Wypiski o pascalu

grzes2a
z bloga „Programowanie to zabawa”

2023

Spis treści

Rozdział 1

Przed trivium

1.1 Startujemy z pascalem

Pascal jest dobrym językiem na początek. Zresztą został wymyślony do nauczania. Miał w prosty sposób pokazywać algorytmy studentom politechniki w Zurychu, uczącym się programowania w roku 1970. Stare dzieje tego języka mogą nas nie interesować, więc od razu przejdźmy do lat 80-tych, kiedy to firma Borland wypuściła całe środowisko do tworzenia programów na komputerach PC. Dzięki kilku fartownym decyzjom tej firmy, pascal miał kilka chwil chwały i potęgi. W XXI wieku dzięki kilku pechowym decyzjom tej samej firmy, pascal znikł prawie całkowicie z rynku. Ta historia źle się kończy, ale nie zmienia to mojego zdania, że do nauki pierwszych kroków nie wymyślono chyba nic lepszego.

Na czym?

We współczesnym świecie, czasami nie jest łatwo napisać swój pierwszy program. Dla niektórych języków programowania, trzeba instalować potężne aplikacje deweloperskie. Gdzież te czasy, gdy wystarczyło po prostu… zacząć pisać. Szczęśliwie próg wejścia w pascalu nie jest wysoki. Damy radę.

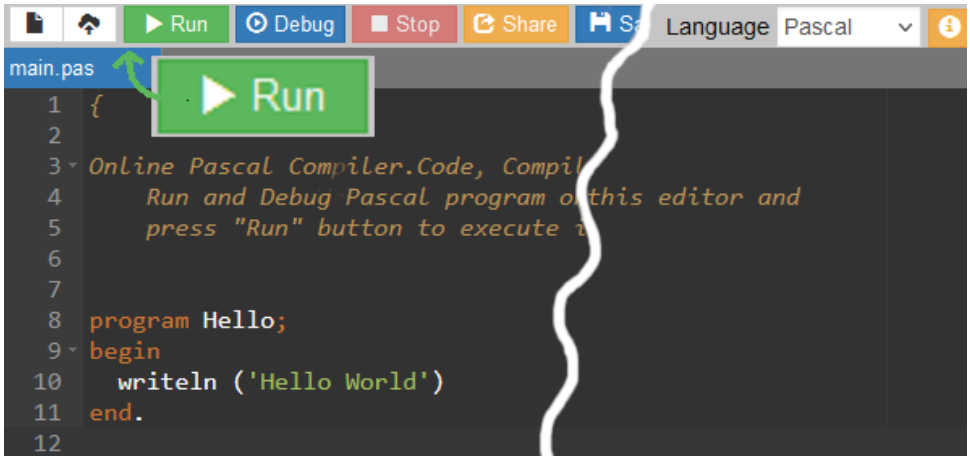
Żeby zacząć, możemy ściągnąć sobie ze strony `https://www.freepascal.org/download.html` odpowiedni plik instalacyjny. Więcej o tym i innych środowiskach przeczytasz na str. 47.

Dla osób, które chcą tylko się pobawić, niekonieczna jest nawet instalacja. W sieci istnieje kilka serwisów które umożliwiają pisanie i uruchamianie programów:

- `https://www.onlinegdb.com/online_pascal_compiler`
- `https://www.tutorialspoint.com/compile_pascal_online.php`

- `https://ideone.com/`

To kilka miejsc w sieci, jakie podpowiedziała mi wyszukiwarka. Pomimo, że stanowią interfejs do podobnej linuksowej instalacji Free Pascala, ich funkcjonalność jest różna – pierwsza pozycja zachowywała się najbardziej poprawnie, więc do niej będę się poniżej odwoływał. Zdaje się, że na wszystkich stronach można wybrać również inne języki programowania, co jest pocieszające, jeśli po uzyskaniu jako-takiej sprawności będziemy się chcieli zwrócić ku „poważniejszym" produktom. Na razie jednak pozostajemy przy pascalu.



Jak tam wejdziesz, to od razu zobaczysz najprostszy program, który już coś robi. Możesz go od razu uruchomić, w tym celu musisz kliknąć przycisk „Run" i wtedy poniżej ukaże się tekst:

```
Free Pascal Compiler version 3.2.2+dfsg-9ubuntu1 [2022/04/11] for x86_64
Copyright (c) 1993-2021 by Florian Klaempfl and others
Target OS: Linux for x86-64
Compiling main.pas
Linking a.out
11 lines compiled, 0.1 sec
Hello World


...Program finished with exit code 0
Press ENTER to exit console.
```

Wśród tych napisów interesuje nas linia zawierająca `Hello World`, reszta to komunikaty które być może w przyszłości dadzą nam dodatkowe informacje o procesie wykonywania napisanego przez nas programu. Na razie zajrzyjmy, co też ciekawego znajduje się w kodzie programu.

Pierwszy program

Tekst programu nazywa się kodem. Często programiści mówią o sobie „koderzy". Rymuje się to ze słowem „szpanerzy", ale nie wiem czy słusznie ☺. Cytowane

środowisko proponuje kod składający się z 11 linijek (są tam numerowane na wypadek, gdyby pojawił się błąd – wtedy szybko można go odnaleźć, jeśli wiemy, w której linii się znajduje). Oto owe 11 linijek:

```
{

Online Pascal Compiler.Code, Compile,
    Run and Debug Pascal program online.Write your code in this editor and
    press "Run" button to execute it.}


program Hello;
begin
  writeln ('Hello World')
end.
```

Pierwsze pięć zawiera tzw. komentarz. Jest to tekst, który nie będzie wykonywany. Po co więc go pisać? Kod programu jest uruchamiany na komputerze – tu komentarze nie są potrzebne. Ale zanim zostanie uruchomiony, czytają go programiści – czasami zapominają, po co napisali to, a nie co innego. Żeby sobie odświeżyć pamięć, zaglądają do komentarzy. O ile wcześniej je napisali ☺. Zawodowi koderzy piszą komentarze także z innych powodów. Żeby przyzwyczaić cię do komentowania, czasami będę o nich przypominał.

Komentarze można umieszczać na trzy sposoby. Zamykając tekst w nawiasy klamrowe:

```
{
To program do liczenia tego i owego.
Nie jest skończony - należy uzupełnić to i tamto }
```

albo w parę `(* *)`

```
(*
Inna postać komentarza
*)
```

Istnieje możliwość wpisywania komentarzy jednolinijkowych:

```
// Poniższe miejsce jest warte uwagi...
```

W następnej linijce nie będzie już komentarza.

Teraz pora na dalsze linijki. O tym, że mamy do czynienia z programem – bo w pascalu można pisać nie tylko programy – informuje nas linia:

```
program Hello;
```

Każdy program będzie zaczynał się od takiej instrukcji. Pojedyncze polecenia w programie nazywane są instrukcjami albo komendami. Instrukcje oddziela się od siebie średnikami. Będzie z tym trochę zamieszania, ale o tym później.

Treść programu jest umieszczona między słowami `begin` i `end`. Zwróć uwagę na kropkę po słowie `end` – ona tam musi być. Jak widać owa treść ogranicza się do jednej linijki:

```
writeln ('Hello World')
```

Jak można się domyślić oznacza ona wypisanie napisu `Hello World`. Jak będziemy chcieli umieścić jakiś napis w kodzie programu, będziemy go oznaczać apostrofami. Czyli w przypadku spolszczenia treści napiszemy:

```
writeln ('Witaj świecie')
```

Ponieważ korzystam z gotowego pakietu formatującego postać kod programu, może cię zdziwić oznaczanie spacji w napisach. Zauważ dziwny znaczek pomiędzy `Witaj` a `świecie`. To właśnie spacja wewnątrz napisu.

Wielkość liter w kodzie

Pascal jest językiem, w którym wielkość liter w kodzie nie ma znaczenia. Oznacza to, że komendę tę można zapisać tak:

```
WRITELN ('Witaj świecie')
```

albo tak

```
WriteLn( 'Witaj świecie' )
```

I ten sposób jest najbardziej przejrzysty, bo pokazuje, że oznacza ona dosłownie „Write Line”. Jeśli więc będziemy chcieli utworzyć jakąś wielkość, którą wygodnie da się opisać za pomocą dwóch wyrazów, to ten sposób będzie akuratny: `RozmiarTablicy`, `DlugoscZycia` itp.

Osobiście wolę nieco inaczej formatować funkcje i procedury – bo polecenie `WriteLn()` to właśnie procedura – niż proponuje środowisko zacytowane wyżej. Mam jeszcze jedno przyzwyczajenie dotyczące formatowania: Ponieważ par `begin end` może w programie pojawić się bardzo dużo, te które zawierają teść programu – mówiło się na to program główny – piszę wielkimi literami:

```
Program HaloPoPolsku;
BEGIN
  WriteLn( 'Ahoj, przygodo' )
END.
```

Skąd wzięło się `Ahoj, przygodo` innym razem.

1.2 Zmienne

Zanim przejdę do wiadomościach o zmiennych, jedna informacja: instrukcje w programie oddzielamy średnikami. Podobnie, ale jednak inaczej niż w językach z rodziny C (C++, java, C#) i jest z tym trochę zamieszania. Na razie, dla prostoty przyjmijmy, że po każdej instrukcji stawiamy ; (co w ogólności okaże się nieprawdą).

W drugiej części poznamy sposoby na przetwarzanie danych w pisanym programie. Co może być danymi? Na przykład liczby i napisy. Na początek tyle nam wystarczy.

Każda pojedyncza dana będzie miała swoje miejsce w programie, nazwane konkretną nazwą. Możemy sobie wyobrazić, że to jakieś miejsce w pamięci komputera. W programie będzie ono dostępne poprzez tzw. zmienną. W przypadku dużej liczby danych, będziemy musieli je pogrupować, ale na razie pobawmy się pojedynczymi zmiennymi.

Liczby

Jeśli więc potrzebna nam będzie liczba, to trzeba poinformować o tym program – mówi się: zadeklarować zmienną. Robi się to przed słowem `BEGIN` w następujący sposób:

```
var liczba : integer;
BEGIN
  ...
END.
```

Zapis ten oznacza, że w programie możemy posługiwać się zmienną `liczba`, będącą liczbą całkowitą (`integer`). Tak właśnie deklarujemy zmienne.

Jak już napisałem pascal nie rozróżnia wielkości znaków w nazwach. Oznacza to, że nazwy `liczba`, `Liczba` i `LICZBA` oznaczają tę samą zmienną. Nazwy mogą składać się z liter, cyfr i podkreślnika `_`, przy czym cyfra nie może być pierwszym znakiem.

Wiedząc, co ma robić program, możemy sobie zaplanować ile zmiennych będzie nam potrzebne. Załóżmy, że potrzebujemy policzyć odwrotność jakiejś liczby całkowitej. Odwrotność jest już liczbą zmiennoprzecinkową. Popularnie (choć z matematycznego punktu widzenia niepoprawnie) mówi się o liczbach rzeczywistych. Zwykle używa się typów: `real` i `double`. Historycznie `real` był podstawowym typem dla tego typu liczb, i ten właśnie zastosujemy:

```
var
  liczba : integer;
  odwr : real;
BEGIN
```

```
  ...
END.
```

Pozostaje nam tylko nadać jakąś wartość zmiennej `liczba` i policzyć ową odwrotność. Są dwie możliwości nadawania wartości. Można napisać w programie ile ona ma być równa, albo poprosić użytkownika o podanie owej liczby.

Pierwszy sposób na nadanie wartości:

```
liczba := 10;
```

Symbol `:=` składający się z dwukropka i znaku równości nazywa się operatorem przypisania. Ma on za zadanie nadawać i zmieniać wartości zmiennych. On również będzie brał udział w nadaniu wartości zmiennej `odwr`:

```
odwr := 1/liczba;
```

Jak widać pojawił się kolejny operator / (dzielenie). Zobaczmy jak wygląda cały program do liczenia odwrotności:

```
program LiczenieOdwrotnosci;
var
  liczba : integer;
  odwr : real;
BEGIN
  liczba := 10;
  odwr := 1/liczba;
  WriteLn( 'Odwrotność␣liczby␣', liczba);
  WriteLn( 'jest␣równa␣', odwr );
END.
```

Wynik jest dość zaskakujący:

```
Odwrotność liczby 10
jest równa 1.0000000000000001E-001
```

Powyższy zapis jest równoważny stwierdzeniu, że wyliczona odwrotność to: $1{,}0000000000000001 \times 10^{-1}$, czyli inaczej 0,10000000000000001.

Po pierwsze powinno być 0,1 – komputer ewidentnie coś pokręcił przy wyliczaniu. W przypadku liczb zmiennoprzecinkowych trzeba pamiętać, że każde działanie ma swoją dokładność. Ta jedynka na końcu wyniku, to właśnie objaw tego, że komputer nie liczy z nieskończoną dokładnością. Nawiasem mówiąc, jeśli chodzi o liczby zmiennoprzecinkowe, to poprawnie/dokładnie zapisywane, są tylko liczby wymierne postaci: (jakaś niewielka liczba całkowita)/(jakaś niewielka potęga 2). Można się domyślać, że wypisana na ekranie liczba też jest przybliżonym przedstawieniem wartości przechowywanej w zmiennej `odwr`.

Po drugie zamiast przecinka jest kropka. Tak zapisuje się ułamki dziesiętne w krajach anglosaskich, skąd pochodzi duża część kultury informatycznej. Dlatego właśnie w kodzie programu przecinek w ułamkach zapisuje się kropką.

Po trzecie widzimy, że liczba jest nieładnie napisana. Można jednak zażyczyć sobie, by napisać ją inaczej. W tym celu stosuje się odpowiednie formatowanie – za nazwą zmiennej, po dwukropkach piszemy jak ma być przedstawiona liczba. Dla programu:

```
WriteLn( liczba );
WriteLn( liczba:3 );
WriteLn( liczba:10 );
WriteLn( liczba:30 );
WriteLn( liczba:2:1 );
WriteLn( liczba:12:5 );
```

uzyskamy następujące wyniki (gdy `liczba` będzie równa 123,456789):

```
 1.23456789000000E+002
 1.2E+002
 1.23E+002
        1.23456789000000E+002
123.5
   123.45679
```

Pierwsza liczba oznacza ile minimalnie ma zajmować miejsca wypisywana zmienna (jak się nie zmieści, to się „rozepcha”). Jeśli po dwukropku pojawi się druga liczba – będzie to oznaczało ile cyfr po przecinku zostanie wypisane.

Skoro pojawił się operator dzielenia, podam pięć kolejnych operatorów arytmetycznych działających na liczbach całkowitych. Przy założeniu, że zadeklarowano dwie zmienne `liczba1` i `liczba2`, wykonanie kodu:

```
liczba1 := 7;
liczba2 := 3;
WriteLn( 'suma ', liczba1+liczba2 );
WriteLn( 'różnica ', liczba1-liczba2 );
WriteLn( 'iloczyn ', liczba1*liczba2 );
WriteLn( 'dziel. całkowite ', liczba1 div liczba2 );
WriteLn( 'reszta z dzielenia ', liczba1 mod liczba2 );
```

da następujący wynik:

```
suma 10
różnica 4
iloczyn 21
dziel. całkowite 2
reszta z dzielenia 1
```

Drugi sposób na nadanie wartości: Użytkownik podaje ile ma wynosić wartość zmiennej. Służy temu polecenie `ReadLn()`. Zobaczmy jak to działa:

```
Write( 'Podaj liczbę: ' );
ReadLn( liczba );
```

Najpierw poinformowaliśmy użytkownika, żeby wiedział co ma wpisać. Potem użytkownik wpisuje co trzeba i po zatwierdzeniu ENETRem, zmienna liczba ma już właściwą wartość.

Zauważmy tylko, że użyliśmy polecenia `Write()`, a nie `WriteLn()` – różnica jest taka, że kursor nie przejdzie do następnej linii. Podobnie ma się z poleceniem `ReadLn()` – istnieje jego wersja `Read()`, szczególnie użyteczna przy czytaniu danych z pliku o czym będzie mowa w przyszłości.

W uzupełnieniu napiszę, że istnieje kilka typów do przechowywania zmiennych całkowitych i zmiennoprzecinkowych. Małe podsumowanie znajduje się na końcu tego odcinka.

Trzeba przyznać, że `ReadLn()` jest dość dobrze pomyślane dla początkującego programisty. Spodziewa się on, że dane wpisane przez niego z klawiatury zostaną wczytane przez program po zatwierdzeniu ENTERem.

Pewna cecha tej komendy będzie widoczna w poniższym przykładzie. Uruchomienie linii:

```
Write( 'Napisz kilka liczb: ' );
ReadLn( liczba1, liczba2 );
```

gdy użytkownik wpisze cztery liczby i zatwierdzi ENTERem:

```
Napisz kilka liczb: 10 20 30 40
```

poskutkuje to nadaniem wartości 10 zmiennej `liczba1` i 20 zmiennej `liczba2`. Teksty 30 i 40 zostaną zignorowane.

* * *

Jeszcze jedna uwaga związana z operatorem przypisania. Ponieważ zastanawiająco często zdarzają się następujące operacje zwiększenia wartości zmiennej całkowitej:

```
liczba := liczba + 1;
```

postanowiono wprowadzić osobną operację inkrementacji – czyli zwiększenia o 1:

```
inc( liczba );
```

Podobnie wprowadzono dekrementację (zmniejszenie o 1):

```
dec( liczba );
{ to samo co: liczba := liczba -1 }
```

Będę stopniowo i niekonsekwentnie wprowadzał ten zapis w niniejszym kursie.

Napisy

Do przechowywania napisów służy typ `string`. Długość takiego napisu nie może być dłuższa od 255 znaków, a często jest jeszcze krótsza. Zwykle to nie przeszkadza, bo trudno o tak długie imię, nazwisko czy inną daną tekstową.

Typ `string` jest wygodny i intuicyjny:

```
program Nazwy;
var polowa1, polowa2, razem :string;
BEGIN
  polowa1 := 'Wars';
  polowa2 := 'zawa';
  razem := polowa1+polowa2;
  WriteLn( razem );
END.
```

Po sklejeniu napisów i wpisaniu ich do zmiennej razem, zostanie wypisany wynik:

```
Warszawa
```

Wartości napisów można też odczytywać z klawiatury – i tu ważna informacja: do `string`-a wpadną wszystkie wpisane znaki aż do wystąpienia ENTERA. Tak jak w poniższym przykładzie, który warto sobie przeanalizować:

```
program Odczytywanie;
var napis :string;
    liczba: integer;
BEGIN
  Write( 'Napisz␣kilka␣liczb:␣' );
  Readln( liczba, napis );
  WriteLn( 'Liczba:␣', liczba );
  WriteLn( 'Napis:␣', napis );
END.
```

Uruchomienie powyższego programu, gdzie użytkownik wpisuje `10 20 30 40`:

```
Napisz kilka liczb: 10 20 30 40
Liczba: 10
Napis: 20 30 40
```

Napisy składają się ze znaków. Typ znakowy to `char`. Obiecuję, że poświęcę cały odcinek na ten temat.

* * *

Niezależnie od możliwości jakie dają procedury `Read()` i `ReadLn()` chyba najprościej jest odczytywać każdą wartość parą instrukcji:

```
Write( 'Podaj␣wartość:␣' );
ReadLn( zmienna );
```

* * *

Poniżej obiecane podsumowanie typów liczbowych Free Pascala.

Typ `integer` jest domyślnym typem całkowitym, jego rozmiar i zakres zależą od środowiska. Powinien być stosowany w przypadkach, gdy nasza zmienna jest niezbyt dużą liczbą całkowitą i nie musimy się zastanawiać nad jej postacią binarną. Czyli w zdecydowanej większości zmiennych całkowitych w niniejszym kursie. Typ `integer` jest identyczny z `SmallInt` lub `LongInt` – to zależy od konfiguracji środowiska Free Pascala. Gdy zakres i zawartość binarna zmiennej ma znaczenie, należy posługiwać się typami:

rozmiar [bajty]	**nazwa**	**zakres**
1	`byte`	$0..255$
	`ShortInt`	$-128..127$
2	`Word`	$0..65535$
	`Smallint`	$-32768..32767$
4	`LongWord` `cardinal` `DWord`	$0..4294967295$
	`LongInt`	$-2147483648..2147483647$
8	`QWord`	$0..18446744073709551615$
	`Int64`	$-9223372036854775808..9223372036854775807$

W podstawowej konfiguracji Free Pascala `integer` jest dwubajtowy (16-bitowy) jak w starożytnym Turbo Pascalu. Konfiguracje bardziej zaawansowane wymuszają rozmiar czterobajtowy. Jeśli posługujesz się dużymi liczbami, niewielką pomocą może być odczytanie stałej `MaxInt`, by zorientować się czy grozi ci przekroczenie zakresu.

Jeśli chodzi o liczby zmiennoprzecinkowe, to podstawowy typ `real` również jest zależny od wersji środowiska i ustawień. Niemniej jeśli mamy dokonywać jakichś działań, to zwykle typ ten powinien nam wystarczyć. W poszczególnych przypadkach może być on identyczny z którymś typem wbudowanym. Tablicę tych typów umieszczam poniżej, przy czym trzy pierwsze są zgodne ze specyfikacją IEEE

754. Ostatni zaś `currency` przeznaczony jest do zastosowań finansowych – jego zadaniem jest poprawne przetwarzanie danych podobnych do ułamka dziesiętnego. Jest to w zasadzie typ całkowity ze zmodyfikowanymi działaniami arytmetycznymi – porównaj jego zakres z całkowitym `Int64`.

rozmiar [bajty]	**nazwa**	**zakres**
4	`single`	$\pm 1.5 \times 10^{-45} .. 3.4 \times 10^{38}$
8	`double`	$\pm 5.0 \times 10^{-324} .. 1.7 \times 10^{308}$
10	`extended`	$\pm 1.9 \times 10^{-4932} .. 1.1 \times 10^{932}$
8	`currency`	$-922337203685477.5808 .. 922337203685477.5807$

Do tych typów dołożymy znaki – na razie powinien nam wystarczyć typ `char` – oraz napisy – typ `string`. Dokładniejszy opis umieszczony jest na str.90-97. Dodatkowo pamiętajmy o typie logicznym: `boolean` opisanym na str. 60.

1.3 Kilka porad przy okazji `if`-a

Trzeci odcinek zawiera kilka spostrzeżeń, jakie powinny się przydać początkującym programistom. I nie ma co udawać, że poprzednie dwa odcinki były jakieś fascynujące – koniecznie trzeba poznać kolejne elementy języka pascal, żeby napisać coś fajniejszego. Zajmiemy się mianowicie instrukcją sterującą `if then else`.

Instrukcja ta pozwala na podejmowanie decyzji, a jej użycie jest następujące:

```
if warunek then
  instrukcje_jesli_prawda
else
  instrukcje_gdy_nieprawda;
```

Przy czym człon z `else` („w przeciwnym razie”) nie jest obowiązkowy. To czy będziemy go wykorzystywać, zależy od logiki programu.

Warunek może być wyrażeniem logicznym, albo zmienną logiczną typu `boolean`. Co może być takim warunkiem? Pascal jest tu bardzo intuicyjny – jeśli wynik wyrażenia da się opisać jako tak/nie albo prawda/fałsz, to prawdopodobnie będzie ono wyrażeniem logicznym w sensie języka. Na przykład stwierdzenie, że jedna liczba jest większa od drugiej: `liczba1>liczba2` jest takim warunkiem. Można wtedy określić, która z liczb jest większa:

```
program KtoraWieksza;
var liczba1, liczba2 : integer;
BEGIN
  Write( 'Podaj␣dwie␣liczby:␣');
```

```
  ReadLn( liczba1, liczba2 );
  if liczba1>liczba2 then
    WriteLn( 'Większa␣liczba␣', liczba1 )
  else
    WriteLn( 'Większa␣liczba␣', liczba2 );
END.
```

Najczęściej w takich przypadkach będzie można natknąć się na specyficzne zachowanie średnika. Na początku nauki programiści wstawiają średniki na końcu każdej instrukcji. Ale średnik nie kończy instrukcji, ale je rozdziela. Jeśli napiszemy taki kod:

```
if liczba1>liczba2 then
  WriteLn( 'Większa␣liczba␣', liczba1 ); //ŹLE
else
...
```

pojawi się błąd kompilatora:

```
Fatal: Syntax error, ";" expected but "ELSE" found
```

Dlaczego tak się stało? W wersji podanej powyżej `if then else` potrafi wykonać pojedynczą instrukcję, a wyrażenie:

```
WriteLn( 'Większa␣liczba␣', liczba1 );
```

oznacza, że za średnikiem jest kolejna instrukcja. Jeśli nie ma jej napisanej, to jest to tzw. instrukcja pusta. A dwóch instrukcji, nawet jeśli jedna z nich jest pusta, `if then else` nie umie obsłużyć. W każdym razie należy zapamiętać, że przed `else` nie powinno być średnika.

Bodaj najczęstszym stosowanym operatorem w konstrukcji `if then` jest operator porównania:

```
if liczba=1 then ...
```

Pasuje on nie tylko do typów liczbowych, ale również do znaków (`char`) a nawet napisów:

```
napis := 'Ala␣ma␣kota';
if napis = 'Ala␣ma␣kota' then
  WriteLn('Odgadłeś␣napis');
```

Warto tu zauważyć różnicę między operatorem przypisania `:=` a porównania `=`. Pierwszy nadaje wartość zmiennej, a drugi tylko sprawdza jej wartość. Gdybyśmy pomylili te operatory w zastosowaniu do `if`, kompilator odnajdzie błąd.

Inne operatory to:

- < mniejszy
- <= mniejszy lub równy
- > większy
- >= większy lub równy
- <> różny – ten operator też często się wykorzystuje.

Popatrzmy jeszcze na kod napisanego programu i zauważmy, że pomimo wypisania większej liczby, program nigdzie nie „zapisał" sobie, która z nich jest większa. A jeśli będziemy tej wartości jeszcze potrzebować? Na razie program ma kilka linijek, więc problem jawi się jako teoretyczny, ale przechowanie wyniku, pozwoli na coś więcej. Przechowajmy więc większą wartość w zmiennej `max`:

```
program ZnajdzWieksza;
var liczba1, liczba2, max : integer;
BEGIN
{ odczytywanie }
  Write( 'Podaj␣dwie␣liczby:␣');
  ReadLn( liczba1, liczba2 );
{ wyliczenie }
  if liczba1> liczba2 then
    max := liczba1
  else
    max := liczba2;
{ wypisywanie }
  WriteLn( 'Większa␣liczba␣', max )
END.
```

Teraz nasz program dorobił się pewnej cechy, która w poważniejszych programach jest bardzo pożądana: chodzi o separację funkcjonalności kodu. Widoczne komentarze dzielą program na trzy w miarę niezależne części: czytanie danych z klawiatury, wyliczenie wyniku i wyświetlenie danych. Ważność tej zasady docenisz przy tworzeniu większych programów.

Zadanie ZTS: Przeanalizuj algorytm znajdujący większą wartość i spróbuj poszerzyć go, żeby wyliczał największą wartość z trzech liczb – trzeba będzie zadeklarować kolejną zmienną `liczba3`.

Czasami trzeba będzie napisać bardziej skomplikowane warunki – na przykład będące iloczynem, alternatywą lub zaprzeczeniem innych warunków. Choćby sprawdzian, czy liczba jest jednocyfrowa, będzie sprawdzał czy liczba jest jednocześnie większa od -10 i mniejsza od 10. Składanie warunków dokonuje się operatorami `and` oraz `or`. Zaprzeczenie uzyskuje się przez wpisanie `not` przed warunkiem. I tu kolejna uwaga – gdybyśmy tak napisali wyrażenie:

```
if liczba>-10 and liczba<10 then //ŹLE!
```

to pojawi się błąd:

```
Error: Incompatible types: got "Boolean" expected "Int64"
```

Chodzi o to, że operatory logiczne mają bardzo wysoki priorytet i powyższy zapis oznacza, że na początku wywołuje się: `-10 and liczba`, co jest bez sensu, bo ani `-10` ani zmienna `liczba` nie są typu `boolean`. Rada jest następująca: warunki składowe zamykamy w nawiasy:

```
if (liczba>-10) and (liczba<10) then
```

Wykorzystajmy ten warunek, jako wstęp do następnego programu: Napiszemy zestaw informacji nt. podanej liczby. Program oceni czy liczba jest duża czy mała, poda znak liczby i być może coś jeszcze. Wtedy jednocyfrowość oznaczałoby, że liczba jest mała (co do wartości bezwzględnej). I tu pojawia się wątpliwość – czy liczby małe to te z przedziału od -9 do 9? A może to te pomiędzy -15 a 15? Oczywiście to autor wybiera granice, ale dobrze byłoby, gdyby mógł gdzieś sobie je zapisać w wyróżnionym miejscu, żeby łatwo było je znaleźć, gdy będzie chciał zmienić ich wartość. Po to w pascalu stosuje się tzw. stałe. Umieszcza się je na początku programu i korzysta w następujący sposób.

```
program InfoOLiczbie;
const Mala = 10;
var liczba : integer;
BEGIN
  Write('Podaj␣liczbę:␣');
  ReadLn( liczba );
  if (liczba>-Mala) and (liczba<Mala) then
    WriteLn( 'To␣mała␣liczba' );
END.
```

Gdybyśmy chcieli zmienić przedziały oznaczające małą liczbę, zrobimy to dokonując tylko jednej zmiany, w dodatku w miejscu, które łatwo znaleźć, bo na początku programu. Gdyby nasz program miał 1000 linii i trzeba byłoby zmienić 25 wystąpień takiej wartości, łatwo byłoby o pomyłkę.

Jeśli wyborów jest kilka, są one rozłączne i obejmują wszystkie możliwości, można składać instrukcje `if` ze sobą. OK, to nie zabrzmiało zrozumiale, więc może przykład: Mamy ocenić znak liczby. Mamy do dyspozycji trzy przedziały: ujemne, zero i dodatnie. Są rozłączne, a ich suma obejmuje wszystkie możliwości. Składamy więc dwa `if`-y ze sobą: pierwszy sprawdzi czy liczba jest ujemna, drugi czy jest zerem:

```
if liczba<0 then
  WriteLn( '-␣ujemna' )
else if liczba=0 then
```

```
  WriteLn( '- zero' )
else WriteLn( '- dodatnia' );
```

W trzeciej linijce widzimy charakterystyczną zbitkę `else if`, ale nie jest to jakaś specjalna konstrukcja – po prostu, jeśli pierwszy warunek nie jest spełniony, dokonujemy kolejnego sprawdzenia.

Do tej pory mieliśmy rzadko występującą sytuację w typowych programach: po sprawdzeniu warunku wykonywana była tylko jedna instrukcja. Zwykle tych instrukcji jest więcej. Mały przykład poniżej.

Skoro sprawdziliśmy, że liczba jest dodatnia, możemy wyliczyć pierwiastek:

```
if liczba>0 then
  WriteLn('- pierwiastek ', sqrt(liczba) )
```

A co zrobić jeśli będziemy chcieli znaleźć logarytm? Jeśli chcemy zgrupować ileśtam instrukcji zamykamy je w blok `begin end` – tylko bez kropki na końcu. Tak jak tutaj:

```
if liczba>0 then
begin
  WriteLn( '- pierwiastek ', sqrt(liczba) );
  WriteLn( '- logarytm', ln(liczba) )
end;
```

Koniec wieńczy dzieło ☺. Oto cały program do wypisywania metryczki zadanej liczby:

```
program InfoOLiczbie;
const
  Mala = 10;
  Srednia = 100;
var
  liczba : integer;
BEGIN
{ wczytanie liczby }
  Write('Podaj liczbę: ');
  ReadLn( liczba );
  WriteLn( 'Oto jej własności:');
{ znak liczby }
  if liczba<0 then
    WriteLn( '- ujemna' )
  else if liczba=0 then
    WriteLn( '- zero' )
  else WriteLn( '- dodatnia' );
{ wielkość liczby }
  if (liczba>-Mala) and (liczba<Mala) then
    WriteLn( '- mała' )
  else if (liczba>-Srednia) and (liczba<Srednia) then
```

```
    WriteLn( '- średnia' )
  else Writeln( '- duża ' );
{ parzystość liczby }
  if liczba mod 2 = 0 then
    Writeln( '- parzysta' )
  else Writeln( '- nieparzysta' );
{ dodatek dla liczb dodatnich }
  if liczba>0 then
  begin
    Writeln( '- pierwiastek', sqrt(liczba) );
    Writeln( '- logarytm', ln(liczba) );
  end;
END.
```

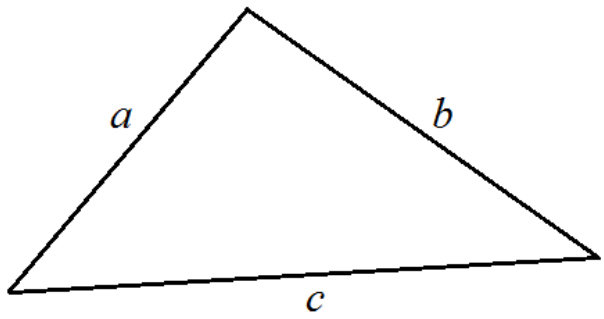
I jego przykładowe wykonanie:

```
Podaj liczbę: 16
Oto jej własności:
- dodatnia
- średnia
- parzysta
- pierwiastek 4.0000000000000000000E+0000
- logarytm 2.772588722239781237 71E+0000
```

1.4 Kilka słów o pisaniu programów

Ten rozdzialik nie będzie pokazywał nowych elementów języka, ale skupi się na kilku przykładach, które być może przydadzą się w praktyce.

Zajmiemy się najpierw problemem, czy trzy podane liczby mogą być długościami boków trójkąta. Zanim zaczniemy pisać kod, należałoby się zastanowić, jakie te liczby muszą być.



Otóż wystarczy, że suma długości krótszych boków będzie większa od najdłuższego boku. Czyli

```
a+b>c
```

tylko nie wiadomo czy użytkownik podając długości, zrobi to w odpowiedniej kolejności. Czyli nie wiadomo, czy c jest rzeczywiście największe. Można oczywiście sprawdzić wszystkie możliwości:

```
(a+b>c) and (b+c>a) and (c+a>b)
```

Chwila skupienia powinna wystarczyć, żeby zdać sobie sprawę, że długości boków muszą spełniać wszystkie trzy warunki jednocześnie – stąd operator and.

Rozwiązanie może się podobać, bo cykliczne zamiany a, b, c miejscami wyglądają OK, ale w praktyce okaże się, że takie rozbudowane warunki logiczne nie są zbyt przejrzyste. Szczególnie, kiedy nazwy zmiennych będą dłuższe niż jednoliterowe – tu pozwoliłem sobie na nazwy a, b, c, bo tak się zwyczajowo określa długości odcinków na lekcjach matematyki. Można jednak postąpić inaczej, w ogóle rezygnując z użycia tego warunku i znajdując najdłuższy bok.

Sposób na znajdowanie największej liczby poznaliśmy w poprzednim odcinku, więc można go zastosować do powyższego problemu:

```
if a>b then
  max := a
else if b>c then
  max := b
else max := c;
```

Teraz można obliczyć sumę krótszych boków:

```
krotsze := a + b + c - max;
```

i mamy warunek zapisany w czytelny sposób:

```
krotsze > max
```

Spostrzegawczy czytelnicy pewnie zauważą, że powyższy warunek jest równoważny takiemu:

```
a+b+c > 2*max
```

W tej wersji nie musimy deklarować zmiennej krotsze. Ale taka „oszczędność” jest dość kontrowersyjna: Według mnie krotsze > max jest o wiele bardziej czytelne. Warto więc chyba utworzyć dodatkową zmienną, by napisany kod łatwo się czytało i rozumiało.

Oto program w całości:

```
program Trojkat;
var
  a, b, c, krotsze, max : real;
BEGIN
```

```
{ Odczytanie danych }
  Write( 'Podaj␣trzy␣długości:␣' );
  ReadLn( a, b, c );
{ Wyliczanie warunku }
  if a>b then
    max := a
  else if b>c then
    max := b
  else max := c;
  krotsze := a + b + c - max;
{ Sprawdzenie }
  if krotsze > max  then
    WriteLn( 'Trójkąt␣możliwy' );
END.
```

Nie jest to jedyne rozwiązanie – można tak pozamieniać wartości trzech liczb, żeby c było największe. Czyli poprawimy nieco podane dane na początku programu, by później uprościć sobie pracę. Sprawdzimy czy b<a i jeśli tak, to zamienimy ich wartości miejscami. Potrzebna będzie pomocnicza zmienna (bufor – ta nazwa dobrze oddaje jej charakter), która przechowa jedną z nich:

```
if b<a then
begin
  bufor := a;
  a :=b;
  b := bufor
end;
```

Teraz wiemy na pewno, że b jest większe (lub równe) od a. Podobny krok dla b i c, zapewni, że największą wartość będzie miało c. Tak to będzie wyglądać w programie:

```
program Trojkat;
var
  a, b, c, bufor : real;
BEGIN
{ Odczytanie danych }
  Write( 'Podaj␣trzy␣długości:␣' );
  ReadLn( a, b, c );
{ Ustawienie c jako najdłuższy bok }
  if b<a then
  begin
    bufor := a;
    a :=b;
    b := bufor
  end;
  if c<b then
  begin
    bufor := b;
    b := c;
```

```
    c := bufor
  end;
{ Sprawdzenie }
  if a+b > c then
    WriteLn( 'Trójkąt możliwy' );
END.
```

Warto tu na chwilę zastanowić się nad „poprawianiem" danych. Istnieje chęć, szczególnie dla początkujących programistów, żeby uczynić swój program „głupoodpornym". To znaczy tak go napisać, żeby użytkownik nie mógł nic „zepsuć". W powyższym przypadku możemy zażyczyć sobie, żeby bok najdłuższy podany był jako ostatni, ale na wszelki wypadek przesortujemy dane, żeby mieć pewność. Tu akurat ma to sens, bo chcemy się nauczyć kliku technik, ale w realnym świecie niekoniecznie będzie to dobry pomysł. Chodzi o to, że skoro dane wejściowe są złe, to nie ma sensu ich przetwarzać. A więc w owym realnym świecie, będziemy raczej sprawdzać czy dane są dobre, a nie je poprawiać.

Uprzedzając pytanie: Techniki weryfikujące poprawność danych są na tyle zaawansowane lub niepasujące do stosowanego przez nas (na razie) trybu tekstowego, iż przyjmijmy założenie: Użytkownik będzie poprawnie wprowadzał dane. Najwyżej od czasu do czasu przeczyta komunikat w rodzaju:

```
Program finished with exit code 106
```

Taki komunikat pojawia się, gdy wpiszemy tekst, który nie może być zinterpretowany jako liczba całkowita, a procedura `ReadLn()` takowej oczekuje.

* * *

W powyższym przykładzie zanim pojawił się kod programu, najpierw opisałem, co ma on robić – dopiero później przedstawiłem gotowy kod, zgodnie z zasadą „najpierw pomyśl, potem zrób". Czasami wystarczy zapisać sobie na kartce ów sposób działania, potrzebne elementy itd. Czasami trzeba będzie dokonać bardziej pogłębionej analizy i dopisać więcej, by program dało się w miarę sprawnie utworzyć. Pisanie programu bez takowej analizy będzie trudniejsze i bardziej narażone na błędy logiczne. Nie opłaca się więc „oszczędzać" na czasie i od razu wklepywać kod, bez wcześniejszego zastanowienia.

Przed przystąpieniem do pisania kodu powinniśmy znać dwie rzeczy:

- zapis algorytmu, czyli ciągu czynności, które prowadzą do celu;
- potrzebne narzędzia programistyczne.

Akurat w przykładzie z trójkątem algorytm był w miarę prosty i wystarczyło, jak się o nim opowiedziało. W bardziej skomplikowanych przypadkach trzeba będzie podejść do niego bardziej formalnie, stosując któreś z zaleceń:

- wypisać słownie w punktach, co po kolei będzie robione;
- wymodelować (pisemnie), jak będą zmieniać się wartości zmiennych;
- narysować schemat blokowy.

* * *

Pora na drugi problem: wyliczanie pierwiastków równania kwadratowego. To szkolne zadanie nie jest jakieś porażające swoim skomplikowaniem, ale dzięki temu lepiej przypatrzymy się, co nam daje analiza problemu.

Starujemy od równania:

$$ax^2 + bx + c = 0$$

Gdybyśmy zajrzeli do książki z matematyki, to okaże się, że rozwiązania są następujące:

$$x_\pm = (-b \pm \sqrt{\Delta})/(2a)$$

gdzie $\Delta = b^2 - 4ac$.

Jeśli pamiętamy zadania typu „jaka jest dziedzina…”, to od razu rzucą nam się w oczy ograniczenia:

$$a \neq 0 \qquad \Delta \geq 0$$

Zauważmy też, że jak a jest równe 0, to nie ma już sensu liczyć delty, bo samo równanie nie jest kwadratowe. Podsumowując te rozważania zaproponuję następujący spis czynności:

- odczytanie a, b, c;
- sprawdzenie czy $a = 0$ – wpisanie komunikatu o błędzie, w przeciwnym razie działamy dalej;
- wyliczenie Δ;
- sprawdzenie czy $\Delta < 0$ – wypisanie komunikatu o błędzie, w przeciwnym razie działamy dalej;
- wyliczenie wartości $x_\pm$;
- wypisanie wartości $x_\pm$.

Teraz trzeba przełożyć to na kod programu.

Widać, że musimy zadeklarować następujące zmienne: `a`, `b`, `c`, `delta`, `pdelta` (pierwiastek z delty), `x1` i `x2`. Proponuję nie bawić się w odpytywanie użytkownika – w przypadku testowania kolejnych przypadków, czynność ta wydaje się dość czasochłonna i nudna. Od razu wpiszmy więc jakieś znane nam dane – te podane niżej powinny dać w wyniku 2 i 3

```
a := 1;
b := -5;
c := 6;
```

W kolejnych próbach testowych będziemy zmieniać te wartości w kodzie programu.

Pomyślmy, jak powinna wyglądać realizacja punktu drugiego? Można tak, jak to zostało rzeczywiście opisane:

```
if a=0 then
  WriteLn( 'Podane␣złe␣dane:␣a␣jest␣równe␣0' )
else
begin
  // tu będziemy działać dalej
  ...
end;
```

A może bardziej przejrzyście będzie tak:

```
if a<>0 then
begin
  // tu będziemy działać dalej
  ...
end
else
  WriteLn( 'Podane␣złe␣dane:␣a␣jest␣równe␣0' );
```

Jak by nie wybierać, zauważ, że lepiej jest jeśli od razu wypiszemy całą instrukcję `if then else`, wraz planowanym zblokowaniem instrukcji poprzez `begin end`. W ten sposób nie zapomnimy, że gdzieś trzeba będzie zamknąć `end`-em jakiś wcześniej rozpoczęty `begin`. A okaże się to ważne, bo przecież po drodze będziemy mieli kolejny `if then else` z kolejnym blokiem. Warto wprowadzić zasadę, że jeśli gdzieś pojawia się `begin`, to od razu wpiszemy odpowiadający mu `end`, kończący blok, a dopiero potem wypisujemy komendy wewnątrz bloku.

Wyliczenie delty nie jest trudne:

```
delta := b*b - 4*a*c;
```

Sprawdzenie znaku delty realizuje się podobnie, jak sprawdzanie zera `a`, więc przejdźmy do wyliczeń. Pierwiastek z delty obliczymy, jeśli pamięta się o funkcji `sqrt()`. Wartości pierwiastków to zastosowanie na wprost wzorów z podręcznika:

```
pdelta := sqrt( delta );
x1 := (-b-pdelta)/(2*a);
x2 := (-b+pdelta)/(2*a);
```

Częsty błąd w podobnych przypadkach polega na opuszczeniu nawiasów:

```
x1 := (-b-pdelta)/2*a;
```

Zapis ten jest równoznaczny z:

```
x1 := ((-b-pdelta)/2)*a;
```

bo mnożenie ma niższy priorytet od dzielenia. Nawiasem mówiąc: jeśli nie wiemy, który operator będzie wykonywał się pierwszy, to nie oszczędzajmy na nawiasach.

Na koniec jedna z możliwych wersji programu do wyliczania pierwiastków równania:

```
program RownanieKwadratowe;
var
  a, b, c, delta, liczba,
  pdelta, x1, x2 : real;

BEGIN
  a := 1;
  b := -5;
  c := 6;
  if a<>0 then
  begin
    delta := b*b - 4*a*c;
    if delta>=0 then
    begin
      pdelta := sqrt(delta);
      x1 := (-b-pdelta)/(2*a);
      x2 := (-b+pdelta)/(2*a);
      WriteLn( 'Pierwiastki równania to:' );
      WriteLn( 'x1: ', x1:5:3 );
      WriteLn( 'x2: ', x2:5:3 );
    end
    else
      Writeln( 'Nie da się wyliczyć: delta ujemna');
  end
  else
    WriteLn( 'Podane złe dane: a jest równe 0');
END.
```

Uprzedzam, że gdy poznamy język pascal lepiej, będziemy w stanie napisać ten program bardziej elegancko.

* * *

I na koniec kilka słów o wcięciach i odstępach. Istnieje zwyczaj, żeby jedną instrukcję umieszczać w jednej linijce kodu. W dodatku jeśli wymaga tego przejrzystość stosuje się tzw. wcięcia, czyli zaczyna się daną linię od jakiejś liczby spacji lub znaków tabulacji. Zauważ, że w dotychczasowych przykładach każdy blok

`begin end` jest wcięty o 2 spacje w stosunku do reszty. Dzięki temu od razu widać, gdzie dany blok zaczyna się i kończy. Szerokość wcięcia może być dobrana inaczej – np. 3, 4 spacje albo jeden tabulator. Sam sobie wybierzesz sposób jaki jest dla ciebie najbardziej przejrzysty.

Możliwość stosowania wcięć wynika z tego, że tzw. białe znaki – spacje, tabulatory i znaki końca linii – są zasadniczo ignorowane przez kompilator. Dlatego też służą przede wszystkim programiście, żeby mu się łatwo czytało tekst programu. Kompilując poniższy kod, uzyskamy program zachowujący się identycznie jak ten kilka akapitów wcześniej.

```
program RownanieKwadratowe;var
a,b,c,delta,liczba,pdelta,x1,x2:real;
BEGIN a:=1;b:=-6;c:=6;if a<>0 then begin
delta:=b*b-4*a*c;if delta>=0 then begin
pdelta:=sqrt(delta);x1:=(-b-pdelta)/(2*a);
x2:=(-b+pdelta)/(2*a);WriteLn('Pierwiastki␣równania␣to:');
WriteLn('x1:␣',x1:5:3);WriteLn('x2:␣',x2:5:3);end
else WriteLn('Nie␣da␣się␣wyliczyć:␣delta␣ujemna');end
else WriteLn('Podane␣złe␣dane:␣a␣jest␣równe␣0');END.
```

Mam nadzieję, że ten (prawie) ekstremalny przykład przekona cię do stosowania wcięć.

Być może zauważyłeś też, że wprowadzam sporo światła – czyli dodatkowe spacje – w różnych instrukcjach:

```
delta := b*b - 4*a*c;
```

zamiast

```
delta:=b*b-4*a*c;
```

Albo

```
WriteLn( 'x1:␣', x1:5:3 );
```

zamiast

```
WriteLn('x1:␣',x1:5:3);
```

Ten rodzaj formatowania doceniłem po pewnym czasie – sam sprawdź, czy ci się spodoba. Mnie taki sposób wydaje się bardziej czytelny, ale to może kwestia indywidualna.

1.5 Wiele razy

Prawdziwa magia programowania zaczyna się, gdy każemy komputerowi robić wiele rzeczy, które nam zajęłyby dużo czasu lub są po prostu nieciekawe. Chyba najlepszym narzędziem, które przydaje się do tego celu, są pętle. Służą one do powtarzania jakiejś czynności.

Pętla `for`

Zacznijmy od przypadku, gdy wiemy ile razy mamy coś wykonać. Na przykład mamy 100 razy napisać „Nie będę przeszkadzał na lekcji”. To nudne zadanie zlecamy komputerowi:

```
program StoRazy;
var i : integer;
BEGIN
  for i:=1 to 100 do
    WriteLn( 'Nie␣będę␣przeszkadzał␣na␣lekcji' );
END.
```

Powyższa konstrukcja jest dość prosta, ale wyjaśnię jej kolejne elementy: Zmienna `i` nazywana jest zmienną sterującą. Odlicza ona kolejne wykonania każdego kroku pętli. Jak widać w pętli `for` najpierw nadaje się wartość zmiennej sterującej (`i:=1`), potem określa się jaka ma być ostatnia wartość `i` (tutaj to liczba 100), a na końcu wykonuje krok pętli (po słowie `do`). Po wykonaniu każdego kroku, wartość zmiennej `i` jest automatycznie zwiększana o 1.

Zwykle krok pętli obejmuje więcej niż jedną instrukcję i wtedy trzeba je zblokować. Jak w tym przykładzie, który wypisuje cyfry i ich kwadraty:

```
program Cyfry;
```

```
var i : integer;
BEGIN
  for i:=0 to 9 do
  begin
    WriteLn( 'Cyfra ', i );
    WriteLn( 'jej kwadrat ', i*i )
  end;
END.
```

Jak widać pascalowe `for` jest dość elastyczne – można dobrać sobie zakres zmiennej sterującej, taki jaki nam odpowiada i najlepiej pasuje do omawianego zagadnienia, choć w praktyce pewnie będzie on najczęściej zaczynał od 1 lub od 0.

```
WriteLn( 'Do naszej szkoły chodzą:' );
for wiek:=6 to 15 do
  WriteLn( '- ', wiek, '-latki' );
```

Nazwy zmiennych sterujących dobiera się zwyczajowo tak, jak nazywane są zmienne całkowite w matematyce: `i`, `j`, `k`... Choć jak widać z powyższego przykładu nie jest to obowiązująca reguła. Pamiętamy, jeśli chodzi o nazewnictwo, to zaleca się stosowanie dłuższych nazw, które odpowiadają roli pełnionej w programie.

