Kurs języka Object/Delphi Pascal

na bazie implementacji Free Pascal.

AUTOR ŁUKASZ STAFINIAK

Email: lukstafi@gmail.com, lukstafi@ii.uni.wroc.pl
Web: www.ii.uni.wroc.pl/~lukstafi

Jeśli zauważysz błędy na slajdach, proszę daj znać!

Wykład 3: Wskaźniki i Funkcje

Oraz biblioteka *Simple Directmedia Layer*.

Zbiory

Zbiory można budować tylko nad typami porządkowymi (liczby całkowite, typy wyliczeniowe i podzakresy).

```
type
   Junk = set of Char;
   Days = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
   WorkDays : set of days;
```

Typ wyliczeniowy.

Operacje na zbiorach:

- stałe zbiory: `[Mon, Tue, Wed]`
- Suma $A \cup B$: `A+B`
- Różnica $A \setminus B$: `A-B`
- Przecięcie $A \cap B$: `A*B`
- Różnica symetryczna $(A \setminus B) \cup (B \setminus A)$: `A><B`
- Włącz x do / usuń ze zbioru A: `Include(A,x)` / `Exclude(A,x)`
- Należenie $x \in A$: `x in A`

Minimum Spanning Tree

```
program MinimumSpanningTree;
uses math;                                  Używamy wagi Infinity jako „brak krawędzi”.
const NumOfNodes = 10;
type Vertices = 0..NumOfNodes - 1;
const
   AllVertices : set of Vertices = [Low(Vertices)..High(Vertices)];
var
   graph : array[Vertices, Vertices] of Real;      Macierz incydencji koduje wagi.
   visited : set of Vertices;
   SpanningTree : array[Vertices] of Vertices;          Drzewo jako przypisanie
   i, j, MinSrc, MinDest : Integer;                     rodziców wierzchołkom.
   MinWeight : Real;
begin                                       Najpierw wypełniamy graf krawędziami.
   for i in Vertices do for j in Vertices do
         if i < j then begin
            if random > 0.5
            then graph[i, j] := random
            else graph[i, j] := Infinity
         end else if i = j
         then graph[i, j] := Infinity
         else graph[i, j] := graph[j, i];          Graf jest nieskierowany.
```

```
   for i in Vertices do
      SpanningTree[i] := i;
   visited := [0];                                  Odwiedzony jest na starcie korzeń.
   MinWeight := 0;              Pierwszy warunek na wypadek gdy graf nie jest spójny...
   while (MinWeight <> Infinity) and not (AllVertices <= visited) do
   begin                                  Czy AllVertices zawiera się w odwiedzonych?
      {Find the minimum edge to unvisited part.}
      MinWeight := Infinity;
      for i in visited do
         for j in AllVertices - visited do
            if graph[i, j] < MinWeight
            then begin
               MinWeight := graph[i, j];
               MinSrc := i; MinDest := j
            end;
      SpanningTree[MinDest] := MinSrc;
      Include (visited, MinDest)   Dodajemy nowy element do zbioru odwiedzonych.
   end;

   {Display the result.}
end.
```

Wskaźniki

- Dane zmiennych zadeklarowanych przez `var` są przydzielane automatycznie (na stosie dla zmiennych lokalnych), takie dane są ustalonego rozmiaru.
- Tablice można przydzielać dynamicznie – już wiemy – przez `SetLength`. Mają one zliczanie dowiązań, nie trzeba ich zwalniać.
- Dane możemy też konstruować ręcznie, wtedy struktura danych może być dowolnego kształtu. Przydzielamy pamięć dla typu `T` przez `new(T)`.
- `new(PT)` zwraca adres – lokację w pamięci przydzielonej komórki rekordu/obiektu typu `T` gdy `PT = ^T`. Adres zwykle zapisujemy do zmiennej – *wskaźnika*. Często mówi się zamiennie o adresach i wskaźnikach...
  - Więc możemy mieć wskaźnik `a` na wskaźnik `b` (czyli na adres zmiennej `b`).
- Rzeczy przydzielone przez `new` zwykle trzeba zwolnić używając `dispose`, która jako argument bierze adres do zwolnienia.
  - W przyszłości dowiemy się więcej o zliczaniu dowiązań.

Typy rekurencyjne

- Struktury danych zaczynają być ciekawe gdy ich rozmiar może być nieograniczony: uzyskujemy to gdy deklaracje typów zawierają „pętle", tzn. typ jednego z pól jest taki sam jak większej całości do której to pole należy.
- W Pascalu użycia typów nie mogą poprzedzać ich deklaracji: nie ma konstrukcji zaznaczającej że deklaracje są wzajemnie rekurencyjne.
- Żeby temu zaradzić wprowadzono wyjątek: typ wskazywany `T` nie musi być zdefiniowany w momencie deklaracji nazwy na typ `^T` wskaźników na `T` / adresów przechowujących `T`.
  - Wyrażenie `^T` jest „gorszej kategorii" niż `PT` dla `type PT = ^T`, dlatego deklaracje nazw dla typów wskaźnikowych są bardzo częste.

