were shown.

Jeśli obiekt spełnia relację o is Atom spełniony jest następujący lemat

to uzyskujemy jedno słowo o długości jeden.
W obu tych przypadkach teza jest □

program Treegen; (* Generates the language defined by a regular expression*)
(* Program written by A. Kreczmar 1982
proof written by A. Salwicki 1990 *)

```
unit REGEXP:coroutine;
(* an object in the hierarchy of subtypes of type REGEXP re-
presents a regular expression *)
(*
Theorem
For every object o in the hierarchy of classes that inherit from
Regexp class the program Pr (see below), when executed will
print all the words of the regular language represented by the
object o and then it will stop.
Pr: I:=0;
do
attach(o);
(* print the WORD *)
for J:=1 to I
do
write(WORD(J))
od;
writeln;
if W.B then exit fi
od

    Lemma
Let i0 be the value of the variable I. Suppose that the some
words of the language L(o) were generated by the earlier acti-
vations of the coroutine o.
An execution of command attach(o) has the following effect:
the subsequent word of the language L(o) is concatenated to
the content of the WORD(1), ... , WORD(I); i.e. the new word
is placed beginning of the position WORD(I+1). The value of
B attribute becomes true iff all the words of the language L(o)
were shown.

    PROOF of the lemma is distributed in the subclasses of the
class regexp, i.e. the proof goes by induction with respect to
the length of a regular expression *)

    var B:BOOL; (* B º all the words of the language were shown
*)
begin
return
inner;
B := true
end REGEXP;

    unit ATOM: REGEXP class(C:CHAR);
(* an atomic regular expression consists of a letter
```

Proposition. An execution of attach statement applied to this object will place the letter C on I+1-th place
in the table WORD and the value of B will be assigned to true.
In this way the whole regular language is displayed at once.
in this way we proved the base of the induction proof of the Lemma. *)
begin
do
I:=I+1; (* update the position *)
WORD(I):=C;
B:=TRUE;
detach
od
end ATOM;

unit UNION: REGEXP class(L,R:REGEXP);
(* represents the expression (L È R) i.e. the union *)
(* Proposition. Assume that objects L and R enjoy the property expressed by the Lemma
then any time this coroutine will be attached we obtain a subsequent word of the union of the languages L and R.

Consider, a regular expresion of the length k. By our definition it is either a union object or a concatenation object.
Let o be a union object i.e. o is UNION. The structure of its commands assures the following
while not exhausted(L)
do
attach(L) – by induction hypothesis this command returns a word of L language
od
(* L.B = true *) – the exhaustion mark for L
while not exhausted(R)
do
attach(R) – by induction hypothesis this command returns a word of R language
od
(* R.B = true
B = true *)

It is evident that in this way by repeated execution of attach(o) one obtains a sequence of words composed from the all words of L language followed by the sequence of all words from the R language. *)

var m: INTEGER;
begin
do (* repeat : store I; generate one word (first from L next

```
from R; detach; restore I until exhausted *)
m:=I;
(* I is the position of the lastly generated letter. *)
(* M+1 is the position where the current UNION object *)
(* will place the letters of the currently generated word. *)
do
attach(L); (* by the inductive assumption this statement causes
that one word will be generated of the language L and it will
be concatenated to the content of WORD(1) , ... , WORD(I)
*)
if L.B then exit fi;
detach;
I:=m (* reestablish the position in the table WORD for the
next word *)
od;
L.B:=FALSE; (* restart language L *)
do
detach;
I:=m; (* reestablish the position in the table WORD for the
next word *)
attach(R); (* by the inductive assumption this statement cau-
ses that one word will be generated of the language R and it will
be concatenated to the content of WORD(1) , ... , WORD(I)
*)
if R.B then exit fi;
od;
R.B:=FALSE; (* restart language R *)
B:=TRUE;
detach;
od;
end UNION;

    unit CONCATENATION: REGEXP class(L,R:REGEXP);
(* represents the concatenation (L·R) of the languages repre-
sented by the regular expressions L and R *)
(* Suppose the object o is of the class CONCATENATION.

    Now the loop of commands of object o assures basically the
following
while not exhausted
do
store (I);
attach(L); – a word from L
attach(R); – followed by a word from R
detach; – hence a word of (L R) is given
restore(I)
od
```

```
    with the necessary reactions to a case when one language (L
or R) ends.
It is clear that if the object L and R enjoy the property men-
tioned in the Lemma then the object o enjoys it too*)

    var N,m:INTEGER;
begin
do
m:=I; (*M stores the begin position of first language word po-
sition *)
do
attach(L);
N:=I; (* N stores the begin position of the second language
word position *)
do
attach(R);
if R.B then if L.B then exit exit else exit fi fi;
detach; I:=N (* restart language R word generation position *)
od;
R.B:=FALSE; (* restart language R *)
detach; I:=m (* restart language L word generation position *)
od;
R.B,L.B:=FALSE; B:=TRUE; detach
od;
end CONCATENATION;



const N=50; (* DIMENSION FOR ARRAY WORD *)

    var A,B,C,D,E,W,V,L,O,G,II,NN:REGEXP,
I,J,N,M:INTEGER;
(* I = GLOBAL POSITION POINTER FOR ARRAY WORD
*)
var WORD: arrayof CHAR; (* BUFFER FOR WORDS GENE-
RATION *)

begin
writeln(ŁANGUAGE GENERATOR USING COROUTINES");
writeln(ŁANGUAGE IS REPRESENTED AS A TREE WITH
OPERATIONS IN NODES");
writeln(ÓUR OPERATIONS ARE SET THEORETICAL JOIN
AND CONCATENATION OF");
writeln(ŁANGUAGES");writeln;
A:=new ATOM('A'); B:=new ATOM('B'); C:=new ATOM('C');
D:=new ATOM('D'); E:=new ATOM('E');
L:=new ATOM('L'); G:=new ATOM('G');
II:=new ATOM('I'); NN:=new ATOM('N');
```

```
O:=new ATOM('O');
W:=new UNION(A,L);
W:=new CONCATENATION(W,new UNION(D,O));
V:=new CONCATENATION(II,C);
V:=new UNION(V,new CONCATENATION(L,new CONCA-
TENATION(A,NN)));
V:=new CONCATENATION(G,V);
V:=new UNION(A,V);
W:=new CONCATENATION(W,V);
writeln("WE HAVE LANGUAGE DEFINED BY THE FOLLO-
WING EXPRESSION");
writeln;
writeln("(AÈL)·(DÈO)·(AÈG·(I·CÈL·A·N))");
writeln; writeln;
array WORD dim(1:N);
do
attach(W);
write();
for J:=1 to I
do
write(WORD(J))
od;
writeln;
if W.B then exit fi
od
end

   (*
Theorem
For every object o in the hierarchy of classes that inherit from
Regexp class the program Pr (see below), when executed will
print all the words of the regular language represented by the
object o and then it will stop.
Pr: I:=0;
do
attach(o);
for J:=1 to I
do
write(WORD(J))
od;
writeln;
if W.B then exit fi
od

   Lemma
Let i0 be the value of the variable I. Suppose that the some
words of the language L(o) were generated by the earlier acti-
vations of the coroutine o.
```

An execution of command attach(o) has the following effect: the subsequent word of the language L(o) is concatenated to the content of the WORD(1), ... , WORD(I); i.e. the new word is placed beginning of the position WORD(I+1). The value of B attribute becomes true iff all the words of the language L(o) were shown.

Proof.
Induction with respect to the length of the expression represented by the object o.

Base. Suppose the actual type of o is ATOM. Then the thesis of the lemma is satisfied.

Induction step. Suppose the lemma holds for every regular expression shorter than an integer k. Consider, a regular expresion of the length k. By our definition it is either a union object or a concatenation object.
case A. Let o be a union object i.e. o is UNION. The structure of its commands assures the following
while not exhausted(L)
do
attach(L) – by induction hypothesis this command returns a word of L language
od
L.B := true – set the exhaustion mark for L
while not
do
attach(R) – by induction hypothesis this command returns a word of R language
od
L.R := true
B := true

It is evident that in this way by repeated execution of attach(o) one obtains a sequence of words composed from the all words of L language followed by the sequence of all words from the R language.

case B Suppose the object o is of the class CONCATENATION.

Now the loop of commands of object o assures basically the following
while not exhausted
do
store (I);
attach(L); – a word from L

attach(R); – precedes a word from R
detach; – hence a word of (L R) is given
restore(I)
od

with the necessary reactions to a case when one language (L or R) ends.
It is clear that if the object L and R enjoy the property mentioned in the Lemma then the object o enjoys it too.
This ends the proof of the Lemma. ¨

9. Przeszukiwanie z nawrotami

Dośc często stosuje się zasadę poszukiwania rozwiazań, ktôra sprowadza sie do przeglądania wszystkich możliwośći. Najczęściej spotykamy sie z takim podejśćiem w zadaniach sztucznej inteligencji oraz w grach. Wystarczy przypomnieć znane zadanie o przewiezieniu trzech kanibali i trzech misjonarzy przez rzekę gdy do dyspozycji jest łôdka pozwalająca przewieźć naraz tylko dwie osoby.

Poniżej podajemy tekst programu znajdującego rozwiązanie problemu kanibali i misjonarzy. Zwrôć uwagę na klasę backtrack. Opisuje ona mechanizm tworzenia drzewa ... Klasa ta nadaje się do wielokrotnego zastosowania.

```
program backtracking;
unit backtrack: class;
hidden se;
var root:node,search:se,found:node,
number_of_leaves,number_of_answers:integer;

unit node: class(father:node);
var nsons,level: integer , deadend:boolean;
unit virtual leaf: function :boolean;
end leaf;
unit virtual answer :function :boolean;
end answer;
unit virtual lastson: function : boolean;
end lastson;
unit virtual nextson: function : node;
end nextson;
unit virtual equal : function (w:node):boolean;
end equal;
unit virtual cost: function :real;
end cost;
begin
if father =/= none
```

```
then
level:=father.level+1
else
level:=0
fi;
end node;

unit ok: function (v:node):boolean;
var w:node;
begin
if v=none then result:=false; return fi;
result:=true; w:=v.father;
while w =/= none
do
if v.equal(w) then result:=false; return fi;
w:=w.father
od
end ok;

unit purge: procedure (v:node);
var w: node;
begin
if v=none then return fi;
do
w:=v.father; kill(v);
if w=none then return fi;
w.nsons:=w.nsons-1;
if w.nsons =/= 0 then return fi;
v:=w
od;
end purge;

unit virtual insert: procedure(v:node);
end insert;

unit virtual delete : function :node;
end delete;

unit se: coroutine ;
var i:integer,v,w:node;
begin
return; call insert(root);
do
v:=delete;
if v=none then exit fi;
if v.answer
then
number_of_answers:=number_of_answers+1;
```

```
found:=v; detach; call purge(v);
else
if v.deadend
then
number_of_leaves:=number_of_leaves+1;
call purge(v);
else
do
w:=v.nextson; v.nsons:=v.nsons+1;
if ok(w)
then
w.deadend:=w.leaf; call insert(w);
fi;
if v.lastson then exit fi;
od;
fi;
fi;
od;
found:=none;
end se;

unit optimize: function: node;
var v,w:node;
begin
call insert(root);
do
v:=delete;
if v=none then exit fi;
if v.answer
then
number_of_answers:=number_of_answers+1;
if result=none orif result.cost > v.cost
then
call purge(result); result:=v
fi;
else
if v.deadend
then
number_of_leaves:=number_of_leaves+1;
call purge(v)
else
do
w:=v.nextson; v.nsons:=v.nsons+1;
if ok(w)
then
w.deadend:=w.leaf; call insert(w);
fi;
if v.lastson then exit fi;
```

```
od;
fi
fi;
od;
end optimize;

unit killall :procedure;
var v:node;
begin
do
v:=delete;
if v=none then return fi;
call purge(v);
od;
end killall;

begin
search:=new se;
inner;
kill(search);
end backtrack;
```

```
unit dfs :backtrack class;
var top:elem;

unit elem: class (next:elem, v:node);
end elem;

unit virtual insert: procedure(v:node);
begin
top:=new elem(top,v);
end insert;

unit virtual delete: function :node;
var e:elem;
begin
if top =/= none
then
result:=top.v;
e:=top; top:=top.next; kill(e);
fi;
end delete;
end dfs;
```

```
(* tu się zaczyna program przykładu *)
var n,bc:integer,h1,h2,h3:char;
begin
do
writeln("tell how many canibals(=no missionaries)");
```

```
write(ń= ");
readln(n);
if n=0 then exit fi;
write("boat capacity=");
readln(bc);

pref dfs block
var m,c:integer; (* bc - boat capacity, n- number of cannibals

n- number of missionars *)
unit state: node class(m1,c1:integer);
var m2,c2:integer, left:boolean;
(* m1,m2 number of missionars on both sides of the river
c1,c2 number of cannibals on both sides of the river *)

unit virtual answer: function: boolean;
begin
result:=m1=0 and c1=0
end answer;

unit virtual leaf: function : boolean;
begin
if m1<0 orif m2<0 orif c1<0 orif c2<0 orif
m1>n orif m2>n orif c1>n orif c2>n orif
m1<c1 and m1>0 orif m2<c2 and m2>0
then
result:=true
fi
end leaf;

unit virtual lastson: function :boolean;
begin
if c=0 and m=bc
then
result:=true; m:=0; c:=0;
fi;
end;

unit virtual nextson : function :state;
begin
c:=c+1;
if m=0
then
if c>bc
then
c:=0; m:=1
fi
else
```

```
if m<c orif m+c>bc
then
c:=0; m:=m+1;
fi
fi;
if left
then
if c+m<bc
then
result:=none
else
result:=new state(this state,m1-m,c1-c)
fi
else
result:=new state(this state,m1+m,c1+c)
fi;
end nextson;

unit virtual equal: function(w:state):boolean;
begin
result:=left=w.left and m1=w.m1 and c1=w.c1;
end equal;

unit virtual cost: function :real;
begin
result:=level
end cost;

begin
left:=level mod 2 = 0;
m2:=n-m1; c2:=n-c1;
end state;

unit display: procedure(v:state);
var j,i:integer, w:state,at: arrayof state;
begin
if v=none then writeln(ńo more solutions"); return fi;
i:=v.level;
array at dim (0:i);
w:=v;
for j:=i downto 0
do
at(j):=w; w:=w.father
od;
writeln("move number left side direction right side");
for j:=0 to i
do
write(j); write();
```

```
w:=at(j);
write(w.m1,w.c1,);
if w.left
then
write(->");
else
write(«-");
fi;
writeln(,w.m2,w.c2);
od;
kill(at);
end display;

begin
root:=new state(none,n,n);
write("do you want to optimize ");
writeln(ór to print all the solutions ?");
writeln("(answer opt or all)");
readln(h1,h2,h3);
if h1='o' and h2='p' and h3='t'
then
found:=optimize;
if found =/= none
then
call display(found);
writeln(ńumber of leaves=",number_of_leaves);
writeln(ńumber of answers=",number_of_answers);
else
writeln(ńo solutions");
writeln(ńumber of leaves=",number_of_leaves);
fi;
call killall;
else
if h1='a' and h2='l' and h3='l'
then
do
   attach(search);
   call display(found);
   writeln("ile liści=",number_of_leaves);
   writeln("ile odpowiedzi=",number_of_answers);
   if found=none then exit fi
od;
call killall;
fi
fi;
end;
od;
```

```
end;

end
```

10. Symulacja.

Współprogramy mogą być z sukcesem użyte do symulacji procesów dyskretnych. Np do symulacji epidemii grypy w pewnej populacji, do symulacji ruchu pojazdów w mieście, etc. Czyli, tam gdzie trudno zastosować wzory analityczne, natomiast znane są pewne statystyczne prawidłowości dotyczące procesów. TODO
opisz przykład,
wytłumacz pojęcia: simproces, oś zdarzeń,
operacje: hold, schedule, passivate, run
Schemat programu stosującego klasę Simulation

```
program Bank;
    unit FIFOqueues: class
        ...
    end FIFOqueues;
    unit PriorityQueues: FIFOqueues: class
        ...
    end PriorityQueues;
    unit Simulation: PriorityQueues class
        ...
        unit simproc:FIFOelem class ... end simproc;
         (* procesy symulowane mogą stawać w kolejkach *)
        ...
    end Simulation;
begin
    ...
    pref Simulation block
    (* w tym bloku można wykorzystywać pojęcia i operacje
       opisane w klasie Simulation *)
    begin
        tu umieść Twój program symulacyjny
    end (* blok/program Symulacji *)
end
```

Tu napisz omówienie przykładu z bankiem. Klasy bankteller i klientbanku i jak to działa...
Skomentuj strukture podana powyżej.