Warto też pamiętać, że na czas działania pętli, nie można zarządzać zmienną sterującą. Czyni to ją bardziej bezpieczną, bo w przeciwnym razie łatwo byłoby generować tzw. pętle nieskończone:

```
for i:=1 to 10 do
begin
  i := 1; //BŁĄD!
  WriteLn( i )
end;
```

Taki program się nie uruchomi, bo kompilator wygeneruje błąd:

```
Error: Illegal assignment to for-loop variable "i"
```

Prawdę powiedziawszy, to ograniczenie to wprowadza się raczej ze względów optymalizacji wykonania pętli, a nie z powodów bezpieczeństwa ☺.

Pascal umożliwia też wykonywanie pętli, gdy zmienna sterująca maleje – zwróćmy uwagę na użycie `downto` zamiast `to`:

```
WriteLn( 'Odliczanie:' );
for ile:=10 downto 0 do
  WriteLn( 'Zostało jeszcze: ', ile );
Writeln( 'Nic nie zostało' );
```

Naturalnym wykorzystaniem `for`, jest obsługa tablic – pora więc dowiedzieć się co to takiego.

Tablice

Tablicę służą do przechowywania wielu zmiennych tego samego typu. Deklaruje się je razem ze wszystkimi innymi zmiennymi po słowie `var`, w następujący sposób:

```
nazwa_tablicy : array[min_indeks..max_indeks] of dany_typ;
```

Czyli, żeby zadeklarować tablicę 5 liczb rzeczywistych, potrzeba napisać:

```
liczby : array[1..5] of real;
```

W nawiasach prostokątnych podajemy zakres – podane wartości będą oznaczały pierwszy i ostatni numer elementu tablicy. Wartości te oddzielone są podwójną kropką, charakterystyczną dla pascala. Jak wybrać numerowanie kolejnych elementów tablicy? Jak nam wygodnie, choć pewnie w większości przypadków zaczniemy od 1 a skończymy na wartości oznaczającej rozmiar tablicy. Takie właśnie będą kolejne przykłady.

Jak korzystamy z tablic? Przez tzw. operator indeksowania, czyli nawiasy prostokątne:

```
{ nadanie wartości pierwszemu elementowi tablicy }
liczby[1] := 2.5;
{ dwa przykładowe odczyty drugiego elementu tablicy }
WriteLn( liczby[2] );
zmienna := liczby[2];
```

Liczbę w nawiasie `[]` nazywamy indeksem. Może nią być zmienna typu `integer`:

```
{ wypisanie trzeciego elementu tablicy }
i := 3;
WriteLn( liczby[i] );
```

Podsumujmy te wiadomości w prostym programie odpytującym użytkownika o zawartość tablicy i wypisującym odczytane wartości:

```
program PetleITablice;
var
  i : integer;
  liczby : array[1..5] of real;
BEGIN
  for i:=1 to 5 do
  begin
    Write( 'Podaj ', i, '-tą liczbę: ' );
    Readln( liczby[i] )
  end;
  for i:=1 to 5 do
    WriteLn( liczby[i]:3:2 );
```

```
END.
```

Pascal zezwala na kopiowanie tablic – to dość wyjątkowa cecha wśród języków programowania, czasami znacznie usprawniająca pisanie kodu. Oczywiście obie tablice muszą być tego samego typu:

```
program PetleITablice;
var i : integer;
tab1, tab2 : array[1..5] of real;
BEGIN
{ nadanie wartości do testowania kodu }
  for i:=1 to 5 do
    tab1[i] := i;
{ kopiowanie tablic }
  tab2 := tab1;
{ sprawdzenie, czy tablica się skopiowała }
  for i:=1 to 5 do
    WriteLn( tab2[i]:3:2 );
END.
```

Spójrzmy na powyższy kod i zastanówmy się, ile zmian należałoby w nim wykonać, żeby nasze tablice zawierały nie 5 a 10 liczb. Trzeba zamienić wartości w trzech miejscach. Gdy mamy 10 linijek programu – jak powyżej – nie jest to trudne zadanie, ale gdyby nasz program miał ich tysiąc? Łatwo ominąć jakąś wartość. Dlatego dobrym zwyczajem jest, żeby rozmiary tablic definiować jako stałe.

```
program PetleITablice;
const RozmTab = 5;
var i : integer;
tab1, tab2 : array[1..RozmTab] of real;
BEGIN
  for i:=1 to RozmTab do
    tab1[i] := i;
  tab2 := tab1;
  for i:=1 to RozmTab do
    WriteLn( tab2[i]:3:2 );
END.
```

Zauważmy, że uwaga o stałej może tyczyć się również zakresów pętli `for`, szczególnie jeśli jej wykonanie wiąże się z przetwarzaniem danych z tablic.

Widać już chyba teraz wyraźnie, że rozmiar tablicy musi być znany w momencie kompilacji: Deklaracja tablicy znajduje się przed `BEGIN END.`, więc odpadają przypadki, gdy dopiero w trakcie programu dowiadujemy się ile danych trzeba przechowywać. Na razie musimy się z tym pogodzić i założyć, że będziemy pisać programy w których rozmiar jest jawnie określony.

Warto tu wspomnieć o pewnym, trudnym do wyłapania, błędzie. Chodzi o brak inicjalizacji zmiennych. Dotyczy on wszystkich zmiennych w programie, ale chyba najczęściej daje znać o sobie przy korzystaniu z tablic. Występuje wtedy, gdy

zadeklarujemy zmienną, w programie nie nadamy jej wartości, a chcemy z niej korzystać.

Z punktu widzenia kompilatora program:

```
program Luka;
var
  liczba : integer;
BEGIN
  Writeln( 'Wartość początkowa: ', liczba )
END.
```

jest poprawny, ale wyśle on do nas ostrzeżenie:

```
Warning: Variable "liczba" does not seem to be initialized
```

Ostrzeżenie (warning) to tego typu sytuacja, kiedy kompilator podejrzewa, że coś jest nie tak, ale nie znajduje błędu uniemożliwiającego wykonanie programu. Z zasady powinniśmy eliminować przyczyny ostrzeżeń, a przynajmniej rozumieć jakie jest ich znaczenie w naszym kodzie.

Przyznaję, że błąd tego typu dość trudno popełnić dla pojedynczej zmiennej. Jednak tablice są bardziej narażone na jego wystąpienie, bo pętla inicjalizująca wartości może omyłkowo mieć inne zakresy od indeksów tablicy i coś może zostać pominięte. Zauważmy, że użycie stałej w rodzaju `RozmTab` znacząco ogranicza to niebezpieczeństwo.

Pascal umożliwia tworzenie tablic wielowymiarowych.

```
const DIM =3;
var Macierz3x3 : array[1..DIM, 1..DIM] of real;
```

Wtedy element tablicy ma dwa indeksy, oddzielane przecinkiem:

```
for i:=1 to DIM do
begin
  for j:=1 to DIM do
  begin
    Macierz3x3[i,j] := 0;
  end;
end;
```

Przy okazji widzimy tu również złożenie dwóch pętli `for` – dla lepszej czytelności umieściłem treść kroku każdej pętli w bloku `begin end`, choć każdy zawiera tylko jedną instrukcję. Sam oceń czy taki sposób jest dla ciebie czytelny.

Może się to wydawać niemożliwe, ale w większości zastosowań wystarczy stosować tablice jednowymiarowe. Stanie się to w miarę jasne, gdy poznamy funkcjonalność rekordów (strona 64).

Pętla `repeat until`

Kolejny rodzaj pętli tyczy się sytuacji, kiedy nie wiemy z góry ile razy trzeba będzie wykonać daną czynność, a to czy działać dalej, czy przerwać, będzie wiadomo dopiero po wykonaniu danego kroku pętli. Innymi słowy: jeśli warunek na przerwanie pętli nabiera sensu dopiero w trakcie wykonywania pierwszego (i każdego następnego) kroku pętli, to trzeba stosować właśnie `repeat`.

```
repeat
  //instrukcje
  ...
until warunek_przerwania_petli;
```

Jako przykład napiszemy program na obliczanie odwrotności wielu liczby. Obliczenia zostaną przerwane, kiedy użytkownik poda 0.

```
repeat
  Write( 'Podaj liczbę całkowitą. Podanie zera kończy zabawę. ');
  Readln( liczba );
  if liczba<>0 then
    Writeln( 'odwrotność to: ', 1/liczba );
until liczba=0;
```

Przed wykonaniem pętli, nie wiadomo ile będzie wynosić wartość zmiennej `liczba`. To się dopiero okaże, kiedy użytkownik wpisze jakąś wartość.

Cechą charakterystyczną pętli jest brak grupowania `begin end`, ponieważ samo `repeat until` stanowi sobą oznaczenie grupowania.

Pętla `while do`

Trzecim rodzajem pętli jest `while do`, której używamy, jeśli warunek na wykonywanie pętli jest dobrze określony jeszcze przed jej rozpoczęciem.

```
while warunek_wykonywania_petli do
begin
  //instrukcja
  ...
end;
```

Jak przykład podam program do wypisywania zadanej liczby znaków:

```
Write( 'Ile gwiazdek mam napisać? ');
Readln( liczba );
while liczba>0 do
begin
  Write( '*' );
  liczba := liczba -1
end;
```

Jeśli użytkownik wpisze 0, pętla się nie uruchomi, bo warunek `liczba>0` nie będzie spełniony już na samym początku. Dlatego też czasami w podręcznikach opisuje się różnicę między `repeat` a `while` następująco: „Pętla `repeat` wykona się przynajmniej raz, a `while` może nie wykonać się ani razu".

Trzeba na pewno zwrócić uwagę na asymetrię w warunkach stosowanych przez obie pętle: Dla `repeat until` mamy warunek na przerwanie pętli, a w `while` wręcz przeciwnie – dopóki będzie on spełniony pętla będzie się wykonywać.

W odróżnieniu od `for`, dla obu wariantów zdecydowanie łatwiej uzyskać pętle nieskończone. Mała pomyłka w treści pętli – co się przecież zdarza – może spowodować, że program z niej nie wyjdzie.

```
program PetlaZBledem;
var
  liczba, wynik : integer;
begin
  Write( 'Podaj␣liczbę,␣program␣będzie␣ją␣dzielił␣przez␣2:␣' );
  ReadLn( liczba );
  repeat
    wynik := liczba div 2;
    WriteLn( liczba, '␣div␣2␣=␣', wynik);
    { programista zapomniał na koniec
      kroku pętli zmienić wartość liczba:
    liczba := wynik;}
  until liczba=0;
end.
```

Równie częstym błędem jest przedwczesne kończenie pętli z powodu pomylonego warunku:

```
{ wpisanie liczb od 10 do 1 }
liczba := 10;
repeat
  Writeln( liczba );
  liczba := liczba - 1;
until liczba>0;
{zły warunek - miało być liczba=0}
```

* * *

Kolejny wypisek będzie zawierał kilka zadań sprawdzających znajomość poznanych instrukcji sterujących, a na razie – w ramach podsumowania – zadanie: trzeba policzyć sumę liczb w danej tablicy.

Zgodnie z zasadami opracujemy algorytm, a potem go zrealizujemy. Jak się zabrać do zadania? W zadaniu nie napisali ile będzie elementów – rozwiązanie musi pasować do dowolnego rozmiaru. Czyli trzeba będzie w pętli odwiedzić każdą komórkę tablicy, odczytać jej wartość i dodać do specjalnie przygotowanej zmiennej

oznaczającej sumę. Zanin zaczniemy dodawać kolejne wartości zmienna przechowująca sumę powinna być równa zero (no bo jeszcze nic nie zostało dodane).

- inicjalizacja tablicy;
- wyzerowanie zmiennej suma;
- uruchomienie pętli – w każdym kroku zwiększmy sumę o wartość komórki;
- wypisanie całej sumy.

Tu mała podpowiedź do pierwszego kroku. Uruchamianie komend:

```
Write( 'Podaj ', i, '-tą wartość' );
ReadLn( tablica[i] );
```

przy kolejnym testowaniu programu jest naprawdę nudne, a przecież powinniśmy sprawdzić nasz program dla różnych rozmiarów tablicy i różnych jej wartości. Da się to ominąć poprzez jawną inicjalizację (jeszcze przed blokiem `BEGIN END.`):

```
const Rozm = 5;
var ile : integer;
  tablica: array[1..Rozm] of real = (9, -2.5, 5 , 0, 1.5);
```

Ponieważ w praktycznych zastosowaniach raczej nie będziemy znali wymaganych wartości, potraktujmy to jako wygodną sztuczkę, działającą w tym akurat przypadku, i nie przejmujmy się tym sposobem inicjalizacji tablic.

Ponieważ spis czynności naszego algorytmu jest dość prosty, nie będę już roztrząsał reszty kodu, tylko od razu podam wynik do samodzielnej kontemplacji, jeśli nie masz pomysłu jak go napisać samemu.

```
program SumowanieTablicy;
const Rozm = 5;
var i : integer;
  tablica: array[1..Rozm] of real = (9, -2.5, 5 , 0, 1.5);
  suma : real;
BEGIN
  suma := 0;
  for i:=1 to Rozm do
    suma := suma + tablica[i];
  WriteLn( 'Suma: ', suma:3:2 );
END.
```

Powinno wyjść 13 ☺

* * *

I już na koniec, aspekt wychowawczy: Skoro kilka akapitów wyżej, napisałem (jako przykład) omyłkowo zapętlony program, to teraz poprawię go:

```
program PolowienieCalkowite;
var
  liczba, wynik : integer;
begin
  Write( 'Podaj␣liczbę,␣program␣będzie␣ją␣dzielił␣przez␣2:␣' );
  ReadLn( liczba );
  repeat
    wynik := liczba div 2;
    WriteLn( liczba, '␣div␣2␣=␣', wynik);
    liczba := wynik;
  until liczba=0;
end.
```

żeby zobaczyć jak wyglądają kolejne całkowite dzielenia przez 2:

```
Podaj liczbę, program będzie ją dzielił przez 2: 25
25 div 2 = 12
12 div 2 = 6
6 div 2 = 3
3 div 2 = 1
1 div 2 = 0
```

1.6 Lista zadań do modułu „Przed trivium”

W sześciu poprzednich częściach kursu poznaliśmy kilka podstawowych elementów języka pascal. Może i były one opisywane nieco rozwlekle i niekompletnie, ale to, co zostało umieszczone wcześniej, posłużyć może do samodzielnego napisania kilku programów. Pora zatem na listę zadań.

Zadanie 1 – tabliczka mnożenia

Składając dwie pętle `for` wypisz na ekranie tabliczkę mnożenia. Wewnętrzna pętla będzie służyła do wypisywania pojedynczej linii. Tu mała podpowiedź – możesz użyć formatowania dla liczb, dzięki temu wyniki jedno-, dwu- i trzycyfrowe poukładają się w pionowe kolumny:

```
Write( liczba:5 );
```

Powyższa komenda oznacza, że do wypisania liczby zostanie zarezerwowane miejsce szerokości 5 znaków. Jeśli liczba np. jest dwucyfrowa, zostaną przed nią wstawione 3 spacje. Linię kończymy poleceniem `WriteLn`, wywołanym bez argumentów – kursor przejdzie wtedy do następnej linii.

Jak uda ci się wydrukować wyniki („tabelka” 10 na 10), to spróbuj dołożyć dodatkowe wiersze i kolumny, żeby wiadomo było czego dotyczą. Na przykład tak, jak widać to poniżej:

```
 X |    1    2    3    4    5    6    7    8    9   10
 --+---------------------------------------------------
 1 |    1    2    3    4    5    6    7    8    9   10
 2 |    2    4    6    8   10   12   14   16   18   20
 3 |    3    6    9   12   15   18   21   24   27   30
 4 |    4    8   12   16   20   24   28   32   36   40
 5 |    5   10   15   20   25   30   35   40   45   50
 6 |    6   12   18   24   30   36   42   48   54   60
 7 |    7   14   21   28   35   42   49   56   63   70
 8 |    8   16   24   32   40   48   56   64   72   80
 9 |    9   18   27   36   45   54   63   72   81   90
10 |   10   20   30   40   50   60   70   80   90  100
```

Zadanie 2 – zdolności parapsychologiczne

Tyle się mówi o specjalnych zdolnościach niektórych osób. Jasnowidztwo, bioenergoterapia, telepatia. Zajmijmy się tą ostatnią dziedziną. Pamiętamy pierwsze sceny filmu „Ghostbusters”, gdy Bill Murray kopał prądem studentów za nieodgadnięte rysunki. Wczujmy się w jego rolę i napiszmy program, który wylosuje liczbę z jakiegoś zakresu, a użytkownik będzie ją zgadywał tak długo, aż nie zgadnie. Na koniec program poinformuje użytkownika ile razy próbował.

Do zrobienia tego zadania potrzebna będzie funkcja `random()`. Użycie jej bez argumentów daje liczbę zmiennoprzecinkową z przedziału $(0, 1)$. Użycie jej z argumentem całkowitym:

```
losowanie := random(10) ;
```

wywołuje losowanie liczby całkowitej z zakresu od 0 do 9. Jest to tzw. generator liczb losowych. Zanim zaczniemy wywoływać `random()`, należy zainicjować proces losowania poleceniem `randomize`:

```
randomize;
los := random( 10 );
WriteLn( 'Wylosowałeś:␣', los );
```

żeby każde wywołanie programu losowało inne liczby.

Możliwa modyfikacja: Program informuje użytkownika, że podana liczba jest za duża/za mała. Wtedy odgadnięcie liczby nie powinno przekroczyć $\log_2(\text{zakres})$ zapytań (dlaczego?).

Zadanie 3 – operacje na tablicy

Przypominamy sobie wyliczenie sumy elementów tablicy:

```
suma := 0;
for i:=1 to Rozm do
  suma := suma + tablica[i];
```

Należy rozbudować tamten program o znajdowanie elementu maksymalnego, minimalnego i wyliczenie średniej. W przypadku wyszukiwania wartości minimalnej i maksymalnej, można podglądnąć, jak odnajdowaliśmy największy bok trójkąta – będzie to fragment algorytmu dla całej tablicy.

Zadanie 4 – choinka

Narysować choinkę o podanej przez użytkownika wysokości:

```
    *
   ***
  *****
 *******
*********
```

Tu obrazek dla wysokości równej 5. Najpierw – dla treningu – można próbować utworzyć rysunek prostokąta i trójkąta.

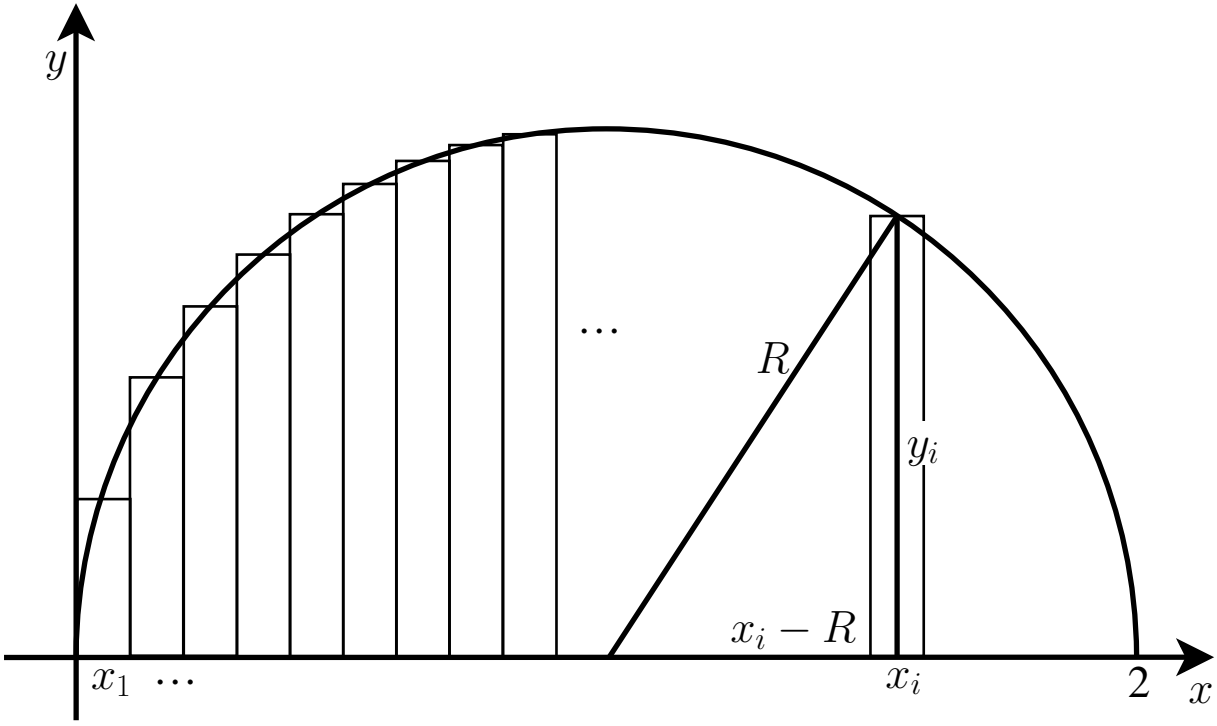
Zadanie 5 – pole koła (lub liczba π)

Jak dokładnie w pascalu reprezentowana jest liczba π, powie nam komenda:

```
WriteLn( pi );
```

Możemy spróbować sami policzyć tę liczbę, oczywiście z mniejszą dokładnością. Na początek nieco teorii.

Wyliczymy pole połówki koła o promieniu 1 i pomnożymy przez 2. W tym celu podzielimy je na kawałki i posumujemy pola prostokątów:



Te prostokąty nie pokrywają się z obszarem koła, co jest dość oczywiste. Mamy jednak nadzieję, że jeśli weźmiemy dostatecznie wąskie paski, to różnica nie będzie znacząca. Tak więc przedział $(0, 2R)$ – u nas $R = 1$ – dzielimy na N odcinków i wtedy szerokość prostokąta będzie równa:

$$\Delta x = 2R/N$$

Z wysokością będzie trochę więcej wyliczeń. Zaczynamy od twierdzenia Pitagorasa:

$$R^2 = (x_i - R)^2 + y_i^2$$

gdzie $x_i = 2R(i - 1/2)/N$. Stała $1/2$ wynika z uwzględnienia szerokości paska – punkt x_i mieści się w jego połowie. Rozwiązując powyższe równanie, dostaniemy:

$$y_i = \sqrt{R^2 - (R - 2R(i - 1/2)/N)^2} = R\sqrt{1 - (1 - 2(i - 1/2)/N)^2}$$

Teraz pozostaje nam już tylko wyliczyć sumę:

$$\sum_{i=1}^{N} \Delta x \cdot y_i = \Delta x \sum_{i=1}^{N} y_i$$

Na lekcjach matematyki czy fizyki praktykuje się wyprowadzenie formuły końcowej (tak jak powyżej zostało to zapisane) i podstawienie danych. Przy pisaniu programu możemy poluźnić tę procedurę i wykonywać obliczenia w kilku krokach. Trudniej się wtedy pomylić, bo formuły są prostsze. W naszym przypadku lepiej będzie więc, jak utworzymy zmienną na przechowywanie x_i:

$$x_i = 2R(i - 1/2)/N$$

następnie znajdziemy y_i:

$$y_i = \sqrt{R^2 - (R - x_i)^2}$$

i z tego wyliczymy sumę

$$\Delta x \sum_i yi$$

Wzory się uproszczą, gdy podstawimy $R = 1$.

Przypominam, że żeby wyliczyć π, powyższą sumę należy pomnożyć przez 2, bo szacowaliśmy pole połowy koła.

Zadanie 6 - funkcja eksponencjalna

Wśród wielu funkcji wykładniczych, ta o podstawie e (liczba Eulera) jest specjalna – zapisuje się ją jako e^x lub $\exp(x)$. W pascalu też przewidziano jej używanie:

```
x := -10;
WriteLn( exp( x ) );
```

Zadanie ma polegać na wyliczeniu tej wartości ze wzoru:

$$e^x = 1 + x + \frac{x}{2} + \frac{x^2}{6} + \cdots + \frac{x^n}{n!} + \cdots = \sum_n \frac{x^n}{n!}$$

Liczenie trzeba przerwać w jakimś miejscu, bo nie wykonamy sumy nieskończonej.

Pierwszy pomysł jaki przychodzi do głowy, to osobno wyliczać każdy wyraz $x^n/n!$ i dodawać go do sumy. Ale nie dodamy w ten sposób zbyt wielu składników, bo $n!$ bardzo szybko rośnie i przekroczy nam zakres każdego typu całkowitego. W takich przypadkach postępuje się inaczej: Zaczyna się od wyraz równego 1, dodaje go do sumy, potem przemnaża przez x i dodaje do sumy, potem przemnaża przez $x/2$ itd. Powinniśmy więc przechowywać zmienną `skladnik` i modyfikować go wraz z każdym krokiem.

Przykład pokazuje pewne aspekty związane z tzw. metodami numerycznymi. Dla dodatnich i niewielkich ujemnych x suma w miarę sprawnie przybliża się do wartości docelowej. Ale dla dużych liczb ujemnych, ze względu na przybliżenia, nigdy nie otrzymamy w ten sposób poprawnej wartości. Poeksperymentuj trochę z typami (`real`, `double`, `extended`) i wartościami x oraz ilością sumowanych wyrazów.

Zadanie 7 – sortowanie tablicy

Sortowanie tablic jest jednym z ważniejszych zagadnień istniejących w programowaniu. Spróbuj napisać program sortujący tablicę przez wstawianie. Algorytm jest następujący: znajdujemy największą wartość w tablicy (to treść zadania nr 3) i zapamiętujemy miejsce wystąpienia tego elementu. Na koniec zamieniamy miejscami element ostatni i największy. W kolejnym kroku powtarzamy te czynności, ale przeszukujemy tylko część tablicy – zakres jest o 1 mniejszy. W każdym następnym kroku będziemy analizować fragment o 1 mniejszy od poprzedniego.

Dla przeprowadzenia procedury sortowania potrzebne są dwie pętle `for`. Malejąca zmienna sterująca pętli zewnętrznej będzie górnym zakresem pętli wewnętrznej.

Zanim napiszemy program, warto rozrysować sobie na kartce, jak będą się zmieniać się zakresy w kolejnych iteracjach pętli zewnętrznej.

Zadanie 8 – trójki pitagorejskie

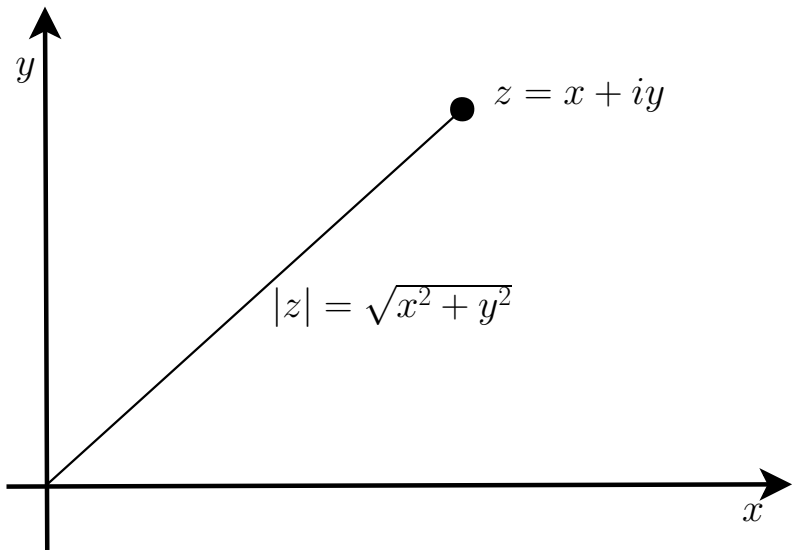
Tytułowe trójki pitagorejskie, to dodatnie liczby całkowite a, b, c, spełniające:

$$a^2 + b^2 = c^2$$

Czyli to takie liczby, które są długościami boków trójkąta prostokątnego. Najpierwsza trójka to 3, 4, 5. Wydawałoby się, że znalezienie innych trójek nie jest zadaniem prostym, ale jak się okaże wcale tak nie jest.

> Wyprowadzenie oparte jest o liczby zespolone i jeśli nie cierpisz matematyki, to od razu przejdź klika akapitów niżej i od razu zapoznaj się w wzorami końcowymi.

Jeśli weźmiemy liczbę zespoloną, której składowe są liczbami całkowitymi dodatnimi, to jej moduł nie musi być całkowity:



ale kwadrat takiej liczby:

$$z^2 = (x + iy)^2 = x^2 - y^2 + 2ixy$$

już spełnia ten warunek:

$$|z^2| = \sqrt{(z^2 - y^2)^2 + 4x^2y^2} = \sqrt{z^4 - 2x^2y^2 + y^4 + 4x^2y^2} =$$

$$= \sqrt{z^4 + 2x^2y^2 + y^4} = \sqrt{(x^2 + y^2)^2} = x^2 + y^2$$

Czyli składowe z^2 i jej moduł są trójką pitagorejską:

$$a = x^2 - y^2$$

$$b = 2xy$$

$$c = x^2 + y^2$$

Żeby nie bawić się problematycznym znakiem a, ustalmy że: $x > y$.

Napisz program, który na podstawie dwóch liczb całkowitych, generuje trójkę pitagorejską.

Narzuca się pytanie, czy w ten sposób wygenerujemy wszystkie trójki pitagorejskie? Odpowiedź brzmi: nie, chyba że nauczymy się skracać – czyli z trójki 30, 40, 50 uzyskamy: 15, 30, 25 oraz 6, 8, 10 oraz 3, 4, 5. Ale to temat na inną opowieść.

Zadanie 9 – pierwiastek

Na koniec zadanie relaksowe: oszacować pierwiastek danej liczby metodą babilońską. W tym celu generujemy ciąg kolejnych przybliżeń:

$$a_{n+1} = \frac{1}{2}\left(a_n + \frac{x}{a_n}\right)$$

Jako a_1 przyjmowało się jakieś szacowanie pierwiastka – np. dla liczby x=426,33 można przyjąć 21, bo 21^2=441 jest dość bliskie szukanej wartości. Wtedy trzeba dokonać mniej iteracji powyższego wzoru. W programie możemy wziąć $a_1 = 1$, komputer dłużej będzie liczył, ale przecież się nie męczy, więc nie musimy się tym przejmować. tym bardziej, że algorytm jest naprawdę bardzo szybko zbieżny.

* * *

Na tym kończymy pierwszą część nauki języka pascal. Jeśli spodobało ci się rozkazywanie komputerowi i wykonywanie przez niego różnych zadań, być może zajrzysz do kolejnych modułów…

1.7 Rozwiązania listy zadań modułu „Przed trivium”

To, że taki a nie inny kod pojawił się tu, jako rozwiązanie zadania, nie oznacza, że jest najlepszy. Tak nie jest – jeśli chodzi o naukę, własny kod jest zawsze lepszy od tego przeczytanego w internecie. Albowiem jeśli sam doszedłeś do wyniku, to czegoś się przy okazji nauczyłeś, a o to przecież chodzi. Potem możesz się zastanowić, czy można w nim jeszcze coś poprawić. Dlatego powinieneś tu zajrzeć dopiero, gdy napiszesz swoją wersję i chcesz porównać rozwiązanie.

Zadanie 1 – tabliczka mnożenia

```
program Tabliczka;
{ x nr wiersza, y nr kolumny }
var x, y :integer;
BEGIN
{ dwie pierwsze linie „opisowe" }
  Write( ' X |' );
  for y:=1 to 10 do Write( y:5 );
  WriteLn;
  Write('---+');
  for y:=1 to 10 do Write( '-----' );
  Writeln;
{ właściwe wyniki }
  for x:=1 to 10 do
  begin
    Write( x:2, ' |');
    for y:=1 to 10 do
      Write( x*y:5 );
    WriteLn
  end;
END.
```

Zadanie 2 – zdolności parapsychologiczne

```
program Telepatia;
const Max = 10;
var zgadula, liczba, licznik : integer;
BEGIN
  randomize;
  liczba := random( Max )+1;
  licznik := 0;
  WriteLn( 'Odgadnij liczbę jaką dla ciebie przygotowałem' );
  WriteLn( 'z przedziału między 1 a ', Max );
  repeat
```

```
    licznik := licznik + 1;
    Write( 'Podaj␣swój␣typ:␣');
    ReadLn( zgadula );
    if zgadula<>liczba then
      Writeln( 'To␣nie␣ta␣liczba,␣próbuj␣dalej.' );
  until zgadula=liczba;
  WriteLn( 'Brawo!');
  WriteLn( 'odgadłeś␣za␣', licznik, '␣razem.')
END.
```

Zadanie 3 – operacje na tablicy

```
program DzialaniaNaTablicy;
const Rozm = 5;
var i : integer;
  tablica: array[1..Rozm] of real = (9, -2.5, 5 , 0, 1.5);
  suma, srednia, max, min : real;
BEGIN
{ suma i średnia }
  suma := 0;
  for i:=1 to Rozm do
    suma := suma + tablica[i];
  WriteLn( 'Suma:␣', suma:3:2 );
  srednia := suma/Rozm;
  WriteLn( 'Średnia:␣', srednia:3:2 );
{ maksymalna i minimalna }
  max := tablica[1];
  min := tablica[1];
  for i:=2 to Rozm do
  begin
    if tablica[i]>max then max := tablica[i];
    if tablica[i]<min then min := tablica[i];
  end;
  WriteLn( 'Największa:␣', max:3:2 );
  WriteLn( 'Najmniejsza:␣', min:3:2 );
END.
```

Zadanie 4 – choinka

```
program Choinka;
var wys, puste,
  nrGwiazdki, nrLinii, nrSpacji : integer;
BEGIN
{ Odczytanie wysokości }
  Write( 'Podaj␣wysokość␣choinki:␣');
  ReadLn( wys );
```

```
{ Rysowanie }
  for nrlinii:=1 to wys do
  begin
    puste := wys-nrlinii;
    for nrSpacji:=1 to puste do
      Write( ' ' );
    for nrGwiazdki:=1 to 2*nrLinii-1 do
      Write( '*' );
    Writeln;
  end;
END.
```

Zadanie 5 – pole koła

```
program LiczbaPI;
{ program nie uwzględnia zmiennej R
  ustawiając, że R=1, co nieco gmatwa wzory}
const N=1000;
      dx = 2/N;
var xi, yi, suma, WylPi : real;
    i : integer;
BEGIN
  suma := 0;
  for i:=1 to N do
  begin
    xi := 2*(i-0.5)/N;
    yi := sqrt( 1 - sqr(1 - xi) );
    suma := suma + yi;
  end;
  WylPi := 2*dx*suma;
  WriteLn( 'Wyliczone pi: ', WylPi );
  WriteLn( 'Wartość z komputera: ', pi)
END.
```

Zadanie 6 – funkcja eksponencjalna

```
program Exponenta;
const IleWyrazow=100; { do zmiany }
var x, skladnik, suma : real;
    n : integer;
BEGIN
  x:=10;                { do zmiany }
  skladnik := 1;
  suma := skladnik;
  for n:=1 to IleWyrazow do
  begin
```

```
    skladnik := x*skladnik/n;
    suma := suma + skladnik;
    Writeln( 'Przyblizenie ', n:4, ' ', suma )
  end;
  WriteLn( 'Funkcja pascalowa ', exp(x) )
END.
```

Zadanie 7 – sortowanie tablicy

```
program SortowanieTablicy;
const Rozm = 5;
var i, nr, pozycja : integer;
  tablica: array[1..Rozm] of real = (9, -2.5, 5 , 0, 1.5);
  max, bufor : real;
BEGIN
{ pętla zewnętrzna wyznaczająca koniec
  działania pętli wewnętrznej - zmienna nr}
  for nr:=Rozm downto 2 do
  begin
    { pętla wewnętrzna }
    { wyszukanie elementu maksymalnego }
    max := tablica[1];
    pozycja := 1;
    for i:=2 to nr do
      if tablica[i]>max then
      begin
        max := tablica[i];
        pozycja := i
      end;
    { zmiana el. ostatniego i maksymalnego }
    bufor := tablica[nr];
    tablica[nr] := max;
    tablica[pozycja] := bufor
  end;
  for i:=1 to Rozm do WriteLn( i, ' ', tablica[i] )
END.
```

Zadanie 8 – trójki pitagorejskie

To trochę co innego niż w zadaniu, ale może będzie ciekawsze niż tylko wypisanie 3 liczb:

```
program TrojkiPitagorejskie;
var max, x, y, a, b, c : integer;
BEGIN
  max := 10; { maksymalna wartość x}
  WriteLn( 'Trójki pitagorejskie c<=',max*max+(max-1)*(max-1) );
```

```
  for x:=2 to max do
    for y:=x-1 downto 1 do
    begin
      a := x*x-y*y;
      b := 2*x*y;
      c := x*x+y*y;
      WriteLn( a:5, b:5, c:5 );
      //WriteLn( 'Sprawdzenie:␣', a*a+b*b, '␣', c*c );
    end;
  WriteLn( 'Gdyby␣jeszcze␣je␣poskracać...' );
END.
```

Zadanie 9 – pierwiastek

```
program Pierwiastkowanie;
const N=10;     { do zmiany }
var x, wyraz : real;
    i : integer;
BEGIN
  x := 48;
  wyraz := 1;
  for i:=1 to N do
  begin
    wyraz := 0.5*( wyraz + x/wyraz );
    WriteLn( 'Przybliżenie␣', i:3, '␣', wyraz )
  end;
  WriteLn( 'Wartość␣z␣kompa␣␣', sqrt(x) );
END.
```

Rozdział 2

Trivium

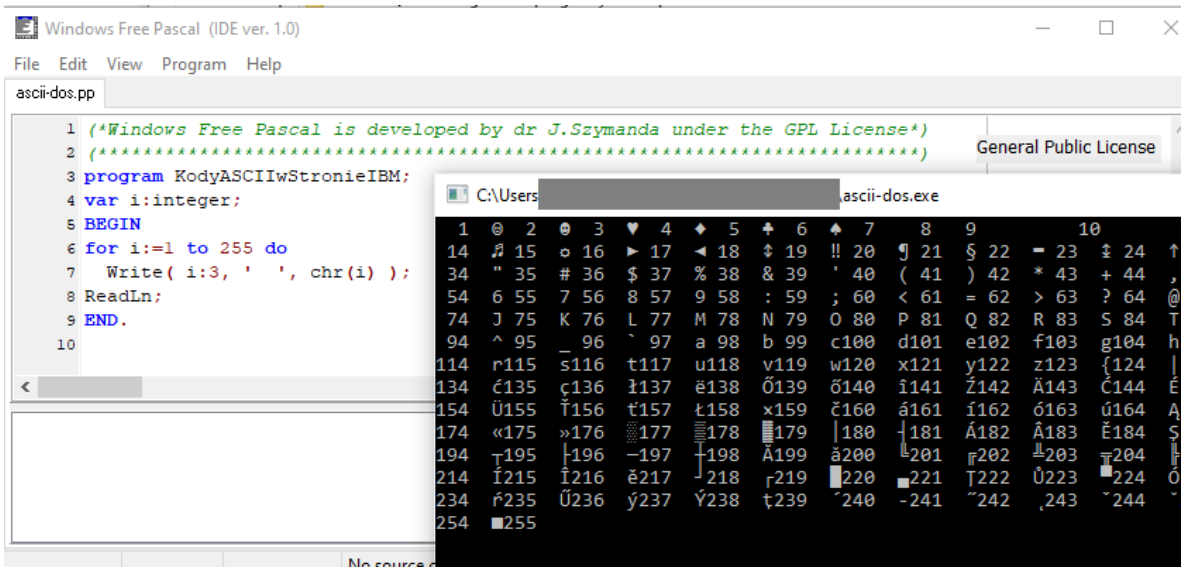
W tym module zapoznasz się z podstawowym i w miarę kompletnym zakresem wiedzy dotyczącym pascala. Zakładam, że skoro poznałeś i przećwiczyłeś dotychczasowy materiał, to nie ma sensu utrzymywania łopatologicznego stylu kursu i można nieco zwiększyć tempo.

2.1 Kompilatory pascala

Nie pisałem do tej pory, gdzie można pisać i kompilować programy w pascalu. Podałem kilka miejsc z edytorami online, ale w końcu chciało by się, mieć u siebie wytwory własnej twórczości. Pora więc na opis kilku środowisk, nie do końca słusznie zwanych kompilatorami.

Windows Free Pascal IDE

Odkryłem ten wynalazek przypadkiem. Niewielki programik, którego nie trzeba instalować, wystarczy rozpakować i… używać: `https://sourceforge.net/projects/windowsfreepascalide/`. Jego autorem jest niejaki Jarosław Szymanda z Wrocławia. Jak sam napisał: *Windows Free Pascal to darmowy edytor kodu źródłowego dla programistów aplikacji konsolowych. Przeznaczony przede wszystkim dla studentów, którzy rozpoczynają naukę programowania. Mogą oni szybko pisać i kompilować oprogramowanie przy użyciu kompilatora Pascala open source opracowanego przez „Free Pascal Team”.*



Jego wielką zaletą, w początkowej fazie nauki (ale i wadą dla zaawansowanych), należy uproszczone do granic możliwości menu. W zasadzie oprócz opcji edycyjnych, mamy do dyspozycji kompilację i uruchamianie programu z pozycji menu **Program**.

> Przypomniały mi się czasy, gdy uzbrojony w jedną dyskietkę ze skopiowanym Turbo Pascalem mogłem siąść przy dowolnym komputerze PC, uruchomić IDE i pisać programy. Wspomniane tu WFP stanowi bardzo podobne rozwiązanie. Jest małe, lekkie, szybko się kopiuje, nie trzeba nic instalować, konfigurować, pobierać i uzupełniać. Po prostu działa.

Free Pascal

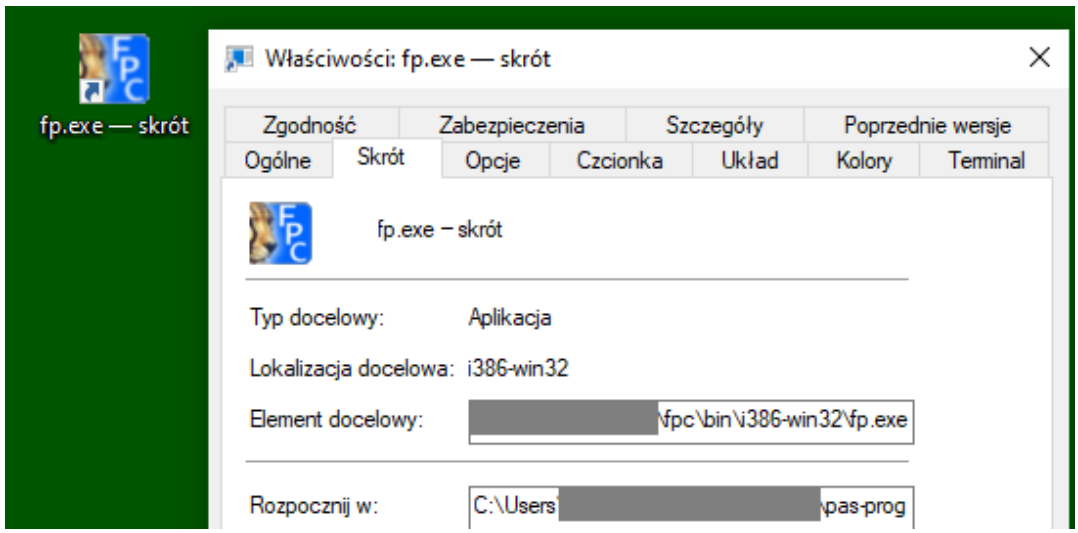
Wydaje się, że najbardziej dostępnym środowiskiem jest IDE Free Pascala. Można je ściągnąć z `https://www.freepascal.org` i zainstalować. Niestety, środowisko to jest wzorowane na starożytnym Turbo Pascalu (jeszcze z czasów DOSa), przez co sposób obsługi jest **inny** niż w większości programów windowsowych! Szczególnie korzystanie ze schowka może się wydawać denerwujące. Większość operacji w edytorze można uzyskać stosując kombinację klawiszy **Ctr+K** i jakiś dodatkowy klawisz:

- **Ctr+K B** – zaznaczenie początku bloku (Begin);
- **Ctr+K K** – zaznaczenie końca bloku (bacK);
- **Ctr+K C** – skopiowanie zaznaczonego fragmentu (Copy);
- **Ctr+K V** – przesunięcie bloku w inne miejsce (moVe);
- **Ctr+K Y** – skasowanie zaznaczonego bloku (nic mi nie przychodzi do głowy dlaczego Y)

- **Ctr+K H** – „odznaczenie” bloku (Hide).

Warto wiedzieć, że menu uaktywnia klawisz **F10**, a kompilację wywołuje wciśnięcie **F9**. Uruchomienie skompilowanego programu następuje po **Ctr+F9**. Do oglądnięcia wyników służy **Alt+F5** – przełącza ono widok edytora na środowisko uruchomieniowe. Ponieważ można mieć pootwieranych kilka tekstowych okienkopodobnych paneli, umożliwiono ich przełączanie za pomocą **F6** i zamykanie aktywnego panelu **Alt+F3**. Edytor i cała reszta zamyka się za pomocą **Alt+X**. W menu można znaleźć więcej możliwych operacji i odpowiadających im skrótów klawiszowych.

Jeszcze jedna uwaga: żeby nie mieć kłopotu z konfiguracją ścieżek, warto utworzyć katalog ze swoimi „dziełami” i wpisać go do skrótu uruchamiającego Free Pascala:



Warto sobie ustawić większą wysokość okna niż zwyczajowe 25 linii. Opcja ta jest dostępna w zakładce skrótu **Układ**.

Podsumowując – każdy, kto kiedyś miał coś wspólnego z niebieskim IDE Turbo Pascala, po latach ociera łezkę widząc jego kopię. Nie wiem jednak, czy niekompatybilność z windowsowymi edytorami, nie stanowi istotnej wady dla współczesnych amatorów programowania. Niemniej, to wartościowy i profesjonalny produkt, a treści z dwóch kolejnych modułów będą uruchamiane i testowane właśnie w tym środowisku.

Lazarus

Na stronie `https://www.lazarus.freepascal.org` można znaleźć wersję podobnego do Delphi środowiska. Co prawda jest ono o uboższe od komercyjnego odpowiednika, ale za to całkowicie darmowe. Po ściągnięciu programu instalacyjnego i zainstalowaniu, mamy możliwość tworzenia programów okienkowych.

Można (i dlatego program pojawił się w tym spisie) tworzyć aplikacje konsolowe. Wybiera się z menu **Plik** → **Nowy…** → **Projekt** → **Prosty program**. Nazwa mówi sama za siebie ☺.

> Tworzenie aplikacji okienkowych opiera się na technikach programowania obiektowego. Mimo to zarówno Delphi jak i Lazarus umożliwiają tworzenie prawdziwych programów, nawet jak umie się tylko postawy pascala. Powstają wtedy użyteczne i działające produkty, choć ich kod jest na bakier z zasadami programowania. Zachęcam więc, do zaznajomienia się z większym zakresem wiedzy, zanim uruchomimy narzędzie RAD. Ale kilka windowsowych programików, choćby po to, by wygenerować swoje pierwsze okienko – **zawsze warto spróbować!**

Delphi

Na stronie `https://www.embarcadero.com/products/delphi/starter` można się zarejestrować i pobrać pakiet Delphi w wersji Community. Jest darmowa, jeśli nie osiągasz przychodów z działalności programistycznej w wysokości 5000$ rocznie. Jeśli jesteś amatorem, to pewnie poziom przychodów jest równy 0$, więc czemu nie skorzystać?

W niniejszym kursie nie odnoszę się jednak do oryginalnego produktu Embarcadero, uznając, że bardziej standardowe są jednak kompilatory spod znaku Free Pascala.

Tryby i kodowania

Na koniec tego krótkiego przeglądu pora na kilka słów wyjaśnień co do kwestii kodowania tekstów. Będzie na ten temat więcej na stronie 90, więc możesz tam zajrzeć, jeśli to co napiszę wyda ci się niejasne.

Edytor internetowy `https://www.onlinegdb.com/online_pascal_compiler` oraz Lazarus używają unikodu UTF8. Tekst napisany w edytorze WFP jest zgodny z kodowaniem CP1250 (tzw. strona kodowa Windows dla Europy Środkowej). Tak samo wyglądający tekst w IDE Free Pascala używa strony IBM 852. Dobrym sprawdzianem co zawierają nasze napisy, jest uruchomienie programu:

```
napis := 'Mały ksiażę';
for i:=1 to length( napis ) do
  WriteLn( i, ' ', napis[i], ' ', ord(napis[i]) );
```

Zaproponowane programy różnią się też domyślnie ustawionymi trybami kompilacji. Przykładowo tylko IDE Free Pascala ma włączone sprawdzanie błędu wyjścia poza zakres. Innymi słowy, gdy chcemy użyć nieistniejącej komórki tablicy,

albo zwiększyć liczbę poza jej maksymalny zakres, program zostanie przerwany. To optymalne ustawienie dla początkującego programisty (str. 70).

2.2 Procedury jako podprogramy

Jedną z technik używanych w programowaniu jest tworzenie tzw. podprogramów. Polega ona na wydzielaniu fragmentów kodu – sekwencji poleceń – w osobne jednostki. Takie podejście nazywa się programowaniem proceduralnym i zdaje się, że swą nazwę wzięło właśnie od języka pascal, w którym podstawowe rodzaje podprogramów to procedury. Swego czasu mówiło się, że pascal jest „definicją" programowania proceduralnego.

> Jeśli zajrzysz do jakiejś publikacji sprzed ćwierć wieku, a takie przecież wciąż istnieją w bibliotekach – okaże się, że mówiono wtedy głównie o programowaniu strukturalnym. Nie oznacza bynajmniej, że oba terminy oznaczają to samo. Podprogramy były jednym ze środków stosowanych w programowaniu strukturalnym. Nie chcę tu jednak wnikać w terminologię – może wrócę do tego tematu.