Podsumowanie składni wskaźników / adresów

- `var ptr : ^TElem;` — deklaracja wskaźnika
- `ptr := @elem;` — adres zmiennej `elem`
- `ptr^ := elem;` — skopiuj `elem` pod adres `ptr`
- `type PElem = ^TElem;` — typ wskaźnikowy
- `WriteLn (ptr^);` — wartość wskazywana przez `ptr`
- `WriteLn (ptr^.field);` — pole rekordu/obiektu wskaz. przez `ptr`
- `ptr := new (PElem);` — zaallokuj pamięć na `TElem`, zwróć adres
- `dispose (ptr);` — zwolnij komórkę wskazywaną przez `ptr`
- `new (ptr);` — zaallokuj i zapisz adres w `ptr`
- `new (ptr, Init);` — zaallokuj i wywołaj konstruktor `Init` obiektu
- `dispose (ptr, Done);` — wywołaj destruktor `Done`, potem zwolnij

Nomenklatura dla deklaracji typów

- Nazwy dla typów często, a w szczególności dla prototypów obiektów, zaczynają się literą `T`, jak `TElem` powyżej.
- Nazwy dla typów wskaźnikowych zazwyczaj zaczynają się od `P`, i potem nazwa typu bez `T` (jeśli było dodane), jak `PElem` powyżej.
- Jeśli potrzebujemy tylko jednej zmiennej typu `TElem` w danej procedurze, to często nazywamy ją `Elem` (jak powyżej – Pascal nie jest wrażliwy na wielkość liter).

Przykład: uporządkowana lista wiązana

```
{Ordered linked uni-directional list representation of sets.}
program OrderedIntList;
const
   data1 : array[0..9] of Integer = (2, 4, 5, 8, 11, 17, 21, 22, 31, 32);
   data2 : array[0..9] of Integer = (2, 6, 7, 8, 9, 17, 27, 29, 30, 32);
type
   PNode = ^Node;
   Node = record
      elem : Integer;
      tail : ^Node
   end;
var
   list1, list2, previous, l1, l2 : PNode;
   first : Node;
   num : Integer;
```

Dane testowe, jako stałe.

Typ wskaźnikowy „forwarding pointer” zadeklarowany przed definicją typu na który wskazuje.

Węzeł pomocniczy `first`, docelowo zmienna lokalna dla odpowiedniej funkcji (wtedy allokowana na stosie).

```
begin
   previous := @first;
   for num in data1 do
   begin
      previous^.tail := new (PNode);
      previous^.tail^.elem := num;
      previous := previous^.tail
   end;
   previous^.tail := nil;
   list1 := first.tail;
   {Initialize list2. Initialization should be a function of data.}
```

Używamy zmiennej pomocniczej jako „dummy” żeby nie traktować pierwszego elementu inaczej niż pozostałe.

Funkcyjny wariant operatora `new`.

Nasze listy są „`nil`-terminated”.

```
   {Merge list2 into list1.}
   l1 := list1; l2 := list2;                          Przebiegamy po list1 i list2
   first.tail := l1; previous := @first;                          wskaźnikami l1 i l2.
   repeat                                          Niezmiennik: previous^.tail = l1
      if l1^.elem < l2^.elem
      then begin                                  Przesuwamy l1: element już w list1.
         previous := l1; l1 := l1^.tail
      end else if l1^.elem = l2^.elem
      then begin                  Przesuwamy l1 i l2 żeby nie wprowadzić powtórzenia.
         previous := l1; l1 := l1^.tail; l2 := l2^.tail
      end else begin                              Wkładamy nowy element do list1.
         new (previous^.tail);                    Proceduralny wariant operatora new.
         previous^.tail^.elem := l2^.elem;
         previous^.tail^.tail := l1;
         previous := previous^.tail; l2 := l2^.tail
      end
   until (l1 = nil) or (l2 = nil);
   list1 := first.tail;

   {Display the resulting list1.}
end.
```

Biblioteka *Simple Directmedia Layer*

- SDL jest przenośną (między systemami Linux/Windows/Mac) lekką biblioteką multimediów: do wyświetlania grafiki i animacji (zarówno w okienku jak i pełnoekranowych), odtwarzania dźwięku, obsługi klawiatury, myszki i joysticka, dostarcza stopery („timers") i proste API dla wątków.
- SDL ma „bindings" (interfejsy) dla praktycznie wszystkich języków.
- Jeśli *dll* biblioteki SDL nie jest w standardowym katalogu systemu ani w katalogu z programem, podajemy jego ścieżkę `SDL_SetLibraryPath`.
- Popularniejszy z dwóch interfejsów SDLa dla Pascala jest JEDI-SDL, w większości włączony do dystrybucji Free Pascala.
- Trzon biblioteki SDL jest minimalistyczny, dlatego warto używać „bibliotek satelitarnych" jak „SDL_image" (odpowiednio moduły `SDL` i `SDL_image`).
- Najlepiej zgłębiać programowanie SDLa w Pascalu przeglądając źródła interfejsu: wtedy wiadomo jak zaadoptować dokumentację SDL oraz informacje na sieci dla innych języków do Pascala.

Wyświetlanie obrazków z przezroczystością

Wersja podstawowa. Moduł SDL obsługuje tylko obrazy w formacie BMP.

```
{Overlay an image with transparency over a background.}
program ImageAlphaSDL1;
uses SDL;
const
   BackgroundFile = 'background.bmp';
   BallFile = 'ball.bmp';
   pixX = 200; pixY = 200;
var
   screen, background, ball : PSDL_Surface;
   SrcRect, DstRect : SDL_Rect;

begin
   WriteLn ('Initializing SDL.');
   if SDL_Init(SDL_INIT_VIDEO) < 0
   then begin
      WriteLn ('Couldn''t initialize SDL : ', SDL_GetError);
      SDL_Quit; exit
   end;
   background := SDL_LoadBMP (BackgroundFile);
```

Powierzchnie do rysowania.

Zaznaczanie obszarów na powierzchniach.