11. Poprawność klasy Simulation

W tym rozdziale pokażemy jak można zrealizować klasę na podstawie jej specyfikacji, konstruując przy tym dowód poprawności. Rozważać będziemy dwie specyfikacje: specyfikację Symulacja docelowej klasy Simulation, oraz specyfikację ATPQ

kolejek priorytetowych – tj. bazę na której zbudowana będzie klasa Simulation. In these articles specifications are viewed as somewhat theoretical, abstract beings. In fact, we talk of formalized algorithmic axiomatizations. Yet, the practice of programming brings the concept similar to specifications, it is interface module.

Now, we illustrate that in some circumstances one may build a class using only knowledge of two specifications: specification $\mathcal{S}_A$ of the base class A and a specification $\mathcal{S}_B$ of a target class B that inherits the class A. No knowledge on the body of the inherited class is needed.

The meaning of this result is the following: the created class B will be correct with any class A provided it correctly implements the specification $\mathcal{S}_A$. One possible application of this result is a quick construction of software. A prototype class A may be used in order to quickly construct the system consisting of classes A and B. Later, one may replace the class A by a more efficient implementation.

A comment is in place here: our specifications can be compared with the interface modules of Java. It turns out that:

1° Interfaces limit themselves to the signature part of an specification. The specification files .spec of the SpecVer system contain signatures as well as properties (or invariants, or axioms) of specified classes.

2° Interfaces can not prevent misinterpretation of the signature. Imagine, an interface

```
interface Stacks  {
    Stacks push(Element e, Stacks s) {  }
    Stacks pop(Stacks s) {  }
    Element top(Stacks s) {  }
}
```

What will happen if a class Stacks implemnts this specification in the manner of FIFO instead of LIFO? This and similar examples demonstrate that specifications of Java do not garantee anything but that the class implementing an interface has declared some methods of given names and parameters.

3° Interfaces are not complete. Obviously, one interface I may be implemented in many ways. There is no way to express that any implementation of I must ...

In section 12 we give the specification for the class Simulation. Section 12.1 presents step by step work on implementation of the class Simulation. Appendix A contains the specification of the base class PQS. Appendix B contains an informal, yet sufficiemtly complete, specification of coroutines.

12. Specification of Simulation class

Now we are going to describe and to axiomatize a system of discrete event simulation. There are numerous situations in which we have to deal with processes to be simulated, for example:

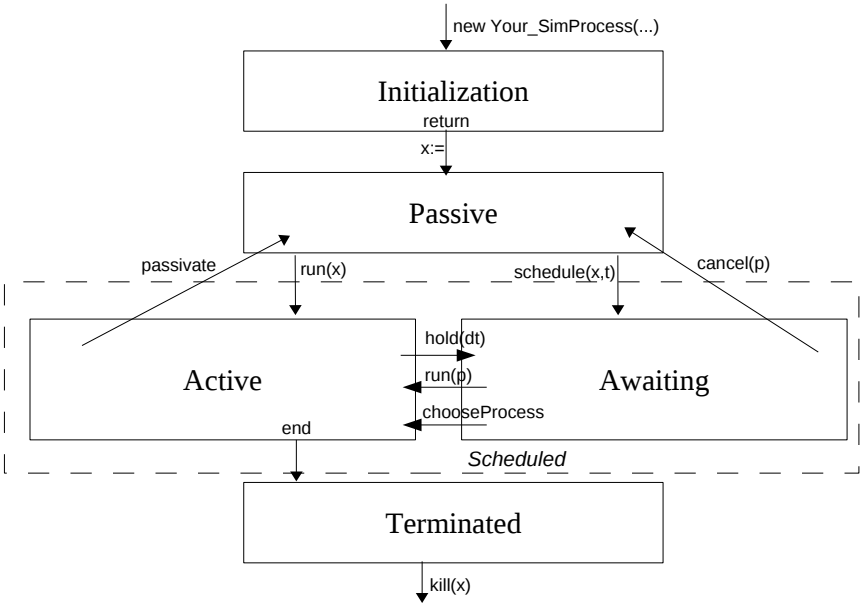
- system of patients and medecine doctors,
- system of vehicles and street lamps,
- system of ...kupno-sprzedaż

Class Simulation is meant as a base for further extensions. It offers a prototype of simulated processes and defines a set of basic operations on processes. The universe of the system *Simulation* of simulated processes and discrete events consists of four disjoint subsets: $SimProcess, EventNotice, Time$ and $SimulationPlans$. Objects of type *SimProcess* are quasi-threads, more precisely they are coroutines. Each object of type *SimProcess* has a thread, however only one coroutine is executed in a moment. The control passes from one coroutine to another grace the direct command attach(x).

The type *Time* may be a predefined class.

The type *EventNotice* has two fields: process and time.

The type *SimulationPlan* is the data structure of priority queues of *EventNocices*



12.1. Constructing Simulation class.

12.1.1. From specification's signature to the skeleton of the class Simulation. The first step is a simple,almost automatic, translation of the specification's signature onto the skeleton of

the class Simulation.

Specification's signature	Class' skeleton

Types:
simprocess
$plan\_of\_simulation \subset PQ$
time
eventnotice
 	
Operations:
current - this process is currently active,
$current : PQ \to SP$
time - the value of currently simulated time,
$time : PQ \to T$
schedule - enables planning of events,
$schedule : (SP \times T) \times PQ \to PQ$
hold - suspend a current process for a while,
$hold : T \times PQ \to PQ$
run - immediately execute the process,
$run : SP \times PQ \to PQ$
passivate - suspends the current process,
$passivate : PQ \to PQ$
cancel - removes a process from plan,
$cancel : SP \times PQ \to PQ$
idle?
$idle : SP \to Boolean$
terminated?
$terminated : SP \to Boolean.$
	unit Simulation: PriorityQueues class
	
	unit Simprocess: elemFIFO coroutine;
	unit isIdle: function: Boolean;
	unit isTerminated: function: Boolean;
	end Simprocess;
	
	unit EventNotice: elemPQ class; ...
	end EventNotice;
	
	unit PlanSymulacji: QueueHead class;
	unit schedule : proc(p: SimProcess, t: ti
	unit hold: procedure(dt: time);
	unit run: procedure(p: SimProcess);
	unit passivate: procedure;
	unit cancel: procedure;
	unit chooseProcess: procedure;
	unit currentProcess: function: SimProce
	unit currentTime: function: time;
	var currProcess: SimProcess, currTime:
	end PlanSymulacji;
	
	unit Time: class ... end Time;
	
	var SQS: PlanSymulacji;
	end Simulation;

Below we gathered the postulates (invariants, axioms) the class Simulation should obey.

(S1)

SQS *is a finite set.*

(S2)

$EventNotice = SimProcess \times Time$

(S3)

$SQS.currentProcess = (SQS.min\ qua\ EventNotice).p$

(S4)

$SQS.currentTime = (SQS.min\ qua\ EventNotice).t$

(S5)

$\neg\, idle(p, pq) \Rightarrow (\exists t) member((p, t), pq)$

(S6)

$\forall_{pq \in SimulationPlan} \forall_{p \in Simprocess} member((p, t_1), pq) \wedge member((p, t_2), pq) \Rightarrow t_1 = t_2$

(S7)
$pq = c \wedge idle(p, pq) \wedge \neg terminated(p, pq) \Rightarrow$
$$[callschedule((p,t), pq)]\neg idle(p, pq) \wedge pq = insert((p,t), c)$$
(S8)
$(pq = o \wedge \neg idle(p, pq) \wedge \neg terminated(p, pq)) \Rightarrow$
$$[callschedule((p,t), pq)](idle(p, pq) \wedge pq = insert((p,t), delete((p, time), o)),$$
(S9)
$terminated(p, pq) \Rightarrow [callschedule((p,t), pq)]\{ERROR\},$
(S10)
[**call** $hold(t, pq)$] $\alpha \equiv$ [**call** $schedule((current(pq), time + t), pq)$] α,
(S11)
$(current(pq) = p' \wedge \neg terminated(p, pq)) \Rightarrow$
$$\{[callrun(p, pq)]\alpha \equiv [callschedule(p, time, pq)]\alpha\}$$
(S12)
$(p = current(pq) \wedge pq = o) \Rightarrow$
$$[\textbf{call } passivate(pq)](idle(p, pq) \wedge pq = delete((p, time), o))$$
(S13)
$pq = o \Rightarrow$ [**call** $cancel(p, pq)](idle(p, pq) \wedge pq = delete((p,t), o))$.

We made the following decisions:

- We introduce the class PlanSymulacji derived upon the class Queuehead which implements priority queue
- Instead of functions $schedule, run, hold, passivate, etc.$ of type PQ with one argument of type PQ we declare methods $schedule, run, hold, passivate, etc.$ within class PlanSymulacji. In this way we spare transferring argument and receiving the result. This simplifies the body of class Simulation and the usage of it.

The full text of the first version is here.
Now we ought to fill the bodies of methods schedule, run, hold, etc in such a way that the properties S1 – S13 are valid.

12.1.2. Własność S1.

(S1) $$SQS \text{ is a finite set.}$$

Remark that the property S1 is garanteed. For the variable SQS points to a priority queue object.

12.1.3. Własność S2: EventNotice = Simprocess × Time.

(S2) $$EventNotice = SimProcess \times Time$$

Własność S2 says: objects of type $EventNotice$ are pairs $\langle s, t\rangle$ where $s \in SimProcess$ and $t \in Time$. $EventNotice$ objects are inserted into priority queue. Hence they need an ordering relation. We use the following definition:
$e1 \leq e2 \stackrel{df}{\equiv} e1.t \leq e2.t$

The class *EventNotice* takes the following form

```
unit EventNotice: elemPQ class(p: SimProcess, t: Time);
    unit less: virtual function(e: EventNotice): Boolean;
    begin
        result:= t ≤ e.t
    end less;
end EventNotice;
```

The full text of the second version is here.

12.1.4. Własności S3 i S4. We shall prove that in each state of SimulationPlan SQS and hence in each moment of a simulation experiment the following two properties hold.

(S3) $SQS.currentProcess = (SQS.min\ qua\ EventNotice).p$

and

(S4) $SQS.currentTime = (SQS.min\ qua\ EventNotice).t$

To assure this, we put the instruction
call *chooseProcess*;
as the last instruction in the operations: $hold, run, passivate$.
For example,

```
unit hold: procedure(dt: time);
begin
    ...
    call chooseProcess;
end hold;
```

The instruction *chooseProcess* has to select from the priority queue *SQS* the eventnotice of the minimal time and to activate the process named in this eventnotice. Moreover information on the chosen process and on the time of chosen eventnotice are to be accessible as the values of function designators: *cuurrentProcess* and *currentTime*. We achieve this by declaring private variables: *currProcess* and *currTime* and makking their values available through the methods *currentProcess* and *currentTime*.

```
unit chooseProcess: procedure;
    var e: EventNotice;
begin (* value of SQS.min is the least element of priority queue *)
    e:=SQS.min qua EventNotice; (* projection qua EventNotice is needed here *
    (* variables currTime i currProcess are private variables of the object SQS *)
    currProcess:= e.p;
    currTime := e.t;
    attach(e.p);
end chooseProcess;
```

We can not not forget to declare method *currentProcess*.

```
unit currentProcess: function: SimProcess;
begin
    result := currProcess;
end currentProcess;
```

In a similar way we declare method *currentTime*. The reader sees that properties S3 and S4 are valid. The full text of the third version is here.

12.1.5. Własność S5.

(S5) $$\neg idle(p, pq) \Rightarrow (\exists t) member((p, t), pq)$$

In words, every not suspended process is planned for certain time t. This property requires that information whether an object s of type *SimProcess* has an eventnotice in the *SimulationPlan* or not were known to the object itself. The simplest way to achieve this is to put the eventnotice $\langle s, t\rangle$ in the object s. Now, the function *idle* answers correctly by checking the value of the variable *event*.

```
unit SimProcess: elemFIFO coroutine;
    var event: EventNotice; (* make sure that, event.p = this SimProcess *)
    unit isIdle: function: Boolean;
    begin
        result := (event=none); (* not scheduled iff event = none*)
    end isIdle;
end SimProcess;
```

The full text of the fourth version is here.

12.1.6. Własność S6.

(S6)
$$\forall_{pq \in SimulationPlan} \forall_{p \in Simprocess} member((p, t_1), pq) \wedge member((p, t_2), pq) \Rightarrow t_1 = t_2$$

In words, in every simulation plan every process can be planned at most once. Własność S6 will follow from the analysis of methods *schedule*, *hold*, *run*, for they are inserting an eventnotice to the plan of simulation.

12.1.7. Własność S7. Własność S7 reads:

(S7)
$$pq = c \wedge idle(p, pq) \wedge \neg terminated(p, pq) \Rightarrow [call schedule((p, t), pq)] \neg idle(p, pq) \wedge pq = insert((p, t), c)$$

In words, a suspended process can be scheduled to be reactivated at time t. Note it is the time of simulation system.

$$terminated(p) \equiv objectpexecutedallitsinstructions$$

We shall write a constructor (empty) and a thread of the coroutine *SimProcess*.

```
begin
    return; (* end of constructor, constructor is empty *)
    inner; (* it is a place for the thread of derived class *)
    finished :=true; (* thread is terminated *)
    call passivate;
    raise Error; (* at the attempt to activate a terminated simprocess object *)
end SimProcess;
```

A provisory variant of procedure *schedule* may look as follow:

```
unit schedule: procedure(p:SimProcess, t: time);
    var ev: EventNotice;
begin
    ev := new EventNotice(p, t);
    if p.idle and not p.terminated then call SQS.insert(ev); p.event:=ev; endif;
end schedule;
```

The full text of the fifth version is here.

12.1.8. Własności S8 i S9.

(S8)

$(pq = o \wedge \neg idle(p,pq) \wedge \neg terminated(p,pq)) \Rightarrow [callschedule((p,t),pq)](idle(p,pq) \wedge pq = insert((p,t),delete((p,ti$

A scheduled already process p can be scheduled again for another time, this will delete an earlier eventnotice for p.

(S9) $terminated(p,pq) \Rightarrow [callschedule((p,t),pq)]\{ERROR\},$

An attempt to schedule a terminated process results in an error. Properties S8 and S9 require the correct, full version of operation *schedule*.

```
unit schedule: procedure(p:SimProcess, t: time);
    var ev: EventNotice;
begin
    if p. terminated
    then
        raise ErrorDo_not_ScheduleTerminatedProcess;
    else
        ev := new EventNotice(p, t);
        if not p.idle
        then
            call SQS.delete(p.event);
        endif;
        call SQS.insert(ev);
        p.event:=ev;
    endif;
end schedule;
```

As it is easy to observe the operation *schedule* defined in this way satisfies properties S7 and S8. We should react when the parameter t has the value less than *currentTime*.

The full text of the sixth version is here.

12.1.9. Własność S10.

(S10) [call $hold(t, pq)$] $\alpha \equiv$ [call $schedule((current(pq), time + t), pq)$] α, for arbitrary formula α.

A hold operation suspends the current process and schedules its activation after t units of time. The property leads directly to the following body of the procedure hold.

```
unit hold: procedure(dt: time);
begin
    call SQS.schedule(currentProcess, currentTime+dt);
    call SQS.chooseProcess;
end hold;
```

The first instruction causes suspension of the current simprocess' thread for dt units of time. The second instruction will choose a new active simprocess object. The full text of the seventh version is here.

12.1.10. Property S11.

(S11)
$(current(pq) = p' \wedge \neg terminated(p, pq)) \Rightarrow \{[callrun(p, pq)]\alpha \equiv [callschedule(p, time, pq)]\alpha\}$ for arbitrary formula α.