Na początek bardzo prosty przykład, pokazujący gdzie i jak będziemy zapisywać procedury w kodzie programu. Wyobraźmy sobie, że nasz program potrzebuje potwierdzenia od użytkownika, że może działać dalej – wypisuje napis `Wciśnij ENTER` i czeka na ruch użytkownika. W treści programu potrzeba taka pojawia się kilkakrotnie:

```
program ProstyPrzyklad1;
...
BEGIN
  ...
  Write( 'Wciśnij ENTER' );
  ReadLn;
  ...
  Write( 'Wciśnij ENTER' );
  ReadLn;
  ...
END.
```

Bardzo ogólną zasadą dotyczącą programowania jest niedopuszczanie do kopiowania kodu i tyczy się wielu paradygmatów w programowaniu – nie tylko proceduralnym. Żeby tego uniknąć, w naszym przypadku dwie powtarzające się linie kodu umieścimy w treści procedury `pauza`, a w programie głównym (między `BEGIN END.`) będziemy ją wywoływać:

```
program ProstyPrzyklad2;
```

```
...
procedure pauza;
begin
  Write( 'Wciśnij ENTER' );
  ReadLn;
end;
...
BEGIN
  ...
  pauza;
  ...
  pauza;
  ...
END.
```

Zauważmy trzy rzeczy: Po pierwsze treść procedury umieszczona jest w bloku `begin end` (bez kropki). Po drugie cały kod procedury pisze się przed programem głównym, natomiast jego wywołanie sprowadza się do umieszczenia nazwy procedury (tutaj `pauza`) w programie głównym. Po trzecie po zadeklarowaniu procedury (chodzi o jej pierwszą linijkę `procedure pauza;`) stawiamy średnik.

Grupowanie/wydzielanie powtarzającego się kodu w podprogramach jest najprostszą techniką w programowaniu proceduralnym. Podstawową zaletą dla programisty, jest fakt, że pisze on dane instrukcje tylko raz, mogąc wywołać je wielokrotnie. To nie jedyny powód dla którego powinniśmy stosować podprogramy o czym więcej za chwilę.

Powyższy przykład jest prosty, ale rzadko spotykany. O wiele częściej mamy do czynienia z kodem, który zależy od jakichś zmiennych. Dlatego umieszczając instrukcje w treści procedury musimy umożliwić im korzystanie z owych zmiennych. Z grubsza rzecz biorąc, są trzy sposoby na przekazywanie danych do procedury (na razie skupimy się na dwóch) – służą temu tzw. argumenty, które kiedyś nazywało się (raczej niesłusznie) parametrami. Przeanalizujmy zachowanie programu, który ma dwie procedury – obie pobierają liczbę całkowitą i zwiększają ją o 1.

```
program ArgumentyPodprogramow;

procedure DoOdczytu( x : integer );
begin
  x := x+1;
  WriteLn( 'Liczba wewnątrz procedury: ', x )
end;

procedure DoZapisu( var x : integer );
begin
  x := x+1;
  WriteLn( 'Zmienna wewnątrz procedury: ', x )
end;

var liczba : integer;
```

```
BEGIN
  liczba := 1;
  WriteLn( 'Zmienna liczba to: ', liczba );
  DoOdczytu( liczba );
  WriteLn( 'Zmienna liczba to: ', liczba );
  DoZapisu( liczba );
  WriteLn( 'Zmienna liczba to: ', liczba );
END.
```

Program ten wypisze:

```
Zmienna liczba to: 1
Liczba wewnątrz procedury: 2
Zmienna liczba to: 1
Zmienna wewnątrz procedury: 2
Zmienna liczba to: 2
```

W notacji procedury te różnią się użyciem słowa `var` w deklaracji parametru.

Procedura `DoOdczytu()` pobiera wartość zmiennej `liczba`. Można powiedzieć, że posługuje się kopią tej zmiennej. Natomiast procedura `DoZapisu()` ma dostęp do oryginalnej zmiennej `liczba` i wszelkie działania, jakie wykonuje, odbywają się właśnie na niej. W pewnym sensie można powiedzieć, że pierwsza ma dostęp do odczytu, a druga do zapisu. Sformułowanie to nie jest to do końca ścisłe, bo możliwe jest wywołanie procedury `DoOdczytu()` bez zmiennej:

```
DoOdczytu( 31 );
```

Podobne wywołanie `DoZapisu()` spowoduje błąd, bo potrzebuje ona właśnie zmiennej, a nie samej wartości:

```
DoZapisu( 31 ); //BŁĄD!
```

Niektórzy krytycy języka pascal uważają, że brak informacji o sposobie użycia argumentu przy wywołaniu procedury, może skutkować błędami. Nie wiemy bowiem, czy mamy do czynienia z kopiowaniem czy używaniem zmiennej i w takich sytuacjach musimy zaglądać do kodu procedury, żeby sprawdzić czy użyto `var` przy argumencie czy nie.

Czasami pisze się, że argumenty deklarowane z użyciem `var` są wyjściowe („out”), a te bez `var` określane są jako wejściowe („in”).

* * *

Pozostaje jeszcze trzeci sposób na przekazywanie danych do procedury: argument jest dostępem do stałej. Chodzi o to, że dla dużych zmiennych (np. tabel), których nie chcemy zmieniać, kopiowanie wartości w argumencie zużywa niepotrzebnie pamięć:

```
procedure NieZmieniaZmiennej( zmienna : DuzaZmienna );
...
```

W powyższym przypadku dane są przekazane co prawda „do odczytu”, ale w podprogramie istnieje ich kopia. Żeby tego uniknąć, przewidziano dostęp do zmiennej (tak jak z użyciem `var`) ale z gwarancją, że procedura jej nie zmieni:

```
procedure NieZmieniaZmiennej( const zmienna : DuzaZmienna );
...
```

Teraz procedura operuje na oryginalnej zmiennej, nie ma kopiowania (oszczędzamy pamięć komputera), a kompilator pilnuje, żeby nie można jej było zmienić.

* * *

Uzupełniając kwestie argumentów: procedura ma możliwość używania wielu argumentów wszystkich trzech rodzajów.

```
procedure DuzoArg( var x :real; var a,b :integer; c,d :integer );
```

Argumenty oddziela się średnikami, z tym że dla tego samego typu można je grupować, rozdzielając przecinkami. W powyższym przykładzie `a` i `b` oznaczają dostęp do zmiennych, natomiast przy `c` i `d` następuje kopiowanie wartości. I tak będę to dalej nazywał. W literaturze spotkamy się ze sformułowaniami „parametry przekazywane przez wartość” i „parametry przekazywane przez zmienną”.

Procedury mogą mieć swoje zmienne, a nawet stałe:

```
{ procedura rysująca kwadrat o zadanym boku }
procedure KwadratZGwiazdek( rozm : integer );
const SzerLinii=80;
var i, j : integer;
begin
  if rozm>SzerLinii then
    WriteLn( rozm, ' znaków nie zmieści się na ekranie' )
  else
  begin
    for i:=1 to rozm do
    begin
      for j:=1 to rozm do Write( '*' );
      WriteLn
    end
  end;
end;
```

I tu ważna pojęcie – zmienne zdeklarowane w procedurze nazywa się zmiennymi lokalnymi procedury.

Procedury można nawet zagnieżdżać – procedura może mieć własne (pod)procedury, które można wywoływać tylko w treści procedury nadrzędnej. Przykłady takiego działania być może pojawią się później.

I tu dochodzimy do spostrzeżenia: Postać procedury jest w pewnym sensie kopią struktury programu:

- deklaracja początkowa programu/procedury
- stałe
- procedury
- zmienne
- treść programu/procedury

Niektórzy uważają to podejście za bardzo eleganckie i według nich, również z tego powodu, pascal jest wzorcowym językiem do programowania proceduralnego.

2.3 Wymiana danych między podprogramami. Funkcje

Zanim spróbujemy sił w tworzeniu procedur, należy zastanowić się jak będzie wyglądała wymiana danych między daną procedurą i programem głównym.

> Procedura może być wywołana nie tylko przez program główny, ale również przez inną procedurę, którą nazwiemy wtedy procedurą nadrzędną. W takim przypadku także występuje wymiana danych, ale dla prostoty w dalszej treści kursu, będzie mowa o programie głównym i procedurze. Miejmy jednak w pamięci, że to co tyczy się układu procedura $\leftrightarrow$ program główny, obowiązuje również dla układu procedura $\leftrightarrow$ procedura nadrzędna.

W poprzednim odcinku opisany jest taki sposób: dane wejściowe są przekazywane przez zwykłe argumenty, a dane wyjściowe przez argumenty z użyciem `var` (dostęp do zmiennych nadrzędnych). Nie zawsze taki sposób jest naturalny, więc musimy uzupełnić naszą wiedzę o pojęcie funkcji.

Pomyślmy o podprogramie, który wylicza trzecią potęgę zadanej liczby. Bazując na dotychczasowej wiedzy możemy zaproponować rozwiązanie:

```
procedure Potega3( var wynik : real; x : real );
begin
  wynik := x*x*x;
end;

...
//wywołanie:
```

```
liczba := 1.2;
Potega3( szescian, liczba );
WriteLn( 'Trzecia potęga ', liczba, ' to ', szescian );
```

Można jednak zbudować funkcję, która zwróci nam potrzebą potęgę liczby, na tej samej zasadzie, jak `sqrt()` zwraca pierwiastek:

```
function Potega3( x : real ) : real;
var wynik : real;
begin
  wynik := x*x*x;
  { na końcu funkcji nadajemy jej wartość: }
  Potega3 := wynik;
end;

...
//wywołanie:
liczba := 1.2;
szescian := Potega3( liczba );
WriteLn( 'Trzecia potęga ', liczba, ' to ', szescian );
```

Nadmiarowy kod funkcji `Potega3()` – wszak wystarczyłaby treść: `Potega3 := x*x*x;` – miał nam pokazać, że struktura funkcji jest bardzo podobna do procedury. Bo funkcje też mogą mieć swoje stałe, zmienne i podprogramy zagnieżdżone. Dwie różnice to: nazwa zwracanego typu wpisana w deklaracji po dwukropku:

```
function nazwa_funkcji( argumenty ) : zwracany_typ;
```

i przypisanie zwracanej wartość, zwykle na końcu treści:

```
nazwa_funkcji := zwracana_wartość;
```

Nadawanie wartości funkcji musi wystąpić przynajmniej raz, ale może też mieć miejsce więcej razy. Przykładem niech będzie funkcja licząca silnię:

```
function silnia( n : integer ) : integer;
var  i, iloczyn : integer;
begin
  if n<2 then silnia := 1
  else
  begin
    iloczyn := 1;
    for i:=2 to n do iloczyn := i*iloczyn;
    silnia := iloczyn
  end;
end;
```

Zasadniczo nazwa funkcji zachowuje się jak zwykła zmienna, ale zanim poznamy pojęcie rekurencji, ustalmy, że może pojawiać się w naszych programach

wyłącznie po lewej stronie operatora przypisania `:=`. Innymi słowy możemy jedynie nadawać jej wartość, choć można to robić wiele razy w treści funkcji. Wkrótce okaże się, że istnieją jednak wyjątki od tej zasady.

Wynika z tego, że na pewno stosujemy funkcje w sytuacji, gdy mamy jeden prosty wynik:

(różne dane wejściowe) $\rightarrow$
(pojedyncza liczba, napis, znak czy wartość logiczna)

Dobrym przykładem są tu wszelkie funkcje matematyczne.

* * *

Przetestujmy teraz poznane prawidła dla kilku zagadnień. Co do wprowadzania danych do procedury lub funkcji mamy jasność: będziemy to robić przez kopiowanie zmiennych w argumencie. Pojawia się wątpliwość co do odbierania wyników. Można to robić na dwa sposoby: przez wartość funkcji:

```
function WaznaFunkcja( ... ) : integer;
```

lub przez dostęp do zmiennej w argumencie (`var`):

```
procedure WaznaProcedura( ...; var wynik : integer );
```

Wybór jest często kwestią przyjętej konwencji lub zwyczaju. Czasami nie mamy wyboru, bo dany typ nie może być na przykład zwracany przez funkcję. Zobaczmy, jak to jest dla kilku przykładów.

Przykład: Wypisanie komunikatu w ramce

Chodzi o to, że podprogram powinien dostać informacje o napisie, by wypisać go ramce. Czyli po otrzymaniu napisu `'Ala ma kota'` ma wypisać na ekranie:

```
+----------------+
|  Ala ma kota   |
+----------------+
```

W tym przypadku odpowiedź jest prosta: podprogram nie tworzy żadnych danych, więc na pewno będzie to procedura. Podawany napis nie jest zmieniany, więc w argumencie będziemy mieli kopiowanie wartości. Podsumowując, podprogram będzie miał postać:

```
procedure Komunikat( tresc : string );
...
```

Zadanie. Napisz treść powyższej procedury. Do jej napisania potrzeba jest funkcja zwracająca długość napisu:

```
length( napis )
```

Nie przejmuj się, jeśli okaże się, że ramka źle działa dla napisów zawierających znaki diakrytyczne, np. polskie litery z ogonkami. Problem ten zostanie omówiony na stronie 93.

Przykład: zmiana miejscami

Czasami chcemy zamienić zmienne wartościami. W tym przypadku procedura jest naturalnym rozwiązaniem: Przekazujemy dwie zmienne jako argumenty, a podprogram zamienia ich wartości:

```
procedure Zamien( var x, y : integer, );
var buf : integer;
begin
  buf := x;
  x := y;
  y := buf;
end;
```

Funkcja nie zrealizuje naszej potrzeby, bo może zwrócić tylko jedną wartość, a my w wyniku potrzebujemy otrzymać dwie.

Przykład: liczba zapisana słownie

Jednym z zagadnień, na jakie możemy trafić w praktyce, to zapisanie liczby, podanej jako zmienna typu `integer`, słownie. Zadanie nie jest z tych najprostszych, ale już teraz możemy zaproponować sposób przekazywania danych:

```
function KwotaSlownie( liczba : integer ) : string;
...
```

Funkcja będzie pobierać liczbę, a zwracać napis.

Przykład*: Obrót punktu

Jest to przypadek bardzo podobny do pierwszego – (pod)program nadrzędny przekazuje procedurze dane do zamiany (dwie współrzędne). Czyli deklaracja procedury wyglądałaby następująco:

```
procedure Obrot( var x, y :real; kat : real );
```

Podprogram ma dokonać obrotu punktu – dwóch współrzędnych x i y – o zadany kąt α . Wzory znane z lekcji matematyki wyglądają następująco:

$$x' = x \cdot \cos\alpha - y \cdot \sin\alpha$$

$$y' = x \cdot \sin\alpha + y \cdot \cos\alpha$$

Gdybyśmy te formuły wpisali na wprost do treści procedury, narazimy się na dość ciekawy błąd:

```
procedure Obrot( var x, y :real; kat : real );
{ zmiana jednostek ze stopni na radiany }
const ZmJedn = pi/180;
var alfa : real;
begin
  alfa := ZmJedn*kat;
  { BŁĄD! }
  x = x*cos(alfa) - y*sin(alfa);
  y = x*sin(alfa) + y*cos(alfa)
end;
```

Zauważmy, że we wzorze książkowym x i y są oznaczone primem ($'$). Nie bez powodu: po lewej stronie formuł mamy nowe współrzędne, po prawej stare. A w kodzie do wyliczenia „nowego” y używa się „starego” y i „nowego” x. Czyli wynik będzie nieprawidłowy. Możesz to sprawdzić podstawiając zmienne równe 0 i 1 przy kącie 90. Nowe współrzędne powinny być równe -1 i 0, a powyższy program wyliczy coś innego.

Żeby zlikwidować ten błąd, należy wprowadzić tymczasowe zmienne na współrzędne końcowe, wyliczyć je i przekopiować do zmiennych x i y:

```
procedure Obrot( var x, y :real; kat : real );
{ zmiana jednostek ze stopni na radiany }
const ZmJedn = pi/180;
var alfa, noweX, noweY : real;
begin
  alfa := ZmJedn*kat;
  { użycie zmiennych lokalnych eliminuje błąd }
  noweX = x*cos(alfa) - y*sin(alfa);
  noweY = x*sin(alfa) + y*cos(alfa);
  x := noweX;
  y := noweY
end;
```

Spostrzegawczy zauważą, że potrzebna jest tylko zmienna noweX, ale według mnie oszczędności tego typu spowodują, że kod będzie mniej czytelny.

Ostatni przykład był z gwiazdką, tak jak w zbiorach zadań niektóre zadania też są z gwiazdką. Sugeruje to, że problem jest nieobowiązkowy. Niemniej, według mnie, dobrze zdać sobie sprawę z istoty tego błędu, żeby mieć się na baczności przy używaniu argumentów z dostępem do zmiennej.

2.4 Typy: wyliczeniowy, logiczny, rekordy. Rzutowania

Typ logiczny boolean

Wiemy już, że warunki które wpisujemy do instrukcji sterujących mają wartości logiczne. Możemy posługiwać się też zmiennymi tego typu. I wartościami są stałe `true` i `false`:

```
var warunek : boolean;
...
warunek := true;
...
warunek := liczba>0;
if warunek then ...
```

Do zmiennych tego typu stosujemy operatory `and`, `or` i `not`. Przydaje się, gdy końcowy warunek jest dość skomplikowany do wyliczenia i nie można tego zrobić jednym wyrażeniem:

```
...
warunek := warunek and (liczba>0);
```

Można się domyślać, że `true/false`, to odpowiedniki liczb 1 i 0. I tak jest rzeczywiście, jeśli potrzebne byłyby nam wartości liczbowe, można je dostać za pomocą funkcji `ord()`:

```
WriteLn( 'Wartość warunku: ', ord(warunek) );
```

Bardzo ciekawym zastosowaniem typu logicznego są funkcje o wartościach logicznych. Są one często stosowane i mają dość ciekawą konstrukcję: Mają do wykonania jakieś zadanie, które może się nie udać. Wykonują owe zadanie, zwracając informację czy udało się je wykonać (tak/nie), natomiast właściwe wyniki działania są przekazywane przez zmienną w argumencie.

Tak właśnie działa poniższa funkcja do wyliczania pola trójkąta ze wzoru Herona. Mając trzy liczby `a`, `b` i `c` wyliczymy pole pod warunkiem, że mogą one być bokami trójkąta. Jeśli dobierzemy złe liczby (np. 3, 1, 1), nie uda się uzyskać wyniku. Zobaczmy, że wynik (pole) dostępny jest przez zmienną `pole`:

```
function CzyPoleTr( a, b, c: real; var pole : real ) : boolean;
var p, kwPola : real;

begin
{ Wzór Herona }
  p := 0.5*(a+b+c);
  kwPola := p*(p-a)*(p-b)*(p-c);
```

```
  if kwPola<0 then CzyPoleTr := false
  else
  begin
    pole := sqrt( kwPola );
    CzyPoleTr := true;
  end;
end;
```

Typowe użycie tego typu funkcji, to wykorzystanie jej w konstrukcji `if then else`:

```
if CzyPoleTr( 1, 1.5, 2, pole ) then
begin
  WriteLn( 'Pole wynosi ', pole:3:2 );
  //tu kolejne działania z użyciem pola
end
else WriteLn( 'To nie trójkąt' );
```

Typ wyliczeniowy

Przykład z trójkątem zakładał dwie możliwe opcje: uda się albo nie. A co robić, gdy opcji jest więcej? Pomyślmy sobie, że piszemy podprogram do rozwiązywania równania kwadratowego. Mamy wtedy cztery (choć będę się upierał, że matematycznie trzy) możliwości: a jest równe zero, delta jest ujemna, istnieje jeden pierwiastek, istnieją dwa pierwiastki. Można wtedy sobie zdefiniować własny typ zawierający wszystkie opcje.

Własne typy definiuje się po użyciu słowa `type`. To już trzecie oznaczenie dzielące nasze program na sekcje: mieliśmy już `const` i `var`, oprócz tego tworzyliśmy też podprogramy:

```
{ struktura pliku programu }
program JakisProgram;
{ stałe }
const
 ...
{ własne typy }
type
 ...
{ podprogramy: funkcje i procedury }
 ...
{ zmienne }
var
 ...
{ program główny }
BEGIN
  ...
END.
```

Taka kolejność sekcji w programie nie jest obowiązkowa, co więcej, każda z nich może wystąpić więcej niż raz. Mimo to, dobrze byłoby gdybyśmy jednak zachowali taką kolejność. Przynajmniej na początku nauki.

Wracając do przykładu z równaniem kwadratowym definicja typu zawierającego opcje jest następująca:

```
type
  OpcjeRownKw = ( AZero, DeltaUj, JedenPierw, DwaPierw );
```

Od tej pory możemy tworzyć zmienne typu `OpcjeRownKw` i używać ich w programie:

```
var opcja : OpcjeRownKw;
...
if opcja=DwaPierw then ...
```

Jak widać definiowanie takiego typu jest proste – wystarczy wypisać po kolei kolejne opcje w nawiasie. Takie typy nazywa się wyliczeniowymi (można skojarzyć z wyliczanką).

* * *

Dobrym przykładem na zastosowanie typu wyliczeniowego jest obsługa menu w programie. Dajemy użytkownikowi wybór, co ma być zrobione, i na podstawie wyboru uruchamiamy konkretną akcję. Zwykle pierwsze podejście polega na użyciu liczby całkowitej. Tak jak w „niby menu" jakiegoś programu:

```
Wybierz co mam robić (podaj liczbę):
1 otwórz plik z danymi
2 dopisz rekord
3 zapisz plik z danymi
4 wygeneruj raport
0 zakończ program
```

Takie intuicyjne (naiwne?) podejście jest zgodne z tym co oferuje typ wyliczeniowy, albowiem wypisane stałe:

```
OpcjeProgramu = ( zakoncz, otworz, dopisz, zapisz, raport );
```

oznaczają ciąg liczb 0, 1, 2, 3 i 4. Tyle, że nie mamy do nich jawnego dostępu, zamiast liczb używamy nazw. Taki sposób jest lepszy dla programisty, bo nie musi on pamiętać, że użycie liczby 3 ma uruchamiać zapisywanie pliku z danymi – stała `zapisz` jest bardziej czytelna. Niemniej można odczytać jaka wartość liczbowa „ukryta" jest pod daną stałą:

```
opcja := otworz;
WriteLn( ord(opcja) ); // wypisze 1
```

Działa to też w drugą stronę, mając liczbą można przekonwertować ją typu wyliczeniowego:

```
opcja := OpcjeProgramu( 3 ); // opcja := zapisz
```

ale trudno mi podać jakiś użyteczny przykład takiego postępowania. W przypadkach gdy nadajemy wartość zmiennej z użyciem nazwy typu, mówimy o inicjalizacji za pomocą konstruktora.

To jeszcze kawałek hipotetycznego kodu, który mógłby odpowiadać opisanej powyżej sytuacji:

```
{ procedura z menu odpytująca użytkownika }
menu( opcja );
{ wybór działania na podstawie opcji }
if opcja = otworz then OdczytZPliku( dane, sciezka )
else if opcja = dopisz then DopiszRekord( dane )
...
```

* * *

Zwróćmy uwagę na wybór nazw. Przy niewielkich programach nie ma to znaczenia, ale może się okazać, że przy utworzeniu wielu różnych typach wyliczeniowych, warto wprowadzić konwencję, by stałe tych typów zawierały w sobie element związany z nazwą typu. Na przykład dla równania kwadratowego:

```
OpcjeRownKw = ( RKwAZero, RKwDeltaUj, RKwJedenPierw, RKwDwaPierw );
```

* * *

Warto również tak dobierać stałe typu wyliczeniowego, żeby pierwsza wartość odpowiadała sytuacji nieprawidłowej czy błędowi. Oczywiście jeśli w ogóle tak się da – nasz przypadek wcale nie musi odpowiadać sytuacji: jedna opcja zła, wszystkie inne dobre. Ale jeśli tak właśnie jest, to można sprytnie użyć naszego typu w instrukcjach sterujących. Zobaczmy – dla typu:

```
Opcje = ( blad, ok, super );
```

prawidłowe będą instrukcje:

```
opcja := super;
WriteLn( opcja, ' to inaczej liczba ', ord(super) );
warunek := boolean(opcja);
WriteLn( 'a odpowiada mu logiczne ', warunek );
```

Powyższy kod wygeneruje:

```
super to inaczej liczba 2
a odpowiada mu logiczne TRUE
```

Opcja o wartości `blad` wygeneruje stałą `false`. To jeszcze wypiszmy jak wygląda użycie typu wyliczeniowego do sprawdzania warunku logicznego:

```
if boolean( opcja ) then Writeln( 'Można działać' );
```

Rekordy

Rekordy służą do grupowania danych różnych typów tyczących się jednego obiektu. Typowym przykładem jest zestaw danych określających osobę. W skład tych danych wejdzie imię, nazwisko, stanowisko, numer osobowy itd. Definicja takiego typu wygląda jak spis zmiennych, zamknięty w bloku `record end`:

```
type
  Osoba = record
    imie : string[25];
    nazw : string[50];
    nr   : integer;
  end;
```

Każdy element rekordu nazywamy polem. Innymi słowy, pojedynczy rekord typu `Osoba` zawiera trzy pola: dwa napisy i liczbę całkowitą. Możemy więc traktować rekord, jako grupę trzech zmiennych.

Dostęp do pól odbywa się poprzez operator oznaczony kropką. Dla zmiennej

```
var os : Osoba;
```

nadanie wartości poszczególnym polom, wygląda następująco:

```
os.nazw := 'Kowalski';
os.imie := 'Adam';
os.nr := 23;
```

Odczyt wartości pola dokonuje się podobnie:

```
WriteLn( os.nazw );
```

Warto jednak zwrócić uwagę, że istnieje sposób na skrócenie zapisu i ominięcie operatora pola, przez wykorzystanie instrukcji `with` do:

```
with os do
begin
  WriteLn( nr, '␣', imie, '␣', nazw );
  // inne komendy dotyczące rekordu os
end;
```

* * *

Zauważmy, że wybór terminu „rekord" nie jest przypadkowy i sugeruje związek z tabelą bazodanową. I tak rzeczywiście może być. Przypomnijmy sobie jak wygląda tabelka w Excelu – jest podzielona na linie i kolumny, możemy ją odnieść do tablicy rekordów w pascalu: każda linia jest wtedy odpowiednikiem rekordu, a nawa kolumny odpowiada nazwie pola.

Okazuje się, że tablica (albo ogólniej jakiś zbiornik) z rekordami jest na tyle elastycznym rozwiązaniem, że obsługuje wiele zastosowań praktycznych służących do przechowywania danych w programach. Przypominam o uwadze ze strony 31, że stosunkowo rzadko wymagane są tablice inne niż jednowymiarowe:

```
const N = 100;
var
  zaloga : array[1..N] of Osoba;
...
```

* * *

Rekordy mają jeszcze jedną właściwość: można tworzyć różne ich warianty – nazywa się to rekordem zmiennym. Wariantowość wprowadza się przez wprowadzenie tzw. części zmiennej. Opiera się ona o konstrukcję `case of`, której dokładnie przyjrzymy się na stronie 89. Przeanalizujmy poniższy przypadek:

```
type
  Rodzaj = ( pracownik, kandydat );
  Osoba = record
    imie : string[25];
    nazw : string[50];
    { część zmienna }
    case rodz : Rodzaj of
      pracownik : ( pesel : string[11] );
      kandydat  : ( adres : string[50] )
  end;
```

Zapis taki oznacza, że rekord `Osoba` ma pole `rodz` typu wyliczeniowego `Rodzaj`. Ma ono służyć do sprawdzenia z jakim wariantem rekordu mamy do czynienia. Jeśli wartość pola jest równa `pracownik`, rekord będzie miał jeszcze jedno pole pesel (napis o długości 11 znaków), w przypadku, gdy pole rodz to `kandydat`, będziemy wpisywać adres. Pola zmienne umieszczone są w nawiasach, dla danego wariantu może być ich więcej niż jedno, może też nie być żadnego.

> Przykład proszę potraktować z przymrużeniem oka, bo zakres danych jakie się przetwarza w przypadku kandydatów do pracy czy pracowników jest o wiele szerszy.

Zobaczmy jak działa pole wyboru:

```
{ nadawanie wartości }
os.imie := 'Jan';
os.nazw := 'Kowalski';
os.rodz := kandydat;
{ skoro kandydat, to wpisujemy adres }
os.adres := 'ul. Długa 21 Warszawa';
```

Odczyt danych będzie wyglądać następująco:

```
WriteLn( os.imie, ' ', os.nazw );
if os.rodz = pracownik then
  WriteLn( os.pesel )
else WriteLn( os.adres );
```

Fenomen polegający na tym, że jeden typ może mieć wiele wariantów, jest prosty do wyjaśnienia. Każdy rekord ma część stałą i zmienną. Rozmiar części zmiennej jest równy rozmiarowi wariantu maksymalnego – w naszym przypadku, będzie to 50 bajtów. W tym obszarze zapisywane będą różne dane o różnych typach. Możemy się o tym przekonać, kiedy „pomylimy się" w rodzaju pola zmiennego:

```
os.nazw := 'Kowalski';
...
os.rodz := kandydat;
os.adres := 'ul. Długa 21 Warszawa';
...
with os do
begin
  WriteLn( imie, ' ', nazw );
  WriteLn( pesel );
end;
```

Powyższy kod nie wygeneruje błędu:

```
Adam Kowalski
ul. Długa 21 Warszawa
```

* * *

Zadanie. Zadeklaruj rekord Punkt zawierający dwie liczby rzeczywiste – współrzędne x i y. Napisz procedurę przesuwającą współrzędne o Δx i Δy:

```
Przesun( A, dx, dy );
```

oraz inne procedury które uważasz za przydatne do posługiwania się punktem.

Kiedy doda się pole kierunek, można symulować coś w rodzaju języka LOGO:

```
Punkt A;
A.x := 0;
A.y := 0;
{ kierunek zgodnie z kierunkiem osi X }
A.kierunek := 0;
{ przesuń punkt o 1.5 w prawo }
Przeun( A, 1.5 );
{ obróć kierunek o 30 stopni lewoskrętnie }
Obroc( A );
{ przesuń punkt o 1 w zadanym kierunku }
Przesun( A, 1 );
{ wypisz współrzędne po zmianach }
Wypisz( A );
```

Co prawda Free Pascal zezwala na zwracanie rekordu przez funkcję, więc zamiast procedury z argumentem z dostępem do zmiennej, można napisać funkcję:

```
A := Przesun( A, dx, dy );
```

Wydaje mi się jednak, że ten akurat przypadek lepiej pasuje do procedury.

Rzutowania między typami prostymi

Ponieważ pojawiły się w tym odcinku rzutowania pomiędzy typami `integer`, `boolean` i wyliczeniowym, to dobre miejsce, by przedstawić jak wygląda problem konwersji między innymi typami dostępnymi w pascalu.

Przede wszystkim trzeba powiedzieć, że pascal jest językiem o silnej typizacji, co oznacza, że jak mamy dwie zmienne różnych typów, to zwykle nie da się ich porównać, czy nadawać wartości jednej używając drugiej:

```
ZmiennaA := ZmiennaB; // zwykle błąd kompilatora
```

„Zwykle” nie znaczy „zawsze”. Czasami automatyczne konwersje są dopuszczalne, szczególnie wtedy, gdy nie budzą wątpliwości.

W poniższych przykładach zakładam, że utworzono zmienne następujących typów:

```
var
  ulamek : real;
  liczba : integer;
  duza : longint;
  slowo : word;
  bajt : byte;
  znak : char;
```

Zacznijmy od konwersji między liczbami całkowitymi i rzeczywistymi. Operacja:

```
liczba := 3;
ulamek := liczba;
```

wykona się, bo nie ma wątpliwości co do wartości zmiennej `ulamek`. W drugą stronę już się nie uda:

```
liczba := ulamek; // BŁĄD!
```

bo nie wiadomo co robić z częścią ułamkową. Dlatego mamy dwie możliwości działania, za pomocą funkcji `round()` i `trunc()`:

```
ulamek := 5.999;
{ zaokrąglenie }
liczba := round( ulamek ); // 6
{ obcięcie części ułamkowej }
liczba := trunc( ulamek ); // 5
```

Pewnym zaskoczeniem mogą być automatyczne konwersje między różnymi typami całkowitymi. Kompilator nie zgłasza tu problemów, ale przy przekroczeniu zakresu pojawia się błąd wykonania. Kod:

```
duza := 33000;
liczba := duza;  { maksymalna wartość to 32767 }
```

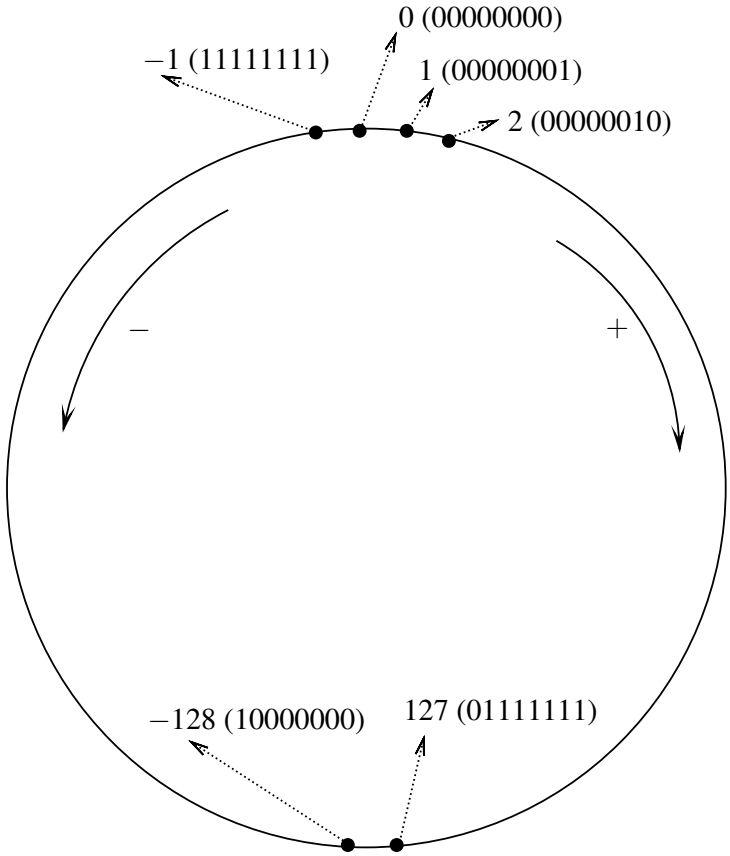
wygeneruje komunikat `Runtime error 201` (z podanymi adresami wystąpienia błędu). Ciekawym jest to, że przy rzutowaniu przez konstruktor:

```
duza := 33000;
liczba := integer(duza);
WriteLn( duza, ' obcięte do 2 bajtów to ', liczba );
```

błąd wykonania się nie pojawi, a uzyskana wartość może na pierwszy rzut oka wydawać się zaskakująca:

```
33000 obcięte do 2 bajtów to -32536
```

By wyjaśnić to zachowanie, należy zdać sobie sprawę, jak realizowane jest przechowywanie liczb w postaci binarnej. Wiemy, że kolejne liczby całkowite są zapisywane binarnie jako ciągi zer i jedynek. I tak $0_{10} = \ldots 00_2$, $1_{10} = \ldots 001_2$, $2_{10} = \ldots 010_2$ itd. Rzecz jasna nie mamy nieskończonej ilości miejsca i każda zmienna ma ograniczony rozmiar. Poniżej rozpatrzono jednobajtowy (ośmiobitowy) `shortint`. Największą, jeśli chodzi postać binarną, zmiennej tego typu jest wartość: 11111111_2. A gdzie wartości ujemne? Otóż dzielimy zakres na pół i liczba 01111111_2 jest największą liczbą dodatnią (127), a o jeden większa, czyli 10000000_2 jest najmniejszą liczbą ujemną (-128). W miarę obrazowo widać to na poniższym rysunku:



Dla typów o większej liczbie bajtów jest podobnie, tylko granice zakresów są większe. W przypadku typów bez znaku (jednobajtowy `byte` i dwubajtowy `word`) możemy narysować podobne koło, tyle, że (dla typu `byte`) najmniejszą wartością będzie $0_{10} = 00000000_2$, a największą $255_{10} = 11111111_2$.

Można się zastanowić, czy przechodzenie poza zakres:

```
liczba := 32767;
liczba := liczba+1;
```

będzie adekwatne ze zwiększeniem wartości binarnej o 1 i czy liczba zmieni są wartość na: -32768? Otóż przy domyślnych ustawieniach kompilatora działanie

powyższe wygeneruje wspomniany wcześniej błąd nr 201. Podobnie będzie dla typów bez znaku. Zauważmy, że rzutowanie przez konstruktor omija ten problem. Wynika to z założenia, że skoro programista użył jawnie rzutowania, to wie co robi. To znaczy, że z rozmysłem zaproponował rozwiązanie, zachowujące wartość binarną zmiennej.

Błąd wykonania nr 201 powstaje, gdy mamy włączoną kontrolę przekroczenia zakresu. W moim kompilatorze (czyli tym, który ściągnąłem ze strony Free Pascala) jest ona włączona domyślnie. Oznacza to, że kompilacja odbywa się z opcją `{$R+}`. Mogę zażyczyć sobie, żeby kontrola nie była włączona – wystarczy dopisać w kodzie programu opcję kompilacji przeciwną do `{$R+}` czyli `{$R-}`:

```
program SprawdzanieKonwersji;
{$R-}
...
```

i wtedy kod

```
liczba := 32767;
liczba := liczba + 1;
Writeln( 'Zwiększenie liczby o 1: ', liczba );
```

wypisze (bez błędu):

```
Zwiększenie liczby o 1: -32768
```

Wyłączania kontroli powinni używać jedynie doświadczeni programiści, którzy wiedzą co robią.

Lista błędów wykonania dostępna jest na stronie: `https://www.freepascal.org/docs-html/user/userap4.html`

* * *

Ponieważ znaki (typ `char`) zajmują właśnie jeden bajt, a dla liczby jednobajtowej mamy typ `byte`, powstaje pytanie czy można używać obu typów zamiennie. Otóż zgodnie z charakterem pascala nie można bezpośrednio:

```
znak := 'A';
// bajt := znak; BŁĄD!
```

Można jednak znajdować nr kodu ASCII znaku:

```
bajt := ord(znak);
```

lub

```
bajt := byte( znak );
```

Działa to również w drugą stronę:

```
bajt := 65;
znak := char( bajt ); // 'A'
```

lub korzystając ze specjalizowanej funkcji:

```
znak := chr( bajt );
```

Oba sposoby działają również dla innych typów całkowitych, przy czym `chr()` potrzebuje liczby z zakresu 0-255.

2.5 Więcej o tablicach

Pora na poszerzenie naszej wiedzy na temat tablic. Przede wszystkim warto wspomnieć, o możliwości zdefiniowania sobie odpowiedniego typu tablicowego w przypadkach, gdy rozmiar tablicy jest wiadomy w momencie kompilacji:

```
const N=10;
type Tablica = array[1..N] of real;
```

Można wtedy deklarować wiele tablic tego typu i każda z nich będzie miała ten sam zakres indeksowania (tutaj od 1 do 10). Taki typ nadaje się do przekazywania jako argument funkcji czy procedury:

```
procedure Zeruj( var tab : Tablica );
var i : integer;
begin
  for i:=1 to N do tab[i] := 0;
end;
```

Tablica może być nawet wartością zwracaną przez funkcję. Tak jak to jest w poniższym przykładzie, gdy tworzymy tablicę zapełnioną wartościami losowymi (też od 0 do 1):

```
function GenTab : Tablica;
var i : integer;
begin
  for i:=1 to N do GenTab[i] := random;
end;
```

Wtedy można tej funkcji używać w następujący sposób:

```
var tab : Tablica;
BEGIN
  randomize;
```

```
  tab := GenTab;  {kopiowanie tablic}
  for i:=1 to N do Write( tab[i]:3:2, ' ' );
  ...
END.
```

Oba sposoby są dopuszczone przez Free Pascala, ale ten pierwszy (przekazywanie tablicy przez argument z dostępem do zmiennej) wygląda dla mnie bardziej naturalnie. Zresztą dla dużych tablic kopiowanie, jakie ma miejsce na końcu działania funkcji `GenTAb`, jest nieuzasadnione, jako zasobo-chłonne. Zauważmy, że w linii oznaczonej komentarzem `{kopiowanie tablic}` – w pamięci mamy dwie tablice z tymi samymi danymi.

Skoro o kopiowaniu tablic mowa: Przekazywanie tablicy do podprogramu bez konieczności dokonywania na niej zmian, jest typowym przykładem zastosowania argumentu `const`:

```
procedure Wypisz( const tab : Tablica );
var i : integer;
begin
  for i:=1 to N do WriteLn( i, ' ', tab[i] );
end;
```

Zadanie. Napisz funkcje wyliczające sumę liczb tablicy oraz średnią wartości przechowywanych w tablicy. Zdefiniuj typ tablicy w sekcji `type` (tak jak w przykładach powyżej).

* * *

Pozostawiona w poprzednim module uwaga, „że będziemy pisać programy, w których rozmiar jest jawnie określony” wskazywała na dość dużą lukę w naszej wiedzy, którą postaram się teraz uzupełnić. Do tej pory posługiwaliśmy się tablicami o stałym rozmiarze (znanym w momencie kompilacji), a jedynym narzędziem usprawniającym było określenie tego rozmiaru jako stałej. Trzeba jednak przyznać, że w zastosowaniach praktycznych takie przypadki są bardzo nieliczne. Zdecydowanie częściej o rozmiarze tablicy dowiadujemy się w trakcie działania programu. Rozwiązaniem stosowanym w takich sytuacjach są tzw. tablice dynamiczne.

Zacznijmy od prostszego przypadku, gdy rozmiar tablicy nie zmienia się w czasie życia programu. Załóżmy, że to użytkownik podaje jego wielkość. Można wtedy utworzyć tablicę w trakcie działania programu, pod warunkiem, że zadeklarujemy ją jako zmienną typu `array of nazwa_typu`. Tak właśnie działa poniższy kod:

```
var tab : array of real;
...
BEGIN
{ na razie tab jest pusta }
  Write( 'Podaj liczbę elementów ');
```

```
  ReadLn( rozmiar );
  SetLength( tab, rozmiar );
{ teraz tablica ma rozmiar elementów }
```

Trzeba jednak pamiętać, że tego typu tablica różni się od przypadku tablic ze sztywno określonym rozmiarem:

- Zakres indeksowania zmienia się od 0 do `rozmiar-1`. Dla początkujących to zwykle najtrudniejsza do pogodzenia się, cecha tablic dynamicznych. Ta konwencja indeksowania od 0, jest zgodna ze zwyczajem panującym w językach z rodziny C. Powtórzmy więc: pierwszy element tablicy to `tab[0]`, drugi to `tab[1]` a ostatni to `tab[rozmiar-1]`.
- Nie można kopiować tablic dynamicznych do tablic statycznych i odwrotnie. Podobnie napisanie wspólnego kodu dla tych obu rodzajów tablic nie uda się, ze względu na niezgodność typów.
- Coś, co wygląda jak kopiowanie tablic dynamicznych, oznacza powiązanie drugiej zmiennej, z tablicą zarządzaną przez pierwszą zmienną.

Ta ostatnia właściwość ma dużo wspólnego z tzw. wskaźnikami, co oznacza, że tablica dynamiczna jest ukrytą formą zmiennej wskaźnikowej – o zmiennych tego typu napiszę więcej w następnym module.

```
var T1, T2 : array of real;
...
BEGIN
  ...
  T2 := T1;
```

Zmienne `T1` i `T2` zarządzają tą samą tablicą.

Funkcja `SetLength()` nie tylko tworzy tablicę, ale może zmienić jej rozmiar. W przypadku, gdy istnieje potrzeba zwiększenia jej rozmiaru np. dwukrotnie, można napisać:

```
rozmiar := lengt( tab );
SetLength( tab, 2*rozmiar );
```

przy czym dotychczas wpisane wartości są zachowane. Funkcja ta może również „skasować” tablicę:

```
SetLength( tab, 0 );
```

* * *

W pascalu istnieje funkcja `lenght()` zwracająca liczbę elementów tablicy. Dla tablic statycznych, kiedy i tak musimy znać/pamiętać granice indeksowania, nie jest aż tak potrzebna, ale bardzo się przydaje dla tablic dynamicznych. Wiedząc, że indeksowanie dla tego typu tablic, zawsze zaczyna się od 0, znajomość rozmiaru tablicy pozwala na bezpieczne jej przetwarzanie:

```
for i:=0 to lenght( tab )-1 do
begin
  WriteLn( tab[i] );
  //inne działania na elementach tablicy
end;
```

Łącząc funkcjonalność `lenght()` z opisaną funkcją `SetLength()` można zrealizować zwiększanie rozmiaru tablicy o pojedyncze elementy:

```
rozm := lenght( tab );
SetLength( tab, rozm+1 );
tab[rozm] := kolejny;
```

To samo krócej (ale za to być może mniej czytelnie):

```
SetLength( tab, lenght( tab )+1 );
tab[lenght( tab )-1] := kolejny;
```

Nawiasem pisząc, to bodaj najczęstszy scenariusz tworzenia tablicy (czy ogólniej zbiornika). Odczytujemy dane, jedna po drugiej i pojedynczo wpisujemy do tablicy za każdym razem zwiększając jej rozmiar. Bardzo typowym przykładem jest odczytywanie pliku tekstowego linia po linii i zapamiętanie jego treści w tablicy napisów. Nie znamy wielkości pliku, kończymy czytanie, kiedy dojdziemy do jego końca.

2.6 Zasady programowania proceduralnego

Ten odcinek będzie zawierał nieco teorii, która może pomóc nie nabierać nieodpowiednich nawyków w czasie nauki. Przy pisaniu dużych programów można się przekonać, że teoria jest przydatna i pożyteczna. Wydaje mi się, że dobrze jest ją poznać, zanim dopadną nas wspomniane złe nawyki, które potem trzeba wypleniać.

Separacja funkcjonalności kodu

Gdy tworzymy podprogram, starajmy się, żeby zajmował się jednym rodzajem funkcjonalności. Prosty przykład zobaczymy poniżej. Będzie to funkcja wyliczająca największy wspólny dzielnik dwóch liczb.

Opowiem, jak wygląda algorytm Euklidesa, służący do znalezienia NWD. Bierzemy dwie liczby dodatnie. Od większej z nich odejmujemy mniejszą. Czynność tę powtarzamy, aż obie będą sobie równe. Dla pary (15, 12) przebieg algorytmu będzie następujący:

(15, 12) → (15−12=3, 12) → (3, 12−3=6) → (3, 6−3=3) czyli NWD=3

Oznacza to, że będziemy musieli użyć pętli. Zauważmy, że równość/ nierówność liczb jest sensowna zanim dokona się pierwszego kroku – w przypadku równości, liczba z pary jest jednocześnie NWD. Skoro tak, to wybierzemy pętlę `while do`.

> Taka postać algorytmu jest dość powolna, można ją zmodyfikować, zamieniając odejmowanie przez branie reszty: w kroku pętli większą liczbą zastępujemy resztą z dzielenia większej przez mniejszą (operator `mod`).

Zadanie: Na podstawie tych danych spróbuj napisać program do wyliczania NWD. Być może jest podobny do poniższego:

```
program NajwiekszyWspolnyDzielnik;
var LiczbaA, LiczbaB, a, b : integer;
BEGIN
{ nadanie wartości }
  LiczbaA := 12;
  LiczbaB := 42;
{ wyliczenie NWD, za pomocą zmiennych pomocniczych a, b }
  a := LiczbaA;
  b := LiczbaB;
  while a<>b do
  begin
    if a>b then a := a-b
    else b := b-a;
  end;
{ wypisanie wyników }
  WriteLn( 'NWD ', LiczbaA, ' ', LiczbaB, ' wynosi ', a )
END.
```

Jak widać w komentarzach program jest podzielony na trzy części: inicjalizacja zmiennych, wyliczenie NWD i wypisanie wyników. Naszym zadaniem będzie umieszczenie części obliczeniowej w osobnym podprogramie, a ściślej rzecz biorąc w funkcji. Nie będzie się ona zajmować nadawaniem wartości danych w programie, ani wyświetlaniem komunikatów. Możemy w takich przypadkach myśleć, że z kodu działającego w trybie tekstowym, wydzielamy kod dający się użyć w programie okienkowym.

Pora zastanowić się jakie zmienne z programu głównego przeniesiemy do podprogramu. W programie mamy dwie pary liczb: `liczbaA`, `LiczbaB` oraz `a` i `b`. Pierwsza para służy do przechowywania informacji o jakie liczby nam chodzi.

Zmienne a i b są zmiennymi pomocniczymi – na końcu każda z nich jest równa NWD. W dalszym ciągu programu (gdyby taki był) nie będą nam potrzebne. To podpowiada nam, że a i b będą używane w funkcji i znikną z programu głównego. Wartości liczbaA, LiczbaB zostaną, ich wartości skopiujemy je do funkcji przez argumenty.

Zadanie: Spróbuj sam wydzielić część kodu do funkcji. Porównaj z poniższą propozycją:

```
program NajwiekszyWspolnyDzielnik;

function NWD( a, b : integer ) : integer;
begin
  while a<>b do
  begin
    if a>b then a := a-b
    else b := b-a;
  end;
  NWD := a;
end;

var LiczbaA, LiczbaB, wynik : integer;
BEGIN
{ nadanie wartości A, B }
  liczbaA := 12;
  LiczbaB := 42;
{ wyliczenie wyniku }
  wynik := NWD( LiczbaA, LiczbaB );
{ wypisanie wyników }
  WriteLn( 'NWD␣', LiczbaA, '␣', LiczbaB, '␣', wynik );
END.
```

Argumenty funkcji (u nas a i b) są jej zmiennymi lokalnymi. Podobnie będzie dla procedury. To ważna cecha o której czasami się zapomina, a warto z niej korzystać.

Sposób polegający na przenoszeniu kodu z programu głównego do podprogramów jest użyteczny na początku nauki kodowania. W przyszłości, gdy nabierzesz sprawności w projektowaniu przepływu danych z/do podprogramu, będziesz raczej od razu pisał procedury i funkcje.

Zauważ, że funkcję NWD() można używać w innych programach – np. w programie dokonującym działań na ułamkach. Już na tak prostym przykładzie widać, że powinniśmy tak planować pisanie podprogramów, żeby można było je testować w innym programie niż docelowy. W miarę możliwości powinny być pisane niezależnie od reszty kodu. Program główny zawiera dane wystarczające do sensownego uruchomienia kodu podprogramu i wypisania wyników jego działania.

Na koniec zadanie uczulające na różnicę między rodzajem argumentów: kopiowaniem wartości a dostępem do zmiennej (z użyciem `var`). Spróbuj zmienić rodzaj argumentu funkcji `NWD()` dopisując słowo `var`:

```
function NWD( var  a, b : integer ) : integer;
...
```

Sprawdź jak zmieni się działania programu i zastanów dlaczego tak się stało. Nieuważne utworzenie argumentu jako dostępu do zmiennej, może być źródłem trudno wykrywalnych błędów w programie – wszak dla kompilatora wszystko jest OK.