Inicjalizujemy co potrzeba:
VIDEO, AUDIO, TIMER, ..., EVERYTHING

Zamykamy SDL przed wyjściem.

```
if background = nil                                          Czy tło załadowane?
then begin
   WriteLn('Couldn''t load ', BackgroundFile, ' : ', SDL_GetError);
   SDL_Quit; exit
end;
screen := SDL_SetVideoMode(background^.w, background^.h,     Rozmiar okna/ekranu.
                           background^.format^.BitsPerPixel,      Głębia kolorów.
                           SDL_SWSURFACE);                  Bufor ekranu: software'owy
                                                          lub SDL_HWSURFACE – sprzętowy.
if screen = nil then begin
   WriteLn ('Couldn''t set video mode : ', SDL_GetError);     Podwójny apostrof ''
   SDL_FreeSurface(background); SDL_Quit; exit          w stringu nie kończy go, tylko
end;                                                    wprowadza apostrof do stringa.
ball := SDL_LoadBMP (BallFile);
if ball = nil
then begin
   WriteLn('Couldn''t load ', BallFile, ' : ', SDL_GetError);
   SDL_FreeSurface(background); SDL_FreeSurface(screen); SDL_Quit; exit
end;
if SDL_SetAlpha(ball, SDL_SRCALPHA, SDL_ALPHA_OPAQUE) <> 0
then begin                    Ustawienie przezroczystości wymaga „gimnastyki"...
 WriteLn ('SDL_SetAlpha error : ', SDL_GetError);
   SDL_FreeSurface(ball); SDL_FreeSurface(background);
   SDL_FreeSurface(screen); SDL_Quit; exit
end;
ball^.format^.Amask := $FF000000;                  wskazania, które bity to kanał alfa.
ball^.format^.Ashift := 24;
```

```
   if SDL_BlitSurface(background, nil, screen, nil) < 0 then
      WriteLn('BlitSurface background error : ', SDL_GetError);
   with SrcRect do begin
      x := 0; y := 0; w := ball^.w; h := ball^.h
   end;
   with DstRect do begin
      x := pixX; y := pixY; w := pixX + ball^.w; h := pixY + ball^.h
   end;
   if SDL_BlitSurface (ball, @SrcRect, screen, @DstRect) < 0
   then WriteLn ('BlitSurface ball error : ', SDL_GetError);
   SDL_Flip (screen);

   ReadLn;
   SDL_FreeSurface(background);
   SDL_FreeSurface(ball);
   SDL_FreeSurface(screen);
   SDL_Quit
end.
```

Przekazujemy „przez wskaźnik” zaznaczenia do skopiowania.

Okno SDL otwarte, ale ciągle mamy dostęp do konsoli.

Zwalniamy zasoby.

Wersja z modułem SDL_image. Kontrolę błędów pominąłem, patrz źródło. Biblioteka SDL_image obsługuje wiele formatów grafiki, nie musi być BMP.

Pod Debianem/Ubuntu potrzebowałem zainstalować `libsdl-image1.2-dev`.

```
{Overlay an image with transparency over a background.}
program ImageAlphaSDL2;
uses SDL, SDL_image;
const
   BackgroundFile = 'background.bmp';
   BallFile = 'ball.bmp';
   pixX = 200; pixY = 200;
var
   screen, background, ball : PSDL_Surface;
   SrcRect, DstRect : SDL_Rect;

begin
   WriteLn ('Initializing SDL.');
   SDL_Init(SDL_INIT_VIDEO);
   background := IMG_Load (BackgroundFile);
   screen := SDL_SetVideoMode(background^.w, background^.h,
                              background^.format^.BitsPerPixel,
                              SDL_SWSURFACE);
```

```
    ball := IMG_Load (BallFile);
    SDL_BlitSurface(background, nil, screen, nil);
    with SrcRect do begin
       x := 0; y := 0; w := ball^.w; h := ball^.h
    end;
    with DstRect do begin
       x := pixX; y := pixY; w := pixX + ball^.w; h := pixY + ball^.h
    end;
    SDL_BlitSurface (ball, @SrcRect, screen, @DstRect);
    SDL_Flip (screen);

    ReadLn;
    SDL_FreeSurface(background);
    SDL_FreeSurface(ball);
    SDL_FreeSurface(screen);
    SDL_Quit
end.
```

Przezroczystość ustawiona automatycznie.

Procedury i funkcje

Często słów „procedura" i „funkcja" używa się zamiennie na oznaczenie „*subroutine*" – wywoływalnego fragmentu kodu. W tym ogólnym sensie są użyte po prawej stronie definicji poniżej.

Procedura to funkcja która zawsze jest wywoływana jako instrukcja – nie może uczestniczyć w wyrażeniach.

Funkcja to procedura która może uczestniczyć w wyrażeniach: jeden z rezultatów obliczeń zwraca jako wynik swojego wywołania w wyrażeniu.

- Typ wyniku funkcji piszemy po `:` za listą argumentów. Wynik funkcji jest wartością automatycznie zadeklarowanej zmiennej która nazywa się tak samo jak funkcja; odwołuje się też do niej słowo kl. `result`.
- Niestrukturalna instrukcja sterująca znana jako `return` w C, nazywa się `exit` w Pascalu. W klasycznym Pascalu służyła tylko do opuszczania procedur, we Free Pascalu można podać jej parametr: `exit(v)` to `return v`.

- Odpowiednikiem `exit` z biblioteki standardowej C, czyli „brutalnego" kończenia procesu (programu), jest `Halt` w Pascalu (z opcionalnym parametrem „exit code"). Jednak lepiej wychodzić z programu głównego również używając `exit`: wtedy sekcje `finally` (m.in. zwalnianie zasobów) będą poprawnie uruchomione.
  - Słowem sprostowania, (Free) Pascal ma instrukcje `break` i `continue`, tak jak C.
- Parametry procedur i funkcji mają kilka *trybów*, które można scharakteryzować jako „wejściowe" – przekazywanie danych do procedury, „wyjściowe" – przekazywanie wyników z procedury, i „dwukierunkowe" – wejściowy i wyjściowy jednocześnie:

domyślny. (wejściowy) Argument przekazany jest przez wartość, tzn. w ramce stosu funkcji utworzona jest nowa zmienna, i wartość argumentu jest do niej skopiowana. Możemy więc korzystać z parametru jako zmiennej lokalnej (odpowiednio zainicjalizowanej).