Procedure run immediately activates the indicated simprocess p.

```
unit run: procedure(p: SimProcess);
begin
    if p.terminated
    then
        raise ErrorDo_not_ScheduleTerminatedProcess;
    endif;
    call hold(0.1); (* wstrzymaj na chwile bieżacy proces *)
    call schedule(p, currentTime);
end run;
```

The instruction call run (x) results in immediate activation of the simprocess object x.

The full text of the eigtht version is here.

12.1.11. Własność S12.

(S12)
$(p = current(pq) \wedge pq = o) \Rightarrow$ [call $passivate(pq)$]$(idle(p, pq) \wedge pq = delete((p, time), o))$

Instruction passivate removes the current simprocess from the plan of simulation.

```
unit passivate: procedure;
    var s: Simprocess;
begin
    s := currentProcess;
    call SQS.delete(s.event);
    s.event :- none;
    call SQS.chooseProcess;
end passivate;
```

The full text of the ninth version is here.

12.1.12. Własność S13.

(S13) $pq = o \Rightarrow [\text{call } cancel(p, pq)](idle(p, pq) \wedge pq = delete((p, t), o)).$

Instruction cancel applies to a scheduled, non-active simprocess.

```
unit cancel: procedure(p: SimProcess);
begin
    call SQS.delete(p.event);
    p.event :- none;
    call SQS.chooseProcess;
end cancel;
```

The full text of the tenth version is here.

12.1.13. Własność S6 ponownie.

(S6)

$\forall_{pq \in SimulationPlan} \forall_{p \in Simprocess} member((p, t_1), pq) \wedge member((p, t_2), pq) \Rightarrow t_1 = t_2$

Własność ta powiada: w każdym momencie obliczeń i dla każdego procesu s, w planie symulacji nie ma dwu zdarzeń e1 i e2 planujacych wznowienie procesu s w dwu różnych chwilach. Można sprawdzić, że procedura schedule zapewnia te własność, a w konsekwencji także pozostałe procedury planowania, które sie na tej procedurze opieraja, zapewniaja te własność

12.1.14. Uwagi. skad sie wział prior?

Unit Mainpr: SimProcess class

12.2. Wnioski. Jako jeden z natychmiastowych wniosków otrzymujemy:

Twierdzenie 24.8 (o relatywnej poprawności). Jeśli klasa bazowa jest modelem specyfikacji Kolejek Priorytettowych to klasa Simulation jest modelem (tj. poprawnie implementuje) specyfikację Symulacja.

Dowód. Wynika z konstrukcji klasy Simulation. □

Kolejne pytanie jakie się nasuwa brzmi: czy istnieje więcej implementacji będących modelem specyfikacji Symulacja? a może wszystkie one są izomorficzne?

Twierdzenie 24.9. Niech klasa C_{PQ} będzie modelem specyfikacji ATPQ kolejek priorytettowych. Każde dwie poprawne implementacje specyfikacji Symulacja (tj. modele) które bazują na klasie C_{PQ} są izomorficzne.

Dla użytkownika duże znaczenie ma następujące

Twierdzenie 24.10. W każdym kroku obliczenia ... współprogramem aktywnym jest current process.

Dowód. Wynika z własności S1 – S13. □

A więc ...

12.3. Appendix A: Specification of Priority Queues. The specification of priority queues class was published in [MS87], we recall it here for the convenience of the reader.

Table 1. Specification $ATPQ$ of priority queues.

Signature	Comments
Sorts	$Universe = E \cup PQ$
E	set of elements
PQ	set of priority queues
Operations	let $e \in E$ and $q \in PQ$
$insert : E \times PQ \longrightarrow PQ$	put e into q
$delete : E \times PQ \longrightarrow PQ$	delete e from q
$min : PQ \longrightarrow E$	find the minimum element
$empty : PQ \longrightarrow \{true, false\}$	is a priority queue q empty?
$member : E \times PQ \longrightarrow \{true, false\}$	does $e \in q$?
$\leq : E \times E \longrightarrow \{true, false\}$	the ordering relation

Axioms

(a1) *The set E ofelements is linearly ordered by the relation* $\leq$.

(a2) [**while** *not* $empty(q)$ **do** $q := delete(min(q), q)$ **done**] *true*

This axiom says for all q program halts, i.e. the priority queue q is finite

(a3) $[q1 := insert(e, q)]\{member(e, q1) \wedge (\forall_{e1 \neq e}\ member(e1, q1) \Leftrightarrow member(e1, q))\}$

(a4) $[q1 := delete(e, q)]\{\neg member(e, q1) \wedge (\forall_{e1 \neq e}\ member(e1, q1) \Leftrightarrow member(e1, q))\}$

(a5) $empty(q) \Rightarrow (\forall_{e \in E}\ \neg member(e, q))$

(a6) $\neg empty(q) \Rightarrow (\forall_{e \in E}\ member(e, q) \Rightarrow min(q) \leq e))$

The operation min finds the least element of the set q.

(a7) $[e := min(q)]true \Leftrightarrow \neg empty(q)$

Axiom (a7) says the result of expression $min(q)$ is defined iff $\neg empty(q)$

(a8) $member(e, q) \Leftrightarrow$ **begin**

```
s1 := q; result := false;
while not empty(s1) and not result do
   if e =min(s1) then result:=true fi;
   s1 := delete(min(s1),s1)
done
end result
```

ROZDZIAŁ 25

$\mathcal{L}_{10}$ Procesy

1. Programowanie współbieżne i rozproszone

W programowaniu niesekwencyjnym występują dwa podstawowe pojęcia: proces i procesor. Należy przy tym odróżniać moduł procesu od obiektu procesu. By sprawę skomplikować jeszcze bardziej niektórzy mówią o wątku procesu.
Program sekwencyjny jest modułem procesu. Program niesekwencyjny zawiera moduły typu process i może utworzyć i uruchomić więcej obiektów procesów. Każdy moduł procesu jest sekwencyjny w tym sensie, że jego instrukcje wykonywane są po kolei, przez odpowiedni procesor. (Równiez instrukcje uruchamiania nowego procesu.) Procesor – fizyczny lub wirtualny zapewnia postęp w obliczeniach wykonując kolejne instrukcje procesu i dokonując zmian w pamięci.
Z tymi, nie całkiem precyzyjnymi pojęciami, możemy podjąć próbę klasyfikacji obliczeń niesekwencyjnych:

współbieżne : Z obliczeniami współbieżnymi wielu procesów mamy do czynienia wtedy, gdy wykonywane są na jednym wspólnym procesorze,

rozproszone : Obliczenia rozproszone wykonywane są na procesorach połączonych siecią. Z konieczności każdy procesor działa w osobnej pamięci.

równoległe : Obliczenia równoległe wykonywane są na komputerze wyposażonym w wiele procesorów(lub rdzeni) i wspólną pamięć. Dzisiaj, każdy procesor wyposażony jest w swoją podręczną pamięć o całkiem sporym rozmiarze.

Definicje przyjęte w Loglanie. W dalszych rozważaniach przyjmujemy następujące definicje.

Definicja 25.1. Moduł programu podobny do deklaracji klasy, odróżniający się od tej ostatniej tym, że w miejsce słowa kluczowego class występuje słowo process nazywamy modułem procesu.

Definicja 25.2. Obiekt utworzony na podstawie modułu procesu nazywamy obiektem procesu lub krótko procesem.

Definicja 25.3. Procesorem jest system wykonawczy Loglanu, tj. Virtual Loglan Processor,w skrócie VLP. W obliczeniach

współbieżnych obiekt procesu (jeśli jest ich więcej niż jeden) dzieli się procesorem VLP z pozostałymi (wykorzystuje pewien ułamek jego czasu).
Jeśli trzy obiekty procesu(-ów) wykonywane są przez jeden procesor VLP, to każdemu z nich, na zmianę, przydziela się parę milisekund czasu procesora fizycznego. Kazdy z nich otrzyma w przybliżeniu 1/3 czasu procesora.

Loglan oferuje jednolity model programowania obliczeń współbieznych i rozproszonych. Model ten obejmuje też rozmaite konfiguracje mieszane. Przez procesor będziemy rozumieć loglanowską maszynę wirtualna VLP. Na każdej takiej maszynie można wykonywac wiele programów naraz, a także można wykonywać zadania wielu procesów jednego programu. Procesory VLP mogą być w łatwy sposób łączone siecią. Zobacz... TU OBRAZKI! Tworzy się w ten sposób klaster (ang.cluster, czyli grono) maszyn wirtualnych. Program P może tworzyć obiekty procesów i alokować je na różnych procesorach. Może też je wznawiać.

Przykład. Zacznijmy od krótkiego przykładu

```
program first;
  #include "classes/gui.inc";
  ┌ (* deklaracja procesu *)
  │  unit writer: process(node: integer, s: string);
  │    var i: integer, A: arrayof char;
  │   begin
  │    A := unpack(s);
  │    return;
  │    for i := lower(A) to upper(A)
  │    do
  │     write(A(i));
  │    od
  └  end writer;
  var w1, w2, w3: writer;
begin
  ┌ (* tworzenie obiektów procesu writer *)
  │   w1 := new writer(0, "Bonjour tristesse Bonjour tristesse");
  │   w2 := new writer(0, "Welcome to London");
  │   w3 := new writer(0, "Willkomen in Berlin");
  ├ (* uruchamianie obiektów w1, w2, w3 *)
  │   resume(w1);
  │   resume(w2);
  │   resume(w3);
  ├ (* oczekiwanie na klawisz 'f'  *)
  │   pref GUI block
  │   begin
  │    while GUI_KeyPressed =/= ord('f') do
  │    od
  └  end (* block *);
end
```

Co się składa na ten program? Elementy programu istotne dla obliczeń współbieżnych zaznaczono kolorem czerwonym. Deklaracje w tym programie sprowadzają się do:

1° deklaracji procesu writer, i

2° deklaracji trzech zmiennych w1, w2, w3 typu writer.

Działanie programu first polega na:

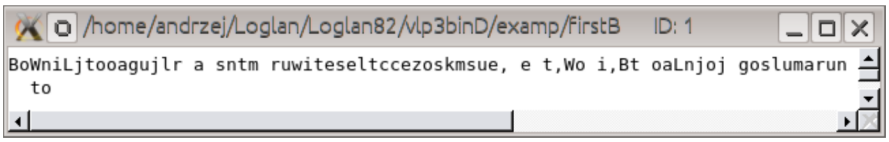
1° stworzeniu trzech obiektów procesu writer i przypisaniu ich do zmiennych w1, w2, w3. Każdy z tych obiektów jest alokowany na tym samym procesorze co program. (Decyduje o tym wartość pierwszego parametru równa 0.) Utworzone obiekty pozostają w stanie Pasywny.

2° Następnie program uruchamia każdy obiekt typu writer wykonując polecenia resume. Każdy z obiektów w1, w2, w3 zaczyna wykonywać swoje zadania niezależnie

od innych, drukując na ekranie odpowienią literę z zadanego tekstu. Łącznie z programem głównym wykonywane są cztery wątki obliczeń sekwencyjnych.

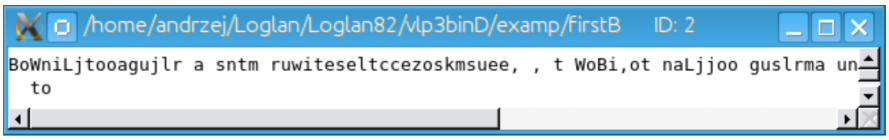
3° Program główny oczekuje, że zostanie naciśnięty klawisz 'f'. Potem program kończy działanie.

Niedeterminizm obliczeń niesekwencyjnych. Obserwując zachowanie tego prostego programu możemy dokonać kilku spostrzeżeń. Po pierwsze, na ekranie pojawi się mieszanina liter.



Wynik programu first (pierwsza próba)

Dlaczego tak się dzieje? To proste, wszystkie trzy obiekty procesów drukują na tym samym ekranie.
Po drugie, jeśli powtórzysz wykonywanie tego programu, kilka lub kilkanaście razy, to zaobserwujesz, że wzór przemieszania liter może się zmienić.



Wynik programu first (druga próba)

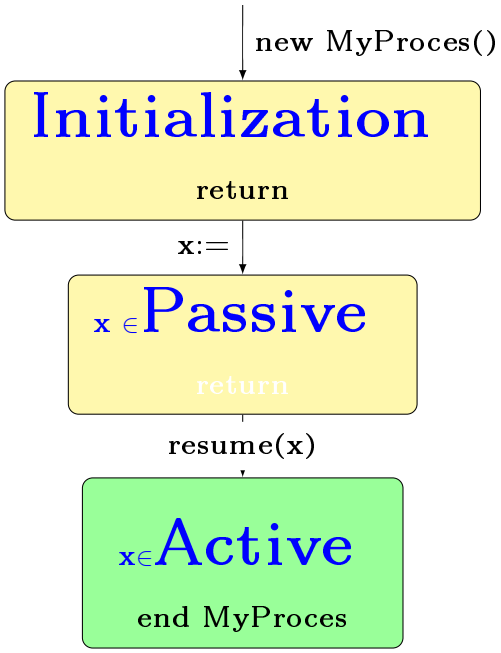
W tym przypadku już w drugiej próbie uzyskaliśmy inny obraz. Ale to ten sam program i nie jest zależny od danych. Dlaczego więc są różnice w wydruku? Aha, obliczenia współbieżne nie są deterministyczne! Ta obserwacja ma doniosłe konsekwencje. W programowaniu sekwencyjnym analizę własności programów opieramy na zasadzie determinizmu: wielokrotne wykonanie programu z tymi samymi danymi początkowymi da zawsze ten sam wynik. Ale jak widać w programowaniu współbieżnym zasada determinizmu nie obowiązuje. Najważniejszą konsekwencją tego co zaobserwowaliśmy, jest całkowita nieprzydatność tzw. testowania programu. Cóż nam bowiem po tym, że podczas testów, na naszych komputerach program współbiezny lub rozproszony nie dał powodu do niepokoju. W nieznacznie zmienionej konfiguracji, np. przy innej temperaturze, lub gdy komputery działają z trochę inną prędkością, są rozmieszczone inaczej niż podczas testów, nieczekiwany błąd może sie pojawić jak diabeł z pudełka.
Oznacza to też, że własność stopu ma dwa oblicza: możemy pytać czy każde obliczenie programu P jest skończone? Ale

możemy też zastanawiać się czy istnieje chociaż jedno obliczenie skończone tego programu? Zamiast zadawać sobie pytanie czy po zakończeniu (deterministycznego) programu P zachodzi warunek β, trzeba teraz pytania formułować inaczej: czy koniecznie po zakończeniu programu M zachodzi warunek β i/lub czy możliwe jest że po zakończeniu programu M zachodzi warunek β. Język opisu zjawisk semantycznych występujących w programowaniu współbieżnym wymaga modalności: konieczne i możliwe. Ujmijmy to jeszcze inaczej: w obliczeniach determinstycznych jeśli jest możliwe, że po wykonaniu programu P zachodzi warunek α, to jest konieczne, ze po wykonaniu tego programu zachodzi warunek α.
Do zjawiska niedeterminizmu będziemy jeszcze powracać. W sekcji 7 omawiamy matematyczne modele obliczeń współbieżnych i równoległych.
Niedeterminizm obliczeń wspólbieżnych (równoległych i rozproszonych) jest OGRANICZONY, tzn. nie uda się otrzymać efektu podobnego do instrukcji niedeterministycznego wyboru pomyślanej jako realizacja kwantyfikatora egzystencjonalnego (por. [HKT02]). Dlaczego tak jest?
1° Zasada MAX maksymalnego niekonfliktowego wyboru eliminuje unfairness
2° Wykonywanie instrukcji atomowej nie może trwać dłużej niz to jest dla niej przewidziane! (mowa o drugim niedeterministycznym wyborze instrukcji do zakończenia) w modelu MAX.