Parametryzacja podprogramów

Gdy trzeba utworzyć różne warianty danej czynności, mamy dwie drogi postępowania: albo tworzymy wiele osobnych i mało różniących się podprogramów, albo piszemy jeden podprogram obejmujący wszystkie przypadki. Trzeba przyznać, że pierwszy sposób jest łatwiejszy, szczególnie, gdy już jedną z wersji mamy napisaną. Kopiuje się wtedy jej kod do kolejnej procedury czy funkcji i wprowadza drobne różnice. Ale nie jest to dobre rozwiązanie. Gdy przyjdzie do wprowadzania zmian lub poprawek w kodzie, musimy to robić osobno dla każdej wersji – prawdopodobieństwo popełnienia pomyłki jest zdecydowanie większe, niż gdybyśmy mieli tylko jeden podprogram do modyfikacji.

Jako przykład podam program do wyszukiwania danych z tablicy. Przykład jest uproszczony, nawet nierealistyczny, ale podobne, bardziej skomplikowane przykłady zdarzają się w zastosowaniach praktycznych. Załóżmy, że mamy tablicę osób,

```
type TZaloga: array[1..N] of Osoba;
...
var zaloga : TZaloga;
```

która zawiera rekordy mające pole `wiek`. Będziemy szukać osób, które są starsze niż podana liczba lat i wykonać dla nich jakieś czynności:

```
procedure Starsi( wart : integer; const zaloga : TZaloga );
var i : integer;
begin
  for i:=1 to N do
    if zaloga[i].wiek > wart then
    begin
      //jakieś czynności na wyszukanym rekordzie:
      ZrobCos( zaloga[i] );
      ...
    end;
end;
```

Jak ta procedura się sprawdzi, zapewne pojawi się potrzeba wypisania młodszych od wskazanego wieku, mających dokładnie tyle-a-tyle lat, oraz będących w zadanym przedziale wiekowym. Każda z tych procedur jest taka sama za wyjątkiem linii sprawdzającej warunek. Mając więc podprogram działający dla zakresu:

```
procedure Wybierz( min, max : integer; const zaloga : TZaloga )
var i : integer;
begin
  for i:=1 to N do
    if ( zaloga[i].wiek >= min ) and
       ( zaloga[i].wiek <= max ) then
    begin
      ZrobCos( zaloga[i] );
      ...
    end;
end;
```

można wykorzystać go w innych, szczegółowych przypadkach:

```
const
  WiekMax = 150; { ludzie dłużej nie żyją }
  WiekMin = 0;

procedure Starsi( wart : integer; const zaloga : TZaloga );
begin
  Wybierz( wart+1, WiekMax, zaloga );
end;
```

* * *

Zauważmy jeszcze typową dla pascala konwencję nazewniczą, według której zdefiniowane typy zaczynają się od litery `T`. Będę ją stopniowo wprowadzał w kursie. Tu znalazła naturalne zastosowanie, bo w programie istniała tylko jedna zmienna typu `TZaloga` i wygodnie było nazwać ją prawie tak samo jak typ.

Programowanie z ukrywaniem zmiennych

Czasami pojawia się potrzeba, żeby utworzyć procedurę lub funkcję mającą wiele argumentów. Takie podprogramy spotyka się w rzeczywistości – można zajrzeć na stronę MSDN gdzie wiele takich kwiatków znajdziemy. Obsługa długiej listy argumentów jest trudna, bo trzeba pamiętać, które co oznaczają. Jeśli pomylimy kolejność argumentów tego samego typu, to kompilator nie zgłosi błędu! Rozwiązaniem kłopotu może być specyficzne użycie rekordu.

Tworzymy rekord zawierający argumenty i zamieniamy całą ich listę na jedną zmienną. Oczywiście robimy tak, jeśli zestaw argumentów, jaki chcemy zastąpić

rekordem, ma ze sobą coś wspólnego. Czyli mając funkcję `CzyRKw` (czy równanie kwadratowe ma rozwiązania) do rozwiązywania równania kwadratowego z pięcioma argumentami:

```
function CzyRKw( a, b, c :real ; var x1, x2 : real ) : boolean;
...
```

z trzech z nich tworzymy typ `DaneRKw` (dane równania kwadratowego):

```
DaneRKw = record
  a, b, c : real
end;
```

a samą funkcję przekształcimy w następujący sposób:

```
function CzyRKw( dane : DaneRKw; var x1, x2 : real ) : boolean;
...
```

Rzecz jasna teraz dostęp do liczb zapisywanych wcześniej jako `a`, `b`, `c` jest inny:

```
delta := dane.b*dane.b - 4*dane.a*dane.c;
...
```

bo są one polami rekordu dane.

* * *

Ukrywanie zmiennych ma jeszcze jedną zaletę – uniezależniamy od siebie kod różnych fragmentów programu.

Rozpatrzmy przypadek, podprogramu sprawdzającego poprawność danych. Niech funkcja logiczna sprawdza czy plik o nazwie `sciezka` istnieje, oraz czy przygotowano już jakieś dane (dla ustalenia uwagi pomyślmy, że `IleDanych` będzie wcześniej odczytanym rozmiarem tablicy dynamicznej):

```
function CzyDaneOK( sciezka:string; IleDanych:integer ) :boolean;
begin
  CzyDaneOK := true;
  if IleDanych = 0 then CzyDaneOK := false;
  if not FileExists( sciezka ) then CzyDaneOK :=false;
end;
```

Można sobie pomyśleć, że od powodzenia tej funkcji zależy wykonanie jakichś czynności związanych z plikiem i zasobem danych. Funkcja ta niech będzie wykorzystywana w kilku miejscach programu. Jednak wraz z rozbudową programu, konieczne jest poszerzenie listy sprawdzanych warunków – niech będzie to po prostu jakaś zmienna logiczna:

```
function CzyDaneOK( sciezka : string;
                    IleDanych : integer;
                    stan : boolean ) : boolean;
...
```

Po zmianie tej funkcji od razu trzeba poprawić kod we wszystkich miejscach, gdzie jest ona wywoływana, co może dezorganizować pracę. Tym bardziej, że nie we wszystkich miejscach zmienna `stan` może odgrywać rolę (skoro do tej pory nie była potrzebna).

Gdy zamiast coraz dłuższej liczby argumentów, zastosujemy rekord:

```
DaneProgramu = record
  sciezka   : string;
  IleDanych : integer;
  stan      : boolean;
end;
```

istnieje możliwość wygodnego wprowadzania zmian. Funkcję w dalszym ciągu da się skompilować:

```
function CzyDaneOK( DaneProgramu dane ) : boolean;
...
```

a jej modyfikację możemy przeprowadzić w dogodnym czasie.

Programowanie z ukrywaniem zmiennych stanowi pierwszy krok do zrozumienia enkapsulacji – jednej z technik programowania obiektowego.

Dzielenie programu na etapy

Kiedy nasz program składa się z kilku osobnych zadań, wygodnie wydzielić je w postaci podprogramów. Zasada ta, zwana też „dziel i zwyciężaj”, zakłada, że jeśli jakiś problem podzielimy na części, to sumaryczna trudność wykonania owych składowych, będzie mniejsza niż gdybyśmy pisali program bez podziału.

Inną zaletą wydzielania składowych jest przejrzystość kodu. Zaglądamy do programu głównego:

```
var dane : table of Osoba;
...
BEGIN
  OdczytDanych( dane, 'd:\katalog\dane.txt' );
  PrzetwarzanieDanych( dane );
  RaportKoncowy( dane, 'd:\katalog\raport.html' );
END.
```

i od razu wiemy, co nasz program robi. Jeśli będziemy ciekawi jak wykonywane są kolejne czynności, zajrzymy do kodu wywoływanych procedur.

Inną zaletą takiego podziału będzie zindywidualizowanie dostępu podprogramów do danych (u nas chodzi o zbiornik `dane`). Dwie pierwsze procedury na pewno potrzebują zmiennej dane do zapisu (użycie `var`), trzecia do odczytu (przez `const`). Mamy wtedy pewność, że jeśli dane nie są poprawne, to na pewno nie zepsuła ich procedura `RaportKoncowy()` – czyli błędu musimy szukać w dwóch pierwszych.

Ograniczanie roli zmiennych globalnych

W powyższym przykładzie widzimy, że tabela `dane` jest przekazywana każdej procedurze poprzez argument. Początkujący programiści zwykle znajdują inne rozwiązanie: deklarują tablicę `dane` przed kodem podprogramów, dzięki czemu mają one do niej dostęp, bez umieszczania jej na liście argumentów. Zmienne deklarowane w ten sposób nazywa się zmiennymi globalnymi.

Generalnie zaleca się ograniczanie ilości zmiennych globalnych do minimum. Trzeba zauważyć, że zmienne globalne są niesamowicie wygodne. Jednocześnie wysoce niebezpieczne, a znalezienie błędu związanego z nieprawidłowym użyciem zmiennych globalnych jest trudne, a dla dużych programów bardzo trudne.

Przeanalizuj poniższy przykład:

```
program ZmiennaGlobalna;
{ zwróć uwagę na lokalizację deklaracji zmiennej nr }
var nr : integer;

procedure Pasek( N : integer );
begin
  for nr := 1 to N Write('-');
  WriteLn
end;

BEGIN
  nr := 3;
  { wypisanie zmiennej nr niby równej 3 w ładnej ramce: }
  Pasek( 10 );
  WriteLn( nr:5 );
  Pasek( 10 );
END.
```

Powinieneś zauważyć, że błąd polegał na „zapomnieniu” zadeklarowania zmiennej lokalnej procedury `Pasek`. Spowodowało to, zmianę wartości zmiennej `nr` w programie głównym. Tu błąd jest łatwy do wykrycia, bo program jest krótki, ale jeśli ma on 1500 linii? Jak wyszukamy miejsce, w którym gdzieś „przy okazji” zmieniliśmy wartość zmiennej globalnej?

Rozwiązaniem eliminującym tego typu przypadki jest deklaracja zmiennych używanych w programie głównym bezpośrednio przed początkiem bloku `BEGIN END.`

```
program BezZmiennejGlobalnej;

procedure Pasek( ... );
...

var nr : integer;
BEGIN
  ...
END.
```

Wtedy zmienna `nr` będzie po prostu zmienną lokalną dla programu głównego.

* * *

Ponieważ możemy zagnieżdżać podprogramy, procedury wewnętrzne mogą korzystać ze zmiennych procedury zewnętrznej, jeśli tylko zostały zadeklarowane przed treścią takiej procedury zagnieżdżonej. W szczególności takimi zmiennymi są argumenty procedury nadrzędnej. Dla procedur wewnętrznych są one jak zmienne globalne.

Poniżej kod do samodzielnej analizy. Przekazujemy wartość do zmiennej `liczba` (argument) w procedurze `Zewnetrzna`, a `Wewnetrzna` ma do dostęp do bieżącej wartości `liczba`.

```
procedure Zewnetrzna( liczba : integer );

  procedure Wewnetrzna;
  begin
    Writeln( liczba );
  end;

begin
  inc(liczba);
  Wewnetrzna;
end;
```

Wywołanie `Zewnetrzna( 10 );` spowoduje wypisanie wartości 11. Innymi słowy argument `liczba` zachowuje się jak zmienna globalna – rzecz jasna „z punktu widzenia” procedury zagnieżdżonej.

* * *

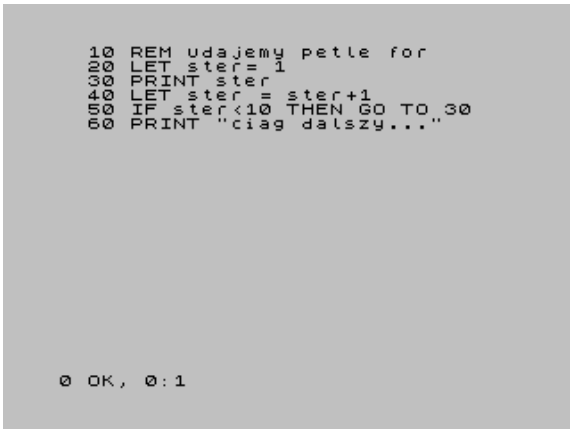
Im wcześniej weźmiemy sobie do serca wymienione w niniejszym odcinku zasady, tym sprawniej będzie wyglądać nasze programowanie. Przydadzą się one zresztą w przyszłości nie tylko w programowaniu proceduralnym.

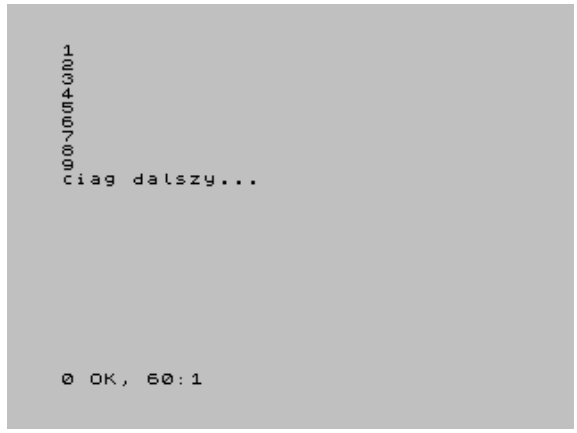
2.7 Poszerzenie wiadomości o instrukcjach sterujących. Skoki

Kiedy w dawnych czasach pisano programy w dawnych językach, sterowanie przebiegiem programu uzyskiwano poprzez kombinację `IF` i `GOTO`. Ta druga instrukcja nakazywała przeskoczyć wykonywanie programu do wskazanej linii. Czyli np. w jakimś archaicznym BASICu mogło to wyglądać następująco:

```
10 REM Tu udajemy pętlę FOR od 1 do 9
20 LET ster = 1
30 PRINT ster
40 LET ster = ster+1
50 IF ster<10 THEN GO TO 30
60 PRINT "ciąg dalszy"
```

Coś podobnego uzyskałem na emulatorze ZX Spectrum, choć przykład nie pasuje do końca, bo w spektrusiowym BASICu `FOR` jednak istnieje, ale inne sposoby sterowania rzeczywiście odbywają się za pomocą `IF` i `GO TO`.





W tym stylu programowania `GOTO` służyło do „wszystkiego”, czyli do uzyskiwania funkcjonalności: instrukcji sterujących, pętli i podprogramów. No i jeszcze do swobodnego skakanie po kodzie, który stawał się w ten sposób trudno czytelny. Nazywa się to uczenie spaghetti-kod i było największym złem starożytnego programowania. Drugim, prawie tak samo wielkim, była słaba kontrola nad zakresem zmiennych – wszystkie były globalne (i tak w pewnym sensie jest do dziś dla wielu współczesnych języków skryptowych). O zmiennych globalnych była mowa – dalej są złe – pora więc napisać coś na temat skoków.

W podręcznikach powstałych w końcówce wieku XX, a nawet na początku XXI, ich autorzy wciąż straszą widmem przerażających skoków. Niesłusznie, albowiem dziś przyczyniają się one do poprawienia czytelności kodu. Można by pomyśleć, że w jakimś sensie niniejszy kurs też cierpi na tę fobię, bo opis pętli pojawił się w poprzednim module, a skoki dopiero teraz.

Skok `break`

Zajrzyjmy na stronę 32, gdzie znajdziemy następujący kod:

```
repeat
  Write( 'Podaj␣liczbę␣całkowitą.␣Podanie␣zera␣kończy␣zabawę.␣');
  Readln( liczba );
  if liczba<>0 then
    Writeln( 'odwrotność␣to:␣', 1/liczba );
until liczba=0;
```

Zauważmy wady: warunek na (nie)zerowość liczby jest sprawdzany dwa razy. Raz robi to `if`, drugi raz pętla na swym końcu. Gdybyśmy chcieli opowiedzieć co robi ta pętla „swoimi słowami", to pewnie opis byłby następujący: W każdym kroku pytamy o liczbę, jeśli jest ona równa zero kończymy pętlę, jeśli różna od zera liczymy odwrotność. Jeśli chcemy wprowadzić w życie słowa „kończymy pętlę", tak by instrukcje w kodzie odpowiadały słownemu opisowi, rzeczywiście trzeba pętlę przerwać. Służy do tego instrukcja `break`. Przerywa ona działanie pętli.

Używając `break`, uzyskujemy kod bliższy językowi naturalnemu:

```
repeat
  Write( 'Podaj␣liczbę␣całkowitą.␣Podanie␣zera␣kończy␣zabawę.␣');
  Readln( liczba );
  if liczba=0 then break;
  Writeln( 'odwrotność␣to:␣', 1/liczba );
until liczba=0;
```

Poprawiła nam się czytelność (jedno wcięcie mniej), ale wciąż warunek jest sprawdzany dwa razy, co podwaja okazję do popełnienia błędu. Pora więc na przedstawienie konstrukcji, która jest powszechnie stosowana, a która na pierwszy rzut oka wygląda wręcz niebezpiecznie:

```
while true do
begin
  Write( 'Podaj␣liczbę␣całkowitą.␣Podanie␣zera␣kończy␣zabawę.␣');
  Readln( liczba );
  if liczba=0 then break;
  Writeln( 'odwrotność␣to:␣', 1/liczba )
end;
```

Warunek jest po prostu stałą `true`, czyli zawsze jest prawdziwy – wygląda to na pętlę nieskończoną. Tego rodzaju pętle muszą gdzieś użyć instrukcji `break`. Warto zaprzyjaźnić się z tą wersją `while do`, gdyż jest użyteczna w zaskakująco wielu przypadkach.

Poleceniem tym możemy przerwać też pętlę `for`, jeśli w pewnym momencie nie będzie potrzeby doprowadzania jej do końca. Bardzo klasycznym przykładem

jest sprawdzanie czy podana liczba jest liczbą pierwszą. I to będzie treść poniższego zadania:

Zadanie. Sprawdzić czy dana liczba jest liczbą pierwszą. Niestety nie ma łatwych sposobów rozwiązania tego zagadnienia. Jednym ze sposobów jest badanie reszt z dzielenia przez kolejne liczby 0 do `liczba-1`. Gdy któraś reszta jest równa zero, to wiadomo, że mamy liczbę złożoną. Weźmy np liczbę 35 i sprawdźmy kolejne dzielniki:

- 35 mod 2 jest równe 1, sprawdzamy dalej,
- 35 mod 3 jest równe 2, sprawdzamy dalej,
- 35 mod 4 jest równe 3, sprawdzamy dalej,
- 35 mod 5 jest równe 0, przerywamy sprawdzanie – liczba nie jest pierwsza.

Dla liczby 7 sprawdzenie wygląda inaczej:

- 7 mod 2 jest równe 1, sprawdzamy dalej,
- 7 mod 3 jest równe 1, sprawdzamy dalej,
- 7 mod 4 jest równe 3, sprawdzamy dalej,
- 7 mod 5 jest równe 2, sprawdzamy dalej,
- 7 mod 6 jest równe 1, doszliśmy do końca, żadna reszta nie jest równa 0 – liczba jest pierwsza.

Informację o pierwszości liczby niech przechowuje zmienna logiczna `CzyPierwsza`, która domyślnie niech będzie równa `true`. Przy uzyskaniu reszty równej zero, należy wykonać dwie instrukcje: zmienić zmienną `CzyPierwsza` na `false` i przerwać pętlę.

Dokonajmy dwóch modyfikacji, które znacząco przyspieszą obliczenia: Zamiast sprawdzać dzielenia przez wszystkie liczby parzyste (2, 4, 6…), sprawdźmy dzielenie przez 2, a następnie używajmy tylko dzielników nieparzystych – trzeba się będzie zastanowić jaki będzie zakres zmiennej sterującej pętli `for`, albo użyjmy pętli `repeat` z dodatkową zmienną. Po drugie nie musimy sprawdzać wszystkich dzielników tylko tych, co są mniejsze lub równe pierwiastkowi z badanej liczby (zastanów się dlaczego). Zmieni to koniec zakres zmiennej sterującej `for`, lub warunku pętli `repeat`. Jedna z propozycji rozwiązania – str. 141.

Skok `continue`

Instrukcja ta przerywa dany krok pętli. Jest ona bardzo użyteczna w sytuacji kiedy mamy zbiornik z danymi, ale mamy zadziałać tylko na niektóre z nich. Jako przykład zobaczmy raport pracowników będących kierownikami, wiedząc, że są wpisani do tablicy z pracownikami.

```
type
Pracownik = record
  imie       : string[25];
  nazw       : string[50];
  kierownik  : boolean;
  stanowisko : string;
  ...
end;
...
var
  zaloga   : array of Pracownik;
  biezacy : Pracownik

...
{ w kodzie programu }
for i:=0 to length(zaloga)-1 do
begin
  biezacy = zaloga[i];
  { najpierw omijamy nie-kierowników }
  if biezacy.kierownik = false then continue;
  { jak trafi się kierownik, to wypisujemy jego dane }
  WriteLn( biezacy.imie, '␣', biezacy.nazw, '␣' );
  ...
end;
```

Zauważmy, że nie stosujemy tu `else` i grupowania instrukcji – jeśli wywołany zostanie skok `continue`, kolejne instrukcje w danym kroku pętli nie zostaną wykonane. W tym przypadku zastosowanie skoku i brak dodatkowego wcięcia zwiększają czytelność kodu.

Skok `goto`

Groza `goto` była tak wielka, że co prawda pozwolono na jej użycie, ale obłożono tę instrukcję dwoma warunkami: Po pierwsze trzeba deklarować wszystkie miejsca do których program ma przeskakiwać w trakcie działania. Deklarujemy je za pomocą słowa kluczowego `label`, w miejscu gdzie deklaruje się zmienne:

```
procedure ...;
label koniec, brak;
var ...
begin
...
end;
```

Miejsca te opisane są dodatnimi liczbami całkowitymi. Można też użyć stałych o czytelnych nazwach (tak, jak w powyższym przykładzie), które w pewnym sensie zachowują się jak stałe typu wyliczeniowego. Miejsce do którego ma przeskoczyć program oznaczamy nazwą/liczbą z dwukropkiem:

```
...
brak:;
...
```

Drugie ograniczenie nie pozwala `goto` wyskoczyć poza daną procedurę/funkcję (albo program główny).

Do czego można używać `goto`? Podobno czasami wygodnie ustawić sobie np. label `koniec` i kierować tam wykonanie programu omijając kolejne instrukcje, których nie ma sensu wykonywać. Ale taki kod zaburza moje poczucie estetyki (choć np. taki MSDN korzysta z takiego sposobu budowania kodu).

```
if not warunek1 then goto koniec;
...
if not warunek2 then goto koniec;
...
koniec:;
```

Na pewno `goto` się przydaje, gdy mamy potrzebę przerwania kilku zgnieżdżonych pętli. Skok `break` przerwałby tylko jedną z nich.

Instrukcja `exit`

Przerywa działanie danej procedury czy funkcji. Dla funkcji ma dodatkowo dość użyteczną własność, bo wprowadza mechanizm zwracania wartości bez używania nazwy funkcji. Czasami jej użycie polepsza czytelność programu. Porównajmy zblokowania i wcięcia w poniższej funkcji do wyliczania pierwiastków:

```
type
  WynRKw = ( RKwOK, Azero, DeltaUj );
  DaneRKw = record
    a, b, c : real
  end;

function CzyRKw( dane : DaneRKw; var x1, x2 : real ) : WynRKw;
var
  delta, pdelta : real;
begin
  if dane.a = 0 then exit( Azero );
  delta := dane.b*dane.b - 4*dane.a*dane.c;
  if delta<0 then exit( DeltaUj );
  pdelta := sqrt(delta);
  x1 := (-dane.b-pdelta)/(2*dane.a);
  x2 := (-dane.b+pdelta)/(2*dane.a);
  CzyRKw := RKwOK;
end;
```

a programem, który robi to samo na stronie 25.

Instrukcja `halt`

Ta instrukcja przerywa cały program, niezależnie gdzie będzie użyta. Można się zastanawiać czy taki killer w ogóle jest przydatny? Otóż odpowiedź brzmi: w pewnych sytuacjach tak.

Chodzi o programy, których wykonanie może się nie udać, i nie zależy to od programisty. Choćby z powodu przygotowania niewłaściwych danych wejściowych – nie ma wtedy sensu, żeby ciągnąć wykonanie programu do końca, bo wyniki też wyjdą złe. Należy poinformować użytkownika o błędzie i skończyć program.

```
{ niech warunek oznacza że dane są OK }
if not warunek then
begin
  WriteLn( StdErr, 'Dane przesłane do programu nie są poprawne');
  halt( 1 )
end;
{ ciag dalszy będzie wykonany jeśli warunek=true }
```

Niezależnie w którym fragmencie programu dojdzie do uruchomienia `halt`, program zakończy działanie, a system operacyjny dostanie tzw. wartość błędu ustawioną na 1. Wartość ta dostępna jest pod Windowsem w zmiennej środowiskowej `errorlevel` i oznacza przypadek porażki programu. Skoro coś może się nie udać, to zwykle może to nastąpić z różnych powodów.

Oto przykład nieudanego uruchomienia programu z linii poleceń:

```
C:\Users\grzes2a\programy>program.exe 2>> program.log
C:\Users\grzes2a\programy>echo %errorlevel%
1
```

Jednocześnie powstał plik `program.log`, do którego zapisano linię:

```
Dane przesłane do programu nie są poprawne
```

Należy się jeszcze wyjaśnienie: pierwszy argument procedury `WriteLn()` czyli `StdErr`, oznacza tzw. standardowe urządzenie błędu – nazwa została zaczerpnięta z języka C. Kierowane są na niego dane wyjściowe niebędące spodziewanymi wynikami, wynikającymi z prawidłowej pracy programu, ale właśnie komunikaty błędu. Można je przekierować (w systemie operacyjnym) za pomocą 2> (tworzenie pliku) lub 2>> (dopisywanie) do wskazanego pliku.

Zmienna środowiskowa `erorrlevel` informuje o sukcesie/porażce ostatnio uruchamianego programu. Jeśli wszystko zadziałało dobrze, zwyczaj każe zwracać 0. Dla programów pascalowych jest to domyślnie zwracana wartość. Jeśli jednak się nie udało, program powinien zwrócić jakąś liczbę niezerową. Dobrze, jeśli w dokumentacji programu autor wypisze, co oznaczają konkretne wartości błędu – w programie można je uwzględnić np. przez odpowiedni typ wyliczeniowy.

Konstrukcja `case of`

Niezwykle lubiana przez programistów konstrukcja `case` swego czasu narobiła prawie tyle samo zła, co okryte złą sławą `goto`. Między innymi dlatego pojawia się dopiero teraz, jako ostatnia ze stosowanych instrukcji sterujących.

Jest ona rozszerzeniem instrukcji `if then else`. O ile `if` musi dostać do sprawdzenia warunek/zmienną logiczną, to `case` potrzebuje tzw. wartości porządkowych, czyli takich, które dałyby się zapisać, jako zmienne całkowite. Ale zobaczmy jak wygląda owa konstrukcja:

```
case zmienna of
  A : instrukcjaA;
  B : instrukcjaB;
  ...
else
  instruckjeInne;
end;
```

Zgodnie z tym co napisaliśmy wyżej zmienna może być typu `boolean`, znakowego, wyliczeniowego lub dowolnego całkowitego. Wpisane tu `A`, `B`…są stałymi – mogą być wypisane jawnie (jako `1`, `2`, `#34`, `true`, `'a'`) albo zdefiniowane w kodzie przez `const`.

Działanie jest następujące: W zależności od tego jaka jest wartość zmiennej, wykonywane są instrukcje wypisane po dwukropku. Jeśli żadna stała nie odpowiada wartości zmienna, zostanie wykonany blok po słowie `else`.

Jeśli chodzi o blokowanie: jeśli więcej niż jedna stała odpowiada naszemu przypadkowi, możemy je oddzielać przecinkiem. Gdy istnieje potrzeba wykonania więcej niż jednej instrukcji w danym przypadku musimy blokować je wpisując w `begin end`, przy czym bloku `else` to nie dotyczy:

```
Write( 'Podaj␣liczbę␣jednocyfrową:␣');
ReadLn( liczba );
case liczba of
 2, 3, 5, 7 : WriteLn( 'to␣liczba␣pierwsza' );
 4, 6, 8    : begin
                Write( 'to␣liczba␣parzysta,␣' );
                WriteLn( 'ale␣nie␣jest␣pierwsza' );
              end;
 0          : Writeln( 'to␣najmniejsza␣liczba' )
else
  WriteLn( liczba, '␣jest␣nieciekawa' );
  Writeln( 'Postaraj␣się␣lepiej␣na␣drugi␣raz' )
end;
```

Blokowanie instrukcji w `case of` zaburza czytelność kodu, co zdaje się jest już widoczne na powyższym przykładzie. Dlatego poważnie potraktuj zalecenie: niech

każdy wybór oznacza wykonanie tylko jednej instrukcji – np. wywołanie jakiegoś podprogramu.

Blok `else` może nam posłużyć do znalezienia błędów w programie. Standardowe działanie `case` powinno polegać na tym, że sekcja `else` nie powinna być nigdy wykonywana. Po prostu powinniśmy przewidzieć wszystkie możliwe przypadki. Jeśli jednak pojawi się jakiś nieprzewidziany, to oznaka, że musimy poprawić program. A skoro program źle działa, to może w ogóle go przerwać?

```
case opcja of
....
else
  WriteLn( StdErr, 'Nieznana opcja: ', ord(opcja) );
  halt( 1 ); { błąd nr 1 oznacza nieznaną opcję }
end;
```

* * *

Słowo wyjaśnienia skąd wzięła się opinia rozrabiaki. Otóż jest to dość wygodna konstrukcja, którą zbyt łatwo wzbogacać o kolejne instrukcje. Szczególnie jeśli zajrzymy do starych programów okienkowych pisanych pod Windows. Program taki sprowadzał się do wieeelkiego `switch`-a – to odpowiednik `case` w języku C – który na podstawie przechwyconego zdarzenia wykonywał daną czynność. Żeby mieć jakieś pojęcie odsyłam żądnych wiedzy do (w miarę) łagodnego wstępu do WinAPI (C/C++) autorstwa p. Z. Kozy – znajdziemy tam dwa przykłady bardzo króciutkich `switch`-y. W typowych programach użytkowych były o wiele, wiele większe. Zwykle nagromadzenie kodu w jednej instrukcji było tak wielkie, iż przestawał on być czytelny.

Dziś takie niebezpieczeństwo pewnie nam nie grozi, bo i programy okienkowe tworzy się zupełnie inaczej.

2.8 Znaki i napisy

Kodowania ASCII

Zacząć chyba trzeba od kilku informacji na temat danych przechowywanych w systemach komputerowych. Tak się ułożyło, że współcześnie podstawową paczką danych jest bajt, czyli osiem bitów. Da się go zapisać jako liczbę całkowita z przedziału 0–255. Bajty służyły też do zapisywania tekstu. Umówiono się jaka liczba odpowiada jakiemu znakowi – przyporządkowanie to nazywa się dziś ASCII i pierwotnie używało liczb siedmiobitowych, obejmując sobą tylko znaki alfabetu angielskiego (wiem, wiem zaraz powiecie, że łacińskiego). Najrozmaitsze działania mające na celu dołączenie do zestawu znaków innych liter czy nawet alfabetów trwają do dziś. I końca nie widać.

Ze względów historycznych typ `char` – czyli ten, który ma przechowywać znaki – też ma rozmiar jednego bajta. Można w nim zmieścić 128 dodatkowych znaków nieuwzględnionych w siedmiobitowym ASCII. Wiele organizacji, które miało jakieś wpływy w światku komputerowym, próbowały pododawać potrzebne sobie znaki. Zestawy, które udało się wprowadzić w życie, nazywają się stronami kodowymi, każda z nich ma swoją nazwę i numer. Współczesny użytkownik w Polsce może spotkać się jeszcze z trzema:

- ISO 8859-2 zwana też ISO Latin 2, która notabene jest polską normą. Zawiera znaki stosowane w alfabetach krajów Europy Środkowej i Wschodniej opartych o alfabet łaciński (bez cyrylicy). Dlatego swego czasu stosowana była w systemach unixowych, gdy potrzebne były polskie ogonki. W latach 90-tych uważana była za tę prawdziwą, „internetową" stronę kodową – używana przez ówczesne linuksy, jako domyślna.
- CP 1250 – „windowsowa". Była chyba lepiej pomyślana niż Latin-2, bo np. przewidywała istnienie cudzysłowów. Przez długi czas, windowsowe programy do edycji tekstu zapisywały go w tej właśnie stronie kodowej. Dopiero od Windowsa 10 Notatnik przy próbie zapisu pliku tekstowego, użyje innego kodowania (UTF-8), wcześniej chciał to robić właśnie w CP 1250.
- CP 852 – zwana DOSową, choć wprowadzona przez IBM (chodzi o czasy, gdy DOS i Windows były tworzone wspólnie przez MS i IBM). Jeśli otworzymy sobie konsolę, czyli program `cmd.exe`, to wpisywane znaki odpowiadają kodowaniu 852.

Ułożenie polskich znaków diakrytycznych przedstawia poniższa tabelka. Podane liczby to numery bajtów, które w danej stronie kodowej są interpretowane jako litery.

małe	ą	ć	ę	ł	ń	ó	ś	ź	ż
ISO 8859-2	177	230	234	179	241	243	182	188	191
Windows (CP1250)	185	230	234	179	241	243	156	159	191
IBM (CP852)	165	134	169	136	228	162	152	171	190
DUŻE	Ą	Ć	Ę	Ł	Ń	Ó	Ś	Ź	Ż
ISO 8859-2	161	198	202	163	209	211	166	172	175
Windows (CP1250)	165	198	202	163	209	211	140	143	175
IBM (CP852)	164	143	168	157	227	224	151	141	189

Jakiej strony używa Free Pascal? – na tym etapie nauki można powiedzieć, że żadnej. Jeśli wygenerowany program zażąda wypisania bajtu, to `cmd.exe` wypisze znak zaczerpnięty ze średniowiecznego CP-852. Jeśli zapiszemy w edytorze, że nasz `znak:='Ś'`, to wartość binarną wybierze dla niego edytor. Czyli jeśli piszemy

w DOSopodobnym IDE Free Pascala, będzie ona równa 151, a w WFP, które używa windowsowej strony kodowej – 140.

Odwzorowanie bajt-litera w używanym środowisku da pętla o treści:

```
for i:=0 to 255 do
  Writeln( i, '␣', chr(i) );
```

* * *

Patrząc na powyższą tabelkę, można by zastanowić się nad napisanie programu do konwersji tekstu z jednej strony do drugiej. Czytałby plik znak po znaku i gdy natrafiłby na wypisany w tabelce bajt, w jego miejsce wstawiałby inny. Cytując Marlenkę: Jest to jakiś pomysł, ale nie wiem jaki. Program taki nie uwzględniłby wszystkich znaków mogących wystąpić w plikach tekstowych (wspominałem choćby o cudzysłowach) – pozostaje pytanie, co miałby zrobić, gdyby trafił na coś, co nie jest literą z tabelki.

Unikod

Wadą stron kodowych w których 1 bajt to 1 znak, było ograniczenie liczby możliwych znaków do 256. A właściwie to jeszcze mniej, bo 32 pierwsze bajty przedstawiają sobą tzw. kody sterujące (między innymi znak końca linii, znak tabulatora i parę innych). A co, jeśli chciało się w jednym dokumencie zapisać teksty w języku polskim, francuskim i jeszcze dołożyć cyrylicę? Na takie przypadki wymyślono tzw. unikod.

> Będę używał spolszczonej nazwy, pomimo, że powszechnie pisze się „unicode". No właśnie – jak to wygląda: angielski wyraz z polskich cudzysłowach?

Nazwa jest myląca, bo sposobów na odwzorowanie: znak – zawartość binarna, jest kilka, większość z nich w dwóch wersjach. Dlaczego więc nazwa jest skrótem od „unikalny sposób na kodowanie"? Bo unikalne jest odwzorowanie: znak – numer znaku. Natomiast sposobów na zapisanie binarne znaku o danym numerze jest, jak napisałem, kilkanaście. Pod Windowsami będą stosowane zazwyczaj dwa:

- UTF-8 – w pewnym sensie rozwinięcie kodowania ASCII. Ponieważ numeracja dla kodów ASII poniżej 128 pokrywa się z numeracją unikodu, znaki te będą reprezentowane przez 1 bajt, taki sam jak w kodowaniu ASCII. W innych przypadkach potrzeba ich więcej – np. dla polskich znaków diakrytycznych będą to dwa bajty, a dla innych języków jeszcze więcej. Jest to domyślny sposób zapisu plików tekstowych dla notatnika Windowsa 10 i wyżej. Jest to to też powszechnie stosowany sposób kodowania stron internetowych.

- UTF-16 – „windowsowe”. Na każdy znak przypadają zwykle dwa bajty, ale bardziej skomplikowane symbole (np. hieroglify starożytnych Egipcjan), będą zlepkiem kilku dwubajtowych paczek. UTF-16 to domyślny sposób zapisu tekstu wyświetlanego przez aplikacje windowsowe. Czyli: jeśli gdzieś widzisz tekst na belce uruchomionego programu (nazwa programu czy nazwa dokumentu) to będzie on pewnie zapisany w tym właśnie formacie.

I znów pytanie – jak się ma do tego Free Pascal? Jeśli chodzi o UTF-8 możemy sprawdzić, bo edytor ze strony `https://www.onlinegdb.com/online_pascal_compiler` używa właśnie tego kodowania:

```
napis := 'Liść';
Writeln( napis, ' ma długość: ', length( napis ) );
for i:=1 to length( napis ) do
  Writeln( napis[i], ' ', ord(napis[i]) );
```

Dowiemy się z tego programu jak to czteroliterowy wyraz zapisany jest przez 6 bajtów ☺. Dowiemy się też, że próba potraktowania bajtu o kodzie większym od 127, jako znaku UTF-8 po prostu się nie uda.

Natomiast dla danych zapisanych w formacie UTF-16, potrzebne są poważniejsze narzędzia niż tylko standardowe znaki i napisy.

Typ `char`

Bardzo spokrewniony z bajtem. Można rzutować jeden na drugi:

```
znak := char( bajt );
bajt := byte( znak );
```

Na oba działa funkcja `ord()`, zwracająca wartość liczbową.

Gdy chcemy użyć jawnej wartości typu char, piszemy znak w apostrofach:

```
znak := 'A';
```

Nie wszystkie znaki da się tak napisać, więc przewidziano możliwość nadawania wartości binarnej:

```
znak := #32
```

co jest równoznaczne z użyciem `' '` (spacja), gdyż bajt 32 wypisany jako char oznacza właśnie spację. Przy okazji – warto wypisać sobie te wartości jako stałe, bo niekoniecznie będziemy pamiętać numerację znaków:

```
const
  zerowy = #0;
```

```
  enter = #13;
  tab = #9;
```

Czasami przydaje się też funkcja `UpCase()`, która zwraca dużą literę. Bo być może zamiast kodu:

```
if (wybor='T') or (wybor='t') then ...
```

lepiej zastosować:

```
wybor := UpCase( wybor ):
if wybor = 'T' then ...
```

`UpCase()` działa też na napisy, ale nie obejmuje innych znaków niż te klawiaturowe – nie zamieni `'ą'` na `'Ą'`, niezależnie od strony kodowej ☺.

Typ `string`

W zasadzie typ `string` zachowuje się jak `array[0..255] of char`. Stąd nie będzie można przeczytać linii dłuższych od 255. W zerowym elemencie przechowuje się informację o zajętości tablicy – czyli o długości przechowywanego napisu. Nie musimy z tego korzystać, bo informację tę możemy uzyskać za pomocą funkcji `length()`.

Przykładowy kod posługujący się zmienną typu `string`:

```
for i:=1 to length( linia ) do
  if linia[i]='.' then linia[i]:=','
```

Jak widać, kod ten zmienia wszystkie kropki na przecinki (można sobie wyobrazić, że tworzymy dane liczbowe, które zostaną zaimportowane do Excela, gdzie część ułamkowa jest oddzielana przecinkiem, więc przykład nie jest tak całkiem bezsensowny).

Warto zaznajomić się kilkoma innymi funkcjami i procedurami działającymi na zmiennych typu string.

- `trim()` – obcina z boków napisu tzw. białe znaki, najczęściej są to spacje.

```
linia := '␣␣Jan␣Kowalski␣';
linia := trim( linia );   // 'Jan␣Kowalski'
```

- `val()` – zamienia napis na wartość liczbową.

```
val( linia, liczba, err );
```

Wynik konwersji napisu linia, umieszczany jest w zmiennej liczba. Wartość błędu `err` wskazuje w którym miejscu postać napisu linia nie pozwala na konwersję. Jeśli konwersja się udała, `err` ma wartość 0.

- `str()` – działanie odwrotne do `val()`, czyli zamiana liczny na odpowiadający jej napis:

```
str( liczba, linia );
```

Obie procedury obsługują dowolne typy numeryczne.

- `pos()` – zwraca pozycję pierwszego wystąpienia próbki w napisie:

```
linia := 'Ala␣ma␣kota';
pozycja := pos( 'ma', linia ); // 5
```

Wartość 0 oznacza, że nie znaleziono próbki.

Inne podstawowe procedury zarządzające typem `string` znajdziemy w dokumentacji Free Pascala.

> Zmienne typu `string` **nie są** inicjowane napisem pustym. Jeśli korzystamy z napisu, który na początku naszych działań ma być pusty, to zainicjujmy go: `napis:=''`.

* * *

Fakt, że każdy napis zabiera 256 bajtów pamięci, może wydawać się marnotrawieniem zasobów komputera, gdyż większość zmiennych tekstowych jest zdecydowanie krótsza. Znając górne oszacowanie rozmiaru napisów, możemy zadeklarować zmienne oszczędniej gospodarujące pamięcią:

```
var
  imie : string[20];
  nazw : string[50];
```

Warto też zastanowić się nad osobnymi typami dla napisów krótszych od 255:

```
type
  PESEL = string[11];
  krotkiStr = string[25];
```

Typ `AnsiString`

Czasami zdarza się konieczność przechowywania napisów dłuższych od 255. Dla takich razów przygotowano o wiele bardziej elastyczny typ `AnsiString`. Posługiwanie się nim jest identyczne jak zwykłym `string`-iem:

```
var napis : AnsiString;
...
napis := 'Ala␣ma';
napis := napis+'␣kota';
for i:=1 to length( napis ) do
  WriteLn( i, '␣', napis[i] );
```

Większość funkcji działających na typie string, będzie zachowywać się poprawnie również dla `AnsiString`. Czym więc różnią się oba typy? „Zwykły" string to 256-znakowa tablica – tyle potrzebuje miejsca dla każdej zmiennej, niezależnie od długości napisu. Zmienna typu `AnsiString`, to ukryty wskaźnik – nie przechowuje treści napisu, ale ma zapisane gdzie (w którym miejscu pamięci komputera) znajduje się tekst napisu. Sprawdźmy, że `sizeof(napis)` jest równy 4. Przy używaniu takiej zmiennej, program rezerwuje tyle pamięci, ile akurat jest potrzebne (+ kilka bajtów na dodatkowe dane, np. długość napisu). Szczegóły implementacyjne są jednak przed programistą ukryte.

Z punktu widzenia programisty oba typy prawie się nie różnią. Można nawet bezpiecznie kopiować napisy obu typów:

```
var
  zwykly : string;
  napis  : AnsiString;
...
  zwykly := napis;
...
  napis := zwykly;
```

Z praktycznego punktu widzenia różnica pojawia się w momencie, gdy zapisujemy `napis` do plików binarnych (jednorodnych lub amorficznych) – zamiast treści zapisze się tzw. adres, czyli wspomniane miejsce w pamięci komputera, w postaci 4-bajtowej liczby. Więcej informacji na temat plików w następnym odcinku.

> Można wyobrazić sobie trudny do znalezienia błąd, polegający na zapisie binarnej wartości zmiennej typu `AnsiString` i ponownym jej odczycie. Zmienna będzie działała poprawnie, jeśli w trakcie wykonywania programu lokalizacja napisu się nie zmieni. Jednak w przypadku gdy program zostanie zamknięty, a wartość zostanie odczytana przy powtórnym jego uruchomieniu, zmienna pokaże przypadkowe dane (jeśli nie wystąpi błąd braku dostępu do pamięci), gdyż adres z poprzedniego uruchomienia będzie już nieaktualny.

Typ `Pchar`

Wzorowany na języku C, w którym napisy były realizowane przez wskaźniki na typ `char`, a koniec napisu wyznaczało pierwsze wystąpienie bajtu zerowego. Jeśli chodzi o zarządzanie pamięcią, to bardzo podobny typ do poprzedniego – różnica polega na tym, że `AnsiString` może posiadać wiele bajtów zerowych, ponieważ jego długość jest przechowywana jako liczba.

```
var
  napis : AnsiString;
  stylC : Pchar;
...
  napis := 'pierwsza␣część'+#0+'druga␣część';
  stylC := Pchar( napis );
  WriteLn( stylC );
```

Powyższy kod utworzy nowy napis (skopiuje), który będzie zarządzany przez zmienną `stylC`, o wartości `pierwsza część`. Kopiowanie treści działa również w drugą stronę:

```
napis := AnsiString( stylC );
```

W obu przypadkach należy korzystać z rzutowania przez konstruktor.

`Pchar` jest o wiele mniej wygodnym typem, bo jest wskaźnikiem:

```
if stylC=nil then stylC := 'początek';
```

ale

```
stylC := stylC + 'ciąg␣dalszy'; //BŁĄD
```

nie zadziała, bo nie można dodać do wskaźnika napisu. Żeby nie dziwić się zbytnio podczas pisania programów, zostawmy typ `Pchar` do momentu lepszego zapoznania się z pojęciem wskaźnika.

2.9 Obsługa plików

Zanim poznamy kilka sposobów na zapis danych do pliku, powiedzmy sobie wprost: W codziennych zastosowaniach zazwyczaj mamy do czynienia z dość specjalizowanymi plikami o skomplikowanych formatach, w których dane zapisywane są w wymyślny sposób: DOCX, XLSX, PDF, JPG, PNG. Żeby utworzyć tego rodzaju plik, należałoby najpierw poznać jego strukturę i oprogramować ją w kodzie swojego programu. Podejrzewam, że jest to zadanie niewiele odbiegające od poziomu pracy inżynierskiej, i nawet jeśli się mylę, to i tak słabo nadaje się dla amatora. Zresztą dziś nie zapisuje się danych w plikach, ale raczej w bazach danych.

Ale, ale! Zabawa nie byłaby zabawą, gdybyśmy nie wygenerowali sobie jakiegoś fajnego formatu. Wydaje się, że najprostszy z nich to HTML. Zanim jednak to zrobimy, mały przegląd czego możemy spodziewać się po pascalu.

Ogólny sposób działania na plikach

Plik jako byt fizyczny dla programu jest dostępny przez swoją nazwę – w poniższych przykładach przechowywana w zmiennej `sciezka` (typ `string`). Dla programisty wszelkie operacje na pliku, będą wykonywać za pomocą zmiennej typu plikowego – poniżej nazwa tej zmiennej to `plik`. Na początku wszelkich działań, należy połączyć zmienną plikową ze ścieżką na dysku. Służy do tego procedura `AsSign()`:

```
AsSign( plik, sciezka );
```

Tworzenie fizycznego pliku odbywa się za pomocą:

```
ReWrite( plik );
```

Natomiast otwarcie istniejącego pliku, bez jego kasowania zapewni:

```
ReSet( plik );
```

Za odczytywanie i zapis odpowiedzialne są znane już procedury `Write()` i `Read()`. Z tymi operacjami wiąże się pojęcie czytnika pliku – inaczej to pozycja w której będą odbywać się operacje na pliku: czytnik ustawiony na początku, będzie działał na pierwszym elemencie. Jeśli chodzi o pozycję czytnika, to najważniejsza jest funkcja `eof()`, która zwraca prawdę, jeśli czytnik jest na końcu pliku i dalej nie ma żadnych elementów.

Gdy skończymy operacje na pliku, należy go zamknąć:

```
Close( plik );
```

Dopiero ten krok porządkuje wszelkie sprawy związane z plikiem – m.in. dopiero wtedy program poleci systemowi operacyjnemu zapis do pliku ostatniej porcji danych. Nie wolno więc o nim zapominać, bo w przeciwnym razie może okazać się, że przed zamknięciem, program nie zdążył przekazać danych do zapisu i dostaniemy niekompletny plik!

To tak z grubsza, bo szczegóły różnią się dla różnych typów plików.

Pliki jednorodne

Kiedyś istnienie plików jednorodnych zapewniało super-wygodną obsługę przetwarzania danych. Dziś ich rolę przejęły bazy danych, ale dla porządku zobaczmy jak to wygląda. Otóż ze względu na to, że (prawie) każdy typ w pascalu

ma ściśle określony rozmiar zmiennej, można zapisywać/odczytywać do/z pliku dane porcjami. Wyobraźmy sobie, że mamy do przechowania na dysku dane osób zawarte w rekordach typu `Osoba`. Wtedy nie musimy się rozdrabniać i osobno zapisywać: imię, potem nazwisko itd. Zapisujemy cały rekord na raz. Podobnie jest z odczytywaniem. Niech więc:

```
type Osoba = record
     ...
end;
```

Pliki przechowujące dane tego właśnie typu będą zadeklarowane jako:

```
var  plikDATA : file of Osoba;
```

Po wywołaniu `AsSign()`, mamy dwa sposoby na otwarcie pliku: `ReWrite()` tworzy plik i kasuje wcześniejszą wersję, jeśli takowa istniała. Natomiast `ReSet()` otwiera plik i ustawia tzw. czytnik pliku na jego początku (pozycja 0). Od tej pory możemy odczytywać dane, jak również je zapisywać.

Procedura `Seek()` przesuwa czytnik na daną pozycję. Komenda:

```
Seek( plikDATA, 3 );
```

ustawi czytnik ZA trzecim rekordem. Wtedy procedura

```
Read( plikDATA, osoba );
```

przeczyta czwarty rekord z pliku i przesunie czytnik przed rekord piąty. Analogicznie przy zapisywaniu, procedura

```
Write( plikDATA, osoba );
```

nadpisze czwarty rekord i też przesunie czytnik o jeden rekord dalej. Jak widać można „jeździć” czytnikiem po zawartości pliku i swobodnie odczytywać i zapisywać.