`const`. (wejściowy) Kompilator może przekazać argument przez wartość lub przez referencję, tzn. przekazać bezpośrednio adres argumentu, ale *nie można parametru modyfikować*. Zawsze moglibyśmy przekazywać dane do funkcji używając tego trybu, ale często używa się trybu domyślnego dla „małych" typów jak typy porządkowe (liczbowe i wyliczeniowe) czy wskaźniki.

`out`. (wyjściowy) Parametr do przekazywania wyników na zewnątrz procedury. Argument musi być zmienną (globalną, lokalną, polem rekordu lub obiektu, komórką tablicy) i jest przekazany przez **referencję**: przypisując do parametru przypiszemy do tej zmiennej; może być niezainicjalizowana. Kompilator zgłosi ostrzeżenie jeśli będziemy odczytywać z tego parametru.

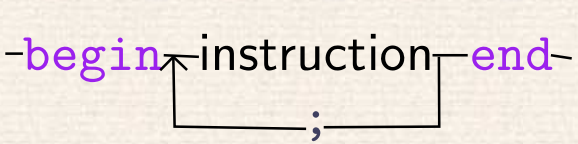
`var`. (dwukierunkowy) Parametr do przekazywania struktur które mają być zmodyfikowane przez procedurę. Podobnie jak przy `out`, przypisanie do parametru zmodyfikuje argument aktualny.

Diagramy składniowe dla procedur i funkcji

Zgrubne diagramy dla fragmentu składni (pomijają wiele konstrukcji i wariantów składni; m.in. klauzulę `uses`).

Plik Pascala:

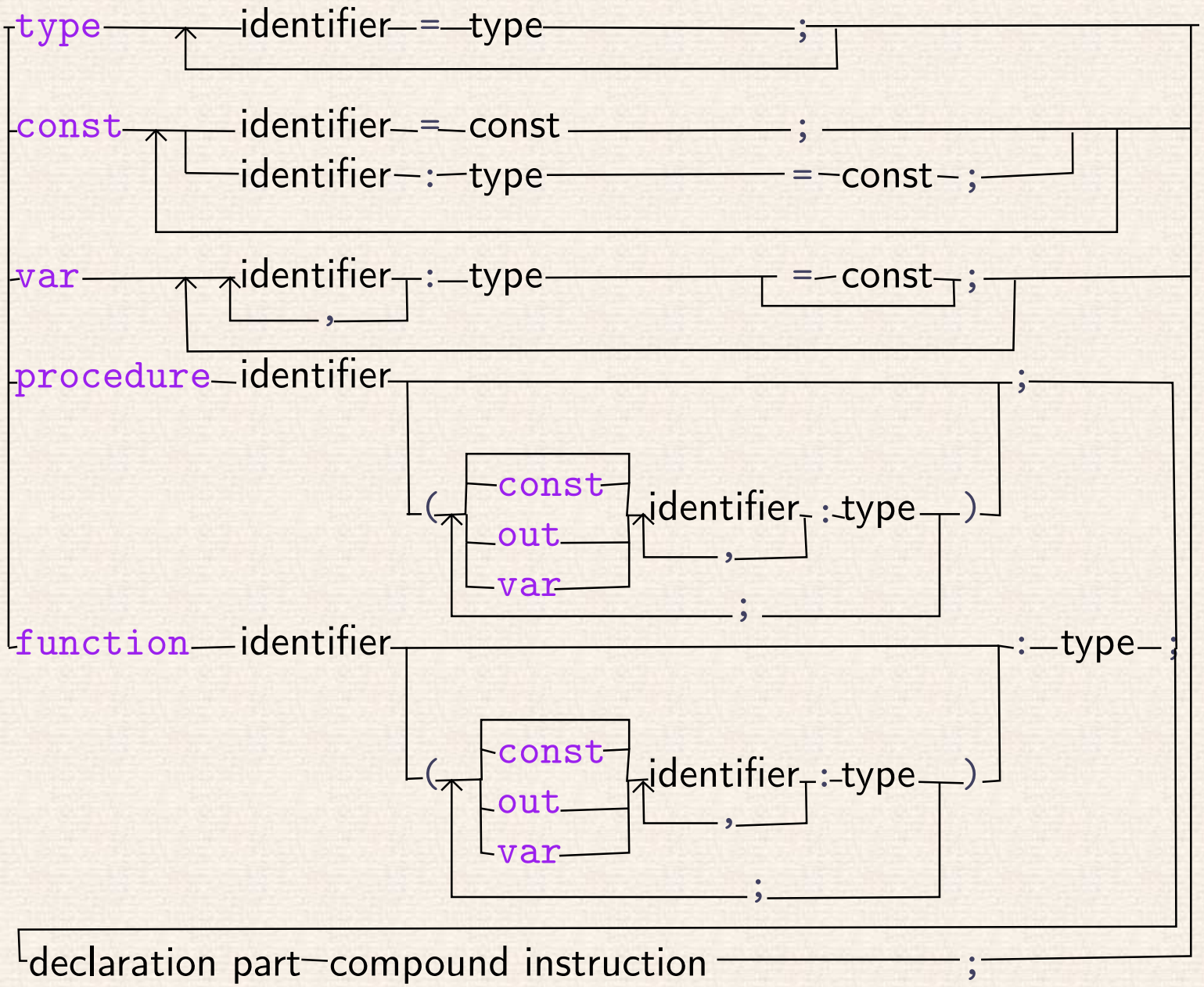
compound instruction:



– zauważ że instrukcje są *oddzielane*,
a nie kończone przez `;`
ale można opcjonalnie użyć `;` przed `end`

(Nie można używać `;` pomiędzy „`then` instrukcja” a „`else` instrukcja”.)

declaration part:


type
identifier = type ;
const
identifier = const ;
identifier : type = const ;
var
identifier , : type = const ;
procedure identifier ;
(const out var identifier , : type ;)
function identifier : type ;
(const out var identifier , : type ;)
declaration part compound instruction ;

Jeszcze o case..of

Nie przedyskutowaliśmy dotąd instrukcji sterującej case..of. Przykład:

```
program CaseOfFigure;
type
   FigureKind = (Point, Circle, Rectangle);
   TFigure = record
      x, y: Real;
      case FigKind : FigureKind of                          Rekord z wariantami.
         Point: ();                                  Wariant bez dodatkowych pól.
         Circle: (radius: Real);
         Rectangle: (width, height: Real)
      end;
      function AsString (const fig : TFigure) : String;
      var sx, sy, sr, sw, sh : String;
      begin
         Str (fig.x:0:2, sx); Str (fig.y:0:2, sy);          Konwersja do napisu
         result := '(' + sx + ',' + sy + ')';      (z formatem :długość:precyzja).
```

```
        case fig.FigKind of
           Point: ;
           Circle:
              begin
                 Str (fig.radius:0:2, sr);
                 result := result + 'circ(' + sr + ')';
              end;
           Rectangle:
              begin
                 Str (fig.width:0:2, sw); Str (fig.height:0:2, sh);
                 result := result + 'rect(' + sw + ',' + sh + ')';
              end
        end
     end;
```

Instrukcja sterująca `case..of`

Wariant `case` bez akcji.

Nie ma potrzeby stosowania `break`.

Wynik funkcji zbudowany używając `result`.

```
var fig : TFigure = (x: 5; y: 5; FigKind: Circle; radius: 2);
begin
   WriteLn (AsString (fig));
end.
```

Procedury pomagają ustrukturować program

```
program SelfAvoidingWalk;
var
   N : Integer;                    {Size of the board.}
   A : array of array of Boolean;    {The board.}
   DeadEnds : Integer; {Counting dead-end walks.}

   procedure Initialize;
   var x, y : Integer;
   begin
      for x := 0 to N-1 do
         for y := 0 to N-1 do
            A[x,y] := false;
   end;
```

```
procedure RandomWalk;
var
   x, y : Integer;
   DeadEnd : Boolean;
   r : Real;
begin
   Initialize;
   x := N div 2; y := N div 2;
   DeadEnd := false;
   while not DeadEnd
         and (x > 0) and (x < N-1) and (y > 0) and (y < N-1) do
   begin
      A[x,y] := true;
      {Check for dead end and make a random move.}
      if A[x-1,y] and A[x+1,y] and A[x,y-1] and A[x,y+1] then
      begin
         Inc (DeadEnds); DeadEnd := true
      end else begin
         r := random;
         if      r < 0.25 then begin if not A[x+1,y] then Inc(x); end
         else if r < 0.50 then begin if not A[x-1,y] then Dec(x); end
         else if r < 0.75 then begin if not A[x,y+1] then Inc(y); end
         else if r < 1.00 then begin if not A[x,y-1] then Dec(y); end
      end
   end
end;
```

```
var
   walk, T : Integer;
   ParseCode : Word;

begin
   Val (ParamStr (1), N, ParseCode); if ParseCode <> 0 then exit;
   Val (ParamStr (2), T, ParseCode); if ParseCode <> 0 then exit;
   SetLength (A, N, N);
   Randomize;
   DeadEnds := 0;
   for walk := 1 to T do RandomWalk;
   WriteLn ((100*DeadEnds) div T, '% dead ends')
end.
```

Animacja BouncingBall, zdarzenia SDLa

```
{Move part of a BMP image bouncing from edges of the window.}
program ImageBlitSDL;
uses SDL;
const
   BackgroundFile = 'background.bmp';
   BallFile = 'ball.bmp';
var
   screen, background, ball : PSDL_Surface;
   procedure DrawBall (const pixX, pixY : Integer);
   var
      SrcRect, DstRect : SDL_Rect;
   begin
      if (pixX >= 0) and (pixY >= 0)
         and (pixX + ball^.w < screen^.w) and (pixY + ball^.h < screen^.h)
      then begin
         with SrcRect do begin
            x := 0; y := 0; w := ball^.w; h := ball^.h
         end;
         with DstRect do begin
            x := pixX; y := pixY; w := pixX + ball^.w; h := pixY + ball^.h
         end;
         SDL_BlitSurface (ball, @SrcRect, screen, @DstRect)
      end
   end;
```

Deklarujemy `Quitted` przed definicją, żeby się ładniej zmieściło na slajdach.

```
function Quitted : Boolean;

procedure Main;
var rx, ry, vx, vy : Real;
begin
   rx := 0.48 * screen^.w / 2;
   ry := 0.86 * screen^.h / 2;
   vx := 0.015 * screen^.w / 2;
   vy := 0.023 * screen^.h / 2;
   repeat
      SDL_BlitSurface(background, nil, screen, nil);
      if (rx + vx < 0) or (rx + vx >= screen^.w - ball^.w)
      then vx := -vx;
      if (ry + vy < 0) or (ry + vy >= screen^.h - ball^.h)
      then vy := -vy;
      rx := rx + vx;
      ry := ry + vy;
      DrawBall (round (rx), round (ry));
      SDL_Flip (screen);
      SDL_Delay (20)
   until Quitted;
end;
```