Załóżmy, że utworzono sieć obejmującą conajmniej trzy komputery (mogą się one znajdować w Paryżu, Londynie i Berlinie). Węzłom sieci nadano numery 2, 12 i 13. Jeśli w programie first.log zamienimy instrukcje tworzenia obiektów procesu writer w ten sposób

```
[ (* tworzenie obiektów procesu writer *)
  w1 := new writer(12, "Bonjour tristesse Bonjour tristesse");
  w2 := new writer(2, "Welcome to London");
  w3 := new writer(13, "Willkomen in Berlin");
```

to każdy pisarz wydrukuje swój tekst na swoim ekranie. Nie będzie efektu przemieszania i trudno mówic o niedeterminizmie tak prostych obliczeń. Tzn. nie warto go rozpatrywać.
Ale nadal możemy zaobserwować niedeterminizm obliczeń rozproszonych. Trzeba jednak napisać nieco dłuższy program.
Drukowanie możemy zlecić jednemu obiektowi procesu ekran. (Zechciej napisać taki proces.) Możemy utworzyć obiekt e typu ekran i przekazać go jako parametr obiektom w1, w2, w3. Obiekty procesu writer przekazywać będą zlecenie drukowania obiektowi e typu ekran. Znowu może dojść do przemieszania liter. Można też drukowanie uporządkować stosując narzędzie znae jako semafor.



Rysunek 1. Diagram (scenariusz) stanów obiektu procesu

2. Składnia i semantyka

Poniżej przedstawiamy składnię i semantyke poleceń dotyczących procesów. Zechciej zauważyc jak niewiele potrzeba, by rozszerzyć język obliczeń sekwencyjnych tak by można było programować obliczenia współbieżne i rozproszone. Narzędzia służące programowaniu obliczeń współbieżnych i rozproszonych są naturalne, w tym sensie, że są zgodne z koncepcją programowania obiektowego.
Język $\mathcal{L}_{10}$ jest rozszerzeniem poprzedniego języka $\mathcal{L}_9$.

$$\mathcal{L}_9 \subsetneq \mathcal{L}_{10}$$

Deklaracje. W tym rozszerzeniu języka znajdujemy deklaracje procesów – modułów i deklaracje zmiennych typu proces. Deklaracja modułu process jest podobna do deklaracji klasy czy współprogramu, z tym, że słowo class należy zastąpić przez process. W treści takiego modułu mogą wystąpić instrukcje zarządzające obiektami procesów (są one wyliczone poniżej).

Definicja 25.4. Niech P będzie identyfikatorem, params – listą parametrów formalnych. Deklaracja modułu procesu ma następującą postać

unit P: process(params); $\mathbb{D}$ begin $\mathbb{I}$ end

Moduł procesu może rozszerzać (tj. dziedziczyć z) modułu klasy lub współprogramu. Niech K będzie nazwą zadeklarowanej klasy.

$$\text{unit K: class(params}_K\text{); } \mathbb{D}_K \text{ begin } \mathbb{I}_K \text{ end K;}$$
$$\text{unit P: K process(params}_P\text{); } \mathbb{D}_P \text{ begin } \mathbb{I}_P \text{ end P}$$

Możliwa jest także sekwencja odwrotna w której moduł klasy dziedziczy z modułu procesu

$$\text{unit P: process(params}_P\text{); } \mathbb{D}_P \text{ begin } \mathbb{I}_P \text{ end P;}$$
$$\text{unit K: P class(params}_K\text{); } \mathbb{D}_K \text{ begin } \mathbb{I}_K \text{ end K}$$

wynik operacji rozszerzania jest modułem procesu.

W Loglanie'82 przyjęto następujące dodatkowe ograniczenia na postać deklaracji modułu procesu:

- Deklaracja procesu musi być zawarta bezpośrednio wśród deklaracji programu głównego (tj. musi być deklaracją top-level).
- Pierwszy parametr formalny na liście params parametrów formalnych modułu procesu musi być typu integer. Wartość pierwszego parametru aktualnego w wyrażeniu new generującym proces, decyduje na którym procesorze będzie alokowany obiekt procesu[1].
- Parametry formalne procesu mogą być typu pierwotnego lub typu proces. Niedopuszczalne jest by parametr aktualny był typu tablicowego lub obiektowego (za wyjątkiem obiektów procesów).[2]
- W module procesu nie występują zmienne nielokalne. To ograniczenie wynika w naturalny sposób, gdy zrozumiemy, że obiekty procesu mogą być alokowane na oddzielnych komputerach (połączonych siecią).

Definicja 25.5. Niech P będzie nazwą modułu procesu. Napis postaci

var a: P;

jest deklaracją zmiennej a typu proces P.

Deklaracje zmiennych tego samego typu P można łączyć, np.
var a,b: P;
jest deklaracją dwu zmiennych a i b typu proces P.

---

[1] Jeśli moduł procesu dziedziczy z klasy K, to ten parametr będzie poprzedzony parametrami aktualnymi konstruktora klasy K.

[2] To wymaganie można łatwo usunąć w nowej wersji kompilatora i zastąpić przez wymóg przesyłania kopii obiektu.

Instrukcje. W języku $\mathcal{L}_{10}$ pojawia sie kilka nowych instrukcji atomowych. Natomiast zachowane zostają konstrukcje programotwórcze: if, for, while oraz średnik.

Definicja 25.6. Następujące napisy są instrukcjami atomowymi w języku $\mathcal{L}_{10}$

- instrukcje atomowe języka $\mathcal{L}_9$ oraz
- instrukcja utworzenia obiektu procesu i przypisania go do zmiennej, np.
  x := **new** *Myproces((*node=*) 8, ...)* ;
- polecenie resume, np.
  resume(*x*)
- polecenie obcego wywołania metody w obiekcie procesu np.
  call x.meth(...) [3]
- polecenie accept, np.
  accept meth1, meth3
- polecenie enable, np.
  enable *meth1, meth2* ;
- polecenie disable, np.
  disable *meth3* ;
- polecenie stop
  stop;
- polecenie return
  w metodach zadeklarowanych wewnątrz modułu process polecenie return może przyjąć postać jak w poniższym przykładzie
  return disable *meth2, meth4* enable *meth1, meth5*;

Klaster procesorów VLP. Procesor VLP jest wirtualną maszyną loglanowską. Możesz mu zlecić różne zadania. zrzut ekr. Procesor VLP może łączyć się z innymi procesorami. W ten sposób możesz utworzyć klaster (ang. cluster).

Deklaracje procesów. Deklaracje są niezbędne by można było utworzyć obiekty procesów.
Deklaracja modułu procesu tylko nieznacznie różni się od deklaracji klasy:

i) zamiast słowa class należy napisać process,
ii) trzeba pamiętać, że wszystkie lokalne zmienne procesu są prywatne,
iii) pierwszy parametr formalny (tj. pierwszy argument) deklaracji jest przeznaczony na przekazanie numeru wirtualnego loglanowskiego procesora w wirtualnym klastrze loglanowskich procesorów (zob. poniżej),

[3] gdy wartością zmiennej x jest obiekt procesu to polecenie call wykonywane jest według protokołu obcego wołania metody – alien call.

Deklaracje zmiennych typu process niczym nie różnią się od deklaracji zmiennych obiektowych. Wartościami takich zmiennych są obiekty procesów.

Polecenia specyficzne procesów. Parę słów o poleceniach. Obiekt procesu tworzymy wykonując polecenie *x*:=**new** *Myproces(p, ...)*. Zwracamy uwagę na to, że pierwszy argument p wyznacza procesor na jakim będzie alokowany obiekt procesu. Typ tego argumentu musi być liczbą całkowitą, nieujemną.
Jeśli wartość p jest równa zero to nowotworzony obiekt procesu jest alokowany na tym procesorze, który wykonuje instrukcję. Ten sam procesor będzie wykonywać instrukcje procesu p po wznowieniu przez polecenie resume(p). Wartość argumentu p różna od zera wskazuje na numer procesora VLP w klastrze wirtualnych procesorów loglanowskich. Nowoutworzony obiekt procesu jest Pasywny. Tzn. jest gotów do rozpoczęcia obliczeń według swojego programu.

Zmiana stanu obiektu procesu na Aktywny tj. uaktywnienie lub wznowienie obiektu procesu x, ma miejsce gdy wykonano polecenie resume(x). Zwróć uwagę na to, że w odróżnieniu od obiektów współprogramów – wiele obiektów procesów może być naraz w stanie Aktywny.
Jeśli polecenie wykonywane jest na procesorze o numerze 7, a obiekt jest alokowany na procesorze 2 to polecenie aktywacji jest przekazywane temu drugiemu procesorowi – w efekcie obiekt jest wykonywany na tym procesorze na którym był alokowany.

Obiekt aktywny może wykonywać wiele poleceń, które poznaliśmy wcześniej: instrukcje przypisania, warunkowe, iteracji etc. Instrukcje te powodują zmianę stanu obiektu procesu. Ważnym elementem stanu obiektu procesu jest dostępność (public) lub niedostępność (private) metod obiektu. Na początku wszystkie metody obiektu procesu są ukryte (private). Polecenie enable met1, met4 udostępnia wyliczone metody innym obiektom. A dokładniej dodaje te metody do zbioru już wcześniej udostępnionych (enabled) metod. Można sobie wyobrazic, że elementem stanu obiektu procesu jest zbiór metod udostepnionych, oznaczamy go MASK. Polecenie disable met2, met3 usuwa nazwy tych metod ze zbioru MASK. Możemy uważać, że odtąd metody te są prywatne i niedostępne z zewnątrz.

Obce wołanie metody. Obiekty procesów mogą współdziałać. Podstawą współdziałania jest oryginalny protokół obcego wołania metod (ang. alien call) wymyślony przez Bolka Ciesielskiego w r. 1988 [Cie89, SW91]. Nie spotkaliśmy podobnej

koncepcji w żadnym innym języku programowania. Podczas obcego wołania metody współpracują ze sobą dwa obiekty procesów. Jeden proces zawiera polecenie wywołania metody, sama metoda jest zadeklarowana w drugim procesie.
Składnia obcego wołania metody ma dwa składniki, jeden w obiekcie y procesu wywołującym metodę methd zadeklarowaną w innym obiekcie x procesu: call x.meth. Drugi składnik występuje w wywoływanym obiekcie x i jest to albo polecenie accept ..., methd, ...; albo polecenie enable methd;. W pierwszym przypadku mówimy o synchronicznym wykonaniu poleceń

y:: call x.methd(args) || x:: accept methd

W drugim przypadku mamy do czynienia z asynchronicznym, także wspólnym, wykonaniem polecenia call x. methd(.) przez oba obiekty procesów.

$$\text{y:: call x.methd(args) } \| \text{ x::} \left\{ \begin{array}{l} \text{enable methd;} \\ \ldots; \\ \ldots; \\ \ldots; \end{array} \right\}$$

Jak wytłumaczyć nazwy synchroniczna lub asynchroniczna odmiana protokołu alien call?

s) zsynchronizowane wywołanie metody meth zadeklarowanej w obiekcie x pewnego procesu, ma miejsce gdy nastąpi spotkanie instrukcji wywołania metody z obcego obiektu procesu z instrukcją accept w tym obcym obiekcie.

a) asynchroniczne wywołanie metody meth zadeklarowenej w obiekcie x pewnego procesu, ma miejsce gdy obiekt ten umożliwił przerwanie wykonywania swego ciągu instrukcji wykonując polecenie enable methd. Przerwanie takie może nastapić w bliżej niekreślonym momencie, zawsze jednak po zakończeniu wykonywania całej instrukcji atomowej (tzn. przy średniku).

Realizacja protokołu alien call. Obiekt y pewnego procesu wykonuje polecenie call x.methd(args):

(1) Sprawdzane są następujące warunki: czy obiekt x istnieje? czy jest Aktywny? czy metoda methd jest udostępniona (enabled)? Gdy te warunki są spełnione przechodzi się do kolejnego punktu. Jeśli nie to obiekt y oczekuje na ich spełnienie.

(2) Obiekt y przekazuje argumenty obiektowi x i oczekuje na sygnał zakończenia protokołu i na wyniki od obiektu x.

(3) Obiekt x
- zapamiętuje zbiór udostępnionych metod (zapamiętuje zmienną systemowa MASK),

- zamyka dostęp do wszystkich metod obiektu (MASK :=∅ tj. wszystkie metody obiektu są zablokowane),
- tworzony jest rekord aktywacji metody methd,
- wykonywana jest treść tej metody,
- po zakończeniu obiekt x odsyła wyniki – argumenty out,
- odtwarza zbiór metod dostepnych przed rozpoczeciem protokołu alien (odtwarzana jest wartość zmiennej MASK),
- wykonywane jest polecenie return ... enable ... disable ...;
- obiekt x wykonuje swoją kolejną instrukcję

(4) obiekt y kontynuuje swoje obliczenie

Komentarza wymaga polecenie return enable ... disable ...; Polecenia tego nie można podzielić, gdyż w takim przypadku część enable ... disable ... nie zostanie wykonana lub jej efekt zostanie zmarnowany. Zastanów sie dlaczego tak by było?
Na czym polega wykonanie polecenia accept met3, met3?

(1) do zbioru udostępnionych metod dołączane sa nazwy metod met2 i met3,
(2) jesli jakiś inny proces wywołał metodę znajdująca sie w zbiorze metod udostępnionych (enabled) to rozpoczyna się wykonywanie tej metody, patrz wyżej. (Zauważ, proces x jest aktywny i spełnione są warunki ..)
(3) w przeciwnym przypadku proces oczekuje na wywołanie którejś z metod udostępnionych.

Można to wysłowić inaczej proces x nie rozpocznie wykonywania instrukcji po instrukcji accept dopóki nie zostanie zrealizowane jedno z poleceń call ...

3. Program Rozmowa

Przedstawiamy poniżej program umożliwiający rozmowę dwu osób poprzez sieć. Dla działania programu niezbędne jest by w klastrze połączonych wirtualnych procesorów loglanowskich znajdowały sie conajmniej dwa procesory VLP. Dla potrzeb tego programu przyjmijmy, że procesory VLP działają na węzłach o numerach 2 i 12. Program główny wykonywany jest na węźle nr 2.
Struktura tego programu jest bardzo prosta:

- program główny tworzy dwa obiekty rozmówca, na węzłach 0 i 12. Obiekt procesu alokowany na węźle 0 jest faktycznie alokowany i wykonywany na węźle nr 2. Tworzone są też dwa obiekty procesu ekran na tych samych węzłach.

- obiekt procesu rozmówca odbiera znaki naciśnięte na klawiaturze i wysyła je do obu ekranów: lokalnego i zdalnego,
- obiekt typu ekran jest serwerem mastępujących usług:
  - wydrukuj przysłany znak w części lokalnej ekranu,
  - wydrukuj przysłany znak w części zarządzanej przez drugiego rozmówcę.
  - skasuj ostatnio wydrukowany znak (Backspace) (lokalnie i zdalnie),
  - zakończ pracę gdy naciśnięto klawisz Esc (escape) i prześlij odpowiedni sygnał do pozostałych procesów.
    To jest sposób na realizację rozproszonego zakończenia całego programu – wszystkich jego procesów.

Program wykorzystuje klasę ANSI, w której znajdują się m.in. następujące procedury:

```
unit ANSI: class;
    unit GotoXY: procedure(wiersz,kolumna:integer);...
    unit SetColor: procedure(color:integer); ...
    unit SetBackground: procedure(color :integer); ...
    unit Bold: procedure; ...
    unit Normal: procedure; ...
end ANSI;
```

Wykonanie instrukcji call GotoXY(3,10) spowoduje, że drukowanie będzie kontynuowane poczynając od 10-tej kolumny w trzecim, od góry, wierszu ekranu. Znaczenie pozostałych operacji wykonywanych przez te procedury jest łatwe do odgadnięcia.

Proces ekran. Najpierw tworzone są dwa obiekty procesu ekran. Argumentem ekranu jest rozmówca, który nim zarządza. Ale w trakcie tworzenia obiektu ekran nie znany jest obiekt rozmowca. (Nie istnieje jeszcze żaden taki obiekt.) Aby temu zaradzić zarządziliśmy co następuje:

- obiekt ekran posiada metodę zarejestruj, która przyjmuje argument typu rozmowca i przypisuje go do lokalnej zmiennej rozm obiektu. Pierwszą instrukcją obiektu ekran jest accept zarejestruj. W ten sposób obiekt oczekuje na przysłanie mu nazwy właściciela.
- Program główny dopilnuje by po utworzeniu obiektów procesu rozmówca przekazać ich nazwy do obiektów procesu ekran. Zob. poniżej.