Gdy ustawimy czytnik na końcu pliku (za ostatnim rekordem), będzie można dopisywać dane. Kod który przestawia czytnik na koniec, bez konieczności czytania całego pliku, może wyglądać następująco:

```
poz := 0;
while not eof( plikDATA ) do
begin
  inc( poz );
  seek( plikDATA, poz )
end;
```

Do kompletu podam jeszcze funkcję, zwracającą aktualną pozycję czytnika:

```
poz := FilePos( plikDATA );
```

Uwaga: Przy zapisie zmiennych typu `Pchar` lub `AnsiString`, do pliku zapiszą się jedynie adresy (≈ numery komórek pamięci komputera, w których aktualnie przechowywane są napisy), a nie treść napisów. Jeśli wydaje ci się to niesprawiedliwe, to pomyśl: zapisywane zmienne/ rekordy muszą mieć te same rozmiary, a `AnsiString` może mieć dowolny rozmiar. Jak to pogodzić? Nie da się – skoro musimy zapisywać dane tego typu, można użyć plików tekstowych, gdzie każda linia może mieć również dowolny rozmiar.

* * *

Trochę się rozpisałem o plikach jednorodnych, bo pomimo swego anachronizmu w dziedzinie zastosowań praktycznych, stanowią bardzo wygodne i miłe wyjście, gdy amator chce zapisać swoje dane w pliku.

Pliki amorficzne

To takie pliki, które zawierają „jakieś bajty". Tu cała odpowiedzialność, żeby poprawnie zapisać i odczytać dane, spada na programistę. Są trochę podobne plików jednorodnych – dane zapisywane są również w postaci binarnej. W pewnym sensie można by o nich pomyśleć, że to typ `file od byte`.

Przy otwieraniu pliku należy zadeklarować rozmiar tzw. bloku. Zapisywane-/odczytywane wielkości muszą wtedy mieć rozmiar będący wielokrotnością bloku. Zdaje się, że dość wygodnie przyjąć go jako 1 – wtedy w operacjach we/wy będziemy się posługiwać rozmiarem w bajtach.

```
AsSign( plikBIN, sciezka);
ReWrite( plikBIN, 1 ) // rozmiar bloku 1
```

albo – jeśli plik istnieje i nie chcemy go kasować:

```
ReSet( plikBIN, 1 );
```

Zapis i odczyt jest dokonywany przez funkcje `BlockWrite()` i `BlockRead()`, które wymaga podania czterech argumentów. Będą to oprócz zmiennej plikowej: zmienna z danymi, liczba bloków planowana do zapisania. Do ostatniego argumentu procedura wpisze ile rzeczywiście bloków zapisano:

```
BlockWrite( plikBIN, liczba, sizeof( liczba ), ile );
```

w powyższym przykładzie będzie ona równa rozmiarowi zapisanej liczby. Odczyt wygląda podobnie:

```
BlockRead( plikBIN, liczba, sizeof(liczba ), ile );
```

Na koniec nie zapomnijmy o zamknięciu pliku za pomocą `Close()`.

Pliki amorficzne otwierane są w trybie wielodostępu – procedura `Seek()` przesuwa czytnik o wielokrotność rozmiaru bloku. Rozmiar bloku może być dość dowolny – to liczba dwubajtowa. Można czytać czy zapisywać dane w większych paczkach od pojedynczej. Używa się wtedy tablic o stałym rozmiarze (nie dynamicznych). Kod do samodzielnej analizy (`tab` jest 10-elementową tablicą przechowującą liczby typu `integer`):

```
AsSign( plik, 'dane.dat' );
ReWrite( plik, sizeof(integer) );
for i:=1 to 10 do
  BlockWrite( plik, i, 1, ile );
Seek( plik, 0 );
BlockRead( plik, tab, 10, ile );
Close( plik );
```

Pliki tekstowe

Zawierają teksty czy liczby w takiej postaci, jakby były wypisywane na ekranie. Użyteczne dla przechowywania danych, do których będzie zaglądał człowiek. Czytanie z takiego pliku odbywa się tak samo, jak odczyt z klawiatury, a zapis wygląda tak samo, jak wypisanie na ekranie. Prawdopodobnie na początku swojej kariery programistycznej, będziesz potrzebował właśnie takiego rodzaju plików.

Deklaracja zmiennej plikowej pliku tekstowego jest następująca:

```
var plikTXT Text;
```

Po użyciu `AsSign()`, wiążącego zmienną ze ścieżką na dysku, możemy taki plik otworzyć do odczytu:

```
ReSet( plikTXT );
```

albo do zapisu od początku (kasujemy poprzedni jeśli taki istniał):

```
ReWrite( plikTXT );
```

albo do zapisu na końcu:

```
Append( plikTXT );
```

Ten ostatni sposób pozwala na zachowanie dotychczasowej treści.

Jak napisałem, posługiwanie się takimi plikami przypomina działania na konsoli, dlatego oprócz procedur `Read()` i `Write()`, możemy stosować `WriteLn()` i `ReadLn()`. Nie ma natomiast ustawiania czytnika, bo każda z linii ma inny rozmiar.

Zobaczmy jak wygląda odczyt całego pliku – tutaj czytamy linia po linii i wypisujemy na konsolę, zakładając, że żadna z linii w pliku nie przekracza 255 znaków:

```
AsSign( plikTXT, 'dane.txt' );
ReSet( plikTXT );
repeat
  ReadLn( plikTXT, linia );
  WriteLn( linia );
until eof( plikTXT );
Close( plikTXT );
```

Zwróćmy uwagę na warunek przerwania pętli `repeat` – funkcja `eof()` zwróci prawdę, gdy dojdziemy do końca pliku.

* * *

Jest jeszcze jedna ważna sprawa związana z plikami tekstowymi. Otóż ze względu na zaszłości historyczne, koniec linii zapisywany jest różnie w różnych systemach operacyjnych. Windows i większość działających na nim programów uznają, że oznaczenie końca linii składa się z dwóch znaków: bajtu `#13` (CR – powrót karetki) i `#10` (LF – koniec linii). Domyślamy się, że nazewnictwo pochodzi z czasów, gdy komputery wypisywały dane nie na konsoli, ale na drukarkach. Jak to jest w innych systemach możemy przeczytać choćby w Wikipedii.

Większość programów przeznaczone do obróbki plików tekstowych domyślnie wpisuje te dwa bajty, gdy użytkownik chce zakończyć linię. Tak też działają programy skompilowane we Free Pascalu na Windows. Gdybyśmy jednak z jakiegoś powodu, zamiast używać `WriteLn()`, chcieli wpisać do pliku tekstowego znak `#13` albo `#10`, to program rzeczywiście zapisze pojedynczy bajt. Wtedy interpretacja takiego „zakończenia linii” zależy od programu, który otworzy plik.

Błędy dostępu do pliku

Programy pascalowe w ustawieniach domyślnych zwykle przerywają swoje działanie, gdy pojawi się błąd związany z plikiem. Mówimy, że jest to błąd wykonania – występuje w czasie uruchomienia programu. To bardzo kłopotliwa cecha, bo przecież programista nie ma kontroli nad tym, jak będzie uruchamiany jego program.

Pewnym rozwiązaniem jest sprawdzenie czy plik istnieje, jeśli chcemy użyć `ReSet()`:

```
if FileExists( sciezka )
```

```
begin
  AsSing( plik, sciezka );
  ReSet( plik );
  ...
end;
```

Funkcja `FileExists()` pomaga ominąć jedno źródło błędu, ale pozostaje wiele innych: możemy na przykład próbować otworzyć plik, gdy jest on zajęty przez inny program albo znajduje się na CD, na którym nie można zapisywać. W takich przypadkach zmienia się tryb wykonania programu i przechwytuje wygenerowane błędy (jeśli powstaną) bez jego przerywania. Sposobem na „znieczulenie" programu na błąd jest ustawienie opcji kompilatora `{$I-}` w kodzie. Przestawienie programu w tryb domyślny odbywa się natomiast za pomocą `{$I+}`:

```
{ będziemy próbowali otworzyć plik, którego nie ma na dysku }
AsSign( plik, 'niematakiegopliku.txt' );
{ Wyłączenie kontroli błędów }
{$I-}
ReSet( plik ); { program działa dalej }
err := IOResult;
{ Włączenie kontroli błędów }
{$I+}
if err <> 0 then
  {jakaś reakcja na wystąpienie błędu}
```

Powstaje pytanie – Skoro `Write()` czy `Read()` też mogą generować podobne błędy, może dla nich też wyłączyć kontrolę? Niekoniecznie. Wyłączenie kontroli zmienia tryb wykonywania programu: każde możliwe wystąpienie błędu powinno być uwzględnione. Czyli zamiast prostego kodu:

```
for i:=1 to 10 do
  Write( plik, i );
```

musielibyśmy napisać:

```
for i:=1 to 10 do
begin
  Write( plik, i );
  err := IOResult;
  if err<>0
  begin
    WriteLn( 'Wystąpił␣błąd␣-␣nie␣udał␣się␣zapisać␣liczby' );
    break; { kończymy pętlę }
  end
end;
{ i pewnie trzeba jakoś inaczej zaplanować resztę programu }
```

Jak widać decyzja czy obsługiwać błędy czy nie, nie jest oczywista. Podejmujemy ją uwzględniając prawdopodobieństwo wystąpienia błędu. Być może skoro plik został prawidłowo otwarty, to umożliwi od resztę działań, bez konieczności sprawdzania ich poprawności.

Warto spojrzeć na listę wszystkich błędów wykonania (tzw. run-time errors) – nie tylko tych związanych z dyskiem. Informacja, jakie rodzaje błędów mogą się pojawić, może być użyteczna.

Plik HTML

To bodaj najłatwiejszy, ale już sprawiający dużo radości format pliku tekstowego. Opiszę poniżej jedynie najprostsze stosowane formatowania, wystarczające jednak do wygenerowania ładnego dokumentu, który bez wstydu będzie można wydrukować, czy nawet dalej edytować w Wordzie.

Jak wyglądają pliki HTML? Z grubsza (w najprostszej postaci) zawierają teksty z komendami, które mówią jak te teksty powinny być sformatowane. Komendy (zwane tagami) zamknięte są w znaki mniejszości i większości: `<komenda>`. Zwykle (choć nie zawsze), komenda otwierająca ma również swój odpowiednik zamykający różniący się użyciem znaku dzielenia: `</komenda>`. Jako przykład można podać komendę wypisującą tytuł widoczny w górnym pasku przeglądarki:

```
<title>Bardzo ważna strona</title>
```

Treść takiego pliku zaczyna się od napisania nagłówka (preambuły). Mieszczą się w nim informacje istotne dla wyglądu, skrypty wykorzystywane na stronie itp. Nagłówek zamknięty jest znacznikami: `<head> ...</head>`. Dla nas istotne będą trzy komendy: tytuł, wskazanie pliku ze stylami (o nim więcej niżej) i zapis określający użyte kodowanie znaków. Oto minimalistyczny początek:

```
<html>
<head>
<title>Bardzo ważna strona</title>
<meta http-equiv="Content-Type" content="text/html; charset=windows-1250">
<link rel="stylesheet" href="formatowania.css">
</head>
...
```

Linijka zawierająca tag `<meta ...>` zawiera deklarację, że zastosowana jest strona kodowa CP-1250. Jest to naturalny wybór, jeśli będziemy działać w systemie Windows i tworzyć tekst zgodny z tym kodowaniem, a wciąż jest to chyba dość częsty sposób generowania plików tekstowych. Zdaje się jednak, że już w niedalekiej przyszłości format UTF8 stanie się powszechny. Dla takiego formatowania parametr `charset` przyjmie wartość `utf-8`:

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
```

Uwaga: Używanie edytora z IDE Free Pascala do tworzenia napisów, zapisywanych później do plików HTML, skazuje nas niejako na ustawienie strony kodowej: `charset=cp852` Przeglądarka zrozumie i wyświetli poprawnie, ale wstyd to będzie gdzieś pokazać. A bez żartów – w bardziej praktycznych sytuacjach, albo opakowujemy tagami HTMLowymi teksty pobrane i utworzone poza naszym programem, albo używamy programów okienkowych – wspomniany wcześniej Lazarus stosuje domyślnie UTF8.

Kolejna linia zawiera odwołanie do pliku `formatowania.css` – zawiera on informacje, jak powinno wyglądać formatowanie treści zawartych w naszym pliku. Można co prawda ustawiać rodzaj czcionki, jej wielkość i inne parametry bezpośrednio w kodzie HTML, ale zdecydowanie lepiej użyć pliku ze stylami – przykład zobaczymy poniżej. W powyższym przykładzie plik `formatowania.css` znajduje się w tym samym katalogu co plik HTML.

Właściwa treść strony (ta widoczna w przeglądarce) zamknięta jest komendami `<body>...</body>`. Pomiędzy nie wpisujemy tekst – jego charakter określają kolejne tagi. Zobaczmy prosty przykład:

```
...
<body>
<h1>Tytuł</h1>
<h2>Podtytuł</h2>
<p>Zwykły akapit
zawierający                napis Ala ma kota.
</p>
...
```

Tagi `<h1>...</h1>` i `<h2>...</h2>` oznaczają nagłówki. Można stosować kolejne nagłówki kolejnych stopni. W tagach `<p>...</p>` umieszczamy zwykły tekst (akapit). Zwróćmy uwagę, że przeglądarka ignoruje nadmiarowe białe znaki – spacje, tabulatory czy znaki końca linii – widzimy je co najwyżej jako pojedyncze odstępy.

W ramach formatowania tekstu możemy zmieniać krój czcionki dla fragmentu napisu:

- `<b>...</b>`, lub `<strong>...</strong>` – pogrubienie;
- `<i>...</i>` lub `<em>...</em>` – pochylenie;
- `<tt>...</tt>` lub `<code>...</code>` – czcionka „maszynowa”, stosowana m.in. do cytowania kodu.

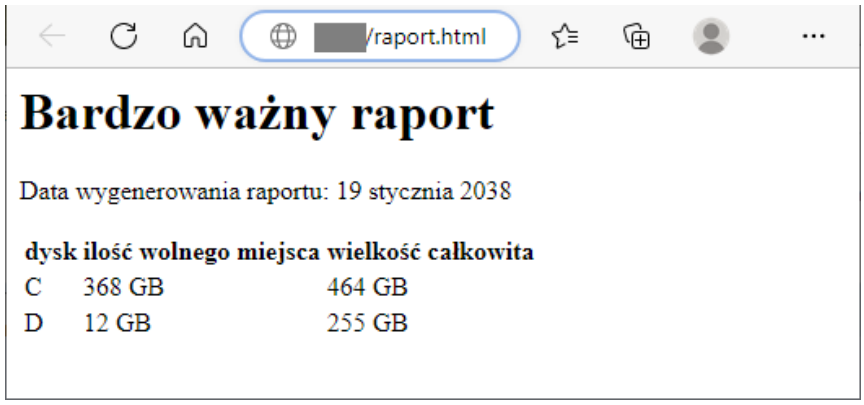
Formatowanie to trudno uwzględnić w programie, który będziemy pisać, więc wspominam o tym tylko dla porządku.

Warto wspomnieć jeszcze o tabelkach. Są one idealnym narzędziem do ładnego wyświetlania danych. Każda tabela (`<table>...</table>`) zawiera linie (`<tr>...</tr>`), które składają się z komórek (`<td>...</td>`). Jeśli pierwsza linia ma się wyróżniać, używamy tagu `<th>...</th>`:

```
<table>
<tr>
<th>dysk</th>
<th>ilość wolnego miejsca</th>
<th>wielkość całkowita</th>
</tr>
<tr>
<td>C</td> <td>368 GB</td> <td>464 GB</td>
</tr>
<tr>
<td>D</td> <td>12 GB</td> <td>255 GB</td>
</tr>
</table>
```

Realizacja tego kodu w przeglądarce wygląda… kiepsko. Żeby ją zmienić, należałoby wpisać do osobnego pliku style, jakie ma zastosować przeglądarka. Nie jest to proste zadanie, ale warto sobie naszykować taki zestaw. Pisze się go używając języka CSS (ang. Cascading Style Sheets czyli kaskadowe arkusze stylów).

Przede wszystkim trzeba sobie powiedzieć, że programowanie w HTML i CSS to dziś obszerna dziedzina wiedzy. Żeby mieć jako-takie rozeznanie, można zajrzeć do samouczka CSS, gdzie można samodzielnie poeksperymentować. Dla przykładu podaję w miarę prosty plik styli oraz wygląd pliku po jego zastosowaniu.

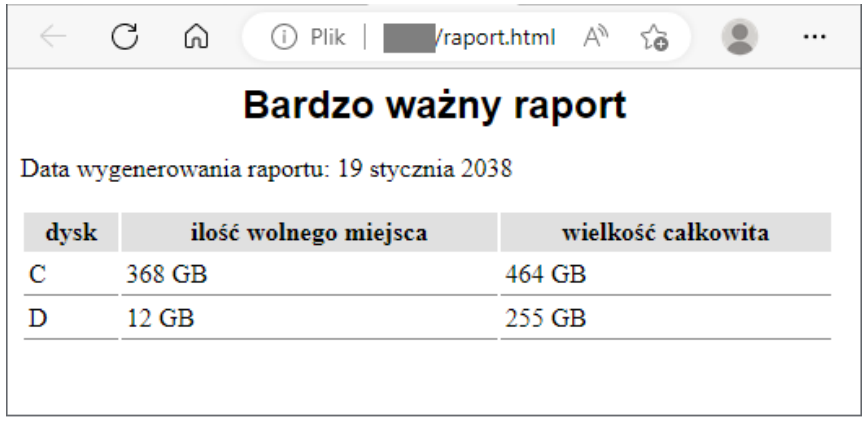



```
h1
{
  text-align:center;
  font-family: Helvetica, Verdana, Arial;
  font-size: 24px;
}
```

```
table
{
  align:center;
  width:500px;
}

th
{
  background-color:#e0e0e0;
  padding: 3px;
}

td
{
  padding: 3px;
  border-bottom: 1px solid #888888;
  empty-cells: show;
}
```



* * *

Skoro już dowiedzieliśmy się co nieco o HTML, to pora na kilka porad, jak takowy stworzyć za pomocą programu. Najprościej rzecz biorąc, trzeba by zapisać kolejne komendy do pliku:

```
WriteLn( plik, '<html>' );
WriteLn( plik, '<head>' );
WriteLn( plik, '<title>Bardzo␣ważna␣strona</title>');
...
WriteLn( plik, '<h1>Tytuł</h1>');
```

Wizja każdorazowego pisania tego kodu w każdym programie, który tworzył będzie HTML nie nastraja optymistycznie. Można jednak napisać sobie szereg podprogramów, które napisane raz, będziemy wykorzystywać wiele razy.

```
procedure PoczHTML( var plik: Text; tytul, style : string );
```

```
begin
  WriteLn( plik, '<html>' );
  WriteLn( plik, '<head>' );
  WriteLn( plik, '<title>', tytul, '</title>' );
  ...

procedure Naglowek1( var plik: Text; tresc : string );
begin
  WriteLn( plik, '<h1>', tresc, '</h1>' );
end;
```

W takim przypadku plik tworzony jest w programie głównym/ procedurze nadrzędnej:

```
AsSign( plik, 'raport.html' );
ReWrite( plik );
PoczHTML( plik, 'Bardzo␣ważna␣strona', 'formatowania.css' );
...
Naglowek1( plik, 'Tytuł␣raportu' );
...
```

Tu ważna uwaga: zmienna plikowa może być argumentem podprogramu tylko z użyciem `var` (dostęp do zmiennej).

Można się też pokusić o napisanie podprogramów, które będą tworzyć napisy, bez konieczności kontaktu ze zmienną plikową:

```
function Naglowek1( tresc : string ) : string;
begin
  Naglowek1 := '<h1>'+tresc+'</h1>';
end;
```

i wtedy w programie nadrzędnym:

```
Writeln( plik, Naglowek1( 'Tytuł␣raportu' ) );
```

albo w dwóch krokach:

```
napis := Naglowek1( 'Tytuł␣raportu' );
Writeln( plik, napis );
```

Jeśli zdecydujemy się na takie rozwiązanie, to należałoby się zastanowić, czy funkcje nie powinny zwracać typu `AnsiString`, bo generowane napisy, mogą być dłuższe niż 255 znaków. Akurat napis tworzony przy zakładaniu pliku, zawierający trzy komendy (tytuł. kodowanie, style) miał ok. 200 bajtów, więc jeszcze mieścił się w rozmiarze typu `string`. Niemniej może pojawić się konieczność opakowywania w tag akapitu dłuższego tekstu. Sprawa do zastanowienia.

Kończąc ten przydługi odcinek, wypiszę jeszcze jakie procedury/funkcje warto sobie napisać, by zautomatyzować tworzenie plików HTML z poziomu programu:

- początek pliku (między <html> a <body>) – częściowo zrobione powyżej;
- serię nagłówków – dla taga <h1> już robione;
- akapit;
- początek tabeli – wypisanie <table>;
- nagłówek tabeli – argumentem ma być tablica z napisami, każdy z nich zamknięty w tagu <th>...</th> (komórka nagłówka), całość zamknięta w <tr>...</tr>;
- linia tabeli – analogicznie jak powyższa, tylko dla komórek użyty będzie tag <td>...</td>;
- koniec tabeli – wypisanie </table>;
- koniec pliku HTML – wypisanie </body></html>.

Zauważmy, że nagłówki i akapit tworzone są identycznie, różnią się tylko tagiem. Być może warto napisać sparametryzowaną wersję funkcji, która wykorzystywana byłaby we wspomnianych procedurach/funkcjach:

```
function naglowek1( tresc string ) : string;
begin
  naglowek := ogolnyTAG( tresc, 'h1' );
end;
```

Napisane funkcje powinny pozwolić nam uruchomienie kodu (jeśli zdecydujemy się na przekazanie zmiennej plikowej do procedur):

```
var ...
BEGIN
  AsSign( plikHTML, 'raport.html' );
  ReWrite( plikHTML );
  PoczHTML( plikHTML, 'Bardzo ważna strona','formatowanie.css' );
  naglowek1( plikHTML, 'Raport' );
  akapit( plikHTML, 'Poniższa tabela zawiera ważne dane' );
  napisy[1] := 'nr';
  napisy[2] := 'imie';
  napisy[3] := 'nazw';
  PoczTabeli( plikHTML );
  NaglTabeli( plikHTML, napisy );
  napisy[1] := '1';
  napisy[2] := 'Jan';
  napisy[3] := 'Kowalski';
  LiniaTab( plikHTML, napisy );
  KonTabeli( plikHTML );
  KonHTML( plikHTML );
END.
```

Co ciekawe: powyższy program główny nie wykonuje konkretnego zadania, ale testuje napisane przez nas funkcje i procedury (m.in. dlatego wypisywana jest tylko jedna linijka tabeli). Jeśli testowanie zakończy się pozytywnie (=znajdziemy i usuniemy błędy w kodzie), będziemy mogli je wykorzystać „produkcyjnie" w innych programach.

* * *

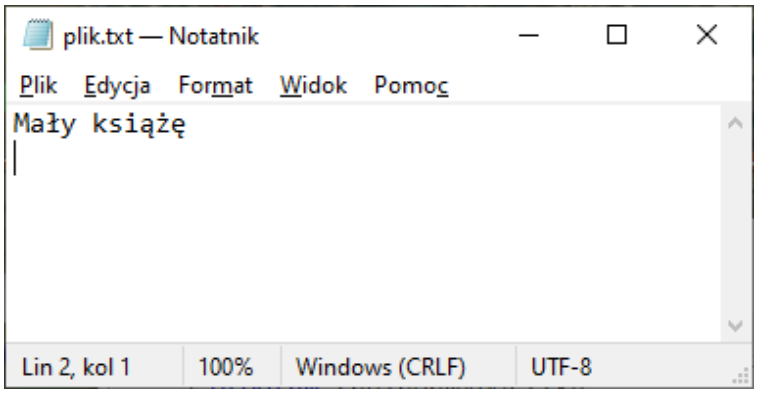
Koniecznie należy wspomnieć, że w nowoczesnym pascalu istnieje jeszcze jeden typ plikowy, a właściwie strumieniowy, podobny do strumieni występujących w C++ czy javie. Ale ponieważ bazuje on na własnościach obiektowych o których do tej pory nic nie napisałem – przenoszę ten materiał do następnych modułów.

2.10 Kilka fajnych zadań z plikami

Ponieważ w poprzedniej części rozpisałem się o teorii na temat plików, przenoszę zadania do niniejszego odcinka. Będę korzystał z napisanych już procedur i funkcji dotyczących formatu HTML.

Zawartość binarna plików

Skoro wiemy, że różne pliki mają różne formaty, to może zajrzyjmy do nich, żeby się dowiedzieć jakie bajty zawierają.



Na początek utworzymy sobie plik tekstowy w Notatniku (jak na obrazku) i będziemy go podglądać. Przeczytamy z niego znak po znaku – dlatego zdefiniujemy sobie zmienną plikową jako `file of char` a nie `Text` – a potem wyświetlimy te znaki wraz z ich numerami ASCII. Ponieważ jednak nie wiemy jaki rozmiar będzie miał nasz podglądany plik, z góry musimy ograniczyć liczbę przeczytanych znaków (poniżej to stała `max`). Na wypadek gdyby było ich mniej przewidziałem zmienną `ile` – w przypadku gdy plik ma `max` lub więcej bajtów, `ile` będzie równe `max`.

Wyświetlanie jest nieco bardziej skomplikowane. Założyłem, że jest sens wypisywać tylko drukowalne znaki ASCII, czyli te pomiędzy #32 a 127. Wszystkie inne będzie reprezentować znak `jakis` (w programie jest to gwiazdka). W dodatku, dla ładniejszego wyglądu, w każdej linii wypiszę taką samą liczbę znaków (stała `wLinii`).

```
program ZagladamyDoPliku;
const
  max = 100;
  wLinii = 8;
  nazwa = 'plik.txt';
  jakis = '*';
var
  znak : char;
  plik : File of char;
  znaki : array[1..max] of char;
  ile, i : integer;
BEGIN
  {otwarcie pliku}
  AsSign( plik, nazwa );
  ReSet( plik );
  {odczytanie początkowych bajtów}
  ile := 0;
  for i:=1 to max do
  begin
    Read( plik, znak );
    znaki[i] := znak;
    inc( ile );
    if eof(plik) then break;
  end;
  {zamknięcie pliku}
  Close( plik );
  {wypisanie zawartości binarnej}
  for i:=1 to ile do
  begin
    znak := znaki[i];
    Write( '|␣');
    if (znak>#31) and (znak<#128) then Write( znak )
    else Write( jakis );
    Write( '␣', integer( znaki[i] ):3, '␣' );
    if (i mod wLinii = 0) or (i=ile) then WriteLn( '|')
  end;
END.
```

Wspomniany plik tekstowy zostanie przedstawiony w następujący sposób:

```
| M  77 | a  97 | * 197 | * 130 | y 121 |    32 | k 107 | s 115 |
| i 105 | * 196 | * 133 | * 197 | * 188 | * 196 | * 153 | *  13 |
| *  10 |
```

Zgodnie z tym co już wiemy, zwykłe znaki są reprezentowane przez jeden bajt, a litery z ogonkami przez dwa. Dodatkowo widzimy, że każdy ENTER w Notatniku przełoży się na dwa bajty: `#13` i `#10`.

Może się jednak zdarzyć, że plik zapisany w kodowaniu UTF8 zawierający napis `Mały ksiażę` po którym następuje znak końca linii, da efekt następujący:

```
| * 239 | * 187 | * 191 | M  77 | a  97 | * 197 | * 130 | y 121 |
|    32 | k 107 | s 115 | i 105 | * 196 | * 133 | * 197 | * 188 |
| * 196 | * 153 | *  13 | *  10 |
```

Widać, że na początku pliku pojawiły się trzy bajty, choć gdybyśmy otworzyli ten plik Notatnikiem, to nie będzie ich widać w edycji. Co oznaczają? Umówiono się, że pliki zapisane w unikodzie, mogą zaczynać się sekwencją dodatkowych dwóch, trzech lub czterech bajtów, zwanych BOM (Byte Order Mark). Między innymi dlatego, żeby podpowiedzieć programowi do edycji z jaką wersją unikodu ma do czynienia.

> Przy korzystaniu z plików tekstowych zapisanych w unikodzie, BOM może sprawić kłopoty: Edytory mogą go nie rozpoznać i wypisywać go w postaci dziwnych znaków. Skrypt napisany w edytorze, który dołożył BOM, może być źle zinterpretowany przez środowisko uruchomieniowe. Program który napiszemy w pascalu nie poradzi sobie z „nadmiarowymi" danymi początkowymi ☺

Patrząc na znaki wypisane w kolumnach, możemy zapragnąć przedstawić je w jeszcze ładniejszej formie. W tym celu przypomnijmy sobie funkcje/procedury do tworzenia plików HTML i utwórzmy tabelkę. Każdy bajt pliku, będzie pokazany w osobnej komórce, zawierającej nr ASCII i sam znak.

Poniżej przedstawiam dwie funkcje – pierwsza z nich ma za zadanie wypisać znak. Osobno znaki drukowalne i reszta. Jak widać niektóre z nich zostały potraktowane specjalnie – cudzysłów, ampersand i porównania, są używane przez HTML i lepiej zapisać je jako komendy HTMLowe. Podobnie wśród znaków sterujących niektóre z nich zostały wyróżnione, reszta będzie reprezentowana jako kwadracik.

```
function OpisZnaku( znak : char ): string;
begin
if (znak>=#32) and (znak<#127) then
  case znak of
    '"' : OpisZnaku := '&quot;';
    '&' : OpisZnaku := '&amp;';
    '<' : OpisZnaku := '&lt;';
    '>' : OpisZnaku := '&gt;'
  else OpisZnaku := znak
  end
else
  case znak of
```

```
    #0  : OpisZnaku := 'NUL';
    #9  : OpisZnaku := 'TAB';
    #10 : OpisZnaku := 'LF';
    #13 : OpisZnaku := 'CR';
  else OpisZnaku := '&square;';
  end;
end;
```

Pomyślałem sobie, że charakter bajtu będzie uwidoczniony przez kolor komórki tabeli. Kody sterujące będą jasnożółte z gradacją (im wyższy numer, tym ciemniejsze pole), znaki drukowalne pokażą się na ciemnożółtym tle (bez gradacji), bajty powyżej 127 na brązowym tle z gradacją. A skoro tła będą jasne lub ciemne, to kolor czcionki musi być kontrastowa, czyli biały lub czarny.

```
function  Komorka( nrAscii : integer; znak : string ) : string;
var
  pocz, numer, skladowa : string;
begin
  { kolor komórki }
  pocz := '<td␣bgcolor="#';
  if nrAscii<32 then
  begin
    pocz := pocz + 'ff';
    pocz := pocz + hexStr(255-nrAscii, 2 )
            + hexStr(127-2*nrAscii, 2 )
  end
  else if nrAscii<128 then
    pocz := pocz + 'ffb200'
  else
  begin
    skladowa := hexStr( 255-nrAscii , 2 );
    pocz := pocz + skladowa + '0000';
  end;
  pocz := pocz + '">';
  { kolor czcionki i uzupełnienie numeru spacjami }
  pocz := pocz + '<font␣color="#';
  if nrAscii<128 then
    pocz := pocz + '000000'
  else
    pocz := pocz + 'ffffff';
  pocz := pocz + '">';
  str( nrAscii, numer );
  if nrAscii<10 then numer := ' '+ numer;
  if nrAscii<100 then numer := numer + ' ';
  Komorka := pocz + znak + '<br>'+ numer +'</font></td>';
end;
```

Jak widać wykorzystałem funkcję `hexStr()`, która przedstawia liczby w notacji szesnastkowej, takiej samej jakiej wymaga HTML. Drugi argument funkcji mówi, jaka ma być długość napisu – puste miejsca są uzupełniane zerami.

Przetestowałem obie funkcje, wypisując kolejne znaki (od 0 do 255) i pakując je w tabelkę:

NUL 0	□ 1	□ 2	□ 3	□ 4	□ 5	□ 6	□ 7	□ 8	TAB 9	LF 10	□ 11	□ 12	CR 13	□ 14	□ 15
□ 16	□ 17	□ 18	□ 19	□ 20	□ 21	□ 22	□ 23	□ 24	□ 25	□ 26	□ 27	□ 28	□ 29	□ 30	□ 31
32	! 33	" 34	# 35	$ 36	% 37	& 38	' 39	(40	) 41	* 42	+ 43	, 44	- 45	. 46	/ 47
0 48	1 49	2 50	3 51	4 52	5 53	6 54	7 55	8 56	9 57	: 58	; 59	< 60	= 61	> 62	? 63
@ 64	A 65	B 66	C 67	D 68	E 69	F 70	G 71	H 72	I 73	J 74	K 75	L 76	M 77	N 78	O 79
P 80	Q 81	R 82	S 83	T 84	U 85	V 86	W 87	X 88	Y 89	Z 90	[91	\ 92	] 93	^ 94	_ 95
` 96	a 97	b 98	c 99	d 100	e 101	f 102	g 103	h 104	i 105	j 106	k 107	l 108	m 109	n 110	o 111
p 112	q 113	r 114	s 115	t 116	u 117	v 118	w 119	x 120	y 121	z 122	{ 123	| 124	} 125	~ 126	□ 127
□ 128	□ 129	□ 130	□ 131	□ 132	□ 133	□ 134	□ 135	□ 136	□ 137	□ 138	□ 139	□ 140	□ 141	□ 142	□ 143
□ 144	□ 145	□ 146	□ 147	□ 148	□ 149	□ 150	□ 151	□ 152	□ 153	□ 154	□ 155	□ 156	□ 157	□ 158	□ 159
□ 160	□ 161	□ 162	□ 163	□ 164	□ 165	□ 166	□ 167	□ 168	□ 169	□ 170	□ 171	□ 172	□ 173	□ 174	□ 175
□ 176	□ 177	□ 178	□ 179	□ 180	□ 181	□ 182	□ 183	□ 184	□ 185	□ 186	□ 187	□ 188	□ 189	□ 190	□ 191
□ 192	□ 193	□ 194	□ 195	□ 196	□ 197	□ 198	□ 199	□ 200	□ 201	□ 202	□ 203	□ 204	□ 205	□ 206	□ 207
□ 208	□ 209	□ 210	□ 211	□ 212	□ 213	□ 214	□ 215	□ 216	□ 217	□ 218	□ 219	□ 220	□ 221	□ 222	□ 223
□ 224	□ 225	□ 226	□ 227	□ 228	□ 229	□ 230	□ 231	□ 232	□ 233	□ 234	□ 235	□ 236	□ 237	□ 238	□ 239
□ 240	□ 241	□ 242	□ 243	□ 244	□ 245	□ 246	□ 247	□ 248	□ 249	□ 250	□ 251	□ 252	□ 253	□ 254	□ 255

Myślę, że wyglądają całkiem dobrze – obramowanie, wyrównanie i czcionki w komórkach i tabeli zrealizowane zostały przez osobny plik ze stylami.

Skoro mamy narzędzie, to aż korci, żeby dla paru różnych rodzajów plików odczytać po kilkadziesiąt początkowych bajtów i przedstawienie ich w podobny sposób. Żeby nie rozpisywać się za bardzo – mam nadzieję, że samodzielne użycie powyższych funkcji nie przekroczy progu umiejętności czytelnika – przejdę od razu do prezentacji. Poniżej pierwsze 64 znaki czterech różnych plików:

Plik tekstowy (program w języku pascal)

p 112	r 114	o 111	g 103	r 114	a 97	m 109	 32	n 110	o 111	w 119	y 121	; 59	CR 13	LF 10	CR 13
LF 10	v 118	a 97	r 114	 32	i 105	 32	: 58	 32	i 105	n 110	t 116	e 101	g 103	e 101	r 114
; 59	CR 13	LF 10	 32	 32	n 110	a 97	p 112	i 105	s 115	 32	: 58	 32	s 115	t 116	r 114
i 105	n 110	g 103	; 59	CR 13	LF 10	B 66	E 69	G 71	I 73	N 78	CR 13	LF 10	 32	 32	n 110

Plik EXE

M 77	Z 90	□ 144	NUL 0	□ 3	NUL 0	NUL 0	NUL 0	□ 4	NUL 0	NUL 0	NUL 0	□ 255	□ 255	NUL 0	NUL 0
□ 184	NUL 0	NUL 0	NUL 0	NUL 0	NUL 0	NUL 0	NUL 0	@ 64	NUL 0	NUL 0	NUL 0	NUL 0	NUL 0	NUL 0	NUL 0
NUL 0	NUL 0	NUL 0	NUL 0	NUL 0	NUL 0	NUL 0	NUL 0	NUL 0	NUL 0	NUL 0	NUL 0	NUL 0	NUL 0	NUL 0	NUL 0
NUL 0	NUL 0	NUL 0	NUL 0	NUL 0	NUL 0	NUL 0	NUL 0	NUL 0	NUL 0	NUL 0	NUL 0	□ 128	NUL 0	NUL 0	NUL 0

Plik PDF

% 37	P 80	D 68	F 70	- 45	1 49	. 46	4 52	LF 10	% 37	□ 208	□ 212	□ 197	□ 216	LF 10	1 49
 32	0 48	 32	o 111	b 98	j 106	LF 10	< 60	< 60	 32	/ 47	S 83	 32	/ 47	G 71	o 111
T 84	o 111	 32	/ 47	D 68	 32	[91	2 50	 32	0 48	 32	R 82	 32	 32	/ 47	F 70
i 105	t 116	 32	] 93	 32	> 62	> 62	LF 10	e 101	n 110	d 100	o 111	b 98	j 106	LF 10	5 53

Plik JPG

□ 255	□ 216	□ 255	□ 224	NUL 0	□ 16	J 74	F 70	I 73	F 70	NUL 0	□ 1	□ 1	□ 1	NUL 0	H 72
NUL 0	H 72	NUL 0	NUL 0	□ 255	□ 225	□ 24	□ 244	E 69	x 120	i 105	f 102	NUL 0	NUL 0	I 73	I 73
* 42	NUL 0	□ 8	NUL 0	NUL 0	NUL 0	□ 7	NUL 0	□ 18	□ 1	□ 3	NUL 0	□ 1	NUL 0	NUL 0	NUL 0
□ 1	NUL 0	NUL 0	NUL 0	□ 26	□ 1	□ 5	NUL 0	□ 1	NUL 0	NUL 0	NUL 0	b 98	NUL 0	NUL 0	NUL 0

Wykres funkcji

Sprawa jest w miarę prosta: dla zdefiniowanej przez siebie funkcji matematycznej wytworzyć plik tekstowy ze współrzędnymi punktów X, Y który będzie otwierany przez Excel. Rzecz jasna trzeba będzie jakąś funkcję sobie napisać. Na przykład taką (każdy może napisać sobie własną):

```
function MojaF( x : real ) : real;
const f = 1.0;
begin
  if x=f then MojaF := 0
  else MojaF := x*f/(x-f);
end;
```

> Uwaga: To rzecz jasna odległość obrazu w równaniu soczewki wypukłej, gdzie ogniskowa $f = 1$. Znaczenie fizyczne mają tylko $x > 0$.
>
> $$\frac{1}{y} + \frac{1}{x} = \frac{1}{f} \;\Rightarrow\; y = \frac{xf}{x-f}$$
>
> Funkcja ma dziurę w dziedzinie dla $x = f$. Wartość ta jest uwzględniona w definicji funkcji, żeby nie było dzielenia przez 0.

Nasz program wygeneruje plik CSV (typ `Text`) zawierający współrzędne punktów – w tabeli będą to dwie kolumny. Plik przeznaczony będzie dla Excela w którym robienie wykresów realizuje się przez: **Wstawianie** → **Wykresy** → **Punktowy**. A oto postać końcowego pliku:

```
 0,00000000000000E+000;-0,00000000000000E+000
 5,00000000000000E-003;-5,02512562814070E-003
 1,00000000000000E-002;-1,01010101010101E-002
 1,50000000000000E-002;-1,52284263959391E-002
 2,00000000000000E-002;-2,04081632653061E-002
...
```

> Słowo o formacie CSV (comma-separated values): stanowi bardzo popularny sposób zapisywania danych tabelarycznych. Poszczególne pola oddzielone są – jak widać wyżej – średnikami. Zauważmy, że nazwa mówi o przecinkach (comma) i tak jest dla lokalizacji anglosaskich. Ponieważ jednak w naszej lokalizacji przecinek jest separatorem dziesiętnym (dla części ułamkowej liczby), wprowadzono zwyczaj, że wartości pól będzie oddzielać się średnikiem.

Dla zadanego przedziału (`xp`, `xk`) i liczby punktów `N`, wyliczymy współrzędne poszczególnych punktów (zmienna `x` i wartość `MojaF(x)`) i zapiszemy je do pliku.:

```
AsSign( plik, 'funkcja.csv' );
ReWrite( plik );
dx := (xk-xp)/N;
for i:=0 to N do
begin
  x:= xp + i*dx;
  WriteLn( plik, x, ';', MojaF(x) );
end;
Close( plik );
```

Warto przećwiczyć kilka modyfikacji tego programu. Postarajmy się wydzielić operacje zapisu do pliku do osobnej procedury. Trzeba będzie przekazać do niej tablicę ze współrzędnymi. Utwórzmy więc typ rekordu, który będzie elementem tablicy:

```
TPunkt = record
  x, y : real
end;
```

Pora podjąć decyzję czy tablica ma być dynamiczna czy zwykła. Jeśli liczba punktów `N` będzie stałą w programie, to korzystamy z tablicy o ustalonym rozmiarze. Jeśli `N` podaje użytkownik – zastosujemy tablicę dynamiczną.

```
TTablica = array of TPunkt;
```

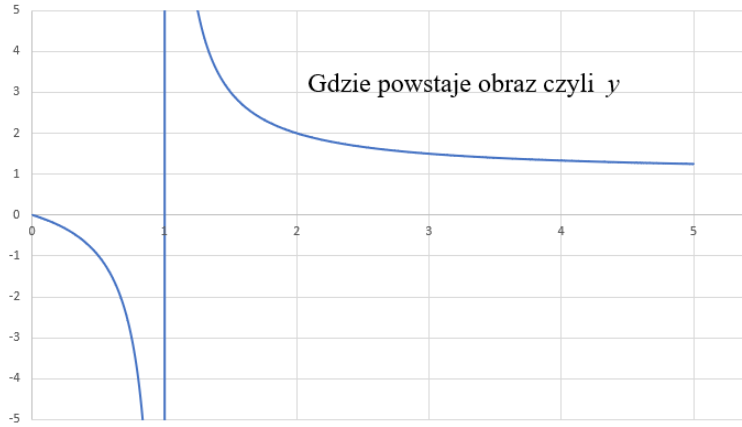
Po utworzeniu pliku i otwarciu jej przez Excel, konieczna jest zamiana kropek na przecinki. Taka zamiana jest dość żmudna, gdy się ją robi więcej niż raz. Dlatego też warto skorzystać z funkcji `FormatFloat()`, zapisując liczbę w formatowaniu zgodnym z naszą lokalizacją. W tym celu należy zadeklarować korzystanie z modułu `sysutils` – nazwę modułu wpisuje się po słowie `uses`, zaraz po deklaracji programu:

```
program WykresFunkcji;
uses sysutils;
```

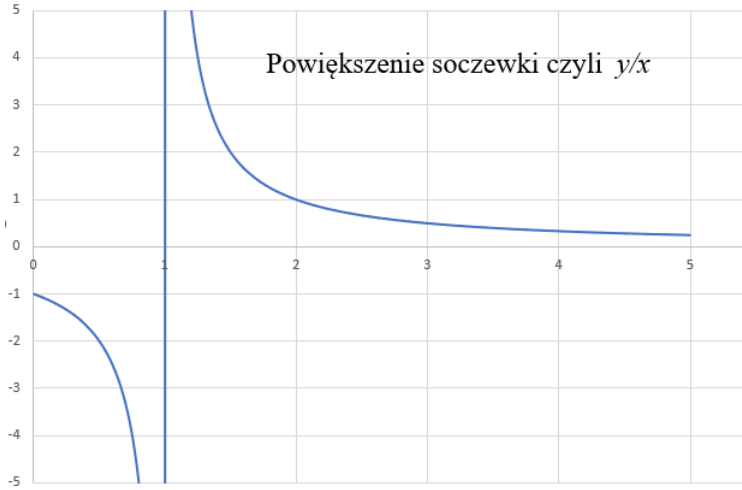
Chyba najprostszym sposobem skorzystania z `FormatFloat()` jest następujący:

```
procedure wypisz( var plik: Text; punkt : Tpunkt );
begin
  WriteLn( plik, FormatFloat( '', punkt.x ),
           ';',  FormatFloat( '', punkt.y ));
end;
```

A jak wyszły wykresy? Ano normalnie – równanie soczewki to w końcu zwykła hiperbola:



Dla położenia przedmiotu $x < f$ dostajemy obraz pozorny (patrzymy przez lupę, żeby zobaczyć powiększenie), dla $x > f$ soczewka skupia światło. Im większa odległość, tym punkt skupienia jest coraz bliżej ogniska f. Jeśli interesuje nas powiększenie soczewki, powinniśmy narysować funkcję: $y/x = f/(x - f)$. Na poniższym wykresie widać, że patrząc przez lupę zobaczymy przedmioty powiększone, a dla obrazów rzeczywistych (za lupą) – im większa odległość tym będą mniejsze. Jak przedmiot będzie bardzo daleko i będzie nim Słońce, to skupienie promieni na super-niewielkim obszarze rozpali nawet ogień:



Fraktale IFS

Korzystając z technik wypracowanych w poprzednim zadaniu zróbmy coś ambitniejszego: fraktal IFS.

Wypadłoby coś powiedzieć na temat czym są takie fraktale. Otóż skrót IFS oznacza iterated function system i jest pojęciem zaczerpniętym z matematyki. Kiedyś wydawało się, że posłużą do kompresji obrazów, ale okazało się, że są lepsze metody. Wracając do definicji: bierzemy kilka przekształceń płaszczyzny (choć jak ktoś chce to może działać w wyższych wymiarach) i w miarę losowo wykonujemy je jedna po drugiej wiele, wiele razy. Każdy wybór dokonywany jest z określonym

prawdopodobieństwem. I teraz niech mnie nie zabiją ludzie matematyczni – jeśli przekształcenia są wystarczjąco „pomniejszające", to działając nimi na dowolny zbiór, przekształcimy go w końcu na samopodobny zbiór fraktalny. Jeśli tego nie zrozumiałeś – nic straconego, może rozjaśni się, jak wykonamy praktyczną realizację.

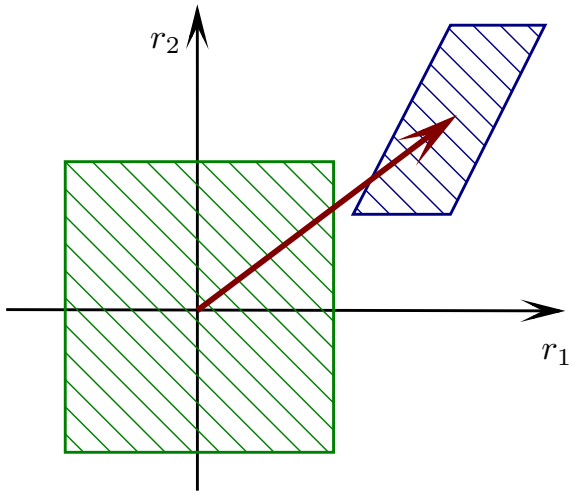
W naszym zadaniu ograniczymy się do tzw. przekształceń afinicznych:

$$\begin{aligned} r'_1 &= a_{11}r_1 + a_{12}r_2 + t_1 \\ r'_2 &= a_{21}r_1 + a_{22}r_2 + t_2 \end{aligned}$$

Składają się one z macierzy A (zmniejsza i obraca) i translacji t (przesuwa).

$$\begin{pmatrix} r'_1 \\ r'_2 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} r_1 \\ r_2 \end{pmatrix} + \begin{pmatrix} t_1 \\ t_2 \end{pmatrix}$$

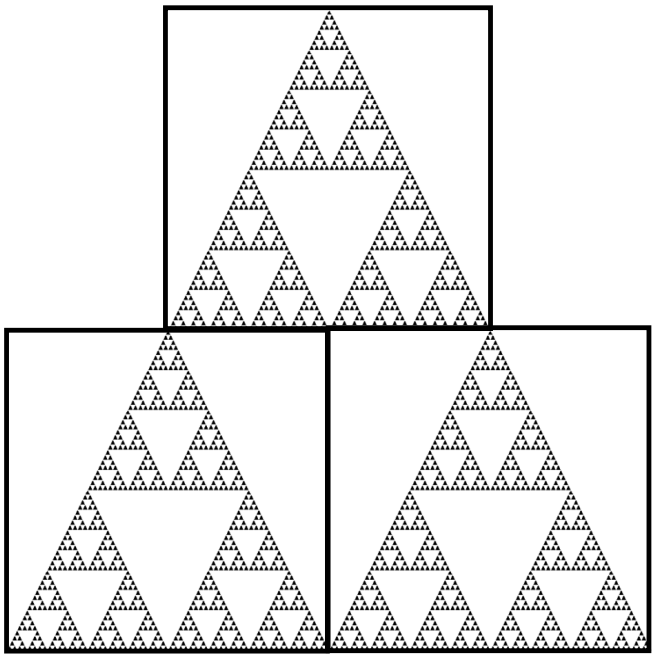
Na obrazku wyglądałoby to tak



Tych odwzorowań może być kilka, żeby uzyskać jako taki obrazek – co najmniej trzy. Algorytm startuje od jakiegoś punktu – w zasadzie nie jest aż tak istotne od którego, bo odwzorowania są tzw. zwężające. Żeby nie fundować sobie niepotrzebnego błądzenia, wybierzemy sobie punkt należący od fraktala czyli środek układu współrzędnych (0,0) – będzie tak, dla większości rozsądnie dobranych IFS. Potem losujemy jedno z odwzorowań, działamy nim na nasz punkt, zapisujemy sobie współrzędne wyniku przekształcenia. Powtarzamy procedurę, tyle że za każdym razem działamy na punkt uzyskany w poprzednim kroku. Dla mądrze dobranych funkcji, zbiór punktów będzie tworzył nam fraktal – tak w wielkim skrócie.

Najprostszym do wytworzenia jest tzw. trójkąt Sierpińskiego. Nie wnikając w ścisłą definicję matematyczną, IFS odtwarza ten zbiór dla trzech funkcji: macierz każdej, nie obracając, zmniejsza obszar dwukrotnie. Odwzorowania różnią się translacjami – pierwsze z nich jest zerowe (nie ma przesunięcia), drugie przesuwa

o 0,5 w kierunku poziomym, trzecie o 0,25 poziomo i 0,5 pionowo. Dla wszystkich trzech prawdopodobieństwa równe są 1/3. Wielokrotne składanie takich odwzorowań powinno nam dać trójkąt z dziurami:



Teraz pora zastanowić się nad programem. Jakie są wymagania? Dobrze byłoby, żeby generował dowolny IFS, a nie tylko trójkąt Sierpińskiego – parametrów funkcji (macierzy i translacji) nie zapiszemy więc w kodzie programu. Trzeba będzie je umieścić w jakimś pliku konfiguracyjnym z którego nasz program sobie je przeczyta. Trzeba też przechować odczytane dane – nie wiadomo ile ich będzie potrzebne. Dochodzi do tego losowanie, przekształcenie punktu, zapamiętanie zbioru punktów i zapisanie ich do pliku CSV. No to do roboty.

Zdefiniujmy potrzebne typy. Punkt moglibyśmy zapisać jako rekord zawierający dwie współrzędne typu `real`. Zaproponuję jednak inny wybór – skoro będziemy przemnażać współrzędne punktu przez macierz, ujednolicę stosowany formalizm i punkty zapiszę w postaci dwuelementowych tablic, a macierz jako dwuwymiarową tablicę:

```
TWekt = array[1..2] of real;
TMac = array[1..2, 1..2] of real;
```

Dane odwzorowania (prawdopodobieństwo, macierz i przesunięcie) zmieszczę w rekordzie:

```
TDane = record
    prog : real;
    A : TMac;
```

```
    trans : TWekt
  end;
```

Będą one przechowywane w tablicy dynamicznej:

```
TDaneIFS = array of TDane;
```

Wyliczone punkty też umieszczę w tablicy dynamicznej:

```
TPunkty = array of TWekt;
```

Zwróćmy uwagę na konwencję nazewniczą: Nazwy typów zaczynają się literą `T`. Umożliwia to m.in. utworzenie zmiennej `daneIFS` typu `TDaneIFS` – szczególnie przydatne, gdy mamy tylko jedną zmienną tego typu – nie trzeba wymyślać nowej nazwy.

Mała próbka kodu, żeby zrozumieć w którym miejscu jesteśmy – to funkcja zwracająca przekształcony punkt:

```
function Funkcja( const A : TMac; trans, punkt : TWekt ) : TWekt;
begin
  Funkcja[1] := A[1,1]*punkt[1] + A[1,2]*punkt[2] + trans[1];
  Funkcja[2] := A[2,1]*punkt[1] + A[2,2]*punkt[2] + trans[2];
end;
```

Mam nadzieję, że widać, iż zapisanie punktu jako wektora „się opłaca”.

Przejdźmy do pliku konfiguracyjnego. Na przykład takiego:

```
(*** Trójkąt Sierpińskiego ***)
(*** Liczba punktów **********)
10000
(*** Liczba przekształceń ****)
3
(*** przekształcenie 1 *******)
0.333333333
0.5 0     0
0   0.5   0
...
```

Linie oznaczane komentarzami jakby-pascalowymi, są przeznaczone dla użytkownika – chodzi o to, żeby ten nie pogubił się we wpisywaniu danych, bo program nie będzie sprawdzał poprawności. To znaczy dobrze byłoby, żeby sprawdzał, ale to możemy zostawić sobie jako modyfikację programu na przyszłość. Na razie zakładamy, że trzecia linia będzie zawierać liczbę punktów do wygenerowania, piąta – liczbę przekształceń. Do pliku zapiszemy też pakiety z parametrami przekształcenia – powyżej są to: krótki opis, prawdopodobieństwo wylosowania, oraz podwójnie: elementy macierzowe i współrzędną przesunięcia dla pierwszej i drugiej

współrzędnej punktu. Postać może być inna, ale ma być w miarę jasna dla człowieka i możliwa do przeczytania przez komputer.