Oddajemy 20 milisekund systemowi (nie idealne).

```
   function Quitted : Boolean;
   var event : TSDL_Event;
   begin
      Quitted := False;
      while SDL_PollEvent (@event) = 1
      do case event.type_ of
         SDL_KeyDown:
            Quitted := Quitted or (event.key.keysym.sym = LongWord ('q'));
         SDL_QuitEv: Quitted := True;
      end;
   end;

begin
   SDL_Init(SDL_INIT_VIDEO);
   background := SDL_LoadBMP (BackgroundFile);
   screen := SDL_SetVideoMode(background^.w, background^.h,
                              background^.format^.BitsPerPixel,
                              SDL_HWSURFACE or SDL_DOUBLEBUF);
   ball := SDL_LoadBMP (BallFile);
   SDL_SetAlpha(ball, SDL_SRCALPHA, SDL_ALPHA_OPAQUE);
   ball^.format^.Amask := $FF000000;
   ball^.format^.Ashift := 24;
   Main;
   SDL_FreeSurface(background); SDL_FreeSurface(ball);
   SDL_FreeSurface(screen); SDL_Quit
end.
```

Pętlimy po zdarzeniach żeby wyczyścić kolejkę.

Przykład `Percolation`, rysowanie pikseli

```
{Percolation of a fluid through a porous material.}
program Percolation;
uses sdl;
function Quitted : Boolean;
var screen : PSDL_Surface;
   procedure PutPixel (const x, y : Integer; const r,g,b : Byte);
   var
      PixelColor : LongWord;
      PixelLocation : ^LongWord;
   begin
      PixelColor := SDL_MapRGB (screen^.format, r, g, b);
      PixelLocation := screen^.pixels +
         y * screen^.pitch + x * screen^.format^.BytesPerPixel;
      PixelLocation^ := PixelColor;
   end;
type Board = array of array of Boolean;
   procedure DrawLayer (const layer : Board; const r,g,b : Byte);
   var i, j : Integer;
   begin
      for i := 0 to High (layer) do
         for j := 0 to High (layer[0]) do
            if layer[i,j] then PutPixel (i, j, r,g,b)
   end;
```

Definicja `Quitted` jak wcześniej.

Free Pascal ma arytmetykę wskaźnikową tak jak C ale `screen^.pixels` jest typu `Pointer` tzn. przeskakuje po jednym bajcie (a nie po czterech bajtach sugerowanych przez `PixelLocation`).

Tablice dynamiczne pamiętają swój rozmiar.

```
var
   open : Board;
   full : Board;
   function Percolated : Boolean;
   var i, depth : Integer;
   begin
      depth := High (full[0]);
      Percolated := False;
      for i := 0 to High (full) do
         Percolated := Percolated or full[i, depth]
   end;
var
   width, depth : Integer;
   procedure Flow (const i,j : Integer);
   begin
      if (i >= 0) and (i < width) and (j >= 0) and (j < depth)
         and open[i,j] and not full[i,j]
      then begin
         full[i,j] := True;
         Flow (i+1,j);
         Flow (i,j+1);
         Flow (i-1,j);
         Flow (i,j-1)
      end
   end;
```

Sprawdzamy czy płyn przeniknął do dolnej krawędzi.

Jeśli płyn może wpłynąć do tej komórki to ją zapełniamy oraz popychamy płyn do sąsiednich komórek rekurencyjnie.

```
var
   i, j : Integer;
   prob : Real;
   trials : Integer = 0;
   successes : Integer = 0;
begin
   WriteLn ('Percolation.');
   Write ('width of material = '); ReadLn (width);
   Write ('depth of material = '); ReadLn (depth);
   SetLength (open, width, depth);
   SetLength (full, width, depth);
   Write ('probability of pores (cool number is 0.593) = ');
   ReadLn (prob);

   SDL_Init(SDL_INIT_VIDEO);
   screen := SDL_SetVideoMode(width, depth, 8,
                              SDL_SWSURFACE or SDL_DOUBLEBUF);
```

Inicjalizacja programu.

```
  repeat
    for i := 0 to width - 1 do                          Inicjalizacja eksperymentu.
      for j := 0 to depth - 1 do
      begin
        full[i,j] := False;
        open[i,j] := random < prob                      Czy pole jest wolne, czy zapełnione.
    end;
    for i := 0 to width - 1 do                          Wlewamy płyn z całej górnej krawędzi.
      Flow (i, 0);
    Inc (trials); if Percolated then Inc (successes);   Aktualizacja statystyk.
    DrawLayer (open, $ff, $ff, 0);                      Pory na żółto.
    DrawLayer (full, 0, 0, $ff);                        Płyn na niebiesko.

    SDL_Flip (screen);                                  Aktualizuj widok.
    SDL_Delay (20)
  until Quitted;

  SDL_FreeSurface(screen);
  SDL_Quit;
  WriteLn ('Trials: ', trials, '; percolation probability: ',
           successes / trials :0:3)
end.
```

Obiekty typów `object`

Na tym wykładzie tylko krótki rzut oka na typy `object`.