Każdy obiekt procesu ekran, oprócz metody rejestruj, wyposażony jest w następujące metody:

- odbiór_lokalny – wydrukuj znak przysłany przez obiekt rozmówca w polu treści tworzonych lokalnie,
- odbiór_zdalny – wydrukuj znak przysłany przez obiekt drugiego rozmówcy w polu treści tworzonych zdalnie,
- kasuj_lokalny – skasuj ostanio wydrukowany znak w polu treści lokalnych,
- kasuj_zdalny – skasuj ostanio wydrukowany znak w polu treści przysłanych,
- koniec – zakończ działanie tego obiektu procesu – tj. przypisz zmiennej knc wartość true.

Działanie obiektu ekran sprowadza się do oferowania tych metod – tj. obiekt procesu ekran jest serwerem usług. Dopóki not knc powtarzaj instrukcję accept jakakolwiek z pięciu wymienionych metod.

Działanie obiektu procesu rozmowca jest bardzo proste:

- podczas inicjalizacji (konstruktor) fizyczny ekran dzielony jest na część do drukowania znaków nadsyłanych (górna część ekranu) i część do drukowania znaków wpisywanych na lokalnej klawiaturze,
- jeśli przeczytany znak to backspace (znak nr 8) to cofnij się o jeden znak w tej samej linii,
- jeśli przeczytany znak to Esc (znak nr 27) to zakończ działanie i wyślij polecenie koniec do obu ekranów lokalnego i zdalnego,
- w pozostałych przypadkach wyślij polecenie drukowania przeczytanego znaku do ekranu lokalnego i ekranu zdalnego.

Deklaracja procesu ekran:

```
unit ekran : ANSI process(node:integer;rozm:rozmowca);

  unit rejestruj : procedure(r:rozmowca);
  begin
    rozm := r;
  end rejestruj;
  unit odbior_zdalny : procedure(s:char);
  begin
    call GotoXy(hy, hisline);
    call Bold;
    writeln(s);
    hy := hy+1;
    if s=chr(13) then hisline := hisline + 1;hy:=1; fi;
    if hy = 80 then hy := 1; hisline:=hisline+1 fi;
    call Normal;
    if hisline = 12 then hisline := 1;
    fi;
  end odbior_zdalny;
  unit odbior_lokalny : procedure(s:char);
  begin
     call GotoXY(my, myline);
     write(s);
     my := my+1;
     if s=chr(13) then myline := myline + 1;my:=1; fi;
     if my = 80 then my := 1; myline := myline+1 fi;
     if myline = 25 then myline := 13;
     fi;
  end odbior_lokalny;
  unit kasuj_lokalny : procedure;
  begin
     my := my - 1;
     if my<0 then my := 0 fi;
     call GotoXY(my, myline);
     write();
  end kasuj_lokalny;
  unit kasuj_zdalny : procedure;
  begin
     hy := hy - 1;
     if hy<0 then hy := 0 fi;
     call GotoXY(hy, hisline);
     write();
  end kasuj_zdalny;
  unit koniec : procedure;
  begin
     knc := true;
  end koniec;

  var knc:boolean, myline,hisline,my,hy:integer, s:char, name:string;
begin
  knc := false;
  myline := 13;
  hisline := 1;
  my := 1;
  hy := 1;
```

Poniżej cytujemy treść procesu rozmowca.

```
unit rozmowca : IIUWgraph process(node: integer;el,ez: ekran);
  (* z klasy IIUWgraph wykorzystamy funkcję  inkey *)
  var knc:boolean, s:char, i:integer;
  [ unit koniec :  procedure;
    |   begin
    |     knc := true;
    |   end koniec;
begin
  knc := false;
  return;
  [ enable koniec;
    |   while not knc
    |   do
    |     i := inkey;
    |     if i<>0 then
    |       s := chr(i);
    |       if i=8 then
    |         call el.kasuj_lokalny;
    |         call ez.kasuj_zdalny;
    |       else
    |         call el.odbior_lokalny(s);
    |         call ez.odbior_zdalny(s);
    |       fi;
    |       if i = 27 then
    |         call ez.koniec;
    |         call el.koniec;
    |       fi;
    |     fi;
    |   od;
end rozmowca;
```

A tu jest program główny.
W odpowiednie miejsca należy wstawić deklaracje modułów ekran i rozmówca. Jeśli chcesz to wykorzystaj polecenie kompilatora #include.

```
program talk;
   #include "classes/ansi.inc" (* dołącz tekst klasy ANSI *)
   unit ekran: process ...
   unit rozmowca: process ...
   var p1,p2:rozmowca, e1,e2:ekran;
begin
   e1 := new ekran(0,none);
   resume(e1);
   e2 := new ekran(12,none);
   resume(e2);
   p1 := new rozmowca(0,e1,e2);
   p2 := new rozmowca(12,e2,e1);
   call e1.rejestruj(p1);
   call e2.rejestruj(p2);
   resume(p1);
   resume(p2);
end
```

4. Producenci i konsumenci

Przedstawiamy analizę kilku odmian zadania producent – konsument. Rozwiązania wykorzystują mechanizm obcego wołania procedury (ang. alien call). Analiza poprawności rozwiazań staje się łatwiejsza dzięki alien call.

4.1. Zadanie - wielu producentów, jeden konsument. Jest to jeden z najczęściej omawianych problemów programowania współbieżnego i rozproszonego. Zakładamy, że pewna liczba aktywnych obiektów procesu Producent wytwarza dane – obiekty i składa je w kontenerze k. Obiekt aktywny procesu Konsument pobiera po kolei dane z kontenera i je przetwarza.
Kontener – kolejka – jest własnością konsumenta
Należy zapewnić by żaden obiekt nie był przetwarzany dwukrotnie, by żaden obiekt nie przepadł i by informacja składana w kontenerze nie uległa deformacji np. podczas równoczesnego odczytywania i zapisu danych.

4.2. Rozwiązanie z wykorzystaniem alien call. Proces Producent:
while needed do
produkuj ;
wyślij do Konsumenta
od

proces Konsument
ma kontener k - kolejka, metodę odbierz i działa w pętli

```
unit queue:class(type element; size:integer);
   (* Ta pomocnicza klas implementuje kolejkę FIFO.
   Element jest nazwą typu elementów kolejki *)
   unit insert:procedure(e:element); ...
   (* wstaw element do kolejki *)
   unit delete:function:element; ...
   (* podaj pierwszy elemnt i usuń go z kolejki *)
   unit isempty:function:boolean; ...
   (* sprawdź czy kolejka jest pusta *)
   unit isfull:function:boolean; ...
   (* sprawdź czy kolejka jest pełna *)
end queue;
```

Poniżej przedstawiamy moduł procesu Konsument.

```
unit Konsument:process(node:integer);
   var Q: queue, f: product;
   ⌈ unit deposit: procedure(f:product);
   |    begin
   |       call Q.insert(f);
   |       if Q.isfull
   |       then
   |          return disable deposit
   |       fi;
   |    end deposit;
   ⌊

begin
   Q := new queue(product, 50);
   return;
   ⌈ do
   |       (* withdraw an element if any *)
   |       (* await for an element if empty *)
   |       disable deposit;
   |       if Q.isempty
   |       then
   |          accept deposit
   |       fi;
   |       f := Q.delete;
   |       enable deposit;
   |       (* consumption of the product f *)
   |       ...
   ⌊    od
end Konsument;
```

Deklaracja modułu procesu Producent wygląda tak:

```
unit Producent: process(node: integer, k: Konsument);
  var p: product;
begin
    return;
[ do
      (* produce product f *)
      ...
      call k.deposit(p)
      (* producer awaits for the completion of call k.deposit(.) *)
[ od
end Producent;
```

A oto program główny, zawiera on moduły procesów Konsument i Producent. Zakładamy, że typ product należy zastąpić przez jedno ze słów kluczowych: integer, real, string lub char lub Boolean. Czyli jeden z typów pierwotnych języka. Utwórz obiekt k procesu Konsument i pewną liczbę obiektów procesu Producent. Wykonaj operację resume() na każdym z tych obiektów. Każdy z nich staje sie Aktywny Wszystkie obiekty procesów działają niezaleznie.

```
program Producenci_i_Konsument;
  unit Konsument: process ...
  unit Producent: process ...
  unit Queue: class ...
  var k: Konsument, p:Producer;
begin
  k := new Consument(0); resume(k);
  for i := 1 to noProducers do
    p := new Producer(nr, k);
    resume(p)
  od;
  (* zakoncz *)
end program
```

4.3. Analiza. Następujące pytania nasuwają się podczas czytania tego programu:

(1) Przypuśćmy, że kilka obiektów procesu Producent równoczesnie przesyła swe produkty do obiektu k procesu Konsument wykonując polecenia dwu producentów p i q, np.

p:: **call** k.deposit(g) || q:: **call** k.deposit(f)

Czy możemy zapewnić, że żadne z tych żądań nie będzie zgubione lub, że dojdzie do równoczesnego wykonywania operacji insert na kolejce i , że w ten sposób

w kolejce pozostanie tylko jeden z dwu wstawianych produktów?

(2) Przypuśćmy, że obiekt k procesu Konsument pobiera produkt z kolejki i że równocześnie jeden (lub więcej) producent p wstawia produkt do kolejki.

k:: f := Q.delete || p:: **call** k.deposit(g)

Mogłoby to spowodować błąd (błędy).
Odpowiedzi na te pytania przynosi następujący

Lemat 25.1. Żadne żądanie nie zostanie pominięte. Nie dojdzie do równoczesnego wykonania dwu instancji procedury *deposit* (na rzecz dwu różnych producentów). Nie dojdzie do równoczesnego pobierania elementu z kolejki przez konsumenta i wstawiania innego elementu do kolejki przez producenta.

Dowód. Należy rozpatrzyć pięć przypadków:

A) Nie dojdzie do równoczesnego wykonywania dwu instancji procedury deposit ponieważ protokół obcego wołania metody zaczyna się od zapamiętania stanu metod udostępnionych (enabled) i od ukrycia (disable) wszystkich jego metod. Tzn. jeśli konsument wykonuje operację deposit na zlecenie jednego z producentów, to wykonywanie tej procedury nie będzie przerwane przez wykonywanie takiej procedury na rzecz innego producenta.

B) Nie dojdzie do równoczesnego pobierania elementu z kolejki przez konsumenta i wstawiania nowego elementu do kolejki przez jakiegoś producenta, ponieważ instrukcje pobierania sa poprzedzone instrukcją disable deposit.

C) W sytuacji przepełnienia kolejki metoda deposit kończąc wykonuje polecenie return disable deposit. Oznacza to, że kolejne żądania call k.deposit(p) wstawienia elementu do kolejki zostaną wstrzymane do czasu gdy w kolejce zwolni się miejsce.

D) W sytuacji pustej kolejki obiekt k procesu Konsument rozpocznie wykonywanie instrukcji accept deposit, która zakończy się, gdy kolejka będzie nie pusta.

E) Nie dojdzie do zagłodzenia jakiegoś producenta - zapewnia to maszyna wirtualna VLP. Wstawiane elementów do kolejki wykonywane jest według kolejności zgłoszeń.

□

4.4. Przekazywanie obiektu z jednego do drugiego procesu. W powyższym przykładzie elementami przekazywanymi z procesu do procesu mogą byc tylko wartośći typów prostych. W

obecnej wersji Loglanu nie można przekazać kopii obiektu z jednego obiektu aktywnego do drugiego[4], to możemy posłużyć się następującą protezą:
proces Producent wywołuje u Konsumenta metodę "przekazuję" z wszystkimi parametrami potrzebnymi do stworzenia kopii obiektu o przez Konsumenta, np. tak:

call k.przekazuję(<params>)

a metoda ta wykonuje polecenie
aux := new Obiekt(<params>) i składuje aux w kontenerze.
Jest to proteza, TAK(!). Zechciej porównać ten scenariusz z koncepcją serializacji i "marshalling" w Javie RMI.

4.5. Jeden producent, wielu konsumentów. W tym przypadku kontener – to jest znowu kolejka – jest własnością producenta. To konsumenci adresują swoje potrzeby do producenta. Rozwiązanie jest symetryczne w stosunku do poprzedniego.

4.6. Wielu producentów, wielu konsumentów. W przypadku gdy jest wielu producentów i wielu konsumentów, warto utworzyć aktywny obiekt procesu Bufor, zarządzający kolejką, z metodami deposit – dla producentów i withdraw – dla konsumentów. Poniżej przytaczamy deklarację procesu Bufor:

```
unit Buffer: process(node:integer);
  var q:kolejka;
  unit deposit: procedure();
  ...
  end deposit;

  unit withdraw: function():product;
  ...
  end withdraw
begin
  q := new kolejka();
  return;
  do
    if q.isempty() then accept deposit
    else
      if q.isfull() then accept withdraw
      else
        accept deposit, withdraw
      fi
    fi
  od;
end Buffer;
```

[4]w kolejnej wersji języka należy wprowadzic wyrażenie copy(o) – kopiuj obiekt

Program główny wygląda teraz tak

```
program Konsumenci_i_Producenci;
  unit Konsument: process ...
  unit Producent: process ...
  unit Buffer: process ...
  var k: Konsument, p: Producent, b: Buffer, i: integer;
  const noProducers=29, noConsumers=15;
begin
  (* create one Buffer and some number of Producers and Consumers *)
  (* put them into action *)
  b := new Buffer(0); resume(b);
  for i := 1 to noProducers do
    p := new Producent(nr, b);
    resume(p)
  od;
  for i := 1 to noConsumers do
    k := new Konsument(nr, b);
    resume(k)
  od;
  ...
  (* zakończ *)
end program
```

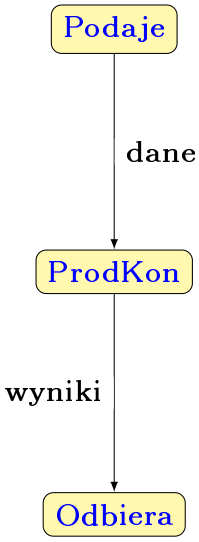
Analiza. W tym programie obiekt b zachowuje się jak monitor. Pełni rolę dostawcy usług deposit i withdraw. Ponadto dopilnuje by w sytuacji pustego bufora konsumenci poczekali, aż któryś producent wstawi produkt do bufora. I w analogicznej sytuacji gdy bufor jest pełny, nasz monitor dopilnuje by producenci poczekali na to by któryś z konsumentów pobrał produkt z kolejki tworząc miejsce na następny produkt.
Z własności obcego wołania metod wynika, że nie dojdzie do równoczesnego wykonywania metod deposit i withdraw. Podsumowując stwierdzamy

Lemat 25.2. W systemie producentów i konsumentów

(i) każde żądanie zostanie obsłużone w odpowiednim czasie,

(ii) nie dojdzie do pomieszania komunikatów.