```
function Odczytaj( sciezka : string;
                   var daneIFS: TDaneIFS ): integer;
var
  i, Lpunktow, Lfunkcji : integer;
  dane : TDane;
  plik : Text;
  pstwo, prog : real;
begin
  Assign( plik, sciezka );
  Reset( plik );
  ReadLn( plik ); {czytanie linii z komentarzem "na pusto" }
  ReadLn( plik );
  ReadLn( plik, Lpunktow );
  ReadLn( plik );
  ReadLn( plik, Lfunkcji );
  SetLength( daneIFS, Lfunkcji );
  prog := 0;
  for i:=0 to Lfunkcji-1 do
  begin
    ReadLn( plik );
    ReadLn( plik, pstwo );
    prog := prog + pstwo;
    dane.prog := prog;
    ReadLn( plik, dane.A[1,1], dane.A[1,2], dane.trans[1] );
    ReadLn( plik, dane.A[2,1], dane.A[2,2], dane.trans[2] );
    daneIFS[i] := dane;
  end;
  Close( plik );
  Odczytaj := Lpunktow;
end;
```

Zwróćmy uwagę na procedurę wyliczania progów prawdopodobieństwa: dla każdej funkcji sumujemy występujące wcześniej wartości – to po to, żeby nie musieć tego wyliczać przy każdym losowaniu. Wybór realizowany będzie następująco: jeśli wylosowana wartość jest mniejsza od p_1 – wykonujemy pierwsze odwzorowanie, jeśli mniejsza od $p_1 + p_2$ – drugie, itd. Dodatkowo `Odczytaj()` zwraca liczbę punktów fraktala – wybrałem tę drogę, bo nie zapisuję tej liczby nigdzie w tablicy `DaneIFS`.

Właściwie to pozostała nam do napisania procedura przekształcająca punkt. Będzie ona losować którą funkcję mamy wybrać, a po wylosowaniu uruchomić ją.

```
procedure KolejnyKrok( var punkt : TWekt;
                       const daneIFS : TDaneIFS );
var nr: integer;
  los : real;
begin
```

```
  los := random;
  for nr:=0 to length( daneIFS ) do
    if los<daneIFS[nr].prog then break;
  with daneIFS[nr] do
    punkt := Funkcja( A, trans, punkt );
end;
```

Pascal w tym miejscu pokazał pazurki: pomimo głębokiego zagnieżdżenia danych dotyczących odwzorowań, dostęp do nich odbywa się dość gładko, dzięki wykorzystaniu funkcjonalności `with`.

Podaję cały kod programu głównego, który jak widać wcale nie jest długi (ma raptem 11 linii).

```
var
  daneIFS : TDaneIFS;
  dane : TDane;
  punkt :TWekt;
  i, Lpunktow : integer;
  punkty : TPunkty;
BEGIN
  punkt[1] := 0;
  punkt[2] := 0;
  Lpunktow := Odczytaj( 'IFS.conf', daneIFS );
  SetLength( punkty, Lpunktow );
  randomize;
  for i:=0 to Lpunktow-1  do
  begin
    KolejnyKrok( punkt, daneIFS );
    punkty[i] := punkt;
  end;
  Zapisz( 'ifs.csv', punkty );
END.
```

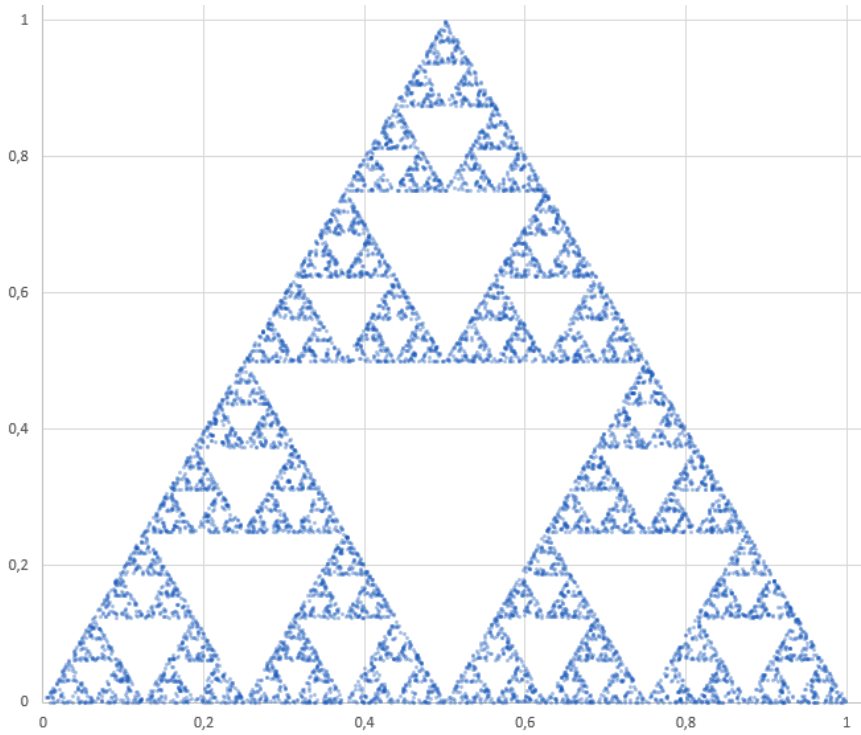
Procedurę `Zapisz()` zostawiam jako zadanie ZTS, tym bardziej, że jest (prawie) identyczna z tą z poprzedniego zadania.

Na koniec kilka obrazków, jakie uzyskałem dla różnych zbiorów odwzorowań. Nie będę narzekał na Excela, który niekoniecznie się nadaje do malowania zbiorów pikselo-podobnych, ale ma za to ma duże możliwości edycyjne.

$$A_1 = \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix} \quad t_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad p_1 = 1/3$$

$$A_2 = \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix} \quad t_2 = \begin{pmatrix} 1/2 \\ 0 \end{pmatrix} \quad p_2 = 1/3$$

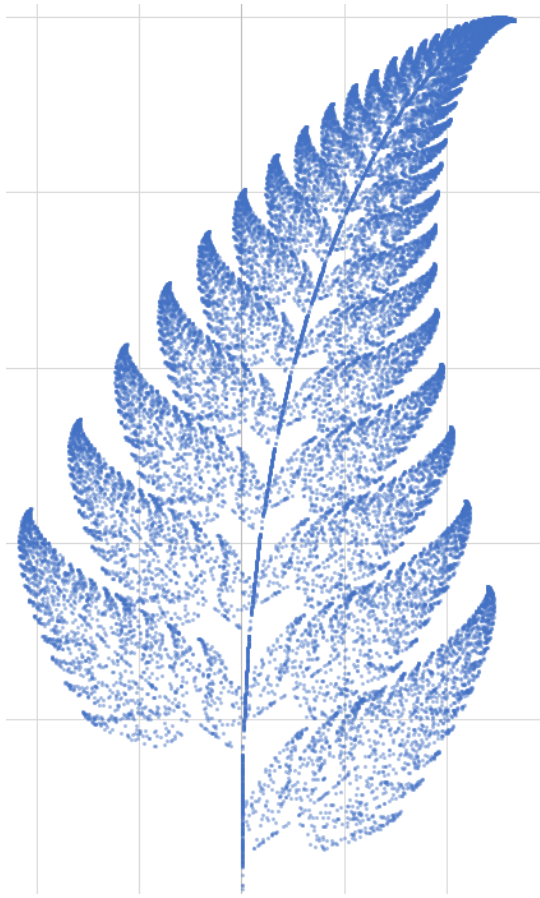
$$A_3 = \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix} \quad t_3 = \begin{pmatrix} 1/4 \\ 1/2 \end{pmatrix} \quad p_3 = 1/3$$



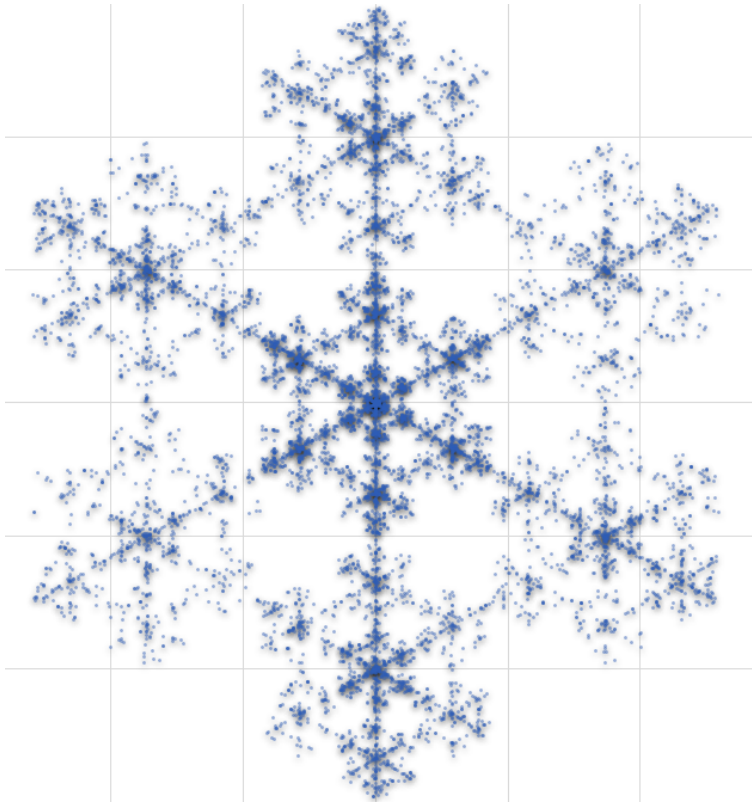
Trójkąt Sierpińskiego i jego funkcje.

A poniżej inny przykład fraktala IFS, bodaj najpopularniejszy:

```
(*** Paproć Barnsleya *****)
(*** Liczba punktów *******)
30000
(*** Liczba przekształceń *)
4
(*** przekształcenie 1 ****)
0.85
0.85   0.04    0
-0.04  0.85    1.6
(*** przekształcenie 2 ****)
0.07
-0.15  0.28    0
0.26   0.24    0.44
(*** przekształcenie 3 ****)
0.07
0.2    -0.26   0
0.23   0.22    1.6
(*** przekształcenie 4 ****)
0.01
0      0       0
0      0.16    0
```



To były dwa bardzo książkowe przykłady. Każdy może sobie popróbować zrobić inne. Na przykład gwiazdkę:



2.11 Lista zadań do modułu „Trivium”

Zadanie 1 – Bisekcja odcinka

Czasami stajemy przed problemem rozwiązania równania

$$f(x) = 0$$

Jeżeli z grubsza znamy przebieg funkcji, tzn. wiemy gdzie mniej więcej znajdują się jej miejsca zerowe, możemy to równanie rozwiązać numerycznie. Warunkiem na rozwiązanie zadania, by w danym przedziale funkcja miała jedno misje zerowe w którym zmienia znak (to ważne!) i była ciągła.

Dla przykładu weźmy funkcję $\sin(x)$. Wiemy że ma miejsce zerowe w okolicach liczby 3,14. Weźmy przedział $(x_p = 3, x_k = 4)$. Wartość $\sin(3)$ jest dodatnia, $\sin(4)$ jest już ujemna. Znajdujemy wartość funkcji dla $x_s = (x_p + x_k)/2$. Po znaku funkcji zorientujemy się, czy miejsce zerowe jest przed czy po x_s. W naszym przypadku $\sin(x_s = 3,5)$ jest mniejszy od zera, więc przecięcie wykresu z osią x znajduje się pomiędzy (x_p, x_s). Powtarzamy procedurę połowienia dla „nowego” odcinka. Każdy krok powoduje zmniejszenie badanego odcinka o połowę – czyli

po kilkunastu krokach otrzymamy dość dokładny wynik, gdzie może znajdować się miejsce zerowe.

Żeby nie bawić się w zbyt wiele sprawdzań znaku funkcji, wystarczy zbadać znak iloczynu: $f(x_p) \cdot f(x_s)$ i na tej podstawie wybierać „nowe" przedziały odcinka.

W programie zdeklarujemy sobie stałą odpowiadającą dokładności – jakaś mała liczba 0,0000001 – i będziemy tak długo dzielić odcinek na połowy, aż długość odcinka będzie mniejsza od dokładności.

Zadanie 2 – Zapałki

Gra z zapałki polega na tym, że spośród N zapałek wyciąga się/spala/zdejmuje ze stołu od 1 do K zapałek ($K > 1$ ale $K < N$). Gra dwóch graczy – użytkownik i program. Przegrywa ten, kto ściągnie ostatnią zapałkę. Podpowiedź co do strategii gry: wygrywa ten, kto za każdym razem zostawi drugiemu graczowi liczbę zapałek będącą wielokrotnością $(K + 1)$ plus jedna zapałka. Czyli ten kto zaczyna, ma zapewnioną wygraną (chyba, że N jest już wspomnianą wielokrotnością).

Zadanie 3 – Papier nożyce kamień

Napisać program grający z użytkownikiem w papier, nożyce i kamień. Program losuje jeden z trzech stanów, użytkownik podaje swój wybór i na tej podstawie ogłaszany jest wynik.

Zadanie 4 – Cyfry rzymskie

Napisać program zamieniający liczbę napisaną cyframi rzymskimi na arabskie. Poniżej wypisano jakie cyfry odpowiadają jakim wartościom liczb:

I	V	X	L	C	D	M
1	5	10	50	100	500	100

Liczba jest sumą cyfr zapisanych jako litery. Symbole ustawione są w kierunku malejącym w prawo, przy czym jeśli symbol I, X lub C poprzedza symbol liczby większej, to wchodzi on do sumy ze znakiem ujemnym. Musi być przy tym przestrzegana zasada, że: I może poprzedzać tylko V i X, X tylko L i C, a C tylko D i M (dlatego np. liczby 999 nie zapisuje się jako IM).

Oprócz wyliczenia wartości liczby, program może sprawdzać, czy liczba w ogóle jest poprawnie zapisana tzn. czy:

- cyfry umieszczone są „malejąco" (z wyjątkiem opisanym powyżej dotyczącym powyżej);

- nie występują inne znaki od I, V, X, L, C, D, M;
- znaki V, L, D występują co najwyżej raz, a I, X, C i M co najwyżej trzy razy.

Zadanie 5 – Systemy pozycyjne

Program pokazujący liczbę w różnych systemach pozycyjnych. Normalnie posługujemy się systemem dziesiętnym, ale matematycy potrafią zapisać liczbę w innych systemach, a informatycy powszechnie zapisują dane w postaci binarnej (system dwójkowy) lub heksagonalnej (system szesnastkowy, gdzie cyfry większe od 9 zapisuje się jak A, B…)

Kolejne cyfry – od końca – otrzymujemy poprzez:

- reszta z dzielenia przez podstawę (bazę systemu);
- do kolejnego kroku bierzemy liczbę będącą wynikiem dzielenia całkowitego liczby przez bazę;
- kończymy jeśli iteracja liczby (czyli dzielenie liczby przez podstawę) jest równa 0.

Zadanie 6 – Podglądanie bitów

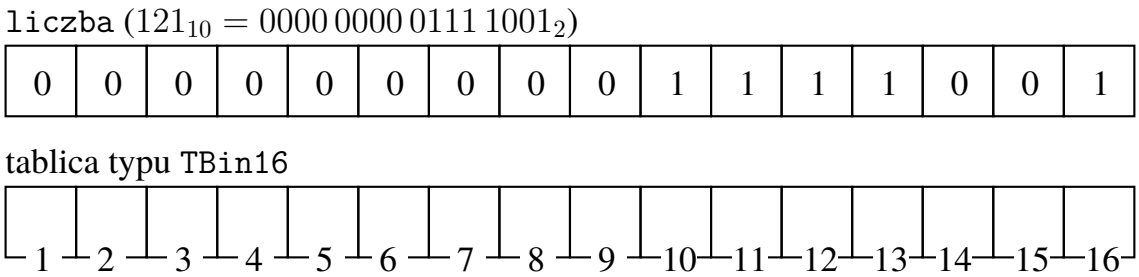
Korzystając z funkcjonalności rekordów zmiennych, w łatwy sposób przedstaw binarną zawartość zmiennych liczbowych (`integer`, `longint`, `real`, `double`, `byte`, `char`). Podpowiedź: jednym polem niech będzie liczba, a drugim tablica bitów.

Czym jest tablica bitów? Typem postaci:

```
TBit16 = bitpacked array [1..16] of 0..1;
```

Taka tablica zajmuje dwa bajty, czyli tyle ile liczba całkowita typu `integer`. Jeśli w części zmiennej rekordu umieścimy dwa pola – jedno typu `TBit16`, drugie `integer`, dowiemy się z jakich 0 i 1 jest zbudowana binarnie, wypisując zawartość tablicy. Dla typów jednobajtowych (`char`, `byte`) utworzymy typ z tablicą ośmioelementową (`TBit8`), podobnie dla typów o większym rozmiarze.

Jeśli zrobisz to zadanie, możesz się nieco zdziwić, ale wyjaśnienie pozostawiam niżej. Tak właśnie „poukładane” są dane w pamięci komputera: najniższy rząd (tutaj dla cyfr w systemie dwójkowym) jest na końcu zmiennej:



Zadanie 7 – Sprawdzanie numeru IP

To zadanie może być ciekawe, jeśli cokolwiek orientujesz się w zagadnieniach sieciowych. Dotyczy numeracji IP w wersji 4 – wciąż jeszcze obowiązującej, poznanej i oswojonej. Każdy komputer w sieci TCP/IP v.4 ma numer IP w skład którego wchodzą dwie liczby 4-bajtowe. Zwykle zapisuje się je tak:

IP: `192.168.0.100`
maska podsieci: `255.255.255.0`

Opcjonalnie dochodzi do tego zestawu trzecia liczba – brama domyślna.

Zadanie polega na tym, żeby policzyć czy dwa numery IP należą do tej samej sieci. Uściślając: Ponieważ nr IP składa się z właściwego IP i maski podsieci, to w powyższym zagadnieniu chodzi o coś ciutkę innego. Wyobraźmy sobie, że nasz komputer ma te swoje IP i maskę, a musi wysłać pakiet do innego komputera o innym IP (nie musimy znać jego maski). W zależności o tego, czy ów drugi komputer jest w tej samej sieci czy innej, pakiet będzie inaczej wysyłany. W drugim przypadku odbiorcą będzie router – host mający numer bramy domyślnej.

Żeby nie grzebać się w czterech zmiennych (bajtach) dla jednego IP, koniecznie trzeba zamknąć je w rekordzie, a ponieważ będziemy dokonywać operacji binarnych, to warto skorzystać z doświadczeń z poprzedniego zadania.

* * *

Warunkiem koniecznym żeby jakkolwiek podejść do tego zadania, jest wiedza jak dokonuje się operacji binarnych w pascalu. Otóż używa się do tego operatorów logicznych: `and`, `or`, `not` i `xor`. Działania takie wykonuje się na typach bez znaku: `byte`, `word`, `longword`. Polegają one na wykonywaniu działań dla każdego bitu osobno. Czyli dla bajtów:

```
A := 3; // 0000 0011
B := 1; // 0000 0001
C := A and B; // 0000 0001
C := A or B;  // 0000 0011
C := not A;   // 1111 1100
C := A xor B; // 0000 0010
```

Jeśli ciekawi się temat przekształceń na bitach, zajrzyj na stronę 182, gdzie rozpatruję inny, ciekawy przypadek.

* * *

Wracając do zadania: Warto sprawdzić czy wpisana przez nas maska podsieci jest postaci (binarnie): najpierw same jedynki, od pewnego momentu same zera.

Zadanie 8 – Losowanie kolejności

Mamy N elementów i musimy je wypisać w losowej kolejności.

Zadanie 9 – Lista katalogów

Wyszukiwanie katalogu o zadanej nazwie. Nie chodzi oczywiście o wpisanie komendy w okienko wyszukiwania eksploratora plików:

```
nazwa:FPC rodzaj:folder
```

żeby wyszukać katalog Free Pascala. Chodzi o odczytanie pliku, który będzie wynikiem uruchomienia programu tree:

```
tree /A > katalogi.txt
```

Jest on w dość czytelnej postaci „drzewko-podobnej” (stąd nazwa programu):

```
Folder PATH listing
Volume serial number is XXXX-XXXX
C:.
+---Ala Ma Kota
|   +---Ala
|   +---Ola
|   \---Pliki Uli
+---Binarki
...
```

Kropka przed `C:` oznacza, że listowaliśmy katalog bieżący. Możemy zażyczyć sobie wypisanie zawartości konkretnego katalogu – wtedy podajemy go jako parametr programu. Wypisanie wszystkich katalogów na dysku może zająć dużo czasu, więc testuj `tree.exe` uruchamiając w jakimś niewielkim folderze. Naszym zadaniem będzie przerobienie pliku z drzewkiem na plik z listą pełnych katalogów:

```
Ala Ma Kota
Ala Ma Kota\Ala
Ala Ma Kota\Ola
Ala Ma Kota\Pliki Uli
```

```
Binarki
...
```

Po co taki zabieg? Gdybyśmy przeszukiwali katalog `C:\Program Files` w poszukiwaniu nazwy podejrzanego programu, to ta druga postać jest wygodniejsza, bo od razu daje pełną ścieżkę.

A teraz jak to zrobić:

- czytamy linię pliku;
- sprawdzamy na jakiej pozycji znajduje się ciąg `+---` lub `\----`; wystąpienie tych napisów może być równe $4k + 1$ ($k = 0, 1 \ldots$)
- wyliczmy k – wystarczy zwykłe dzielenie całkowite odczytanej pozycji;
- zapisujemy resztę linii do k-tego elementu tablicy – tablica indeksowana od 0, bo pierwsze `+---` występuje dla $k = 0$;
- Zapisujemy do pliku wyjściowego napis złożony z elementów tablicy od 0 do k, oddzielanych znakiem \ (backslash).

Ważne uwagi: Ponieważ ścieżki mogą być (i są) dłuższe od 255 znaków, stosujemy typ `AnsiString`. Do wycinania pod-napisu używa się funkcji `copy()` – argumenty liczbowe oznaczają: pozycję początku pod-napisu i jego długość:

```
napis := 'Ala␣ma␣kota';
kawalek := copy ( napis, 5, 4 ); // 'ma␣k';
```

> Okienko cmd.exe używa domyślnie kodowania IBM. Oznacza to, że utworzone pliki nie będą poprawnie pokazywać np. polskich ogonków. Żeby zmienić kodowanie, należy uruchomić (np. na początku skryptu wywołującego tree.exe) komendę:
>
> ```
> mode con cp select=1250
> ```
>
> Dostępne są również inne strony: dla ISO-8859-2 podajemy liczbę 28592 a dla UTF-8 65001.

2.12 Rozwiązania zadań z modułu „Trivium”

Powtórzę tu słowa z poprzedniego modułu: Poniższy kod wcale nie jest najlepszy z możliwych. Może się wręcz okazać, że ten który napisałeś, jest: lepszy, wygodniejszy w konserwacji i mniej na narażony na popełnienie błędu. Powtórzę tu to, co pisałem w poprzednim module: napisany samodzielnie jest najlepszy. W dodatku podane rozwiązania prezentują różne etapy realizacji, niekoniecznie etap końcowy. Można więc je jeszcze modyfikować.

Zadanie 1 – Bisekcja odcinka

Wydzieliłem treść funkcji do funkcji `funkcja` ☺. Gdy będę chciał ją zmienić, to zmiany będą dotyczyć jednego miejsca w programie

```
function funkcja( x : real ) : real;
begin
  funkcja := sin(x);
end;
```

No i sama bisekcja:

```
function MZerowe( xp, xk : real ) : real;
const dokl = 0.00000001;
var xs : real;
begin
  repeat
    xs := (xk+xp)/2;
    if funkcja(xp)*funkcja(xs)>0 then xp := xs
    else xk := xs
  until xk-xp<dokl;
  MZerowe :=  xs;
end;
```

Wywołanie MZerowe(3, 4) zwróci wartość π z zadaną dokładnością.

Zadanie 2 – Zapałki

OK. Nie zaprezentuję całości programu, ale kluczowe procedury poniżej można zobaczyć. Stan gry obejmuje dwie liczby: aktualny stan zapałek i liczbę ściągniętych zapałek w ostatnim kroku – dana może być potrzebna, gdybyśmy chcieli wyświetlić użytkownikowi komunikat: `Komputer zdjął X zapałek`.

```
program GraWZapalki;
const
  N = 17;
  K = 4;

type Tgra = record
    stan, roznica : integer
  end;

procedure init( var gra : Tgra );
begin
  gra.stan := N;
  gra.roznica := 0;
end;

procedure ruchKomp( var gra : Tgra );
```

```
var reszta : integer;
begin
  reszta := gra.stan mod (K+1);
  if reszta=1 then
    {przegrana, ale user może się pomyli}
    gra.roznica := 1+random(K)
  else
  begin
    {już wygraliśmy}
    if reszta = 0 then gra.roznica := K
    else gra.roznica := reszta - 1;
  end;
  gra.stan := gra.stan - gra.roznica;
end;

...
```

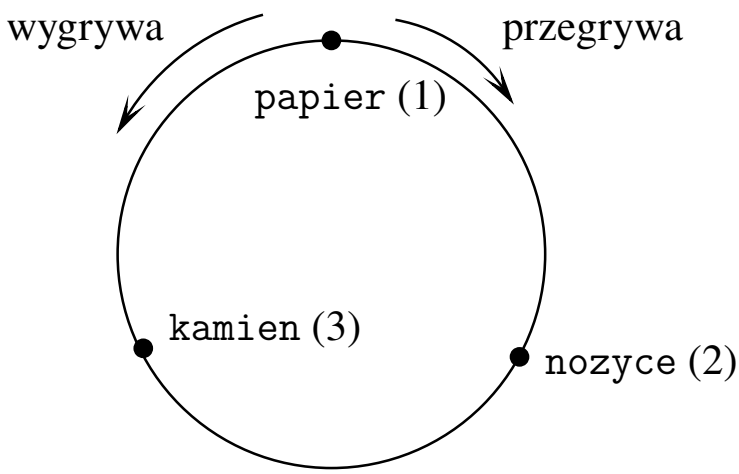
Do napisania zostały: odczytywanie wyboru użytkownika, wypisanie stanu gry (m.in. wspomniany komunikat), naprzemienne kierowanie grą i rozpoznawanie wygranej – te dwa ostatnie elementy można zamieścić w programie głównym.

Zadanie 3 – Papier nożyce kamień

Pierwsza myśl, to zbudować konstrukcję if-ów, która przewidzi 9 możliwych przypadków – na każde 3 możliwości wyboru jednej strony przypada 3 możliwości drugiej. Podam jednak inne, bardziej abstrakcyjne rozwiązanie.

Każda strona rozgrywki może wybrać trzy stany. Jeśli wyobrazimy sobie je jako punkty na równiku oddalone o 120°, to wygrana jest podobna do stwierdzenia, czy dany obiekt jest położony na wschód czy na zachód od nas. W końcu odpowiedź na pytanie w jakim kierunku trzeba kierować się do Japonii, zależy od pierwotnej lokalizacji – co innego powie Europejczyk, co innego Amerykanin.

Ustawię stany gry na okręgu:



i odpowiem na pytanie, czy wybrany stan drugiej strony jest na lewo czy na prawo od mojego stanu. Innymi słowy wyliczę odległość $+1$ albo -1. Dla takich samych stanów ogłoszę remis.

```
program PapierNozyceKamien;
type Stan = ( papier=1, nozyce=2, kamien=3 );
var
  stanKomp, stanUser : Stan;
  los, wynik : integer;
  wybor : char;
begin
  WriteLn( 'To gra w papier, nożyce i kamień.');
  WriteLn( 'Program będzie losował swój wybór,');
  WriteLn( 'ty podasz swój: znak P/N/K zatwierdzisz ENETRem.');
{ losowanie stanu dla komputera }
  randomize;
  los := random(3)+1;
  stanKomp := Stan(los);
{ odczytanie wyboru użytkownika }
  Write( 'Podaj swój wybór: ');
  ReadLn( wybor );
  wybor = UpCase( wybor );
  if (wybor='P') then
    stanUser := papier
  else if (wybor='N') then
    stanUser := nozyce
  else stanUser := kamien;
{ wyliczanie wyniku }
  wynik := ord(stanUser) - ord(stanKomp);
  if wynik<-1 then wynik := wynik+3
  else if wynik>1 then wynik := wynik -3;
{ podanie odpowiedzi }
  WriteLn( 'Wybrałeś: ',stanUser, ', program wybrał: ',stanKomp);
  if wynik>0 then WriteLn( 'wygrałeś!' )
  else if wynik<0 then WriteLn( 'program wygrał :(' )
  else WriteLn( 'remis' );
end.
```

Ponieważ program główny ma w zasadzie podzieloną funkcjonalność, aż się prosi by napisać funkcję do wyliczania wyniku. Jak się można domyślić chodzi o wydzielenie do podprogramu kodu znajdującego się po komentarzu `{ wyliczanie wyniku }`.

Zadanie 4 – Cyfry rzymskie

Wydzielamy funkcjonalność zamiany wartości cyfr rzymskich do funkcji:

```
function WartCyfry( znak : char ) : integer;
begin
```

```
  case znak of
    'I' : WartCyfry := 1;
    'V' : WartCyfry := 5;
    'X' : WartCyfry := 10;
    'L' : WartCyfry := 50;
    'C' : WartCyfry := 100;
    'D' : WartCyfry := 500;
    'M' : WartCyfry := 1000;
  end;
end;
```

Sama procedura zamieniająca wygląda nieco marudnie:

```
function Rzym2Arab( liczbaR : string ) : integer;
var i, wynik, znak : integer;
   cyfra, poprz : char;
begin
  wynik := 0;
  for i:=length( liczbaR ) downto 1 do
  begin
    znak := 1;
    cyfra := liczbaR[i];
    if (i<length( liczbaR )) then
    begin
      poprz := liczbaR[i+1];
      if (cyfra='I') and ( (poprz='V') or (poprz='X') )
        then znak := -1
      else if (cyfra='X') and ( (poprz='L') or (poprz='C') )
        then znak := -1
      else if (cyfra='C') and ( (poprz='D') or (poprz='M') )
        then znak := -1;
    end;
    wynik := wynik + znak*WartCyfry( cyfra )
  end;
  Rzym2Arab := wynik;
end;
```

więc może wprowadzić zmienną logiczną, zamiast drabinek z if-ami:

```
function Rzym2Arab( liczbaR : string ) : integer;
var i, wynik : integer;
   cyfra, poprz : char;
   ujemna : boolean;
begin
  wynik := 0;
  for i:=length( liczbaR ) downto 1 do
  begin
    cyfra := liczbaR[i];
    ujemna := false;
    if (i<length( liczbaR )) then
    begin
```

```
      poprz := liczbaR[i+1];
      ujemna := ((cyfra='I') and ( (poprz='V') or (poprz='X') ))
             or ((cyfra='X') and ( (poprz='L') or (poprz='C') ))
             or ((cyfra='C') and ( (poprz='D') or (poprz='M') ));
    end;
    if ujemna then wynik := wynik - WartCyfry( cyfra )
    else wynik := wynik + WartCyfry( cyfra )
  end;
  Rzym2Arab := wynik;
end;
```

Co do funkcji sprawdzającej, to z wykazu trzech warunków zrealizowałem dwa łatwiejsze:

```
function CzyRzym( liczbaR : string ) : boolean;
var
  cyfra : char;
  i, liczV, liczL, liczD, liczI, liczX, liczM, liczC : integer;
begin
  liczV := 0;
  liczL := 0;
  LiczD := 0;
  liczI := 0;
  liczX := 0;
  liczM := 0;
  liczC := 0;
  for i:=length( liczbaR ) downto 1 do
  begin
    cyfra := liczbaR[i];
    case cyfra of
      'I' : inc( LiczI );
      'V' : inc( LiczV );
      'X' : inc( LiczX );
      'L' : inc( LiczL );
      'C' : inc( LiczC );
      'D' : inc( LiczD );
      'M' : inc( LiczM )
    else exit( false )
    end;
  end;
  if (LiczI>3) or (LiczV>1) or
     (LiczX>3) or (LiczL>1) or
     (LiczC>3) or (LiczD>1) or
     (LiczM>3) then exit( false );
  // to nie koniec...
  CzyRzym := true;
end;
```

Został najmarudniejszy warunek do samodzielnego napisania, bo IXI wciąż jest według programu poprawną liczbą (wylicza 10).

Zadanie 5 – Systemy pozycyjne

```
function ZmPodstawy( liczba: longint; baza : integer ) : string;
var
  cyfra, wynik : string;
  reszta : integer;
begin
  wynik := '';
  while true do
  begin
    reszta := liczba mod baza;
    if reszta<10 then str( reszta, cyfra )
    else cyfra := chr( ord('A') + reszta - 10 );
    wynik := cyfra + wynik;
    liczba := liczba div baza;
    if liczba=0 then break;
  end;
  ZmPodstawy := wynik;
end;
```

Spróbuj zmodyfikować program, żeby dla liczb ujemnych wynik też był pokazywany poprawnie. Wtedy -1 byłoby zapisywane dwójkowo jako właśnie jako `-1` a nie 1111…1 – tak jak w następnym zadaniu ☺ .

Zadanie 6 – Podglądanie bitów

Opierając się na przedstawionym materiale zbudowalibyśmy typy:

```
const N=16;
type
  TBit16 =  bitpacked array [1..N] of 0..1;
  TRodzaj = ( liczbowy, binarny );
  TBinaria16 = record
    case rodzaj : TRodzaj of
      liczbowy: (liczba : integer);
      binarny: (bity: TBit16);
  end;
```

Do tego warto dopisać funkcję przedstawiającą tablicę zer i jedynek w postaci napisu:

```
function bin2str( tablica : TBit16 ):string;
var i : integer;
  wynik, znak: string;
begin
  wynik := '';
  for i:=1 to N do
  begin
    str( tablica[i], znak );
```

```
    wynik := znak + wynik;
  end;
  bin2str := wynik;
end;
```

Zauważmy jednak ciekawą właściwość: w ogóle nie korzystaliśmy z pola `rodzaj` – ani go nie ustawialiśmy, ani nie odczytywaliśmy. Tak naprawdę w ogóle można go pominąć, a wartości w `case of` przyjąć jako kolejne liczby całkowite. Określa się wtedy typ pola, ale go nie deklaruje:

```
TBinaria16 = record
  case byte of
    0: (liczba : integer);
    1: (bity: TBit16);
end;
```

Wtedy cały program:

```
program Konwersje;
const N=16;
type
  TBit16 =  bitpacked array [1..N] of 0..1;
  TBinaria16 = record
    case byte of
      0: (liczba : integer);
      1: (bity: TBit16);
  end;

function bin2str( tablica : TBit16 ):string;
...

var
  dana: TBinaria16;
begin
  dana.liczba := 127;
  Writeln( 'liczba ', dana.liczba, ' to binarnie:' );
  WriteLn( bin2str( dana.bity ) );
end.
```

Zadanie 7 – Sprawdzanie numeru IP

Najpierw przeróbmy rekord do przechowywania liczb 4-bajtowych:

```
const N=32;
type
  TBit32 =  bitpacked array [1..N] of 0..1;
  TBinaria32 = record
    case byte of
```

```
      0: (oktD, oktC, oktB, oktA : byte);
      1: (bity: TBit32);
      2: (licz: longword)
  end;
```

Jak widać od razu uwzględniłem potrzebę przechowywania czterech oktetów składowych. Ich kolejność odpowiada ulokowaniu ich w czterobajtowej liczbie:

192.168.0.1 w postaci „zwykłej”, to binarnie i dziesiętnie:
$(11000000101010000000000000000001)_2 = (3232235521)_{10}$

Dopisałem zestaw funkcji i procedur do łatwiejszego posługiwania się rekordami typu TBinaria32. Począwszy od wygodnej inicjalizacji:

```
procedure LadujIP( oA, oB, oC, oD : byte; var IP : TBinaria32 );
begin
  with IP do
  begin
    oktA := oA;
    oktB := oB;
    oktC := oC;
    oktD := oD;
  end;
end;
```

poprzez mnożenie binarne – wynikiem jest również zestaw czterej bajtów:

```
function MnoBin( IP, maska : TBinaria32 ) : TBinaria32;
var
  wynik : longword;
begin
  wynik := IP.licz and maska.licz;
  MnoBin.licz := wynik;
end;
```

oraz znaną z poprzedniego zadania funkcję wypisującą ciąg zer i jedynek:

```
function bin2str( tablica : TBit32 ):string;
var i : integer;
  wynik, znak: string;
begin
  wynik := '';
  for i:=1 to N do
  begin
    str( tablica[i], znak );
    wynik := znak + wynik;
  end;
  bin2str := wynik;
end;
```

Teraz sprawdźmy jak to działa. W programie głównym określmy sobie maskę i zobaczmy czy rzeczywiście wygląda poprawnie:

```
LadujIP( 255, 255, 255, 0, maska );
WriteLn( bin2str( maska.bity ) );
```

Teraz określmy dwa numery IP i sprawdźmy czy dwa numery IP są w tej samej sieci:

```
LadujIP( 192, 168, 0, 100, IP1 );
siec1 := MnoBin( IP1, maska );
LadujIP( 192, 168, 0, 90, IP2 );
siec2 := MnoBin( IP2, maska );
if( siec1.licz = siec2.licz ) then
  WriteLn( 'IP2␣jest␣w␣tej␣samej␣sieci␣co␣PI1' )
else
  WriteLn( 'IP2␣nie␣jest␣w␣tej␣sieci␣co␣IP1' );
```

Zadanie 8 – Losowanie kolejności

Zadanie nie precyzuje jakiego typu dane mamy losować. Załóżmy, że będą to napisy:

```
type Tablica = array of string;
```

Losowanie polega na wylosowaniu jednego z elementów i zamianie miejscami z ostatnim elementem tablicy. Kolejne losowanie odbywa się w zakresie o mniejszym o 1. Ostatnie losowanie obejmuje dwa elementy.

```
procedure losowanie( var tab : Tablica );
var i, idx : integer;
begin
  for i:= length(tab) downto 2 do
  begin
    idx := random(i);
    zamiana( tab[idx], tab[i-1] );
  end;
end;
```

Inicjalizacja tablicy i jej wyświetlanie – ZTS.

Zadanie 9 – Lista katalogów

```
program PrzerabianieKatalogow;
const MaxZagl = 25;
```

```
var
  linia, sciezka, katalog : AnsiString;
  plikWe, plikWy : Text;
  pozycja, k, i : integer;
  tablica : array[0..MaxZagl ] of AnsiString;
BEGIN
  AsSign( plikWe, 'katalogi.txt' );
  ReSet( plikWe );
  AsSign( plikWy, 'pelnekatalogi.txt' );
  ReWrite( plikWy );
  while not eof( plikWe ) do
  begin
    ReadLn( plikWe, linia );
    { tylko jedna próbka może mieć poz()<>0 }
    pozycja := pos( '\---', linia ) + pos( '+---', linia );
    if pozycja=0 then continue;
    { wyliczam pozycję do wycięcia }
    pozycja := pozycja + 4;
    k := pozycja div 4;
    katalog := copy( linia, pozycja, length(linia)-pozycja+1 );
    tablica[k] := katalog;
    sciezka := '';
    for i:=1 to k do
      sciezka := sciezka + tablica[i] + '\';
    WriteLn( sciezka );
    Writeln( plikWy, sciezka );
  end;
  Close( plikWy );
  Close( plikWe );
END.
```

Nie jestem do końca zadowolony tego programu, ponieważ zastosowałem dwie sztuczki – obie zaznaczone komentarzem. Po pierwsze szukane próbki są dwie, ale w jednej linii może wystąpić tylko jedna (nie jest to prawda, ale prawdopodobieństwo, że ktoś umieści w nazwie katalogu ciąg znaków +--- jest znikome). Dlatego sumowanie wyszukanych pozycji jest ekwiwalentem skomplikowańszej formuły logicznej, sprawdzającej wystąpienie pozycji dwóch próbek. Po drugie „zaoszczędziłem" stosując jedną zmienną pozycja do szukania pozycji próbki i oznaczenia położenia pod-napisu do wycięcia.

Można zauważyć, że pomieszałem też skutecznie funkcjonalności przetwarzania napisów i operacji plikowych. Sprawę poprawiłaby funkcja przetwarzająca napisy, zwracająca (przez argument) stopień zagłębienia = indeks w tabeli. Wartość funkcji zawiera informację, czy w ogóle udało się wyciąć nazwę podkatalogu.

```
function Wytnij( AnsiString linia,
                 var AnsiString katalog,
                 var idx: integer ) : boolean;
var pozycja : integer;
begin
```

```
  pozycja := pos( '\---', linia ) + pos( '+---', linia );
  if pozycja=0 then exit( false );
  pozycja := pozycja + 4;
  idx := pozycja div 4;
  katalog := copy( linia, idx, length(linia) - idx + 1 );
  Wytnij := true;
end;
```

Podobnie można napisać funkcję do tworzenia całej ścieżki. Funkcja pobierałaby tablicę i stopień zagłębienia (w programie powyżej zmienna k).

Liczby pierwsze

Do listy dołożyłem jeszcze jedno z zadań, które pojawiło się w module „Trivium”. Tak na deser.

```
function CzyPierwsza( n : longint ) : boolean;
var biez : longint;
    granica : double;
begin
  if n<2 then exit( false );
  if (n=2) or (n=3) then exit( true );
  if n mod 2 = 0 then exit( false );
  CzyPierwsza := true;
  granica := sqrt( n );
  biez := 3;
  repeat
    if ( n mod biez )=0 then exit( false );
    biez := biez + 2;
  until biez>granica;
end;
```

Rozdział 3

Po trivium

W tym module znajdziesz kilka zagadnień bez których przez długi czas można żyć przy używaniu pascala, ale prędzej czy później możesz na nie trafić. Z punktu widzenia amatora można traktować je jako zaawansowane. Co nie znaczy, że nie warto ich poznać lepiej.

3.1 Rekurencja

Kiedy pierwszy raz stykamy się z rekurencją, możemy się zdziwić jak to w ogóle działa! Funkcja wywołuje samą siebie – czy to nie tak jak baron Münchhausen, co to wyciągał się za włosy z bagna? Nie, w rekurencji nie ma magicznych sztuczek, ale po kolei…

Zobaczmy jak wygląda rekurencyjna wersja programu do liczenia silni:

```
function silnia( n : integer ) : integer;
begin
  if n < 2 then silnia := 1
  else silnia := n * silnia( n-1 );
end;
```

Spotykamy się tu pierwszy raz z przypadkiem, by nazwa funkcji pojawiła się po prawej stronie operatora przypisania `:=`. Oznacza to, że w tym miejscu następuje kolejne wywołanie funkcji `silnia()`. Wyjaśnijmy co się dzieje: W czasie kompilacji, każdy podprogram jest kompilowany do postaci rozumianej przez komputer. Wywołanie procedury lub funkcji polega na uruchomieniu odpowiadającemu mu kawałka kodu. Może się zdarzyć, że jednym z zadań skompilowanego podprogramu będzie kolejne uruchomienie tego samego kodu. Na pierwszy rzut oka wygląda to tak, jakby funkcja wywoływała samą siebie.

Jeśli przeanalizujemy kod `silnia()`, to zauważymy, że kolejne wywołania skończą się, gdy dojdzie do uruchomienia `silnia( 1 )`. Warunek taki musi być spełniony w każdym wykorzystaniu rekurencji.

Zwykle algorytmy rekurencyjne działają dłużej i potrzebują więcej pamięci od swych odpowiedników iteracyjnych. Ale stosuje je się, ze względu na elegantszy zapis i większą elastyczność. Czasem nierekurencyjna wersja algorytmu jest dramatycznie bardziej skomplikowana.

Na czym polega większe zapotrzebowanie na pamięć? Otóż algorytmy rekurencyjne powodują wiele wywołań podprogramów – każdemu trzeba zapewnić odpowiednie zasoby w tzw. stosie programu. Nie wnikając dokładnie co to takiego: między innymi zapisuje się tam jakie są argumenty podprogramu i jakie wartości są zwracane, jeśli mamy do czynienia z funkcją. Jeśli ciąg wywołań będzie zbyt duży, może zabraknąć miejsca lub czasu (częściej spotykane). Dla wyobrażenia: wywołanie `silnia(10)` spowoduje odłożenie na stos programu 10 paczek danych dotyczących 10 wywołań funkcji `silnia()`. Poniżej przykład dla którego liczba wywołań rośnie prawie wykładniczo wraz ze wzrostem wartości argumentu.

Zadanie: Wypisać trójkąt Pascala.