- Na potrzeby tego wykładu, typy `object` są takie jak `record`, tylko pozwalają by część pól była procedurami lub funkcjami: **metodami** manipulującymi obiektem.
- W prototypie obiektu, tzn. w definicji typu obiektowego, pola (zwykłe) poprzedzają metody (dla danej deklaracji widzialności – o tym na kolejnym wykładzie).
- Metodę deklarujemy w definicji obiektu podając jej sygnaturę: część definicji procedury/funkcji poprzedzającą bloki deklaracji, zawierającą tryby i typy parametrów (argumentów), i w przypadku funkcji typ wyniku.
- Oprócz metod `procedure` i `function` są też metody: konstruktory `constructor` oraz destruktory `destructor`.
- Do pól i metod odnosimy się notacją kropkową: `AnObject.AField`.
- Definiujemy metody poniżej deklaracji prototypu obiektu (tutaj, typu `object`), używając nazwy kwalifikowanej, np.:
  `procedure TAnObject.AMethod`.

- Konstruktory i destruktory wywołujemy ręcznie: jeśli (jeden z) konstruktor(ów) obiektów `TObj`, ze wskaźnikami `PObj`, nazywa się `Init` i bierze argument `arg`, to mamy kilka możliwości:

```
program ObjectCreation;
type
   TObj = object
      num : Integer;
      constructor Init (arg : Integer);
   end;
   PObj = ^TObj;
   constructor TObj.Init (arg : Integer);
   begin
      num := arg - 1
   end;
var
   o : TObj;
   p1, p2, p3 : PObj;
begin
   o.Init (1);
   p1 := new (PObj, Init (2));
   new (p2, Init (3));
   new (p3); p3^.Init (4);

   WriteLn (o.num, ', ', p1^.num, ', ', p2^.num, ', ', p3^.num)
end.
```

Ten wariant nie jest polecany! Daje ostrzeżenie.

Moduł `matrix` do operacji 2D,3D,4D

- Moduł `matrix` udostępnia operacje na przydatnych w grafice komputerowej macierzach (kwadratowych) i wektorach 2-, 3- i 4-wymiarowych.
- Typ `Real`, pomimo atrakcyjnej (choć trochę mylącej) nazwy, jest uważany za przestarzały. Używa się raczej typów `Single`, `Double`, `Extended` w zależności od potrzebnej precyzji. Moduł `matrix` jest wyspecjalizowany dla tych trzech typów.
- Obiekty modułu `matrix` posiadają tylko jedno pole danych, `data`: tablicę liczb odpowiednio jedno- (`array of`) lub dwu-indeksową (`array of array of`).
- Mają konstruktory `init` biorące jako argumenty wszystkie liczby do wypełnienia tablic, oraz konstruktory bezargumentowe `init_zero`.
- Schemat nazw typów: `Tvector`*N*`_`*precision* oraz `Tmatrix`*N*`_`*precision*, gdzie *N*=2,3,4 to wymiar, a `precision=single, double, extended`.

- Operatory przypisania i arytmetyczne są przeciążone dla tych typów na „wszystkie sensowne sposoby". Przy czym: `v**w` liczy iloczyn skalarny $v \cdot w$, `v><w` liczy iloczyn wektorowy $v \times w$, a `v*w` liczy iloczyn po współrzędnych $(v_0 w_0, \ldots, v_{n-1} w_{n-1})$.
- Macierze mają metody:
  - zwracającą odpowiednią liczbę: `determinant`,
  - zwracające nowe macierze: `inverse` i `transpose`,
  - zwracające opdowiednie wektory: `get_row` i `get_column`,
  - modyfikujące macierz: `set_row` i `set_column`,
  - oraz konstruktory: `init` biorący wszystkie liczby,
  - bezargumentowe: `init_zero` i `init_identity`.
- Wektory mają metody:
  - zwracające odpowiednią liczbę: `length` i `squared_length` (długość euklidesowa)
  - konstruktory: `init`, bezargumentowe `init_one` i `init_zero`.

Przykład: fraktale *Iterated Function Systems*

```
{Draw a fractal and save it as a BMP image.}
program ChaosGameSDL;
uses SDL, matrix;
const
   WIDTH = 640;
   HEIGHT = 480;
   IFS_VectorsNum = 4;
   IFS_Barnsley_fern_probs : array[0..2] of Real = (0.02, 0.15, 0.13);
   IFS_fern_1: array[0..2,0..2] of double = ((0, 0, 0.5),
                                             (0, 0.27, 0),
                                             (0, 0, 1));
   IFS_fern_2: array[0..2,0..2] of double = ((-0.14, 0.26, 0.57),
                                             (0.25, 0.22, -0.04),
                                             (0, 0, 1));
   IFS_fern_3: array[0..2,0..2] of double = ((0.17, -0.21, 0.41),
                                             (0.22, 0.18, 0.09),
                                             (0, 0, 1));
   IFS_fern_4: array[0..2,0..2] of double = ((0.78, 0.03, 0.11),
                                             (-0.03, 0.74, 0.27),
                                             (0, 0, 1));
```

Dane do wypełnienia macierzy.

Rysowanie pikseli na ekranie.

```
var
   screen : PSDL_Surface;
   procedure PutPixel (const x, y : Integer; const r,g,b : Byte);
   var
      PixelColor : LongWord;
      PixelLocation : ^LongWord;
   begin
      PixelColor := SDL_MapRGB (screen^.format, r, g, b);
      PixelLocation := screen^.pixels +
         y * screen^.pitch + x * screen^.format^.BytesPerPixel;
      PixelLocation^ := PixelColor;
   end;
```