Gdy producenci tworzą duże obiekty. np. z dużymi tablicami.
Proponuję by wtedy nie przekazywać kopii takich obiektów, lecz by były to obiekty aktywne (dokładniej – obiekty procesów, niekoniecznie z własnym wątkiem). Mogą to być serwery



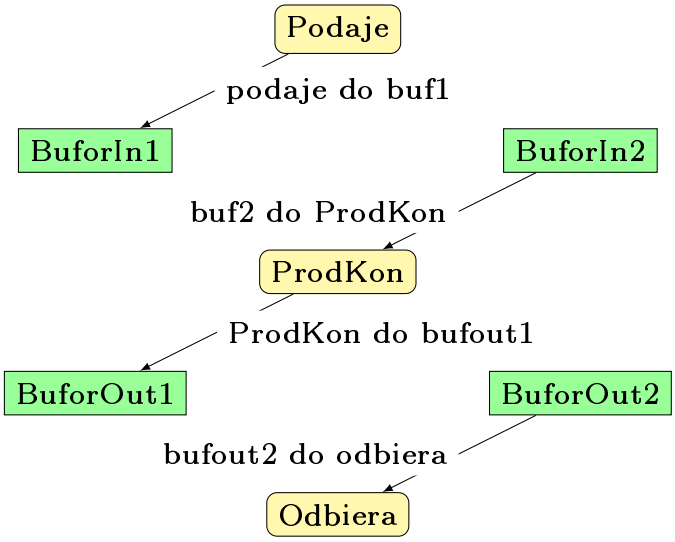
Rysunek 2. Proces ProdKon i dwaj partnerzy

usług i danych. Wtedy przekazywanie od producenta do konsumenta – poprzez bufor – jest łatwe: przekazujemy wskaźnik do takiego obiektu aktywnego. Bufor może stworzyć strukture danych jaką programista mu wybierze – tablice lub drzewo. To w niczym specjalnie nie przeszkadza. Można mieć
var A: arrayof Produkt;
gdzie produkt jest processem np.
unit Produkt: process(...); ... end Produkt;

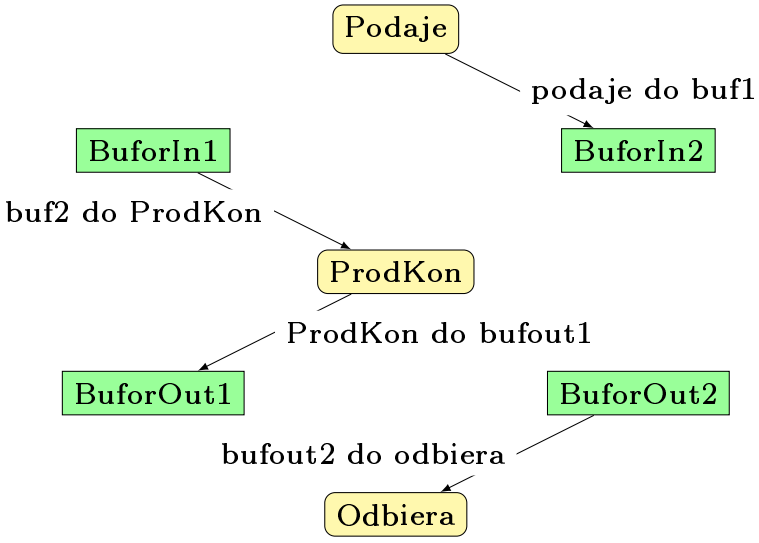
A co z obliczeniami rôwnoległymi? Nie mamy implementacji dla obliczeń rôwnoległych gdy obiekty procesôw są alokowane na procesorach/rdzeniach i wykonują się we wspôlnej pamięci.
Podejrzewam, że stosuje się wtedy rozwiązania właściwe dla obliczeń współbieżnych. Może to i dobre podejście.
Czy rozproszenie może imitować rôwnoległość?

Gdy proces jest i konsumentem i producentem. Rozważmy sytuację, w której jeden proces PASS wykonuje się na szybkim procesorze i pobiera (powoli) dane z procesu INP. Ten sam proces PASS wysyła dane (powoli) do procesu OUTP. obrazek1

Ustanowienie pojedynczego bufora pomiędzy PASS a INP nie zapewni istotnego przyśpieszenia. Jeśli jednak powołamy do życia dwa obiekty procesu BUFORIN to być może zyskamy na czasie. obrazek2
Teraz obiekt PASS



Rysunek 3. Proces ProdKon i dwie pary buforów



Rysunek 4. Inna chwila

5. Czytelnicy i pisarze

Omawiamy znany problem czytelników i pisarzy, jak zapewnić bezpieczny dostęp do wspólnego, globalnego zasobu unikając przy tym zagłodzenia pisarzy i czytelników.Materiał tu

przedstawiany pochodzi z pracy Bolka Ciesielskiego [Cie89]. Zechciej porównać z rozwiązaniem stosującym semafory [BH76].

5.1. Rozwiązanie z możliwością zagłodzenia. OPISAĆ – naszkicować straightforward solution –
strukturą może być tablica lub drzewo np. drzewo BST.
operacje czytania to szukaj lub modyfikuj rekord, bez zmiany klucza
operacje pisania to insert i delete - zmieniają strukturę

```
unit STR: process();
  unit czytaj: procedure ...
  begin
    enable czytaj
    ...
  end czytaj;
  unit pisz: procedure ...
begin
  return
  do
    accept czytaj, pisz
  od
end STR;
```

W systemie może istnieć wiele aktywnych obiektów typu Klient. Np.

```
unit Klient: process(node: integer, srw: STR);
...
end Klient
```

Każdy aktywny obiekt typu Klient może wykonać polecenie bądź call srw.czytaj() bądź call srw.Pisz(). Posługując się wlasnościami protokołu alien call można wykazać, że nie dojdzie do konfliktu czytelników z pisarzami. Umieszczona na początku procedury Czytaj instrukcja enable Czytaj "otwiera drogę" następnemu czytelnikowi.
Rozwiązanie takie nie wyklucza jednak zagłodzenia pisarza(y).

5.2. Rozwiązanie bez zagłodzenia pisarzy. W tym rozwiązaniu wprowadzony jest obiekt procesu Supervisor, który kontroluje dostep do zasobu. W celu uniknięcia zagłodzenia pisarzy do czytania wpuszcza się czytelników grupami. Gdy wszyscy czytelnicy z grupy zakończą czytanie sprawdza się czy jakiś pisarz zgłosił chęć pisania.
Poniżej przedstawiamy schemat procesu Reader.

```
unit Reader:process(node: integer, sv:supervisor);
begin
  return;
  do
    call sv.stamp; (* zarejestruj się *)
    call sv.startread; (* żądanie dostępu do zasobu *)
    (* Czytanie *)
    call sv.endread; (* czytanie zakończone *)
    (* Inne czynności *)
  od
end Reader;
```

Proces Writer ma podobną strukturę.

```
unit Writer:process(node:integer, sv:supervisor);
begin
  return;
  do
    call sv.startwrite; (* żądanie dostępu do zasobu *)
    ...
    (* Pisanie *)
    ...
    call sv.endwrite; (* zakończenie pisania *)
    ...
  od
end Writer;
```

Utrzymaniem porządku zajmuje się obiekt procesu Supervisor. *

```
unit Supervisor:process(node:integer);
  var waiting_readers:integer, (* liczba zarejestrowanych czytelników *)
    writing:boolean, (* TRUE if a writer is writing *)
  (* the following variables concern the group of readers serviced
  in a single supervisor cycle *)
  cr:integer, (* total number of readers *)
  br:integer, (* number of readers which started reading *)
  er:integer; (* number of readers which finished reading *)
  unit stamp:procedure;
  begin
    waiting_readers := waiting_readers+1;
  end stamp;
  unit startread:procedure;
  begin
    br := br+1; (* next reader started the reading *)
    writing := false; (* no writer is writing *)
  end startread;
  unit endread: procedure;
  begin
    er := er+1; (* a reader finishes the reading *)
  end endread;
  unit startwrite:procedure;
  begin
    writing := true; (* a writer is writing *)
  end startwrite;
  unit endwrite:procedure;
  end endwrite;
begin
  return;
  do
    br := 0; er := 0;
    accept startread, startwrite;
    if writing (* if a writer was first *)
    then (* then we wait until it finishes the writing *)
      accept endwrite; (* and finish the service cycle *)
    else
      disable stamp;
      cr := waiting_readers+1; (* the number of waiting readers*)
      waiting_readers := 0; (* additional readers will be *)
      (* serviced in the next readers group *)
      enable stamp;
      while br < cr
      do
        accept startread, endread;
      od;
      while er < cr
      do
        accept endread
      od;
    fi;
  od
end supervisor;
```

Struktura programu głównego wygląda tak:

```
program Czytelnicy_i_Pisarze;
  unit Reader: process ...
  unit Writer: process ...
  unit Supervisor: process ...
begin (* the main program *)
  c := new Supervisor(0);
  resume(c);
  for i := 1 to num_readers
  do
    resume(new Reader(nr(i),c));
  od;
  for i := 1 to num_writers
  do
    resume(new Writer(nr(i),c));
  od;
  (* zakończ *)
end
```

Analiza. Rozwiązanie przedstawione przez Bolka (powyżej) zakłada współbieżny (i konfliktowy) dostęp do zasobu.
Można udowodnic, że przedstawione tu rozwiązanie jest wolne od konfliktówych dostępów do zasobu znajdującego się w gestii supervisora, można też udowodnić, że nie wystąpi zagłodzenie klientów pisarzy.
Ale trudno sobie wyobrazić równoczesne odczytywanie (przeszukiwanie) danych. Każdy proces jest (lub może być) alokowany na innym komputerze w sieci. Każdy proces jest sekwencyjny. Metody procesu są wykonywane na zasadzie protokołu alien call czyli po kolei. Oznacza to, że proces serwer może zgodzić się na wykonywanie kilku na raz instancji procedury Czytanie (która musiałaby byc metodą serwera!), ale każde wywołanie metody Czytanie przerywa wykonywanie wcześniejsze tej metody bądź instancje te wykonywane sa po kolei.

5.3. Rozwiązanie z równoczesnym czytaniem. Spróbujemy naszkicować inne rozwiązanie. Z rozwiązania poprzedniego zachowamy protokół zapobiegający zagłodzeniu pisarzy. Natomiast czytanie staje się treścią metody supervisora. Nowa wersja procesu Reader zawiera instrukcję (NOWA, nie występuje w Loglanie'82!)

activate <serwer>.<metoda>(args).

```
unit Reader:process(node: integer, sv:supervisor);
begin
  return;
  do
    call sv.stamp; (* zarejestruj się *)
    call sv.startread; (* żądanie dostępu do zasobu *)
    activate sv.Czytanie(...);
    (* Czytanie, proces sv uruchomi procproc Czytanie *)
    (* jako nowy proces równolegle bądź współbieżnie *)
    call sv.endread;
    (* czytanie zakończone *)
    (* Inne czynności *)
  od
end Reader;
```

Instrukcja activate jest nową proponowaną instrukcją[5]. Jej semantyka to uruchomienie nowego procesu na tym samym komputerze co proces sv. Wymagane do tego jest zezwolenie allow Czytanie ze strony procesu sv. Wątek nowego procesu to treść metody Czytanie. Po wykonaniu ostatniej instrukcji proces taki znika automatycznie. Podczas wykonywania swych instrukcji procesprocedura ma dostęp do wszystkich zmiennych procesu sv.
Proces writer pozostawiamy bez zmian.
W procesie supervisor pojawi się procedura-proces Czytanie.

[5]Właściwie można by użyc słowa send, przewidzianego przez Bolka w 1988.

```
unit Supervisor:process(node:integer);
  var waiting_readers:integer, (* liczba zarejestrowanych czytelników *)
    writing:boolean, (* TRUE if a writer is writing *)
  (* the following variables concern the group of readers serviced
  in a single supervisor cycle *)
  cr:integer, (* total number of readers *)
  br:integer, (* number of readers which started reading *)
  er:integer; (* number of readers which finished reading *)
  unit stamp:procedure;
  begin
    waiting_readers := waiting_readers+1;
  end stamp;
  unit startread:procedure;
  begin
    br := br+1; (* next reader started the reading *)
    writing := false; (* no writer is writing *)
    allow Czytanie;
  end startread;
  unit endread: procedure;
  begin
    er := er+1; (* a reader finishes the reading *)
  end endread;
   unit Czytanie: processprocedure(...);
  (*  TAK! wywołanie call Czytanie uruchamia proces procedury wewnątrz procesu supervisor *)
   end Czytanie;
  unit startwrite:procedure;
  begin
    writing := true; (* a writer is writing *)
  end startwrite;
  unit endwrite:procedure;
  end endwrite;
begin
  return;
  do
    br := 0; er := 0;
    accept startread, startwrite;
    if writing (* if a writer was first *)
    then (* then we wait until it finishes the writing *)
      accept endwrite; (* and finish the service cycle *)
    else
      disable stamp;
      cr := waiting_readers+1; (* the number of waiting readers*)
      waiting_readers := 0; (* additional readers will be *)
      (* serviced in the next readers group *)
      enable stamp;
      while br < cr
      do
        accept startread, endread;
      od;
      while er < cr
```

Nowy rodzaj modułu wymyśl nazwę jest:

- procesem, może być uruchomiony na rdzeniu komputera (równolegle), lub współbieżnie z procesem obejmującym i innymi instancjami procesami takich modułów.
- jego instancja ma dostęp do procesu obejmującego
- nowa instrukcja activate zamiast instrukcji call

Zasobem, którym procesy mogą się dzielic jest ekran. Ale ... Czy takim zasobem może byc np. plik?

Czy jest sens rozważać problem czytelników i pisarzy? TAK. Proces Zasób ma metodę czytanie. Ta metoda rozpoczyna się od enable startread. Tzn. w trakcie czytania nowy proces może zgłosić chęć czytania. Proces Zasób może zgodzić się na czytanie – przez enable czytanie? lub accept czytanie.

Obiekt *sv* procesu supervisor zarządza dostępem do dzielonego zasobu. W jakich stanach może się znaleźć ten obiekt:

początek W tym stanie zarządca oczekuje na zgłoszne startread lub startwrite **accept** startread, startwrite,

pisanie W tym stanie zarządca oczekuje na operację pisania **accept** endwrite,

czytanie W tym stanie wyróżniamy trzy kolejne stany składowe tego stanu
  - zapisz liczbę zarejestrowanych czytelników,
  - dopóki liczba zarejestrowanych czytelników jest większa od liczby czytelników, którzy rozpoczęli czytanie oczekuj na startread lub start end,
  - dopóki liczba czytelników, którzy zakończyli czytanie jest mniejsza od liczby zarejestrowanych czytelników oczekuj na start end,

...

Komentarze. Zauważ, że można się obejść bez instrukcji enable i disable. Wymaga to wprowadzenia dodatkowego procesu, który zajmowałby się wyłącznie rejestrowaniem zgłoszeń czytelników i na żądanie udostępniałby tę informacje procesowi superwizor.

Powazniejszy problem mamy z wykorzystaniem zezwoleń na operacje czytania. Procesy czytelników i proces zarządcy zasobu nie mają wspólnych danych. Zmienne każdego procesu są prywatne i niedostępne z zewnątrz. Na czym ma więc polegać równoczesne czytanie danych z zasobu przez różne obiekty procesu czytelnik? Czy to w ogóle jest możliwe?

Notatka 1 maja 2017
Będzie to możliwe gdy serwer potrafi uruchomić kolejne "wątki" procedury czytanie na swoich rdzeniach lub współbieżnie. Po

rozpoczęciu procedury czytanie w serwerze. Bo procedura ta musi być w serwerze. Zaczynamy od instrukcji enable czytanie w treści tej procedury. Wiemy, że nie ma konfliktu z poprzednio uruchomionymi wątkami czytania i możemy uruchomić kolejny wątek. Problem polega na tym, że proces wykonywania procedury czytanie, ma się zachowywać jak by był procesem. Alokacja na węźle? nie musimy się tym przejmować niech zajmie sie tym interpreter int. Ale trzeba go nieco zmienić! Najtrudniejsze jest wskazać int-owi, że ma postąpić inaczej. A jak ma postępować? Stworzyć nowy obiekt procesu anonimowego i uaktywnić go.
W nowej wersji Loglanu mogłoby to wyglądac tak:

unit Czytanie: wątek procedura (...);
...
end Czytanie;

gdzie wątek jest procesem. Obecna wersja Loglanu nie akceptuje takiej składni, niestety. (7 ERROR 303 COROUTINE/PROCESS ILLEGAL HERE AS PREFIX WATEK)

Miałem pomysł na ciekawy przypadek zadania czytelnicy i pisarze. Co to było?
Aha, czy potrafimy opisać (z sensem) pracę przebiegu kompilatora (wieloprzebiegowego) tak jak to zrobili autorzy GIERAlgolu?
Każdy przebieg pobiera dane z jednego pliku i wysyła wyniki do kolejnego pliku. Jeśli pobieranie odbywa się z bufora A, a tymczasem komputer (inny procesor) ładuje do bufora B. I podobnie wysyłane sa wyniki do Bufora C podczas gdy komputer wysyła zawartość bufora D na dysk. Narysować układ tych procesów. Trochę podobny do pierścienia producentów i buforów pomiędzy nimi.

Wydaje się wskazanym, obmyślenie innego modelu procesu. Zauważmy, że w pierwszym programie jaki obejrzeliśmy w tym rozdziale wiele obiektów procesu pisarz wypisywało na ekran. (Co skutkowało chaosem.) Zasób może mieścic się w pamięci dyskowej np. na pliku. Wtedy równoczesne czytanie ma sens.