```
                1
            1       1
        1       2       1
    1       3       3       1
    .   .   .   .   .
```

Trójkąt Pascala buduje się w następujący sposób: w każdym kolejnym wierszu wypisuje się liczby, będące sumą dwóch liczb z poprzedniego wiersza. Istnieje twierdzenie, że liczby te są równe:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

gdzie n to numer wiersza, a k to pozycja w wierszu. Oba indeksy zaczyna się liczyć od 0:

k=	0	1	3	3	4	5	6	7	8	9	10
n=0	1										
n=1	1	1									
n=2	1	2	1								
n=3	1	3	3	1							
n=4	1	4	6	4	1						
n=5	1	5	10	10	5	1					
n=6	1	6	15	20	15	6	1				

n=7	1	7	21	35	35	21	7	1			
n=8	1	8	28	56	70	56	28	8	1		
n=9	1	9	36	84	126	126	84	36	9	1	
n=10	1	10	45	120	210	252	210	120	45	10	1
.....											

Zauważmy, że kolejne wyrazy tego trójkąta rosną o wiele wolniej niż $n!$, więc powinno dać się je policzyć bez użycia silni, która jak pamiętamy rośnie tak szybko, że już dla niewielkich n przekracza wszelkie zakresy przeznaczone na typy całkowite. Pamiętając, jak konstruuje się trójkąt Pascala, możemy oprzeć się na innej formule $\binom{n}{k}$:

$$\binom{n}{k} = \begin{cases} 1 & \text{jeżeli } n = 0 \\ 1 & \text{jeżeli } k = 0 \\ 1 & \text{jeżeli } k = n \\ \binom{n-1}{k} + \binom{n-1}{k-1} & \text{w reszcie przypadków} \end{cases}$$

Po napisaniu okaże się, że można poprawnie liczyć wielkości $\binom{n}{k}$ dla $n < 18$, jeśli przyjmiemy, że nasz wynik jest liczbą typu `integer`. Dla typu `longint` jest nieznacznie lepiej bo poprawny wynik dostaniemy dla $n < 34$. Czas wyliczeń może być bardzo długi – liczba wywołań funkcji jest olbrzymia dla dużych n. Dla $\binom{5}{3}$ mamy 19 wywołań, dla $\binom{10}{5}$ – 503, a dla $\binom{15}{7}$ – 12869. Tak więc, zanim użyjemy rekurencji, powinniśmy znać podobne oszacowania dla danego przypadku, żeby nie dostać programu, który nigdy (w skali naszego życia, albo jednego wieczornego posiedzenia) się nie skończy.

Quick sort – ważny przykład

Jednym z algorytmów, które prezentują się bardzo dobrze w rekurencji, jest tzw. szybkie sortowanie tablic. Polega ono na dzieleniu tablicy na dwa fragmenty: z większymi i mniejszymi wartościami. Rzecz jasna potrzeba poprzerzucać te większe do jednej części a mniejsze do drugiej. Potem każdą z tych części znów dzielimy i tak w kółko. Tak opisany algorytm niewiele tłumaczy, więc pora na dokładniejszy opis.

Przede wszystkim wybiera się jakąś wartość – elementy większe od niej przesuniemy na jedną stronę, mniejsze na drugą. Cały algorytm działa prawidłowo, jeśli wartości w tablicy są rozmieszczone dość losowo z punktu widzenia posortowania. Możemy więc do porównań wziąć ostatni element tablicy, skoro każdy jest równie dobry. Teraz w pętli od pierwszego do przedostatniego elementu tablicy porównujemy z naszą wartością (w kodzie poniżej nazywa się ona `granica`) i jeśli rzeczywiście jest mniejsza to przenosimy ją na początek tabeli – ściślej rzecz

biorąc zamieniamy miejscami. Pierwszą liczbą mniejszą od `granica` przenosimy na pierwsze miejsce w tabeli, drugą na drugie miejsce itd. Zapamiętujemy liczbę przesuniętych liczb w zmiennej `biez` i po każdym przeniesieniu zwiększamy ją o 1. Na koniec zamieniamy miejscami ostatni element z bieżącym.

```
procedure podziel( var tab : Tablica );
var biez, i : integer;
    granica : real;
begin
  granica := tab[N];
  biez:=1;
  for i:=1 to N-1 do
    if tab[i]<granica then
    begin
      zamien( tab[i], tab[biez] );
      inc( biez );
    end;
  zamien( tab[N], tab[biez] );
end;
```

Powyższa procedura dzieli tablicę tyko raz. W kolejnym kroku należy ją zastosować osobno do pierwszej części i osobno do drugiej. No a potem powtarzać, aż przedziały staną się dwuelementowe. Po co nam więc procedura licząca tylko krok pierwszy? Czasami napisanie prostszego kodu pozwala na łatwiejsze przejście do docelowego rozwiązania: tu potrzeba uogólnić procedurę na dowolny przedział – powyższa operuje na całej tablicy od 1 do `N`. Powinna ona też zwracać pozycję w której znajduje się nasza `granica` – tego elementu tablicy nie musimy już zmieniać – czyli ta ogólniejsza wersja `podziel()` będzie funkcją:

```
function podziel(var tab :Tablica; pocz,kon :integer):integer;
var biez, i : integer;
    granica : real;
begin
  granica := tab[kon];
  biez:=pocz;
  for i:=pocz to kon-1 do
    if tab[i]<granica then
    begin
      zamien( tab[i], tab[biez] );
      inc( biez );
    end;
  zamien( tab[kon], tab[biez] );
  podziel := biez;
end;
```

Do tego procedura rekurencyjna wywołująca `podziel()` no i samą siebie:

```
procedure SzybkiSort( var tab : Tablica; pocz, kon : integer );
var srodek : integer;
```

```
begin
  if pocz<kon then
  begin
    srodek := podziel( tab, pocz, kon );
    SzybkiSort( tab, pocz, srodek-1 );
    SzybkiSort( tab, srodek+1, kon );
  end;
end;
```

Wywołanie dla całej tablicy:

```
SzybkiSort( tab, 1, N );
```

Wypisana procedura obowiązuje dla jednego typu tablic. Gdybyśmy chcieli ją zastosować do innych typów, należałoby ją zmodyfikować. Szczególnie, że niewiele typów można porównywać operatorem <, bo dla innych konieczna jest specjalizowana funkcja.

Zastosowanie rekurencji nie pociąga za sobą dużo jednoczesnych wywołań procedury `SzybkiSort()`, ponieważ połowiąc nawet dużą tabelę dość szybko dojdziemy do dwuelementowych przedziałów. Oto wyniki jakie uzyskałem dla różnych rozmiarów tablicy – ponieważ początkowy rozkład tablicy był losowy, to wyniki za każdym razem będą nieco inne.

Zwraca się też uwagę, że działa ona najlepiej na nieposortowanych – może nawet lepiej byłoby użyć sformułowania „pomieszanych” – tablicach. Kluczowy jest tu wybór wartości do której będziemy porównywać elementy tablicy przy jej podziale – dobrze byłoby gdyby była ona bliska mediany przechowywanych wartości. W zastosowaniach praktycznych, korzystamy z gotowych bibliotek, gdzie ten i podobne mu problemy są uwzględnione.

3.2 Wskaźniki

Twórcy pascala oraz jego praktycznych realizacji, zrobili dużo, żeby programista nie musiał przejmować się zmiennymi wskaźnikowymi, zwanymi zwykle wskaźnikami. Co to za zmienne? Otóż jeśli wyobrazimy sobie, że każda paczka danych znajduje się gdzieś w pamięci komputera, a każda jednobajtowa komórka pamięci ma swój numerek, to możemy zapytać się jaki jest numer komórki pamięci, gdzie znajdziemy naszą zmienną. Na ów numer mówi się adres i tak od teraz będę go nazywał. Podsumowując: zmienne, które przechowują adresy innych zmiennych to właśnie zmienne wskaźnikowe.

Każdy wskaźnik posiada informację nie tylko o adresie, ale również o rozmiarze zmiennej ma którą wskazuje i jej typie. Zobaczmy jak wygląda deklaracja wskaźnika – do nazwy typu należy dopisać znak daszka ^:

```
var
  liczba : integer;
  wsk    : ^integer;
```

Zmienna `wsk` służy do zarządzania zmienną typu `integer`. Można ją powiązać z istniejącą zmienną:

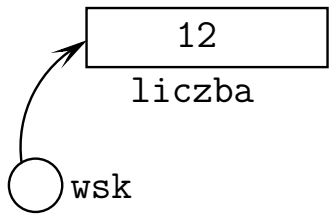
```
liczba := 12;
wsk := @liczba;
```

Operator `@` nazywa się operatorem adresu: działając na zmienną daje jej adres. Ten adres da się odczytać, choć to informacja, która się raczej nie przydaje:

```
WriteLn( 'adres (wartość wskaźnika): ', longword(wsk) );
```

Żeby podejrzeć wartość liczbową adresu, użyłem rzutowania na typ `longword` – nieujemny typ całkowity o rozmiarze 4 bajty, ponieważ `sizeof( wsk )` też jest równe 4.

Sytuację, gdy `wsk` wskazuje na `liczba` symbolicznie przedstawia rysunek:



Mając wskaźnik, możemy zarządzać zmienną z którą jest związany. W tym celu stosujemy operator wyłuskania ^ choć ja wolę go nazywać zawartością wskaźnika:

```
wsk^ := 23;
```

Teraz zmienna `liczba` będzie miała wartość 23.

* * *

Wskaźniki czasami służą do robienia żartów. Na przykład kompilator nie chce byśmy zmieniali zmienną sterującą pętli `for`? No to oszukajmy go i dokonajmy działań poprzez wskaźnik:

```
for i:=0 to 20 do
begin
  wsk := @i;
  wsk^ := wsk^ + 1;
  Write( i, ' ' );
end;
```

Choć wydaje się to nieprawdopodobne, program wypisze: `1 3 5 7 9 11 13 15 17 19 21`. Kompilator nawet się nie zająknie!

> Ten przykład należy do bardzo, bardzo złego stylu programowania. W zwykłych programach nie róbmy takich rzeczy. Tylko w ramach żartu! Z drugiej strony motto bloga brzmi: „Programowanie to zabawa" więc sam nie wiem co powiedzieć ☺.

* * *

Do potencjalnie niebezpiecznych cech należy tzw. arytmetyka wskaźników. Zmienne wskaźnikowe można zwiększyć lub zmniejszyć – będą one wtedy wskazywać na „miejsce obok".

```
wsk := @liczba;
{ zwiększenie wskaźnika o 1 }
inc( wsk );
```

Teraz adres `wsk` będzie większy o 2, bo rozmiar `integer` jest równy właśnie 2. Możemy się więc spodziewać, że dodanie n do wskaźnika, oznacza przesunięcie adresu o $n \times$rozmiar typu.

Arytmetyka wskaźników przydaje się w bardzo specyficznych przypadkach, w niniejszym kursie w ogóle takowych nie będzie. No chyba że do zabawy. Na przykład żeby podejrzeć, jak zmienne programu poukładane w pamięci:

```
var
  liczba, druga, i : integer;
  wsk     : ^integer;

BEGIN
  liczba := 222;
```

```
  druga := 444;
  wsk := @liczba;
  for i:=0 to 20 do
  begin
    WriteLn( i:2, '␣', longword(wsk), '␣', wsk^ );
    inc( wsk );
  end;
END.
```

* * *

Dopóki nie wskażemy na jaką zmienną ma wskazywać dany wskaźnik, środowisko nada mu wartość `nil`. Jest to specjalna stała nazywana wskaźnikiem zerowym. Jego wartość rzeczywiście jest równa 0. Wartość `nil` informuje nas: „na razie jeszcze na nic nie wskazuję, nie powiązano ze mną żadnej zmiennej”.

Zmienne dynamiczne

Wskaźniki pozwalają na tworzenie tzw. zmiennych dynamicznych. Do tej pory deklarowaliśmy zmienne i po uruchomieniu programu, były tworzone automatycznie. Istnieje jednak możliwość, że zmienna powstanie w trakcie działania programu i temu właśnie służą zmienne wskaźnikowe.

Przykład tworzenia zmiennej dynamicznej przedstawia poniższy schemat:

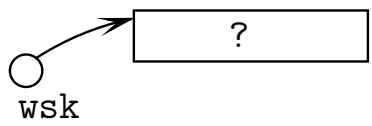
Po wykonaniu kodu

```
var wsk : ^integer;
```

zadeklarowaliśmy wskaźnik. Po rozpoczęciu programu będzie od równy `nil` – na razie nie jest związany z żadną zmienną.

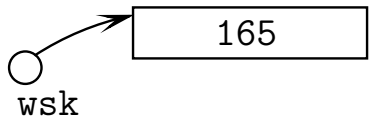
```
new( wsk );
```

Operator `new()` tworzy zmienną dynamiczną. Polega to na tym, że po jego wykonaniu fragment pamięci ma służyć do przechowania wartości. Na razie nie wiadomo co jest w przydzielonej pamięci.



```
wsk^ := 165;
```

Nadajemy wartość zmiennej dynamicznej. Zauważmy, że tak utworzona zmienna nie ma nazwy, jedyna możliwość zarządzania nią to wskaźnik, który jest z nią związany.



W programowaniu istnieje ważna zasada – jeśli gdzieś utworzyliśmy zmienną dynamiczną – mówi się wtedy o alokowaniu pamięci – powinniśmy ją też zwolnić. Kiedy zmienna dynamiczna nie jest już potrzebna, zwalniamy zajmowaną przez nią pamięć operatorem `dispose()`:

```
{ tworzenie zmiennej, alokowanie pamięci }
new( wsk );
{ używanie zmiennej }
wsk^ := 23;
{ niszczenie zmiennej, zwalnianie pamięci }
dispose( wsk );
```

Możemy więc myśleć o zarządzaniu tego typu zmiennymi, jak o pożyczaniu i oddawaniu pamięci z/do zasobów programu.

Ze zmiennymi dynamicznymi związane jest niebezpieczeństwo wycieków pamięci. Polegają one na „gubieniu” zmiennych dynamicznych przez nieuważne zarządzanie wskaźnikami. Spójrzmy na kod:

```
{ utworzenie pierwszej zmiennej dynamicznej }
new( wsk );
wsk^ := 10;
{ utworzenie kolejnej zmiennej }
new( wsk );
{ zmienna przechowująca liczbę 10 nie jest dostępna w programie }
```

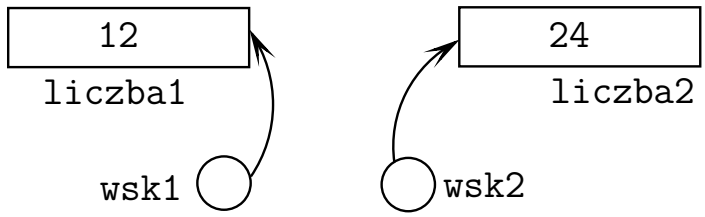
Błędy tego typu są ważne przy działających długo programach, które intensywnie alokują i zwalniają pamięć.

> Nowoczesne języki nie kłopoczą programisty pamiętaniem o zwalnianiu pamięci. Najpopularniejsze z nich czyli java i C# oferują tzw. odśmiecacz, który w trakcie działania programu sam zwalnia nieużywaną pamięć. Pascal jednak w tym względzie jest podobny do C/C++.

* * *

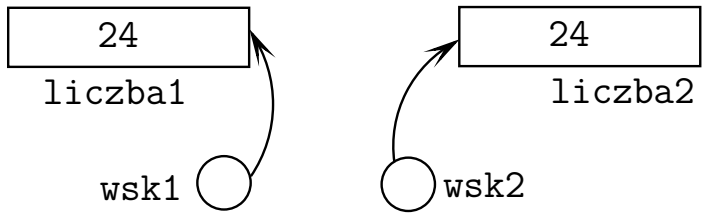
Ponieważ wprowadzenie pojęcia wskaźników stanowi pewne wyzwanie dla osób uczących się programowania, warto prześledzić zachowanie zmiennych przy różnych operacjach. Weźmy przypadek, gdy mamy dwie liczby i dwa wskaźniki:

```
wsk1 := @liczba1;
wsk2 := @liczba2;
```



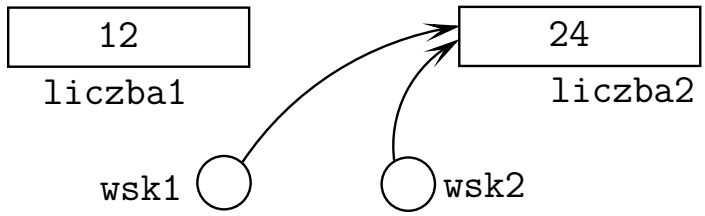
Polecenie kopiowania zawartości wskaźnika poskutkuje:

```
wsk1^ := wsk2^;
```



Natomiast gdyby zamiast tego programista zażyczył sobie kopiowanie wskaźników, to zmieni się przypisanie do zmiennej:

```
wsk1 := wsk2;
```



Zauważmy, że gdyby zmienna związana pierwotnie ze `wsk1` była zmienną dynamiczną, nastąpiłby wyciek pamięci.

* * *

Pojawia się pytanie: Do czego takie zmienne mogą się przydać? Skoro ich użycie jest ryzykowne, to może w ogóle ich nie warto używać? Otóż wbrew tym obawom, istnieją zastosowania zmiennych wskaźnikowych we współczesnym programowaniu, choć w takich językach jak java czy C# nazywa się je inaczej. Do najważniejszych z nich należy tworzenie dynamicznych struktur danych – następny

odcinek będzie poświęcony takim przypadkom. Wskaźniki są też intensywnie wykorzystywane w programowaniu obiektowym – na co zwrócę uwagę w następnych modułach.

3.3 Dynamiczne struktury danych

Wskaźniki miały doniosłe znaczenie w starożytnym programowaniu i w zasadzie jeśli ktoś np. musi wchodzić w jakieś niskopoziomowe zagadnienia, to bez wskaźnika ani rusz. Dopóki jednak nie będziemy zmuszeni do zaprogramowania jakiegoś sterownika w czymś, co może przypominać assembler, to problemy tego typu nie będą nas dotyczyć. Wskaźniki służą jednak pracowicie do dziś, jeśli chce się utworzyć dynamiczne struktury danych. Takim przypadkiem jest choćby dynamiczna tablica.

Tablice dynamiczne – ukryte wskaźniki

Jak pamiętamy (możemy zajrzeć na stronę 72), deklaracja tablicy zadanego typu umożliwia wygodną zmianę tablicy:

```
type
  Osoba = record
    ...
    end;
  Tablica = array of Osoba;

var
  tab : Tablica;
```

Na początku programu ma ona rozmiar 0:

```
Writeln( length( tab ) ); // 0
```

i przyjmuje wartość `nil`, czyli tak naprawdę (to nadużywany zwrot w języku polskim, ale tu wyjątkowo dobrze pasuje) jest wskaźnikiem.

```
if tab = nil then Writeln( 'Tablica␣jest␣pusta' );
```

Co więcej, jeśli opróżnimy tablicę przez komendę:

```
SetLength( tab, 0 );
```

to znów przyjmie ona wartość `nil`.

Skoro tablica ma dużo wspólnego, to osoby znające C, od razu spytają, czy można zarządzać taką tablicą korzystając ze wskaźnika. Odpowiedź na to pytanie

jest zasadniczo negatywna. Jedynie arytmetyka wskaźników zamiast indeksowania (częściowo) obowiązuje, ale nie ma powodów by jej używać, jako techniki bardziej pracochłonnej i o wiele bardziej ryzykownej.

Przykład listy powiązanej

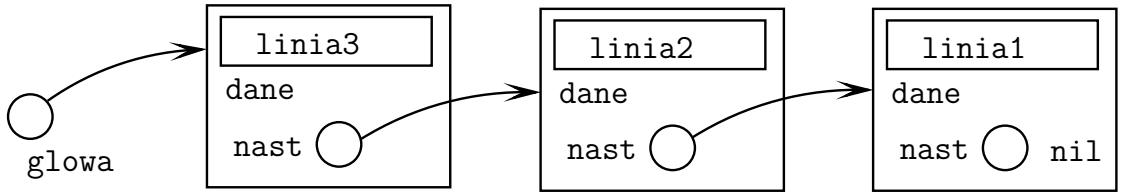
To ćwiczebny przykład, który warto choć raz w życiu przejść samodzielnie, żeby mieć orientację jak wyglądają takie twory. W praktyce korzysta się z gotowych struktur danych – nazywa się to uczenie abstrakcyjnymi typami danych. Więcej na ten temat w kolejnych (na razie planowanych) modułach.

W przypadku listy powiązanej mamy dwie strategie: albo do standardowego rekordu zawierającego poszczególne pola jako pola, dodatkowo dodajemy ostatnie pole jako wskaźnik do następnego rekordu, albo rekord zawiera tylko dwa pola: rekord z danymi i wskaźnik. W przypadku samodzielnego tworzenia typu rozwiązanie pierwsze wydaje się prostsze, przynajmniej na początku. W programowaniu za pomocą szablonów, wybiera się drugą drogę. W przedstawianym tu przykładzie nasze dane, to pojedynczy napis, więc można sobie pomyśleć, że preferujemy pierwsze rozwiązanie.

```
type
  PKostka = ^TKostka;
  TKostka = record
    dane : string;
    nast : PKostka;
  end;
```

Ten zapis może nas trochę mierzić. Typ `PKostka` jest zdefiniowany w oparciu o nieistniejący jeszcze typ `TKostka` – ten będzie zdefiniowany poniżej. Języki umożliwiające tworzenie dynamicznych struktur danych muszą w jakiś sposób udostępniać funkcjonalność tzw. typów niepełnych. Z punktu widzenia kompilatora można przymknąć oko na takie działanie: do zadeklarowania typu wskaźnikowego potrzebuje on tylko nazwy typu na który będziemy wskazywać.

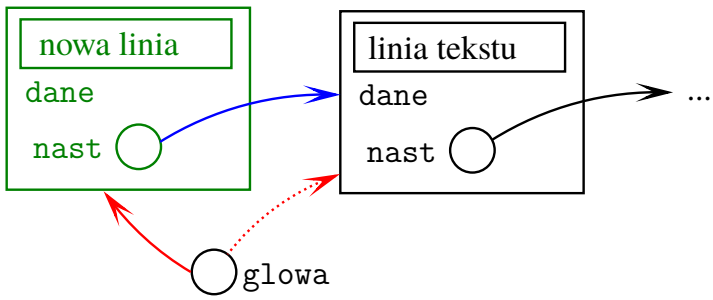
Z takich kostek można utworzyć listę powiązaną, w której pole `nast` wskazuje na następny rekord w liście.



Lista taka zarządzana jest przez zmienną wskaźnikową, która wskazuje na pierwszy rekord. Lista kończy się wtedy, gdy pole nast ostatniego rekordu na nic

nie wskazuje – jest równe `nil`. Ustawienie zera na końcu listy stanowi informację – dalej już nic nie ma.

Spróbujmy dopisać kolejną kostkę na początku istniejącej listy. Zadanie jest w miarę proste, bo doklejamy rekord bezpośrednio do kostki na którą wskazuje wskaźnik `glowa`. Mamy do wykonania: utworzenie nowej zmiennej dynamicznej za pomocą `new()`; wpisanie do niej danych; doklejenie kostki do dotychczas pierwszego elementu; i na koniec zmianę wartości wskaźnika `glowa`, tak by wskazywał na nową kostkę. Rozróżniłem te działania kolorami dla poprawienia czytelności:




```
procedure DodajPocz( var glowa: PKostka; const kostka: TKostka );
var
  bufor : PKostka;
begin
  new( bufor );
  bufor^ := kostka;
  bufor^.nast := glowa;
  glowa := bufor;
end;
```

Zauważmy, że procedura działa poprawnie nawet wtedy, gdy lista jest pusta! Wtedy do pola `nast` tworzonego rekordu wpisywany jest wskaźnik `nil` (początkowa wartość `glowa`).

Spróbujmy wypisać wszystkie elementy listy. Poniżej prezentuję kod wykonujący zadane czynności na kolejnych elementach listy. Tu akurat realizujemy wypisywanie, stąd obecność procedury `WriteLn()`. W innych przykładach będą to inne polecenia działające na rekord `TKostka`.

```
procedure Wypisz( glowa : PKostka );
begin
  repeat
    Writeln( glowa^.dane );
    glowa := glowa^.nast;
  until glowa = nil;
end;
```

Jedną z czynności, jaką można wykonywać w bardzo podobny sposób, jest kasowanie listy (zwalnianie pamięci).

```
procedure kasuj( var glowa : PKostka );
var
  bufor, biez: PKostka;
begin
  biez := glowa;
    repeat
    bufor := biez;
    biez := biez^.nast;
    { linia użyta tylko w testowaniu kasowania:
        WriteLn( 'kasuję: ', bufor^.dane ); }
    dispose( bufor );
    until biez = nil;
  glowa := nil;
end;
```

Na koniec podaję procedurę dopisującą rekord do istniejącej listy, tym razem na jej końcu. Ponieważ wymyśliłem sobie, że powinna działać również dla przypadków gdy `glowa=nil`, jest trochę skomplikowana:

```
procedure DodajKon( var glowa :PKostka; const kostka :TKostka );
var
  bufor, biez : PKostka;
begin
  new( bufor );
  bufor^ := kostka;
  if glowa = nil then
    begin
    glowa := bufor;
    exit;
    end;
  biez := glowa;
  while biez^.nast <> nil do
    biez := biez^.nast;
  biez^.nast := bufor;
end;
```

Przetestujmy napisane funkcje:

```
var lista : PKostka;
    bufor : TKostka;
BEGIN
  bufor.dane := 'pierwszy';
  DodajKon( lista, bufor );
  bufor.dane := 'drugi';
  DodajPocz( lista, bufor );
  bufor.dane := 'trzeci';
  DodajPocz( lista, bufor );
```

```
  bufor.dane := 'czwarty';
  DodajKon( lista, bufor );
  Wypisz( lista );
  Kasuj( lista );
END.
```

* * *

Dwa opisywane przypadki: dynamiczna tablica i lista powiązana nie są jedynymi. Warto wspomnieć o bardzo ważnej roli, jaką odgrywają w informatyce drzewa binarne. Szczególnie wydajne są ich zaawansowane wersje tzw. drzewa czerwono-czarne. Tematyka jest warta samodzielnych studiów.

3.4 Typy funkcyjne

Typy funkcyjne są niesamowite. Wiem, że brzmi to dziwnie, ale tak właśnie jest. Istnieją w różnych wersjach i odmianach chyba we wszystkich liczących się językach programowania. Również w pascalu. O co w nich chodzi?

W poprzednim module mieliśmy dwa charakterystyczne zadania: znajdowanie miejsca zerowego funkcji i tworzenie pliku CSV do wykresu funkcji. W obu przypadkach pisaliśmy jakąś funkcję. Jeśli program miał działać dla innej funkcji, trzeba było ją zmienić. A chciało by się mieć kod pasujący do każdej funkcji.

Przypomnijmy zadanie do znajdowania miejsca zerowego:

```
function MZerowe( xp, xk : real ) : real;
const dokl = 0.00000001;
var xs : real;
begin
  while xk-xp>dokl do
  begin
    xs := (xk+xp)/2;
    if funkcja(xp)*funkcja(xs)>0 then xp := xs
    else xk := xs
  end;
  MZerowe :=  xs;
end;
```

gdzie `funkcja()` była konkretną funkcją zdefiniowaną wcześniej. Dobrze byłoby, gdyby można było sobie jakoś wybrać dowolną funkcję rzeczywistą.

Naszym celem będzie przekazanie przez argument do podprogramu `MZerowe()` dowolnej funkcji. Pierwszy krok to zdefiniowanie sobie typu takiego argumentu:

```
type TFun = function( x: real ): real;
```

Od teraz `Tfun` oznacza tzw. typ funkcyjny, a ściślej rzecz biorąc wskaźnik na funkcję pobierającą jedną wartość typu `real` (zmienna `x`) i zwracająca typ `real`. Od teraz możemy tworzyć zmienne:

```
var f : TFun;
```

które przybierają wartości funkcyjne:

```
f := @funkcja;
WriteLn( f(x) );
```

Podprogram `funkcja()` musi być zdefiniowany wcześniej i pobierać jedną liczbę typu `real` i zwracać wartość typu `real`.

Jak widać nadanie wartości zmiennej typu funkcyjnego polega na podaniu adresu funkcji (skompilowany kod funkcji, podobnie jak zmienna zajmuje jakieś miejsce w pamięci komputera pod jakimś adresem). Jest to chyba jedyne miejsce w pascalu, poza rozwiązaniami obiektowymi, gdzie zmusza się użytkownika do jawnego wykorzystywania wskaźników. Powyższy kod pokazuje też, że typy funkcyjne są ukrytymi wskaźnikami.

Zobaczmy na przykładzie, jak to działa. Najpierw definiujemy funkcje. U nas będzie to funkcja liniowa $y = 2x$ i wartość bezwzględna $y = |x|$:

```
function modul( x : real ) : real;
begin
  if x<0 then modul := -x
  else modul := x
end;

function podwojenie( x : real ) : real;
begin
  podwojenie := 2*x
end;
```

W programie głównym możemy podłączyć te funkcje do zmiennej:

```
f := @modul;
WriteLn( f(-1) ); // równe  1.00000000000000E+000
f := @podwojenie;
WriteLn( f(-1) ); // równe -2.00000000000000E+000
```

> Jak już pisałem na wstępie odcinka, podobne typy stosuje się w wielu językach programowania. W C będą to wskaźniki na funkcję; C++ dołoży do nich jeszcze obiekty funkcyjne; java dysponuje interfejsami funkcyjnymi; w C# stosuje się tzw. delegaty. Pamiętam jednak, jak lata temu czytałem w jakimś podręczniku do C/C++ opis wskaźników na funkcje, przedstawiający

> podobną funkcjonalność do tej przedstawionej w niniejszym rozdziale. Autor podręcznika zachwycał się, że takich rozwiązań próżno szukać w innych językach, szczególnie w pascalu – były to czasy, gdy pascal miał jeszcze coś do powiedzenia. Cóż mogę powiedzieć... Typy funkcyjne istniały w pascalu od samego początku utworzenia standardu tego języka. Notabene droga, jaka została wybrana przez Borlanda w Turbo Pascalu, potem przeniesiona do Free Pascala, różni się od standardu.

Niestety nie podłączymy funkcji matematycznych istniejących w pascalu (`sin()`, `cos()`, `exp()` itd.), ze względu na inną ich adresację. Jedną z prób wyjścia jest opakowanie takiej funkcji:

```
function sinus( x : real ) : real;
begin
  sinus := sin(x);
end;
```

i wykorzystanie opakowania:

```
f := @sinus;
WriteLn( f(0.1) ); // to samo co sin(0.1)
```

Wróćmy jednak do postawionego celu. Dopiszmy do `MZerowe()` dodatkowy argument oznaczający funkcję, której miejsce zerowe powinniśmy znaleźć.

```
function MZerowe( xp, xk : real ; fun : TFun ) : real;
const dokl = 0.00000001;
var xs : real;
begin
  while xk-xp>dokl do
  begin
    xs := (xk+xp)/2;
    if fun(xp)*fun(xs)>0 then xp := xs
    else xk := xs
  end;
  MZerowe :=  xs;
end;
```

Wywołanie tej funkcji będzie następujące:

```
x := MZerowe( 3, 4, @sinus );  //3.14159265905619E+000
```

Przy pomocy tego samego podprogramu możemy obliczyć miejsce, w którym funkcja $y = 2x$ przecina oś X. Będzie to 0 (ale!) z zadaną dokładnością:

```
x := MZerowe( -1, 4, @podwojenie );  //5.58793544769287E-009
```

Zauważmy bardzo ważną rzecz: Typy funkcyjne umożliwiają niesamowicie wygodną parametryzację naszych procedur. Przy czym wyszukiwanie miejsca zerowego nie jest tu szczytem możliwości. O wiele bardziej użyteczny przykład podam poniżej.

Sortowanie po wybranym parametrze

Przypomnijmy sobie algorytm szybkiego sortowania opisany na stronie 144. Było zastosowany dla liczb typu `real`. A gdyby zastosować go do sortowania tablicy zawierającej inny typ danych? Na przykład rekordy z danymi – dla ustalenia uwagi niech będą to dane osób:

```
const   Rozm = 10;
type
  TOsoba = record
    nr   : integer;
    nazw : string[50]
  end;
  TTabOsoba = array[1..Rozm] of TOsoba;
```

Chyba się domyślasz, że typowe rekordy mają zwykle o wiele więcej pól, ale dla naszego ćwiczebnego przypadku wystarczą dwa różne. Dla prostoty przyjąłem też tablicę o stałym rozmiarze, choć w przypadkach praktycznych zwykle potrzebna jest tablica dynamiczna.

Pierwszym krokiem do zastosowania napisanego wcześniej kodu do nowego przypadku, będzie zamiana typu `real` na `TOsoba` i ustawienie typu tablicy na `TTabOsoba`, dla procedur `Zamiana()`, `podziel()` i `SzybkiSort()`. Wtedy jednak pojawi nam się problem z porównywaniem rekordów:

```
function podziel( var tab :TTabOsoba; pocz, kon :integer ) :integer;
var biez, i : integer;
    granica : TOsoba;
begin
  granica := tab[kon];
  biez:=pocz;
  for i:=pocz to kon-1 do
    if tab[i]<granica then // BŁĄD!
...
```

Linijka która ma na celu porównać dwa rekordy wygeneruje błąd, bo dla typu `TOsoba` nie ma zdefiniowanego operatora `<`, bo co to znaczy że jedne dane są mniejsze od drugich? Chcielibyśmy, żeby dało się posortować dane według konkretnego pola: po numerze lub w kolejności alfabetycznej. W tym celu trzeba użyć jakieś funkcji porównującej o wartościach logicznych. Będzie ona zwracać prawdę, gdy pierwszy rekord będzie „mniejszy” od drugiego.

Z funkcją porównującą pole `nr` nie powinno być kłopotów:

```
function PorOsNr( A, B : TOsoba ) : boolean;
begin
  if A.nr < B.nr then PorOsNr := true
  else PorOsNr := false;
end;
```

gorzej z porównaniem nazwisk. I to nie jest prosty problem. Pascal w zasadzie nie ma wsparcia dla sortowania alfabetycznego zgodnego z wymaganiami języka polskiego. I jest tu wiele problemów praktycznych: jak traktować duże i małe litery? co robić z numeracją ASCII (lub binarną reprezentacją w unikodzie) polskich znaków diakrytycznych? jak sortować napisy zawierające dywiz (kreseczkę)?

W tym miejscu kursu nie ma chyba sensu silić się na praktyczną realizację, szczególnie, że chodzi mi o pokazanie sposobu zastosowania typów funkcyjnych. Dlatego też wykorzystam zwykłe porównanie napisów, oparte o numerację kodów ASCII. Realizuje je funkcja `CompareStr()` porównującą dwa napisy. Znajduje się ona w module `sysutils` – żeby z niej skorzystać należy umieścić w programie deklarację:

```
uses sysutils;
```

Funkcja ta zwraca wartość ujemną, gdy pierwszy argument jest wcześniej w porządku pseudo-alfabetycznym, zero gdy oba są równe i większą od zera, gdy pierwszy napis jest „większy”:

```
napis := 'Abacki';
zapis := 'Zetacki';
Writeln( CompareStr( napis, zapis ) ); //-24
```

Właśnie `CompareStr()` wykorzystam w kolejnej funkcji porównującej:

```
function PorOsNazw( A, B : TOsoba ) : boolean;
begin
  if CompareStr(A.nazw, B.nazw)<0 then PorOsNazw := true
  else PorOsNazw := false;
end;
```

Skoro mamy już obie funkcje, to należy zadeklarować dla nich odpowiedni typ:

```
type
  TPorOs = function( A, B : TOsoba ) : boolean;
```

i przekazać zmienną tego typu do procedur `podziel()` i `SzybkiSort()`. Korzystając z dotychczas przedstawionego materiału, składamy wszystko w całość:

```
function podziel( var tab :TTabOsoba;
                  pocz, kon :integer;
```

```
                    por : TPorOs ) :integer;
var biez, i : integer;
    granica : TOsoba;
begin
  granica := tab[kon];
  biez:=pocz;
  for i:=pocz to kon-1 do
    if por( tab[i], granica ) then
    begin
      zamien( tab[i], tab[biez] );
      inc( biez );
    end;
  zamien( tab[kon], tab[biez] );
  podziel := biez;
end;

procedure SzybkiSort( var tab : TTabOsoba;
                      pocz, kon : integer;
                      por : TPorOs );
var srodek : integer;
begin
  if pocz<kon then
  begin
    srodek := podziel( tab, pocz, kon, por );
    SzybkiSort( tab, pocz, srodek-1, por );
    SzybkiSort( tab, srodek+1, kon, por );
  end;
end;
```

Pora na sprawdzian. Po uruchomieniu programu głównego (wykorzystuje on dwie procedury: `Zaladuj()` do inicjalizacji tablicy danymi testowymi i `Wypisz()` do wypisywania jej zawartości na ekranie – obu tu nie cytuję):

```
var tablica : TTabOsoba;
BEGIN
  Zaladuj( tablica );
  Wypisz( tablica );
  WriteLn;
  SzybkiSort( tablica, 1, Rozm, @PorOsNr );
  Wypisz( tablica );
  WriteLn;
  SzybkiSort( tablica, 1, Rozm, @PorOsNazw );
  Wypisz( tablica );
END.
```

dostaniemy trzy listy: początkową, posortowaną po polu `nr` i posortowaną po polu `nazw`.

* * *

Żeby nie było, że kurs buja w obłokach, pokazuję kod funkcji porównującej alfabetycznie napisy (w miarę) poprawnie. Zakładamy, że napisy są zgodne z którymś kodowaniem ASCII, np. CP-1250. Działanie polega na wyszukiwaniu litery w uporządkowanym napisie dla danej próbki i porównaniu wyszukanej pozycji dla dwóch próbek.

```
function Compare1250( pierwszy, drugi : string ) : integer;
{ poniższy napis wygenerowany w edytorze obsługującym CP1250 }
const porzadek = '0123456789aAąĄbBcCćĆdDeEęĘfFgGhHiIjJkKlLłŁmMnNoOóÓpPqQrRsS
var
  zakres, dlPierw, dlDrugi, i, pozPierw, pozDrugi : integer;
begin
  dlPierw := length(pierwszy);
  dlDrugi := length(drugi);
{ krótszy napis jest wcześniej }
  Compare1250 := dlPierw - dlDrugi;
  if dlPierw<dlDrugi then zakres := dlPierw
  else zakres := dlDrugi;
{ pierwsza różnica w znakach zakończy procedurę }
  for i:=1 to zakres do
  begin
    pozPierw := pos( pierwszy[i], porzadek );
    pozDrugi := pos( drugi[i], porzadek );
    if pozPierw<>pozDrugi then
    begin
      Compare1250 := pozPierw-pozDrugi;
      break
    end;
  end;
end;
```

Ale i ta funkcja nie jest idealna. Działa poprawnie dla poprawnie wygenerowanych napisów. Między innymi zakłada, że małe litery są wcześniej od wielkich, co zdaje się nie jest właściwym rozumieniem porządku alfabetycznego. Funkcja nie ma obsługi znaków innych niż litery i cyfry: w pewnym sensie ignoruje wystąpienie jakiegokolwiek innego znaku – ponieważ `pos()` w takich sytuacja zwraca 0, to takie znaki są przed literami i cyframi.

3.5 Moduły

Zdarza się, że wypracowany przez nas zestaw typów i funkcji jest używany w więcej niż w jednym programie. Można wtedy kopiować między dwoma programami, ale jest na to lepszy sposób: Można przygotować gotową „paczkę” i z niego korzystać. Takie „paczki” nazywają się modułami i samo środowisko udostępnia nam ich kilkadziesiąt. Intuicyjnie możemy je traktować jako rozszerzenia języka pascal, choć poprawniej byłoby o nich myśleć jako o dodatkowych funkcjonalno-

ściach dołączanych do naszego kodu. Jako przykład modułu udostępnianego przez środowisko zajrzymy do `crt`. Zobaczymy też, jak samemu napisać moduł – na przykładzie zestawu procedur dotyczących liczb zespolonych.

Moduł `crt`

Jak ktoś pamięta Turbo Pascala z lat 80-tych, to zna ten moduł doskonale. Tworzyło się wtedy programy w trybie tekstowym - konsola miała 25 linii i 80 kolumn. Użytkownicy obywali się wtedy bez myszki, wszelkie działania odbywały za pomocą klawiatury. Pomimo, że to dziś mało wyobrażalne, otwierały się okienka, wybierano opcje itd. Działa tak przecież środowisko Free Pascala.

Do takich programów potrzeba było czegoś więcej niż tylko wypisywania linia po linii, jak robiliśmy to we wszystkich napisanych do tej pory programach. Czasami trzeba było ustawiać kursor w dowolnej części ekranu czy zmienić kolor tła lub czcionki. Klasycznym przykładem było czekanie na wciśnięcie dowolnego klawisza:

```
repeat until KeyPressed;
```

Funkcja `KeyPressed` zwracała prawdę, jeśli jakikolwiek klawisz został wciśnięty. Czasami korzystało się z funkcji `ReadKey`, żeby dowiedzieć się, który to klawisz został wciśnięty:

```
znak := ReadKey;
```

Co ciekawe przy wykorzystaniu tej funkcji, żaden znak nie pojawiał się na ekranie.

Żeby skorzystać z tych nietypowych funkcji i procedur, należy poinformować program, że będzie się korzystać z modułu `crt`. Listę modułów (poniżej tylko jeden) podaje się zaraz po deklaracji programu, po słowie `uses`:

```
program CudaNaKosoli;
uses crt;
...
```

I w dzisiejszych czasach przydać się może kolorowanie wyprowadzanych napisów, oraz ustawianie kursora w zadanym miejscu:

```
ClrScr;
GotoXY( 1, 1 );
Write( 'Gwiazdka na środku konsoli' );
GotoXY( 40, 12 );
TextBackground( green );
TextColor( yellow );
Write( '*' );
```

Użyte funkcje i procedury to:

- `ClrScr` – czyści ekran (podobnie jak `cls` w linii poleceń);
- `GoToXY()` – ustawia kursor w odpowiednich współrzędnych znakowych. Lewy górny róg ma współrzędne (1,1);
- `TextBackground()` i `TextColor()` – ustawiają kolor tła i tekstu. Argumentem tych procedur jest kolor, czyli liczba typu `word`. W przykładzie powyżej użyłem nazw kolorów, ale ponieważ są to nazwy stałych liczb, można użyć wartości liczbowych:

```
for i:=0 to 15 do
begin
  GotoXY( 1, i );
  TextBackground( i );
  Write( '␣␣␣' );
  TextBackground( black );
  TextColor( i );
  Write( i:2, '␣*' );
end;
```

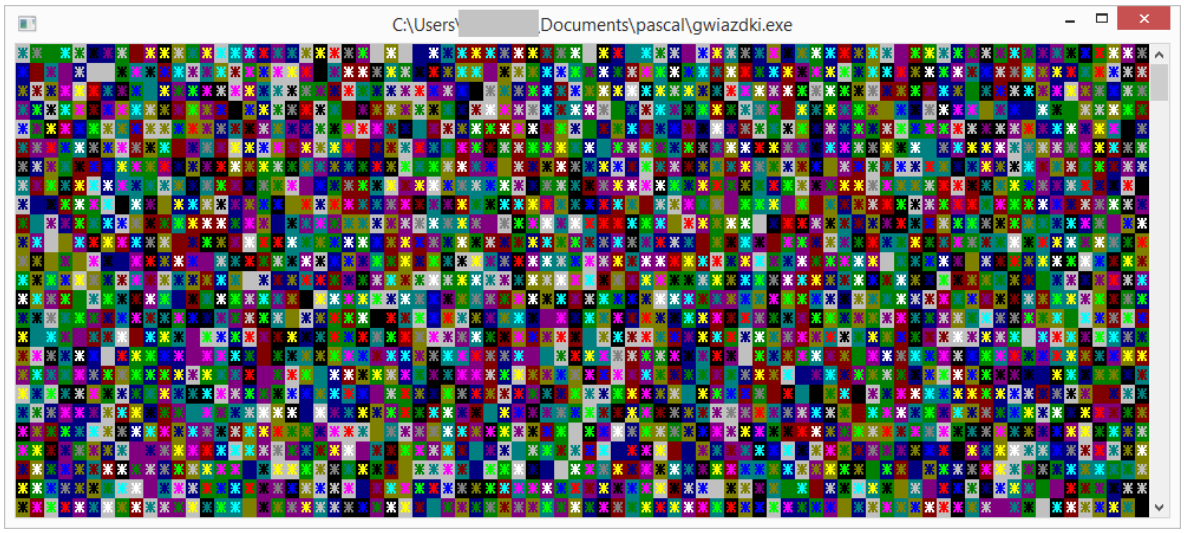
Pełna lista funkcji, procedur i stałych modułu `crt` dostępna jest w dokumentacji Free Pascala: `https://wiki.freepascal.org/Crt`

> We współczesnym Windows (powyżej 8) użycie modułu `crt` wymusza windowsową stronę kodową w okienku konsoli. W przypadku edytora Free Pascala, który pracuje w IBMowskiej (DOSowej) stronie kodowej, spowoduje to wypisanie dziwnych znaków zamiast polskich liter. Jeśli z jakiegoś powodu potrzebujemy `crt`, to: albo piszemy w zewnętrznym edytorze (niewygodne); albo zmieniamy konfigurację (ale nie wiem jak i nie wiem czy to w ogóle możliwe); albo przechodzimy na inne środowisko (np. opisane wcześniej WFP).

Zadanie: Napisz program, który w kółko będzie wyświetlał gwiazdkę (albo inny zadany znak) w losowych miejscach ekranu, gdzie tło i czcionka też będą miały losowy kolor. Program ma rysować znaki do momentu wciśnięcia dowolnego klawisza.

Jeśli rysowanie działa zbyt szybko, zastosuj procedurę `delay()`, która wstrzymuje działanie programu. Argument jest liczbą całkowitą `word` i oznacza czas zatrzymania wyrażony w milisekundach.