Używamy pierwszej ćwiartki układu współrzędnych: $[0,1)^2$

```
   procedure DrawPoint (const x, y, r, g, b : Real);
   begin
      Assert ((0 <= r) and (r <= 1) and (0 <= g) and (g <= 1)
              and (0 <= b) and (b <= 1),
              'ChaosGameSDL.DrawPoint: RGB component outside 0..1 range');
      if (0 <= x) and (x < 1) and (0 <= y) and (y < 1)
      then PutPixel (trunc (x * WIDTH), HEIGHT - trunc (y * HEIGHT) - 1,
                     round ($ff * r), round ($ff * g), round ($ff * b));
   end;
```

weź część całkowitą: `trunc`

zaokrąglij do najbliższej całkowitej: `round`

```
var
   {We use an N+1 dimensional matrix representation
    of affine transformations.}
   IFS_Vectors : array[0..IFS_VectorsNum-1] of Tmatrix3_double;
   {The last vector probability is: 1 - sum of the others.}
   IFS_probs : array[0..IFS_VectorsNum-2] of Real;
   r, c : Real;
   epochs, t, branch : Integer;
   point : Tvector3_double;
   FileName : String;
begin
   IFS_probs := IFS_Barnsley_fern_probs;
   IFS_Vectors[0].data := IFS_fern_1;
   IFS_Vectors[1].data := IFS_fern_2;
   IFS_Vectors[2].data := IFS_fern_3;
   IFS_Vectors[3].data := IFS_fern_4;

   SDL_Init (SDL_INIT_VIDEO);
   screen := SDL_SetVideoMode(WIDTH, HEIGHT, 32, SDL_SWSURFACE);

   Write ('How many iterations: ');
   ReadLn (epochs);
   {The third dimension must always be 1 -- it produces translation.}
   point.init (0, 0, 1);
```

Zazwyczaj nie wywołanie konstruktora jest błędem! Ale uwzględniając specyfikację modułu `matrix` możemy sobie pozwolić.

```
    c := 1;
    for t := 1 to epochs do
    begin
       r := random; c := (c + r) / 2; branch := 0;
       while (branch < Length (IFS_probs))
             and (IFS_probs[branch] < r) do
       begin
          r := r - IFS_probs[branch];
          Inc (branch)
       end;
       point := IFS_Vectors[branch] * point;
       {Print the point with a fancy color.}
       DrawPoint (point.data[0], point.data[1], c, (c + 1) / 2, 1 - c)
    end;

    SDL_Flip (screen);
    Write ('Give name for the image: ');
    ReadLn (FileName);
    SDL_SaveBMP (screen, PChar (FileName+'.bmp'));

    SDL_Quit
end.
```

Kolor wykładniczo zapomina o starych wyborach gałęzi. (Poeksperymentuj ze zmianą wag dla c i r.)

Inicjalizacja macierzy konstruktorem.

```
{Draw a fractal and save it as a BMP image.}
program ChaosGameSDL;
uses SDL, matrix;
const
   WIDTH = 640;
   HEIGHT = 480;
var
   screen : PSDL_Surface;
   procedure PutPixel (const x, y : Integer; const r,g,b : Byte);
   var
      PixelColor : LongWord;
      PixelLocation : ^LongWord;
   begin
      PixelColor := SDL_MapRGB (screen^.format, r, g, b);
      PixelLocation := screen^.pixels +
         y * screen^.pitch + x * screen^.format^.BytesPerPixel;
      PixelLocation^ := PixelColor;
   end;
   procedure DrawPoint (const x, y, r, g, b : Real);
   begin
      Assert ((0 <= r) and (r <= 1) and (0 <= g) and (g <= 1)
              and (0 <= b) and (b <= 1),
              'ChaosGameSDL.DrawPoint: RGB component outside 0..1 range');
      if (0 <= x) and (x < 1) and (0 <= y) and (y < 1)
      then PutPixel (trunc (x * WIDTH), HEIGHT - trunc (y * HEIGHT) - 1,
                     round ($ff * r), round ($ff * g), round ($ff * b));
   end;
```

```
var
   IFS_Sierpinski : array[0..2] of Tmatrix3_double;
   {The last vector probability is: 1 - sum of the others.}
   IFS_Sierpinski_probs : array[0..1] of Real = (1/3, 1/3);

   r, c, x, y : Real;
   epochs, t, branch : Integer;
   point : Tvector3_double;
   FileName : String;
begin
   FillByte(IFS_Sierpinski, SizeOf(IFS_Sierpinski), 0);
   IFS_Sierpinski[0].init (1/2, 0, 0,
                           0, 0.5, 0,
                           0, 0, 1);
   IFS_Sierpinski[1].init (1/2, 0, 1/2,
                           0, 1/2, 0,
                           0, 0, 1);
   IFS_Sierpinski[2].init (1/2, 0, 0.5 / 2,
                           0, 1/2, sqrt (3) / 4,
                           0, 0, 1);
   SDL_Init (SDL_INIT_VIDEO);
   screen := SDL_SetVideoMode(WIDTH, HEIGHT, 32, SDL_SWSURFACE);
   Write ('How many iterations: ');
   ReadLn (epochs);
   point.init (0, 0, 1);
```

Bez tej linijki inicjalizującej pamięć jest ostrzeżenie ale nie jest ona potrzebna...

```
    c := 1;
    for t := 1 to epochs do
    begin
       r := random; c := (c + r) / 2; branch := 0;
       while (branch < Length (IFS_Sierpinski_probs))
             and (IFS_Sierpinski_probs[branch] < r) do
       begin
          r := r - IFS_Sierpinski_probs[branch];
          Inc (branch)
       end;
       point := IFS_Sierpinski[branch] * point;
       x := point.data[0]; y := point.data[1];
       DrawPoint (x, y, c, (c + 1) / 2, 1 - c);
    end;
    SDL_Flip (screen);
    Write ('Give name for the image: ');
    ReadLn (FileName);
    SDL_SaveBMP (screen, PChar (FileName+'.bmp'));
    SDL_Quit
end.
```