The idea of the above solution is that during the reading cycle only those readers are allowed to start reading which were waiting at the beginning of the cycle. All others must wait for the next reading cycle and a writer may be serviced before them. In this way the readers cannot starve the writers. As can easily be seen the solution would be simpler and more elegant if the alien call mechanism made available to a process the number of processes waiting for an alien call of the given procedure. In that case the procedure stamp and registration

of waiting readers could be omitted. Instead a system count of waiting readers should be consulted. A simple extension to the alien call is described below. A solution using only the pure synchronous rendezvous would be more complex because the supervisor should consist of two separate processes: one of them for registering waiting readers and the second for actually controlling reading and writing.

6. Współpraca z bazą danych

Wiele obiektów procesów może współpracować z bazą danych. Dla uproszczenia przyjmiemy, że mamy do czynienia z obiektami procesów jednego z dwu rodzajów:

klient: – obiekt takiego procesu (lub podtypu typu klient) zgłasza do bazy żądania (zadania) np. czy należy?, wstaw, usuń, min, następny – a więc operacje na kolejce priorytetowej,

serwer: – obiekt procesu serwer zarządza swoim poddrzewem, ma sporą pamięc przeznaczoną na ten cel (rzędu 4GB lub więcej), potrafi komunikować się z podobnymi obiektami poprzez sieć

Obiekty – serwery tworzą graf (drzewo). Obiekt s typu serwer może zlecać zadanie innemu serwerowi q, a w tym czasie przyjmować zlecenie od klienta. Obiekt serwer jest więc i serwerem i klientem. Obsługa zleceń i przyjmowanie komunikatów o wynikach zleconych zadań odbywa się zgodnie z protokołem alien call. Równoległość jest zapewniona przez rozproszenie zadań na procesory połączone siecią.

Jeśli pomyślimy, że baza danych jest zorganizowana jako drzewo, np. drzewo BST lub B-drzewo, to operacje dostępu do bazy danych możemy podzielić na grupy w ten sposób:

c czytanie informacji,
m modyfikacja rekordu,
z wstawianie oraz usuwanie rekordu – pociąga zmianę drzewa,
i intencja zmiany - zawsze poprzedza zmianę.

Pary operacji konfliktowych i niekonfliktowych przedstawione są w poniższej tabelce zaczerpniętej z książki [?].

	c	m	i	z
c	+	-	+	-
m	-	-	+	-
i	+	+	-	-
z	-	-	-	-

Problem jest podobny do problemu czytelników i pisarzy. Przyjmujemy więc następujące ustalenia:

- w każdym procesie operacje intencji oraz zmiany (wstawiania lub usuwania) tworzą parę i występują w tej kolejności,
- operacje te występujące w jednym procesie, nie moga byc rozdzielane przez operacje intencji lub zmiany pochodzące z innego procesu,
- dopuszczalne natomiast jest wykonanie operacji czytania lub modyfikacji (zgłaszane przez inny proces) pomiędzy operacjami intencji i zmiany,
- operacje czytania i intencji nie są konfliktowe i mogą być wykonywane równocześnie przez wiele procesów,
- operacje intencji i modyfikacji także nie są konfliktowe i mogą być wykonywane równoczesnie.

Prowadzi to do następującej konkluzji: obiekt procesu supervisor może się znajdować w jednym z ośmiu stanów:

P początek cyklu, accept startread, startmod, startintent,
C czytanie, accept endread, startread, startintent,
CI czytanie oraz intencja accept endread, startread, startintent,
I intencja accept startread, startintent, endintent,
MI intencja oraz modyfikacja accept startmod, endmod
M modyfikacja accept startmod, endmod, startintent
U usun accept startdel,enddel
W wstaw accept startins, endins.

Moduł procesu Klient ma następującą budowę:

```
unit Klient: process(node: integer, sv:Serwer);
      unit DoCzytania: class;
      begin
      call sv.zarejestruj;
      call sv.startread;
      inner;
      call endread;
      end DoCzytania;
      unit DoPisania: class;
      begin
      call sv.intent;
      call sv.startwrite;
      inner;
      call endwrite;
      end DoPisania;
      unit Czytanie: DoCzytania procedure(...);
      ...
      end Czytanie;
      unit Wstawianie: DoPisania procedure(...);
      ...
      end Wstawianie;
      unit Usuwanie: DoPisania procedure(...);
      ...
      end Usuwanie;
begin
   (* treść Klienta *)
end Klient;
```

Moduł Klient może być rozszerzany na kilka sposobów (przez dziedziczenie) w zależności od potrzeb konkretnej aplikacji. Poniżej przedstawiamy moduł zarządzający dostępem do bazy danych.

```
unit Supervisor: process(node: integer);
      var stan: char;
      unit zarejestruj procedure ...
      unit startread: procedure ...
      unit startintent: procedure();
      unit endintent: procesure();
      unit startdel: procedure
      unit enddel: procedure
      unit begin
   stan:='P';
   return;
   do
      (* Początek cyklu: *)
      case stan
      when 'P': accept startread, startintent, startmod;
      when 'C': disable zarejestruj;
           zarejestrowanych:=

      when 'I': accept endintent, startread, startmod;
      when 'J' (* 'IF' *): accept startdel, startins;
      when 'M':
      when 'D' (* CI' *):
      when 'N' (* 'MI' *):
      when 'U': accept enddel;
      when 'W': accept endins;

      esac;
   od
end Supervisor
```

========== uzupełń brakujące metody: startread, ... każda metoda kończąc wyznacza nowy stan ==========

6.1. Propozycja obliczeń równoległych. Jak należy wykorzystać serwer z wieloma rdzeniami?
Myślę, że można nadchodzące wywołania metod serwera realizować nie na zasadzie alien call, lecz nieco bardziej liberalnej i elastycznej.
Jeśli serwer realizuje metodę m i od innego klienta nadchodzi żądanie wywołania metody np. m lub innej m', to nieużywany rdzeń może rozpocząć wykonywanie tej metody. Utworzy rekord aktywacji, odbierze parametry aktualne, etc.
Wymaga to zmiany w interpreterze, ale jest do pomyślenia.

Jak zaprogramowac obsługę bazy danych? Z żądaniami zwracają się klienci $=$

$-$

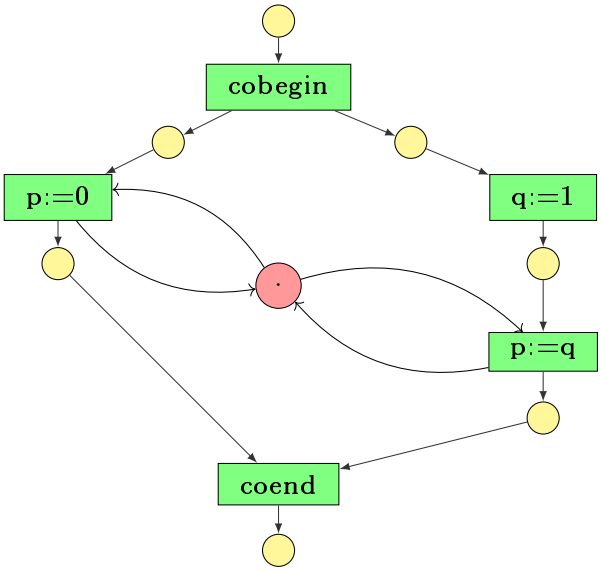
7. Rozmaite modele obliczeń

Widzieliśmy, że obliczenia współbieżne nie są deterministyczne. Powtarzanie obliczeń nie musi przebiegać zawsze tak samo, może dawać inne wyniki. Jakie wyniki są możliwe? Jak mogą przebiegać obliczenia programów z procesami?
Czy istnieje jedna intuicyjna, semantyka obliczeń współbieżnych? Wcześniej dla obliczeń sekwencyjnych stwierdzilismy, że rôżne sposoby zapisywania iteracyjnych programów dzielą tę samą semantykę. Ta obserwacja nie jest prawdziwa w świecie obliczeń niesekwencyjnych. O tym mówi ten paragraf.
Zwracamy uwagę na zjawisko głębsze niż niedeterminizm obliczeń: rozproszonych, współbieżnych, równoległych. Dla obliczeń nie-sekwencyjnych można stosować więcej niż jeden model semantyki. Jest to zaskakujące gdy przypomnimy sobie, że semantyka obliczeń iteracyjnych, sekwencyjnych jest jedna wspólna rôżnym ortografiom pisania programôw while. W tej części rozdziału konstruujemy pewien model obliczeń, nazwaliśmy go MAX model maksymalnego zaangazowania procesów w danym momencie nie będących w konflikcie. W literaturze można spotkać pogląd, że znaczeniem programu nie-sekwencyjnego jest unia wszystkich możliwych przeplotów (ang. interleaving) por. [BA96] akcji wystepujących w procesach danego programu. W modelu, zob. [?] jaki przedstawiamy poniżej liczba możliwych przeplotów jest mniejsza. Może to mieć znaczenie dla analizy programów niesekwencyjnych.
Uważamy, że w sytuacji gdy kilka procesów ma do wykonania pewien zbiór Z atomowych poleceń i gdy komputer (lub klaster komputerów) dysponuje odpowiednią liczbą procesorów wykonany zostaje pewien maksymalny, niekonfliktowy podzbiór $X \subset Z$. Koncepcję maksymalnego zaangażowania procesów przedstawimy posługując się uproszczonym językiem programowania współbieżnego z instrukcją cobegin ... coend. Można by opowiedziec tę historię z pełnym bagazem języków do L_9. Ale po co? Abstrahujemy od bagażu zbędnego w tej chwili. Najpierw dwa przykłady.

Przykład 25.1. Niech M oznacza następujący program

$$\{\textbf{cobegin } p := 0 \parallel q := 1; p := q \textbf{ coend}\}$$

Jaki może być końcowy wynik? Jeśli przyjmiemy, że instrukcje mogą być wykonane w dowolnym przeplocie poleceń atomowych, to zarôwno 0 jak i 1 mogą być końcowymi wartościami



Rysunek 5. Plansza 1 diagram programu

zmiennej p. (W poniższej formule słowo cobegin zastąpiliśmy przez [[, a słowo coend zastąpiliśmy przez]].) Zanotujmy

$$\Diamond\{[\![\ p := 0 \parallel q := 1; p := q\ ]\!]\}p \wedge \Diamond\{[\![\ p := 0 \parallel q := 1; p := q\ ]\!]\}\neg p$$

Skąd to się bierze? Zagraj w grę. Planszą do gry jest diagram tego programu, zob. rysunek 5.
Reguły gry są następujace:

s) Ustaw swój pion w początkowym stanie, tj, przed instrukcją cobegin.

p) Dopóki nie dojdziesz z pionem do stanu końcowego powtarzaj
{jeśli masz potrzebne piony to możesz je przesunąć dalej } W szczególności
- pion przed instrukcją cobegin "rozmnaża się",
- do przejscia przez instrukcję coend musisz mieć dwa piony gotowe do rozpoczęcia tej instrukcji,
- do wykonania instrukcji {p:=0} lub {p:=q} musisz mieć dwa piony, jeden gotów do rozpoczecia tej instrukcji i drugi otwierający dostęp do zmiennej p (zabierasz go z czerwonego kółka).

Jesteś gotowa? Gramy!
Gra może zakończyć sie wynikiem $p = 0$ lub $p = 1$.
Poniższa tabelka stanowi podsumowanie Twoich eksperymentów. Pierwszy stan obliczenia tego programu ma jeden stan bezpośrednio po nim następujący: w tym drugim stanie widzimy dwie gwiazdki symbolizujące piony-procesory. Od teraz

dwa procesory wykonują instrukcje dwu procesów: lewego i prawego. Stan numer dwa jest w relacji bezpośredniego poprzednika z trzema stanami uwidocznionymi w trzecim wierszu tabelki. Każdy z nich jest możliwym bezpośrednim następnikiem drugiego stanu. W efekcie, mozliwe jest, że obliczenie potoczy się według jednego z trzech (nie więcej) wzorów opisanych w poniższej tabeli. Każda z tych historii jest zapisana w odrębnej kolumnie tabelki.

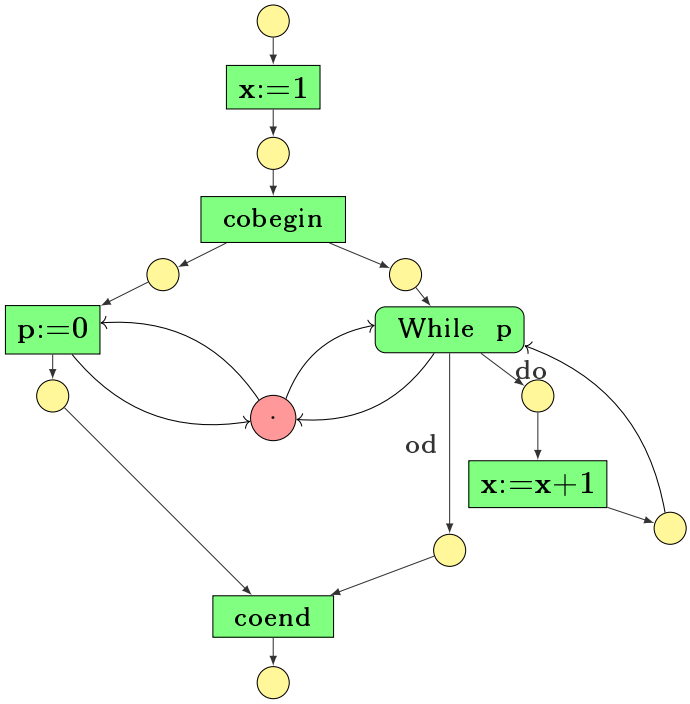
Tabelka obrazująca możliwe historie wykonania programu ⟦ p:=0 ∥ q:=1; p:=q ⟧

	$\langle v$,* ⟦ p:=0 ∥ q:=1; p:=q ⟧$\rangle$	
	$\langle v$, ⟦*p:=0 ∥*q:=1; p:=q ⟧ $\rangle$	
$\langle \frac{p}{0}\frac{q}{v_q}$, ⟦*∥*q:=1; p:=q ⟧$\rangle$	$\langle \frac{p}{v_p}\frac{q}{1}$,⟦*p:=0 ∥*p:=q⟧ $\rangle$	$\langle \frac{p}{0}\frac{q}{1}$,⟦ * ∥ * p:=q ⟧ $\rangle$
$\langle \frac{p}{0}\frac{q}{1}$, ⟦ * ∥ * p:=q ⟧$\rangle$	$\langle \frac{p}{1}\frac{q}{1}$,⟦ *p:=0 ∥ * ⟧ $\rangle$	$\langle \frac{p}{1}\frac{q}{1}$,⟦ * ∥ * ⟧ $\rangle$
$\langle \frac{p}{1}\frac{q}{1}$, ⟦ * ∥ * ⟧$\rangle$	$\langle \frac{p}{0}\frac{q}{1}$,⟦ * ∥ * ⟧ $\rangle$	$\langle \frac{p}{1}\frac{q}{1}$,∅ $\rangle$
$\langle \frac{p}{1}\frac{q}{1}$,∅ $\rangle$	$\langle \frac{p}{0}\frac{q}{1}$,∅ $\rangle$	

A teraz zmodyfikujmy zasady: w każdym stanie MUSISZ rozpocząć wykonywanie ruchu jeśli nie ma konfliktu. A instrukcje {p:=0} i {q:=1} nie są konfliktowe.
Co teraz? Czy końcową wartością zmiennej p może być 0?
Jeśli przyjmiemy, że dane są dwa procesory i że każdy z nich rozpoczyna wykonywanie polecenia gdy tylko jest to dopuszczalne, to jedynym możliwym wynikiem okazuje się $p = 1$. W poniższej tabelce zestawiamy historie obliczeń jakie mogą się zdarzyć. Teraz oprócz * symbolu gotowości do działania pojawia się symbol ∘– w ten sposób oznaczamy rozpoczęcie wykonywania instrukcji.

Tabelka obrazująca historie wykonania programu ⟦ p:=0 ∥ q:=1; p:=q ⟧ w modelu MAX

	$\langle v$,* ⟦ p:=0 ∥ q:=1; p:=q ⟧$\rangle$	
	$\langle v$, ⟦ *p:=0 ∥ *q:=1; p:=q ⟧$\rangle$	
	$\langle v$, ⟦ ∘ p:=0 ∥∘ q:=1; p:=q ⟧$\rangle$	
$\langle \frac{p}{0}\frac{q}{1}$, ⟦*∥* p:=q ⟧$\rangle$	$\langle \frac{p}{v_p}\frac{q}{1}$,⟦∘ p:=0 ∥*p:=q⟧ $\rangle$	$\langle \frac{p}{0}\frac{q}{v_q}$, ⟦ * ∥∘ q:=1; p:=q ⟧ $\rangle$
$\langle \frac{p}{0}\frac{q}{1}$, ⟦ * ∥ ∘p:=q ⟧$\rangle$	$\langle \frac{p}{0}\frac{q}{1}$,⟦ * ∥ ∘p:=q ⟧ $\rangle$	$\langle \frac{p}{0}\frac{q}{1}$,⟦ * ∥ ∘p:=q ⟧ $\rangle$
$\langle \frac{p}{1}\frac{q}{1}$, ⟦ * ∥ * ⟧$\rangle$	$\langle \frac{p}{1}\frac{q}{1}$,⟦ * ∥ * ⟧ $\rangle$	$\langle \frac{p}{1}\frac{q}{1}$,⟦ * ∥ * ⟧ $\rangle$
$\langle \frac{p}{1}\frac{q}{1}$,∅ $\rangle$	$\langle \frac{p}{1}\frac{q}{1}$,∅ $\rangle$	$\langle \frac{p}{1}\frac{q}{1}$,∅ $\rangle$



Rysunek 6. Plansza 2

Opisz to co widać w tej tabelce...
Podsumowując stwierdzamy, że w semantyce MAX nasz program jest równoważny następującemu programowi sekwencyjnemu

$$\{\, q := 1; p := q \,\}$$

Rozpatrzmy teraz kolejny przykład

Przykład 25.2.

$$\{x := 1; \textbf{cobegin } p := 0 \parallel \textbf{while } p \textbf{ do } x := x + 1 \textbf{ od coend}\}$$

Ponownie narysujemy diagram programu. Zagrajmy w grę na tej planszy, por. rysunek 6. W wariancie z dowolnymi przeplotami (tj. ARB) wartość zmiennej x może być dowolnie wielka. Co gorzej, gra może toczyć się bez końca.
W wariancie MAX gra się zawsze kończy i końcowa wartość zmiennej x jest równa 1 lub 2, nie więcej.
Program ten można zastąpić programem równoważnym (niedeterministycznym!)

$$\{p := 0; \{x := 1 \textbf{ or } x := 2\}\}$$

Czy jesteśmy gotowi do analizowania programów z instrukcja niedeterministycznego wyboru or?

Przykłady te mogą Ci się zdawać zbyt proste i mało znaczące.

Przykład 25.3. Rozważ jednak programy

$$\{\textbf{cobegin } p := 0 \parallel q := 1; p := q \textbf{ coend}; \textbf{if } p \textbf{ then } K \textbf{ else } M \textbf{ fi}\}$$

oraz

$$\{x := 1; \textbf{cobegin } p := 0 \parallel \textbf{while } p \textbf{ do } x := x+1 \textbf{ od coend}; \textbf{if } x > 3 \textbf{ then } K \textbf{ fi}\}.$$

Programy K i M mogą być bardzo skomplikowane i trudne w analizie. Jeśli jednak wiesz, że system wykonawczy wykonuje obliczenia zgodnie z modelem MAX, to możesz sobie oszczędzić wiele pracy. Nieprawdaż?

Oczywiście potrafisz sam zbudować wiele podobnych przykładów.
Pytanie, które się nasuwa w naturalny sposób: czy rozprawianie o modelu MAX ma sens w obliczeniach rozproszonych bądź równoległych? Śpieszymy z odpowiedzią: i owszem, w obliczeniach tych zaobserwujemy podobne zjawiska. Trzeba tylko stworzyc odpowiednio więcej procesów i pozwolić by komunikaty wysyłane przez nie konkurowały między sobą.

Problem Otwarty. Napisać zaproszenie do badania czy można podać aksjomaty semantyki MAX?

Relacja bezpośredniego następstwa MAX. Po omówieniu przykładów jestesmy gotowi by opisać semantykę obliczeń równoległych (i współbieżnych) z większą dokładnością.
Ograniczenie. Lepiej wytłumaczymy model MAX rozpatrując język programów iteracyjnych z instrukcją

$$\textbf{cobegin } K_1 \parallel K_2 \parallel \cdots \parallel K_n \textbf{ coend}$$

Rozpatrujemy język programowania w którym instrukcjami atomowymi są instrukcje przypisania.

Definicja 25.7. Zbiór P programów jest to najmniejszy zbiór napisów taki, że

(i) do zbioru P należy każda instrukcja przypisania,
(ii) jeśli do zbioru P należą napisy K i M, a napis γ jest formułą otwartą, to do zbioru P należą także napisy
K; M if γ then K else M fi while γ do K od,
(iii) jeśli do zbioru P należą napisy $K_1, K_2, \ldots, K_n$ to napis cobegin $K_1 \parallel K_2 \parallel \cdots \parallel K_n$ coend także należy do zbioru P. Zamiast słów cobegin i coend pozwolimy sobie używać znaków [[oraz]].

Niech v oznacza jakieś wartościowanie zmiennych. Niech $K \in P$ będzie programem z języka P.

Definicja 25.8. Obliczeniem programu K dla początkowego wartościowania v jest ciąg stanów taki, że

1°) Pierwszym elementem tego ciągu jest stan $\langle v, *K\rangle$,

2°) każde kolejne dwa elementy tego ciągu znajdują się w relacji bezpośredniego następstwa, zobacz poniższą definicję,
3°) stan $\langle v, *\emptyset \rangle$ jest końcowym stanem obliczenia.

Zaczynamy od definicji konfliktowego zbioru instrukcji.

Definicja 25.9. Zbiór Z instrukcji jest konfliktowy wtedy i tylko wtedy gdy zawiera przynajmniej jedną instrukcję przypisania zmieniającą wartość zmiennej x i conajmniej jedną instrukcję atomową odczytująca wartość zmiennej X.

W każdym stanie pewne instrukcje atomowe są w trakcie wykonania (oznaczone znakiem $\circ$), niektóre instrukcje atomowe są gotowe do rozpoczęcia wykonywania (oznaczone znakiem *). Conajmniej jedna instrukcja atomowa została zakończona. Relacja bezpośredniego następstwa stanów

Definicja 25.10.

Ćwiczenia

25.1. Jeden producent – wielu konsumentów. Napisz program i uruchom go. Przeprowadź analize poprawności Twego rozwiązania.

25.2. Podobne zadanie: wielu producentów i wielu konsumentów. Napisz program i uruchom go. Przeprowadź analize poprawności Twego rozwiązania.

25.3. Stwórz program w ADZIE (lub w CSP lub w Javie), który będzie zachowywał się jak program z przykładu 25.1.

25.4. Zastanów się jak powinno wyglądać programowanie równoległe? Czy protokół alien call jest odpowiednim dla tego rodzaju programowania narzędziem?

25.5. Załóżmy, że programy wymienione w przykładzie 25.3 będą wykonywane na komputerze z dwoma procesorami. Czy można je uprościć?

25.1. Spróbuj podać aksjomaty dla obliczeń równoległych w semantyce MAX.

25.2. Czy można podać aksjomaty obliczeń rozproszonych z poleceniem alien call?

Bibliografia

[AA82] S. Alagić and M.A. Arbib. Projektowanie programów poprawnych i dobrze zbudowanych. WNT, 1982.

[Acz98] Amir D. Aczel. Wielkie twierdzenie Fermata. Prószyński i Ska, Warszawa, 1998.

[AdBO10] Krzysztof Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. Verification of Sequential and Concurrent Programs. Springer, Berlin Heidelberg, 2010.

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. The Design and Analysis of Computer Algorithms. AddisonWesley, 1974.

[ALSU07] A. Aho, M. Lam, R. Sethi, and J. Ullman. Compilers: principles, techniques & tools. Addison Wesley, Boston, 2007.

[Ame01] P Amey. Logic versus Magic in Critical Systems. In Reliable Software Technologies - Ada Europe 2001, Lecture Motes in Computer Science 2043. Springer, 2001.

[App98] Andrew W. Appel. Modern Compiler Implementation in Java. Cambridge University Press, 1998.

[AVN+07] Joao Abreu, Vasco T. Vasconcelos, Isabel Nunes, Antonia Lopes, Luis S. Reis, and Alexandre Caldeira. ConGu, The Specification and the Refinement Languages. Technical report, University of Lisbon, March 2007.

[BA96] M. Ben-Ari. Podstawy programowania współbieżnego i rozproszonego. WNT, Warszawa, 1996.

[Bar96] John Barnes. Programming in Ada 95. Addison-Wesley Publ., Boston, 1996.

[Bar03] J. Barnes. High Integrity Software. Addison-Wesley, London, 2003.

[BH76] Per Brinch-Hansen. Podstawy Systemów Operacyjnych. PWN, Warszawa, 1976.

[BKR91] Lech Banachowski, Antoni Kreczmar, and Wojciech Rytter. Analysis of Algorithms and Data Structures. Addison-Wesley, 1991.

[Cie89] Bolesław Ciesielski. Alien call - a synchronization mechanism for distributed processes . Master's thesis, University of Warsaw, 1989.

[CK84] G. Cioni and A. Kreczmar. Programmed Deallocation without Dangling References. Information Processing Letters, 18:179–185, 1984. http://lem12.uksw.edu.pl/Loglan82/Doc/Programmed-deallocation-without-Dangling-Reference-I.pdf.

[Con63] M.E. Conway. Design of a separable transition-diagram compiler. Communications of the ACM, 1963.

[Dij78] E.W.D. Dijkstra. Umiejętność Programowania. WNT, Warszawa, 1978.

[Dil90] A Diller. Z: An Introduction to Formal Methods. J. Wiley, Chichester, 1990.

[EM85] H Ehrig and G. Mahr, editors. Fundamentals of Algebraic Specification 1. Springer, 1985.

[Eng67] Erwin Engeler. Algorithmic Properties of Structures. Math. Systems Theory, 1:183–195, 1967.

[Flo67] Assigning meanings to programs, volume Mathematical Aspects of Computer Science, 1967.

[GM95] J. Gosling and H. McGilton. The Java Language Environment. Technical report, Sun Microsystems Co., October 1995.

[GMP71] A. Gôraj, G. Mirkowska, and A. Paluszkiewicz. On the notion of description of program. Bull. Polish Academy Sciences, Ser. Math. Phys., 18:499–506, 1971.

[Grz05] Andrzej Grzegorczyk. Undecidability without Arithmetization. Studia Logica, 79(2):163–230, 2005.

[HJ98] C.A.R. Hoare and Heng Jifeng. Unifying Theories of Programming. Prentice Hall, 1998.

[HKT02] David Harel, Dexter Kozen, and Jerzy Tiuryn. Dynamic Logic. MIT Press, 2002.

[JW74] K. Jensen and N. Wirth. Pascal user manual and report. Springer Verlag, New York, 1974.

[Kal55] Laszlo Kalmâr. Über ein Problem betreffend die Definition des Begriffes der allgemeinrekursiven Funktion. Zeit. math. Logik, 1:93–96, 1955.

[Kle36] Stephen C. Kleene. General Recursive Functions of Natural Numbers. Mathematishes Annalen, 112:727–742, 1936.

[Knu77] Donald Knuth. The Art of Programming. Addison Wesley, 1977.

[Kre77a] Antoni Kreczmar. Effectivity problems of Algorithmic Logic. Fundamenta Informaticae, page 19–32, 1977.

[Kre77b] Antoni Kreczmar. Programmability in Fields. Fundamenta Informaticae, page 195–230, 1977.

[KU58] A.N. Kolmogorov and W.A. Uspenski. K opredelenii algoritma. Uspechi Matematicheskich Nauk, 13(4):3–28, 1958.

[Lag10] Jeffrey C. Lagarias, editor. The Ultimate Challenge: The 3x+1 Problem. American Mathematical Society, Providence R.I., 2010.

[Mey87] Bertrand Meyer. Eiffel. P, W, 1987.

[Mir71] Grazyna Mirkowska. On Formalized Systems of Algorithmic Logic. Bull. Polish Academy Sciences, Ser. Math. Phys., page 421–428, 1971.

[Mir77] Grazyna Mirkowska. Algorithmic logic and its applications in the theory of programs I. Fundamenta Informaticae, pages 1–17, 147–165, 1977.

[Mos45] Andrzej Mostowski. Axiom of choice for finite sets. Fundamenta Mathematicae, 33:137–168, 1945.

[MS87] Grażyna Mirkowska and Andrzej Salwicki. Algorithmic Logic. PWN and J.Reidel, Warszawa, 1987. [Online; accessed 7-August-2017].

[MS92] Grażyna Mirkowska and Andrzej Salwicki. Logika algorytmiczna dla programistów. WNT, Warszawa, 1992.

[MS96] Grażyna Mirkowska and Andrzej Salwicki. The Algebraic Specification do not have the Tennenbaum property. Fundamenta Informaticae, 28:141–152, 1996.

[MSS08] G. Mirkowska, A Salwicki, and O. Swida. SpecVer - the methodology integrating specification, programming and verification. Fundamenta Informaticae, 85:343–357, 2008.

[MSST00] G. Mirkowska, A. Salwicki, M. Srebrny, and A. Tarlecki. First-Order Specifications of Programmable Data Types. SIAM Journal on Computing, 30:2084–2096, 2000.

[O'D82] Michael J. O'Donnell. A Critique of the Foundations of Hoare-Style Programming. In Proc. Logics of Programs LNCS, volume 131, page 349–374. Springer, 1982.

[Okt82] Hanna Oktaba. On algorithmic theory of references. PhD thesis, Institute of Informatics, University of Warsaw, 1982. [see also Mirkowska and Salwicki 1987, pp. 328 - 341].

[RBSW79] W.M. Ratajczak-Bartol and D. Szczepańska-Wasersztrum. Data Structure for Simulation Purpose in Loglan77. Technical Report 373, Institute of Computer Science, Polish Academy of Sciences, Warszawa, 1979.

[RBSW07] W.M. Ratajczak-Bartol and D. Szczepańska-Wasersztrum. Code of Simulation and other Classes. Technical report, Warsaw University, October 2007.

[RS63] Helena Rasiowa and Roman Sikorski. Mathematics of metamathematics. PWN, Warszawa, 1963.

[Sala] Andrzej Salwicki. A new proof of Euclid's algorithm. http://lem12.uksw.edu.pl/wiki/OnEuclid'salgorithm.

[Salb] Andrzej Salwicki. Historia projektu Loglan. url[http://lem12.uksw.edu.pl/wiki/Historia_projektu_Loglan].

[Sal80] Andrzej Salwicki. On Algorithmic theory of Stacks. Fundamenta Informaticae, 3:311–332, 1980.

[Sav98] J.E. Savage. Models of Computation. Addison Wesley, 1998.

[Sie50] Wacław Sierpiński. Teoria Liczb. Monografie Matematyczne. PWN, 1950.

[Sta] Richard Stallman. Manifest GNU. url[http://www.gnu.org/gnu/manifesto.pl.html].

[Str13] B Stroustrup. The C++ Programming Language. Addison-Wesley Publ., Boston, 2013.

[SW91] Andrzej Szałas and Jolanta Warpechowska. Loglan'82. WNT, Warszawa, 1991.

[Swi02] Oskar Swida. An environment for concurrent and distributed computatio PhD thesis, Institute of Informatics, Warsaw University, 2002.

[Sza91] Andrzej Szałas. Loglan. WNT, 1991.

[Tar33] Alfred Tarski. O pojęciu prawdy w językach nauk dedukcyjnych. Towarzystwo Naukowe Warszawskie, Warszawa, 1933. [Online; accessed 7-September-2017].

[Thi66] Helmuth Thiele. Wissenschatstheoretische Untersuchungen in algorithmis VEB Deutscher Verlag der Wissenschaften, Berlin Heidelberg, 1966.

[Wan78] M. Wand. A New Incompleteness Result for Hoare's Systems. Journal of ACM, 25:168–175, 1978.

[Yan58] Yuri I. Yanov. O logicheskih schemach algoritmov. Problemy Kibernetiki, 1:75–127, 1958.

Indeks