```
{zatrzymanie programu na sekundę}
delay(1000);
```



Przyznam się, że trochę oszukiwałem żeby uzyskać efekt jak na obrazku. Ominąłem mianowicie dolny, prawy punkt konsoli. Kiedy trafiałem tam, by narysować gwiazdkę, kursor przechodził do następnej linii i cały malunek przesuwał się o jedną linię. Nie wiem czemu procedura ukrywająca kursor nie skutkowała. Można się też wykpić od tego problemu, zamalowując mniejszy prostokąt niż rozmiar dostępnego ekranu.

Kodujemy obsługę liczb zespolonych

> Liczby zespolone mogą ci się nie podobać. No bo to matematyka, której nikt nie lubi. Dlatego na końcu postarałem się zaproponować kilka zadań, gdzie te liczby mogą się przydać.

W matematyce wykorzystuje się liczby zespolone do rozmaitych celów. Spotyka się je również w elektronice (przy obliczaniu układów RLC) i fizyce. Nie ma ich natomiast w pascalu. Zadanie będzie polegało na napisaniu procedur, które dodają, odejmują, mnożą i dzielą dwie liczby zespolone. Przydać się może również obliczanie kwadratu modułu jak i samego modułu.

Dla tych co nie wiedzą, co to takiego: cała zabawa wzięła się z niemożności rozwiązania równania: $x^2 = -1$. No niby wszyscy wiedzą, że liczba do kwadratu musi być nieujemna, ale… sami pamiętamy ze szkoły, jak pisaliśmy rozwiązania równania kwadratowego i obliczaliśmy pierwiastek z delty (musiała być dodatnia) – a gdyby tak w jakiś sposób wyciągnąć pierwiastek z ujemnej delty? Wtedy każde równanie kwadratowe miałoby swoje rozwiązanie. Jest to możliwe, gdy wprowadzimy „ekstra” liczbę i, taką, że $i^2 = -1$. Mówimy, że i jest urojone. Teraz każda liczba zespolona będzie się składać z dwóch części: rzeczywistej i urojonej: $z = 5 + 2i$.

Powinniśmy zdefiniować obiekt, który będzie przechowywał część rzeczywistą liczby i część urojoną. Wydaje się, że najlepiej do tego celu nada się rekord, przechowujący dwie liczby rzeczywiste:

```
type
  Zesp = record
    re : double;
    im : double;
  end;
```

Część rzeczywista i urojona dostępna jest poprzez pola rekordu: `z.re:=1.3` oraz `z.im:=-0.3`.

Pamiętając, że $i^2 = -1$ łatwo rozpisać sobie na kartce jak te działania powinny wyglądać. Na przykład weźmy mnożenie: dla $z = z_r + iz_i$ i $w = w_r + iw_i$ po przeprowadzeniu obliczeń otrzymamy:

$$z \cdot w = (z_r w_r - z_i w_i) + i(z_r w_i + z_i w_r)$$

Chciałoby się napisać funkcję `MnoZ()`, która pobiera dwie zmienne typu `Zesp` i zwraca wynik mnożenia:

```
function MnoZ( z1,z2 : Zesp ) : Zesp;
```

> Moduł z liczbami zespolonymi był pierwszym napisanym przeze mnie. Ówczesne środowisko Turbo Pascala nie pozwalało tworzyć funkcji zwracających rekordy. konieczne było wtedy używanie procedur, gdzie jedne argument realizował dostęp do zmiennej (`var`). Ponieważ cały niniejszy kurs dotyczy Free Pascala, gdzie takiego ograniczenia nie ma, utworzymy funkcje, bo są wygodniejsze. Dodatkowo są nieco mniej wrażliwe na błąd opisany na stronie 59.

Myślę, że dodawanie jest dość łatwe do zrealizowania. Odejmowanie to dodawanie liczb, z których druga ma zmieniony znak. Pierwszą nietrywialną funkcją jest mnożenie:

```
function MnoZ( z1, z2:zesp ) :zesp;
begin
  MnZ.re := z1.re*z2.re - z1.im*z2.im;
  MnZ.im := z1.re*z2.im + z1.im*z2.re;
end;
```

Przyznam się, że od takiego zapisu wolę następujący:

```
function MnoZ( z1, z2:zesp ) :zesp;
var w : Zesp;
begin
  w.re := z1.re*z2.re - z1.im*z2.im;
  w.im := z1.re*z2.im + z1.im*z2.re;
  exit( w );
end;
```

Być może dlatego, że w tym drugim troszeczkę trudniej się pomylić. Ale wybór jest tu chyba kwestią indywidualną. Dodawanie i odejmowanie, jako łatwe zostawiam jako zadanie ZST.

Do takiego zestawu potrzebujemy jeszcze:

- dzielenia;
- kilku stałych: 1, i, 0;
- funkcji rzutującej liczbę rzeczywistą na zespoloną.

No to po kolei:

Dzielenie dwóch liczb zespolonych:

$$\frac{z_r + iz_i}{y_r + iy_i} = \frac{(z_r + iz_i)(y_r - iy_i)}{(y_r + iy_i)(y_r - iy_i)} = \frac{\text{mnożenie}(z, \bar{y})}{\text{kwadrat modułu}(y)}$$

Jak widać, skoro mamy już napisaną funkcję mnożenia, trzeba dopisać jeszcze kwadrat modułu i obie je wykorzystać.

Stałe. Trzeba sobie przypomnieć, jak tworzyliśmy stałe rekordy. Przypadki takie należą do wyjątków – to pierwsze miejsce w niniejszym kusie, kiedy rzeczywiście je deklarujemy:

```
const  jedenZ : Zesp = ( re:1; im:0 );
```

Funkcja do konwersji typów przydaje się, gdy mamy działania mieszane – jedną z liczb jest liczba zmiennoprzecinkowa. Można co prawda przed każdym działaniem pisać:

```
liczbaZ.re = liczba;
liczbaZ.im = 0;
MnoZ( z, liczbaZ );
```

ale zdecydowanie wygodniej jest użyć następującej funkcji:

```
function ReZ( x real ) : Zesp;
begin
  ReZ.re := x;
  ReZ.im := 0;
end;
```

W takim przypadku działanie, np. mnożenie liczby zespolonej `z` i rzeczywistej `liczba` wygląda następująco:

```
MnoZ( z, ReZ( liczba ) );
```

Według mniej łatwiej.

* * *

Po napisaniu całego kodu do obsługi liczb zespolonych, nasz program wyglądał będzie następująco:

```
program LiczbyZepolone;

type
  Zesp = record
    re : double;
    im : double
  end;

const
  jedenZ : zesp = ( re:1; im:0 );
  iZ     : zesp = ( re:0; im:1 );
  ...następne stałe...

function DodZ( z1, z2: Zesp ) : Zesp;
begin
  DodZ.re := z1.re + z2.re;
  DodZ.im := z1.im + z2.im
end;

...następne funkcje...

var
  liczba, wynik: Zesp;
begin
  ...
  wynik := DodZ( liczba, iZ );
  WriteLn( 'Wynik wynosi: ', Wynik.re, ', ', Wynik.im );
  ...
end.
```

Napracowaliśmy się, ale pomyślmy co trzeba będzie zrobić, by wykorzystać obsługę liczb zespolonych w następnym programie. Najprościej wystarczy skopiować kod z definicją typu `Zesp` i treścią napisanych funkcji. Pamiętamy jednak, że kopiowanie kodu jest najgorszą techniką programowania! Kiedy okaże się, że gdzieś jest błąd, trzeba go będzie poprawiać, we wszystkich skopiowanych miejscach. Dopisywanie nowych funkcji do zestawu (np. na obliczanie kwadratu) spowoduje, że dostaniemy kilka wersji naszego zbioru funkcji.

Podsumowując poprzedni akapit: zamiast wielu miejsc z tym samym zestawem typów, funkcji i procedur, lepiej mieć jeden komplet wykorzystywany wszędzie. Między innymi po to są właśnie moduły.

Jak tworzy się moduł?

Język pascal umożliwia tworzenie własnych modułów. Jeśli więc często używamy jakichś podprogramów i typów możemy stworzyć moduł, a następnie zamieścić je w nim.

Struktura modułu jest następująca:

```
unit NazwaModułu;

interface
  uses Nazwy_modułów; {jeśli nasz moduł wykorzystuje inne moduły}
  Deklaracje_typów;
  Deklaracje_stałych;
  Nagłówki_procedur_i_funkcji;

implementation
  Deklaracje_potrzebne_na_rzecz_modulu;
  Definicje_procedur_i_funkcji;

end.
```

Słowo kluczowe `unit` oznacza nagłówek modułu. Dzięki temu kompilator wie, że zamiast programu ma skompilować moduł.

Część modułu po słowie kluczowym `interface` służy do definiowania typów, stałych oraz nagłówków procedur i funkcji. One wszystkie będą dostępne w programie, w którym zadeklarujemy dany moduł. Natomiast część kodu po słowie `implemenation` zawiera deklaracje stałych, zmiennych i typów, które będą dostępne tylko w obrębie modułu. Należy tam umieścić kod podprogramów, zadeklarowanych w części `interface`. W części implementacyjnej można też umieścić inne podprogramy, ale nie będą one dostępne na zewnątrz modułu. Moduł – podobnie jak program – kończymy słowem `end.` (z kropką).

Wydzielanie kodu do modułu

Zobaczmy w praktyce, jak podzielić kod. Najpierw tworzymy plik modułu. Powinien nazywać się tak samo jak nazwa modułu czyli na przykład dla `unit zespolone` będzie to plik `zespolone.pas`:

```
/* ****** plik zespolone.pas  ****** */
unit Zespolone;

interface
{definicja typu}
type
  Zesp=record
    re :double;
    im :double
```

```
  end;
{stałe}
  const jedenZ : Zesp = ( re:1; im:0 );
  const zeroZ  : Zesp = ( re:0; im:0 );
  const iZ     : Zesp = ( re:0; im:1 );
{deklaracje funkcji}
  function DodZ( z1, z2: Zesp ) : Zesp;
  function OdjZ( z1, z2: Zesp ) : Zesp;
  ...

implementation
{treści funkcji}
function DodZ( z1, z2: Zesp ) : Zesp;
begin
  DodZ.re := z1.re + z2.re;
  DodZ.im := z1.im + z2.im
end;
...

end.
```

Kompilacja tego pliku spowoduje utworzenie dwóch plików: `zespolone.o` i `zespolone.ppu`. Oba pliki będziemy dołączać do każdego projektu, który będzie korzystał z napisanych w module funkcji i typów. Jedną z możliwości jest przekopiowanie obu do katalogu z naszym programem.

Teraz we właściwym programie deklarujemy jedynie chęć korzystania z modułu zawierającego odpowiednie procedury. Robimy to za pomocą słowa `uses`:

```
/* ****** plik program.pas ****** */
program LiczbyZepolone;
uses zespolone;

var
  licz1, licz2, wynik: Zesp;
begin
  ...
```

I to wszystko.

Uwagi

Kod moduł podzielony jest na interfejs i implementację. Interfejs stanowi specyficzny „spis treści”. Zauważmy, że zostały tam wpisane jedynie nagłówki funkcji/procedur, czyli ich „pierwsze linijki”. Deklaracje te będą widoczne dla każdego pliku `.pas` któremu wpiszemy `uses zespolone;`. Dobrym zwyczajem jest dobrze opisać komentarzami część interfejsu, tak by stała się jednocześnie dokumentacją napisanego przez nas kodu. Każdy kto będzie korzystał z naszego modułu, powinien mieć dostęp do w ten sposób utworzonej dokumentacji.

Ponieważ mamy możliwość definiowania stałych i typów, zwróćmy uwagę, żeby nie dopuścić do dublowania nazw. Przecież kolega programista używający naszego modułu, nie musi dokładnie znać jego treści i przypadkiem utworzy sobie typ czy zmienną o nazwie, która już istnieje. Można się przed tym nieco zabezpieczyć stosując odpowiednią konwencję nazewniczą – w powyższym przykładzie nazwy kończyły się literą Z.

Zadania

Znając funkcjonalność liczb zespolonych, zaproponuj nowe wersje programów do wyliczania trójek pitagorejskich i rozwiązywania równania kwadratowego. W tym drugim przypadku ujemna delta nie będzie przeszkodą w rozwiązaniu równania.

* * *

W przypadku rozpatrywania układów biernych RLC oblicza się tzw. zawadę układu (bardziej uczenie mówią na to moduł impedancji). Najlepiej rozpatrzyć to na przykładzie:

kiedy do takiego układu podłączymy stałe napięcie, prąd nie popłynie – tak jakby układ miał nieskończony opór. Kiedy podłączymy napięcie zmienne kondensator będzie się ładował i rozładowywał w takt zmian napięcia i przez układ prąd popłynie. Im większa częstotliwość napięcia, tym prąd będzie płynął łatwiej. Można więc stosować coś w rodzaju prawa Ohma:

$$I = U/|Z|$$

gdzie U jest napięciem skutecznym, I natężeniem skutecznym prądu, a $|Z|$ to właśnie zawada.

Jak obliczyć zawadę? Każdemu elementowi przypisuje się odpowiedni zespolony „opór”:

- opornikowi rzeczywisty opór R;
- kondensatorowi urojony $-i/(\omega C)$
- indukcyjności też urojony $i\omega L$.

Liczba ω oznacza tzw. częstość i jest „prawie” tym samym co częstotliwość prądu f: $\omega = 2\pi f$. Potem dodajemy do siebie jak zwykłe opory zgodnie ze wzorami na obliczanie oporu zastępczego. Uzyskamy w ten sposób liczbę zespoloną Z. Zawada układu jest wartością bezwzględną (modułem) z tej liczby: $|Z|$.

Dla przykładowego obrazka (tego powyżej):

$$Z = R - \frac{i}{\omega C}$$

Przyjmując stałe wartości R, L i C wygeneruj dane (jak w zadaniu na str. 116) do wykresu zależności $I(\omega)$. Nazywa się to charakterystyka układu RLC.

* * *

Zadanie nie związane z liczbami zespolonymi: Utwórz moduł zawierający zestaw funkcji/procedur do tworzenia plików HTML (str. 108).

3.6 W epoce pisma obrazkowego

Skoro poznaliśmy liczby zespolone, to spróbujmy wygenerować sobie żuka Mandelbrota. To fraktal utworzony w laboratoriach IBM, o którego istnieniu świat został poinformowany przez B. Mandelbrota w 1982 roku. Kto go naprawdę wymyślił, trudno powiedzieć. Więcej na temat autorstwa jak i samego zbioru znajdziemy w Wikipedii.

Zbiór ten jest podzbiorem liczb zespolonych, a uzyskuje się go następująco: Dla każdej liczby zespolonej p sprawdza się czy ciąg:

$$z_0 = p, \qquad z_{n+1} = z_n^2 + p$$

jest rozbieżny. To znaczy nie można sprawdzić wszystkich liczb, więc wybiera się punkty z jakiejś siatki. Podobnie nie można sprawdzić rozbieżności, bo trzeba by przejść z kolejnymi iteracjami – zgodnie z powyższym wzorem – nieskończenie wiele razy. Ustala się więc maksymalną liczbę iteracji i jeśli kolejne wyrazy ciągu nie przekroczą $|z_n| > 2$, to uznaje się, że punkt p z dużym prawdopodobieństwem należy do zbioru.

Pliki graficzne – fcl-image

Język pascal jako taki nie umożliwia tworzenia plików graficznych, chyba że wczytamy się w specyfikację któregoś z nich i sami oprogramujemy co trzeba. Istnieje jednak wiele bibliotek zewnętrznych, to znaczy takich, w których ktoś zadał sobie trud studiowania specyfikacji i napisał kod za nas. Na stronie z dokumentacją: `https://wiki.freepascal.org/Graphics_libraries` podanych jest

kilka gotowych rozwiązań, jakie możemy używać, programując w pascalu. Przejrzałem je i doszedłem do wniosku, że chyba najłatwiejszą biblioteką do zastosowania jest fcl-image. Do najważniejszych zalet należy fakt, że nie trzeba jej instalować, bo dostępna są w standardowej instalacji Free Pascala.

Poniżej kilka przykładów, które pomogą nam rozeznać się w specyfice fcl-image. Jest ona o tyle dla nas nowa, że jawnie używa rozwiązań obiektowych o których nic do tej pory nie pisałem, ale być może będzie właśnie okazją do oswojenia się z tym tematem. Przeanalizujmy więc jak działa owa biblioteka.

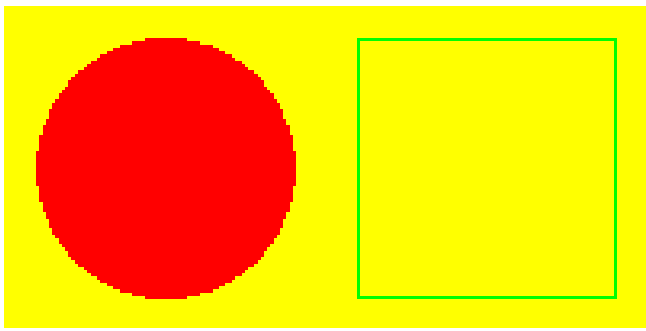
Rysunki tworzy się w oparciu o dwa związane ze sobą obiekty. Pierwszy z nich zarządza pamięcią (typ `TFPMemoryImage`), no bo dane graficzne gdzieś muszą być przechowywane. Zwyczajowo zmienne tego typu nazywa się obrazami (image). Drugi służy do zarządzania treścią rysunku (typ `TFPImageCanvas`). To do tego obiektu zgłaszamy chęć umieszczenia zielonego kwadratu na tle niebieskiego kółka. Zmienne tego typu nazywa się zwykle płótnem (canvas). Z płótnem związane są dwa pola: pióro (pole `Pen`) i pędzel (pole `Brush`). Pierwszy odpowiada za rysowanie linii, drugi za wypełnianie kolorem powierzchni.

W poniższym programie przedstawiłem kilka procedur malujących po płótnie. Uwagę mogą zwrócić przypadki, gdy funkcja/procedura zachowuje się jak pole rekordu. Tak jest na przykład w linii `canvas.Ellipse( 10, 10, 90, 90 );` czy `img.Free;`. Jest to jedna z cech programowania obiektowego – funkcja/procedura może być polem rekordu, który wtedy nazywa się obiektem. Mam nadzieję, że względna prostota przykładu pozwoli na korzystanie z niego. Na razie bez wgłębiania się w tajniki obiektologii.

```
program ProsteFiguryPNG;
uses
  FPImage, FPImgCanv, FPCanvas, FPWritePNG;
var
  img: TFPMemoryImage;
  canvas: TFPImageCanvas;
begin
  { utworzenie głównego obiektu obrazu o wymiarach 200x100 }
  img := TFPMemoryImage.Create( 200, 100 );
  { utworzenie płótna i związanie go z obrazem }
  canvas := TFPImageCanvas.Create(img);
  { najpierw tło }
  canvas.Brush.FPColor := colYellow;
  canvas.FillRect( 0, 0, img.Width, img.Height );
  { coś narysujmy - najpierw bez ramki... }
  canvas.Pen.Style := psClear;
  canvas.Brush.FPColor := colRed;
  canvas.Ellipse( 10, 10, 90, 90 );
  { ...potem bez wypełnienia }
  canvas.Pen.Style := psSolid;
  canvas.pen.FPColor := colGreen;
  canvas.Brush.Style := bsClear;
```

```
  canvas.Rectangle( 110, 10, 190, 90 );
  { na koniec zapis do pliku }
  img.SaveToFile( 'figury.png' );
  { zwalnianie pamięci - to trzeba robić ręcznie }
  canvas.Free;
  img.Free;
end.
```

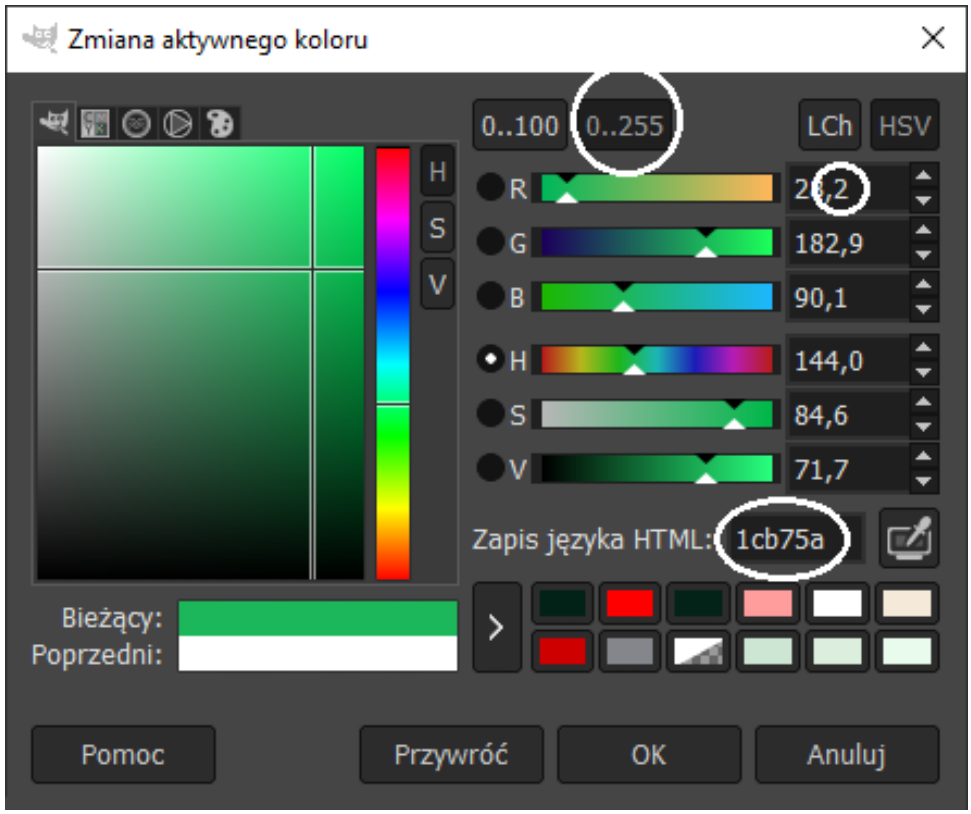
Komentarze powinny wyjaśnić co się maluje. Zwróć uwagę na listę dołączonych modułów. Po nazwie można się zorientować, jaką funkcjonalność oferują. Obrazek który powstaje po uruchomieniu programu jest następujący:



Lista stałych typu `TFPColor` oznaczających kolory dostępna jest na stronie: `https://wiki.freepascal.org/Colors` Jeśli jednak chcemy tworzyć własne kolory, to sprawa jest bardziej skomplikowana niżby się wydawało.

Kolory w fcl-image

Być może jesteśmy przyzwyczajeni do zapisywania koloru za pomocą trzech bajtów RGB. Ewentualnie dodajemy czwarty oznaczający stopień przezroczystości – na razie odpuszczę sobie jej obsługę, ograniczając się do trzech składowych. Jeden bajt na jedną składową daje nam głębię 24-bitową. Format taki wykorzystywany choćby na stronach HTML. Kiedy jednak zajrzymy do jakiegoś programu graficznego:



to zauważymy części dziesiętne. Widoczna powyżej wartość szesnastkowa `1cb75a` to dziesiętnie: 28/183/90. Biblioteka fcl-image również umożliwia osiągnięcie większej głębi i na każdą składową poświęca dwa bajty. Można w łatwy sposób przerobić wartość jednobajtową na dwubajtową, przesuwając ją binarnie o osiem miejsc:

np.: `$A5` → `$A500` (zapis heksagonalny)

czyli przemnożyć przez 256.

Kolory w fcl-image można składać samemu. W tym celu deklarujemy zmienną typu `TFPColor` i wpisujemy składowe. Poniższy oznacza czerwień o największej głębi kanału R:

```
czerwony := FPColor( $FFFF, 0, 0 );
```

Funkcja `FPColor` potrzebuje trzech wartości typu `word` czyli dwubajtowych liczb bez znaku. Tu akurat wpisane są w postaci heksagonalnej, bo dla takiej postaci lepiej widać zawartość binarną.

Zanim przetestujemy w praktyce składanie kolorów, pora dowiedzieć się, jak zamalowywać pliksel po pikselu. Nie ma w tej bibliotece czegoś w rodzaju `PutPixel`. Kolorowanie odbywa się przez dostęp do dwuwymiarowej tablicy kolorów:

```
canvas.colors[i,j] := colRed;
```

Trzeba jedynie pamiętać, że jeśli obrazek ma wymiary `Szer`×`Wys`, to dostępne współrzędne mają zakresy – poziomo: `0..Szer-1`, pionowo: `0..Wys-1`.

Przedstawiam do przetestowania poniższy kod, w którym można uzyskać gradowane kolory z „tradycyjną” jednobajtową głębią. Użyta w programie stała `Glebia` równa jest 256:

```
img := TFPMemoryImage.Create( Glebia, Glebia );
canvas := TFPImageCanvas.Create(img);
for i:=0 to Glebia-1 do
  for j:=0 to Glebia-1 do
  begin
    green := i*Glebia;
    red := j*Glebia;
    kolor := FPColor( red, green,  0 );
    canvas.colors[i,j] := kolor;
  end;
img.SaveToFile('kolorki.png');
canvas.Free;
img.Free;
```

Uzyskany obrazek.

Wracamy do fraktala

Nasze zadanie jest następujące:

- Wybieramy jakiś prostokąt z płaszczyzny zespolonej.
- Dla zadanej rozdzielczości dzielimy prostokąt na sieć, tak by jeden punkt sieci odpowiadał jednemu pikselowi planowanego obrazka.
- Dla każdego punktu p z sieci przeprowadzamy iterację wyrażenia: $z_{n+1} = z_n^2 + p$. Kończymy, gdy aż $|z| > 2$ (to będą kolorowe punkty) lub gdy zaplanowana liczba iteracji nie wyjdzie poza ten zakres (punkty fraktala – kolorowane zwyczajowo na czarno). Małe ułatwienie: ponieważ w celu wyliczenia

$|z|$, najpierw liczymy kwadrat modułu, a potem go pierwiastkujemy, można sprawdzać warunek $|z|^2 > 4$. Nie trzeba wtedy liczyć pierwiastka.

- Skutkiem powyższego będzie liczba dokonanych iteracji dla każdego p.

Podzielmy sobie nasz program na kilka podprogramów:

- `inicjalizacja()` – na podstawie danych początkowych, zostaną zainicjalizowane wszystkie potrzebne zmienne;
- `iteracja()` – wyliczenie pojedynczej iteracji;
- `wyliczenie()` – wyliczenie całości (`iteracja()` może być procedurą wewnętrzną);
- `rysunek()` – wyliczone dane skierowane zostaną do zapisania na dysku w postaci rysunku.

Na początek spakuję całość potrzebnych danych do osobnego typu:

```
type
  TPunkty = array of array of integer;
  TZukMand = record
    RozX, RozY : integer;
    xp, xk, yp, yk : double;
    IleIter : integer;
    punkty : TPunkty;
    skala  : double;
  end;
```

Typ `TPunkty` oznacza dynamiczną tablicę dwuwymiarową. Zdaje się, że nie była stosowana w dotychczasowej treści kursu.

Do inicjalizacji będą potrzebne: wymiary prostokąta, liczba punktów w poziomie (lub pionie) i maksymalna liczba iteracji. W sumie sześć zmiennych. Można przekazać je do procedury `inicjalizacja()` z dodatkowym siódmym argumentem typu `ZukMand`. Można też – i tak zostanie to poniżej zrobione – wstępnie nadać te wartości rekordowi z danymi, który byłby w takim ujęciu jedynym argumentem procedury inicjalizacyjnej.

```
procedure inicjalizacja( var zuk : TZukMand );
{ zakładamy, że współrzędne x i y, oraz RozX są poprawnie określone }
begin
  with zuk do
  begin
    skala := (xk-xp)/RozX;
    RozY := round( (yk-yp)/skala );
    SetLength( punkty, RozX, RozY );
  end;
end;
```

Pojedyncza iteracja wymaga istnienia zestawu funkcji do obróbki liczb zespolonych (tu oznaczanych jako `Zesp`) – potrzebne będą mnożenie `MnoZ()`, dodawanie `DodZ()` i kwadrat modułu `KModulZ()`. Dzięki nim, całą matematyczną wiedzę, potrzebną dla tego zagadnienia, zmieścimy w dwóch linijkach kodu.

```
function iteracja( x, y : double; IleIter : integer ) : integer;
var
  i : integer;
  punkt, biez : Zesp;
begin
  punkt.re := x;
  punkt.im := y;
  biez := punkt;
  for i:=1 to IleIter do
  begin
    biez := DodZ( MnoZ( biez, biez ), punkt );
    if( KModulZ( biez )>4 ) then exit( i );
  end;
  iteracja := 0;
end;
```

Funkcja zwraca liczbę wykonanych iteracji. Jednak jeśli jednak wykonano wszystkie i moduł jest wiąż mniejszy od 2, to ustawiłem zwracaną wartość na 0. Oznacza to, że punkt (x, y) jest prawdopodobnie punktem fraktala. W każdym innym przypadku funkcja zwróci wartość większą od 0.

> Uwaga, moment niewychowawczy: Kiedy będziemy robić kolejne powiększenia żuka, potrzebnych będzie coraz więcej iteracji. Okaże się wtedy, że wywoływanie funkcji: `MnoZ()`, `DodZ()` i `KModulZ()` zajmuje najwięcej zasobów. Przepisanie instrukcji `DodZ( MnoZ( biez, biez ), punkt )` na dwie współrzędne typu `double` przyspieszy działanie programu. Ale nie ma nic za darmo. Zdecydowanie łatwiej się wtedy pomylić – trzeba sobie dokładnie rozpisać, jak będą wyglądać współrzędne punktu po iteracji.

Pojedynczą iterację wkładamy w podwójną pętlę:

```
procedure wyliczenie( var zuk : TZukMand );
var i, j : integer;
   x, y : double;
begin
  with zuk do
  begin
    for j:=0 to RozY-1 do
      for i:=0 to RozX-1 do
      begin
        x := xp + i*skala;
        y := yk - j*skala;
        punkty[i,j] := iteracja( x, y, IleIter )
```

```
      end;
  end;
end;
```

Dzięki funkcjonalności `with` kod nie wymaga jawnego sięgania do struktury `zuk`, przy korzystaniu z jej pól. To przyjemna cecha języka pascal. Zauważmy też różnicę w wyliczaniu współrzędnej X-owej i Y-owej. ZTS – dlaczego tak jest?

Tu jeszcze jedna uwaga na temat zagnieżdżania procedur. Gdybyśmy potraktowali funkcję `iteracja()` jako wewnętrzną względem `wyliczenie()`, to można by pokusić się o zaplanowanie „wspólnych" zmiennych `x` i `y` dla obu podprogramów:

```
procedure wyliczenie( var zuk : TZukMand );
var x, y : double;

  function iteracja( IleIter : integer ) : integer;
  ...
    punkt.re := x;
    punkt.im := y;
  ...
  end;

var i, j : integer;
begin
  ...
    x := xp + i*skala;
    y := yk - j*skala;
    punkty[i,j] := iteracja( IleIter )
  ...
end;
```

Nie wiem czy taki sposób jest bardziej czytelny. Pewnie zależy to od przyzwyczajeń i umiejętności. Osoby „wychowane" na C++ mogą się krzywić, że to jednak zakamuflowana funkcjonalność zmiennych globalnych. Z drugiej strony takie użycie zmiennych `x` i `y` skraca listę argumentów, przez co (według mnie) zwiększa przejrzystość kodu, choć wymaga nieco większej dyscypliny.

Jak już wyliczymy sobie liczbę iteracji we wszystkich punktach, trzeba będzie zapisać te dane w postaci obrazka. Jedyna trudność, to zamienić liczbę iteracji na trzy składowe koloru typu `word`. Najlepiej przygotować sobie kilka takich funkcji i podstawiać je do procedury tworzącej obrazek. Potrzebny typ funkcyjny:

```
type   TFkol = function( nr : integer ) : TFPColor;
```

natomiast sama procedura wygląda następująco:

```
procedure rysunek(const zuk :TZukMand; nazwa :string; f :TFkol);
var i, j : integer;
```

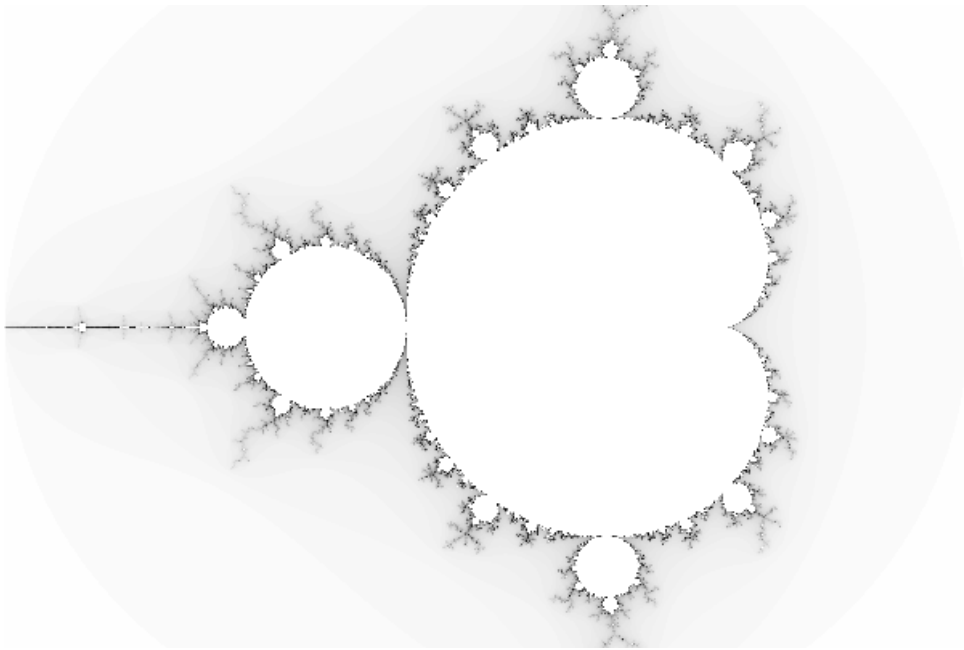
```
  img: TFPMemoryImage;
  canvas: TFPImageCanvas;
begin
  img := TFPMemoryImage.Create( zuk.RozX, zuk.RozY );
  canvas := TFPImageCanvas.Create(img);
  for j:=0 to zuk.RozY-1 do
    for i:=0 to zuk.RozX-1 do
      canvas.colors[i,j] := f( zuk.punkty[i,j] );
  img.SaveToFile( nazwa );
  canvas.Free;
  img.Free;
end;
```

Zauważmy tu pewien przejaw zasady separacji kodu: Wszystkie odwołania do biblioteki graficznej fcl-image zawarte są w procedurze `rysunek()` oraz funkcjach typu `TFkol`. Podobnie obsługa liczb zespolonych zamknięta jest w kodzie `iteracja()`. Tak więc gdybyśmy w pewnym momencie zapragnęli zastosować inną bibliotekę obrazkową, wiemy, że musimy dokonać zmian w jednym miejscu programu. Reszta kodu jest niezależna od tych zmian.

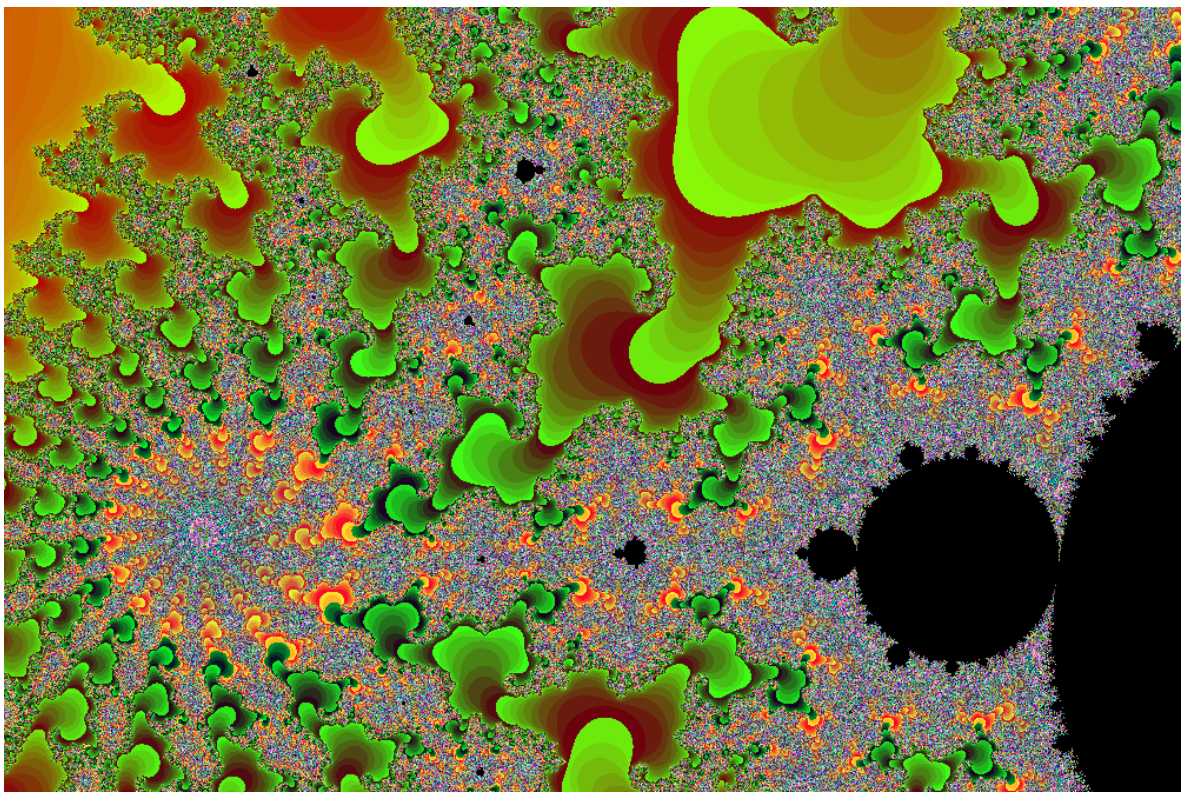
Pewnie się niecierpliwicie, kiedy w końcu pokażę jakieś obrazki. No to dla prostej funkcji `BWkolor()` – domyślacie się, że produkuje różne stopnie szarości – i dla danych jak niżej:

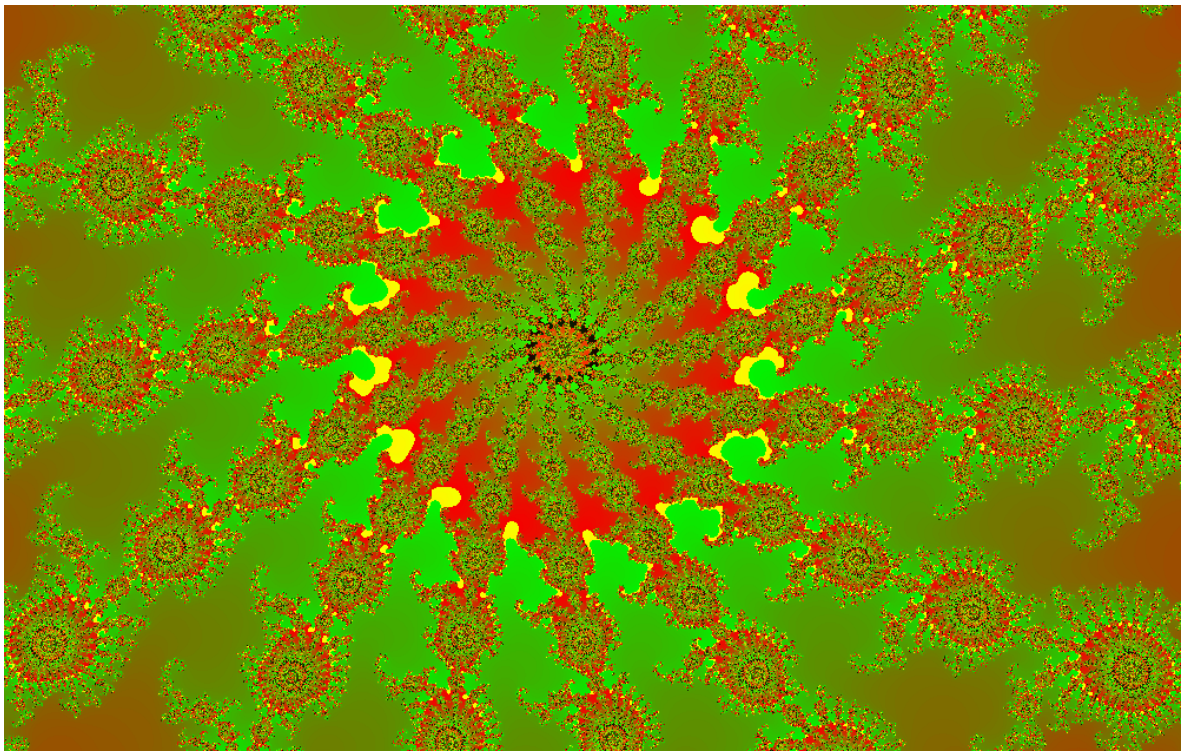
```
var zuk : TZukMand;
BEGIN
   zuk.xp := -2;
   zuk.xk := 1;
   zuk.yp := -1;
   zuk.yk := 1;
   zuk.RozX := 600;
   zuk.IleIter := 200;
   inicjalizacja( zuk );
   wyliczenie( zuk );
   rysunek( zuk, 'zuczek.png', @BWkolor );
END.
```

program główny wygeneruje obrazek:



Startując z przedstawionego punktu wyjściowego będziemy stopniowo ograniczali współrzędne prostokąta, czyli stosowali coraz większe powiększenia. Oczywiście, żeby dostać coś ciekawego, będziemy operować blisko punktów brzegowych zbioru Mandelbrota. Okaże się wtedy, że trzeba zwiększyć liczbę iteracji i napisać dodatkowe funkcje produkujące kolory. Nawet dość przypadkowe funkcje, które na podstawie liczby całkowitej utworzą trzy dwubajtowe, nietrywialne składowe RGB dają całkiem przyjemne efekty:





Kolorowe przesunięcia bitów

Na zakończenie odcinka kilka obrazków z bardziej przemyślaną funkcją tworzącą kolor. Zanim jednak pokażę kod tej funkcji, koniecznie trzeba tu powiedzieć o operatorach przesunięcia bitowego. Na stronie 128 mamy przykład kilku operatorów bitowych, poniżej zapoznamy się z dwoma kolejnymi.

Jeśli spojrzymy na binarną postać liczb 10 i 20 – zauważymy, że w zapisie dwójkowym obie mają te same cyfry:

$$10_{10} = \ldots 00001010_2 \qquad 20_{10} = \ldots 00010100_2$$

ale przesunięte o jedną pozycję. Innymi słowy – mnożenie przez 2 może być zrealizowane przez odpowiednie przesunięcie bitów. W pascalu obywa się ono za pomocą operatora `shl`

```
wartosc := wartosc shl 1;
```

Jak łatwo się domyślić, drugi argument operatora `shl` mówi o ile bitów trzeba przesunąć cyfry. Przesunięcie w prawo, równoznaczne z dzieleniem całkowitym przez potęgę dwójki, wygląda następująco:

```
wartosc := wartosc shr 1; // dzielenie przez 2
wartosc := wartosc shr 3; // dzielenie przez 8
```

Reszta z dzielenia – czyli ostatnie bity – jest rzecz jasna gubiona przy tych działaniach. Te „ostatnie” bity nazywane są najmłodszymi lub najmniej znaczącymi

bitami. Po lewej stronie zapisanej liczby, mamy bity najstarsze albo najbardziej znaczące.

Zwykle używanie operatorów przesunięć nie jest kłopotliwe, ale trzeba pamiętać o kilku rzeczach. Operatory `shl` i `shr` zwracają liczbę czterobajtową. Wynika z tego, że przesuwanie w lewo ma sens co najwyżej o 31 pozycji. Dla większych przesunięć `shl` może cię zaskoczyć. Dla typów `QWord` i `Int64`, wynik będzie ośmiobajtowy, zgodnie z tymi typami.

Żeby móc korzystać z przesuniętej liczby, trzeba ją przypisać do jakiejś zmiennej. Typowym błędem początkujących programistów jest intuicja, że komenda

```
liczba shl 1;
```

powoduje przesunięcie bitów w zmiennej `liczba`. Oczywiście to nieprawda, bo żeby zmienić zmienną, powinniśmy napisać:

```
liczba := liczba shl 1;
```

Skoro pojawił się taki zapis – musimy uważać, żeby nie nastąpiło przekroczenie zakresu, bo pojawi się błąd:

```
Runtime error 201 at ...
```

Oznacza to zwykle, że nie przeanalizowaliśmy problemu do końca. Np. posługujemy się dwubajtowymi zmiennymi typu `word`, i w wyniku przekształceń bitowych wyszliśmy poza dwa młodsze bajty (starsze stały się różne od zera). Błędu przekroczenia nie będzie, jeśli zrzutujemy uzyskaną wartość:

```
liczba := word( liczba shl 1 );
```

ale powinna to być świadoma decyzja, bo oznacza potencjalną utratę najbardziej znaczących bitów – w powyższym przypadku jeden najstarszy bit zostanie obcięty.

Skoro mamy już arsenał narzędzi, pora na zastosowanie ich do naszego programu. Funkcja tworząca kolor, dostaje na wejściu liczbę wykonanych iteracji, i wylicza z niej trzy liczby dwubajtowe. Istotna informacja w składowych RGB umieszczona jest w najstarszych bitach. Z drugiej strony, jeśli maksymalna liczba iteracji jest mniejsza od kilku tysięcy, to te najstarsze bity w przekazywanej zmiennej są równe zero. Pomysł polega na przesunięciu młodszych bitów argumentu w lewo w celu utworzenia składowych RGB.

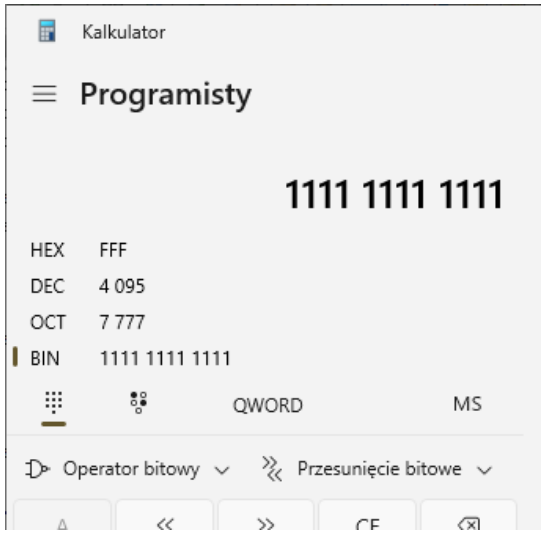
Żeby nie przekroczyć zakresu przy przesunięciu, najpierw przemnożymy binarnie naszą liczbę przez maskę obcinającą starsze bity:

$$\texttt{XXXXXXXX XXXXXXXX and } 00001111\ 11111111 \rightarrow 0000\texttt{XXXX XXXXXXXX}$$

a potem przesuniemy:

$$0000\texttt{XXXX}\ \texttt{XXXXXXXX}\ \texttt{shl}\ 4 \rightarrow \texttt{XXXXXXXX}\ \texttt{XXXX}0000$$

Dla trzech składowych potrzebujemy trzech masek – po wycięciu będą zostawiać 8, 10 i 12 młodszych bitów, a potem przesuwać je odpowiednio o 8, 6 i 4 miejsca. Sposób ten zapewni, że nie przekroczymy zakresu 16 bitów. Maski wyliczymy sobie zwykłym windowsowym kalkulatorem w trybie programisty:

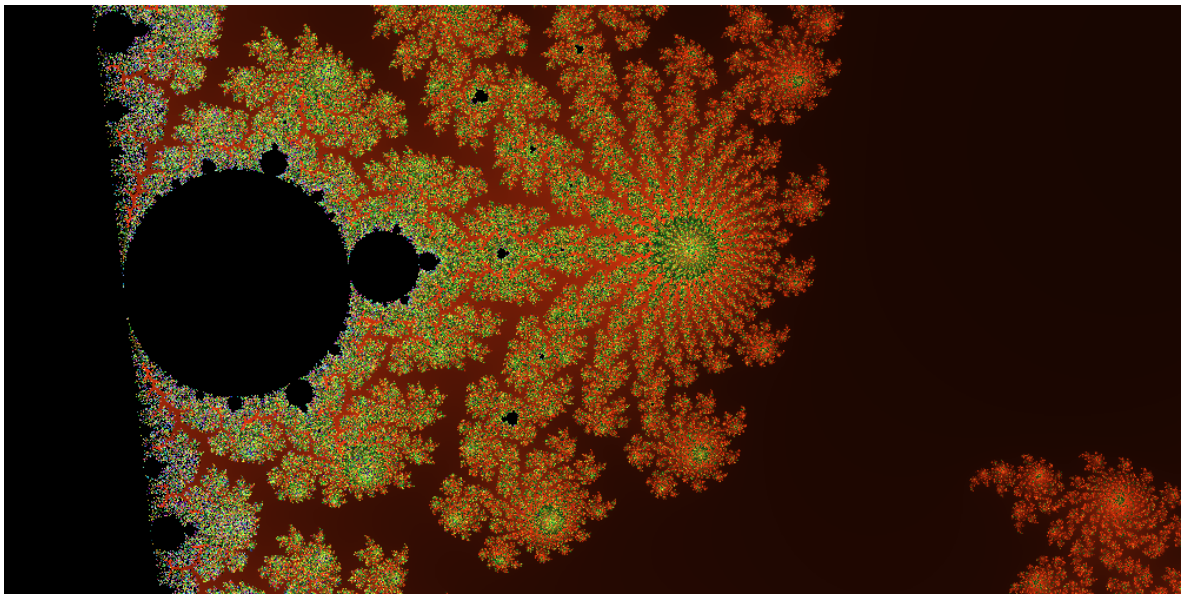


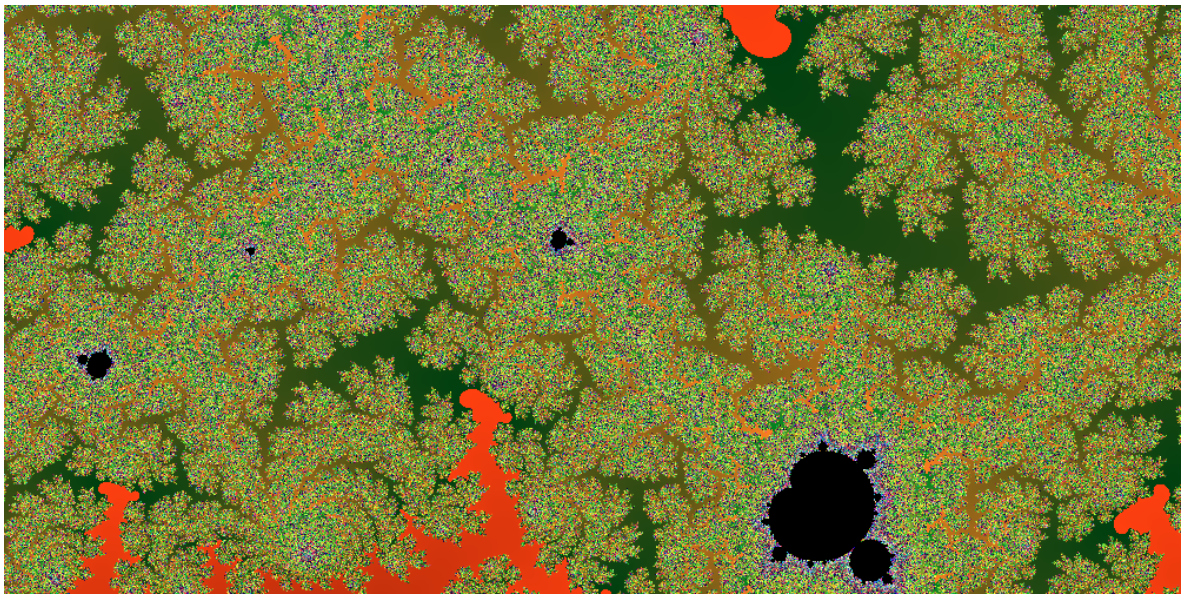
W programie możemy zapisać ich wartości dziesiętne: 255, 1023 i 4095, ale dla podkreślenia, że chodzi o maskę, treściwiej i czytelniej użyć wartości heksagonalnych: `$00FF`, `$03FF` i `$0FFF`.

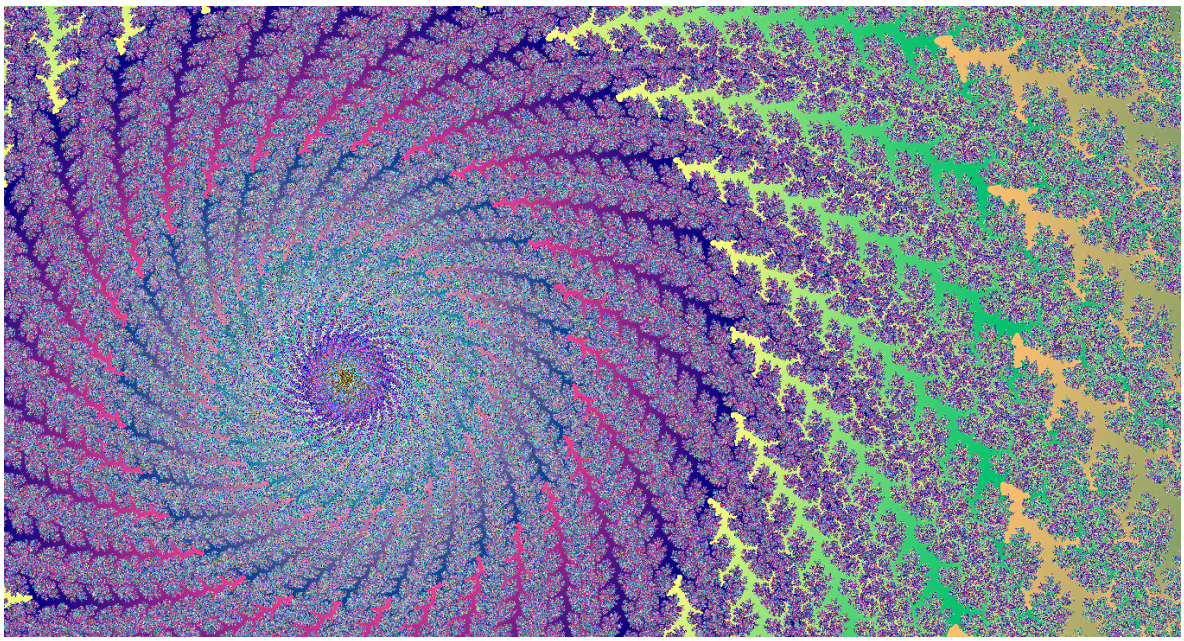
A oto i sama funkcja:

```
function Kolorowo( nr : integer ) : TFPColor;
const maska1 = $00FF;
      maska2 = $03FF;
      maska3 = $0FFF;
var r, g, b : word;
begin
  if nr=0 then exit( FPColor(0, 0, 0) );
  r := ( nr and maska1 ) shl 8;
  g := ( nr and maska2 ) shl 6;
  b := ( nr and maska3 ) shl 4;
  Kolorowo := FPColor( r, g, b );
end;
```

oraz wygenerowane obrazki zaglądające wgłąb zbioru:







* * *

Nasz zbiór składa się z punktów o współrzędnych, będących liczbami zmiennoprzecinkowymi. Nie zajmowałem się ich możliwą dokładnością, bardziej po ludzku zwaną „liczbą miejsc po przecinku". Użyty typ `double` ma zarezerwowane 52 bity na przechowywanie mantysy liczby. Przekłada się to mniej-więcej na 15 miejsc znaczących w systemie dziesiętnym.

$$\text{dokładność}_{10} \approx \log_{10}(2^{52}) \approx 15,6$$

Ponieważ nasze liczby są bliskie 1, oznacza to dosłownie ok. 15 miejsc po przecinku. Osiąganie większych rozdzielczości jest niemożliwe.

3.7 Parametry wywołania programu

Często się zdarza, że wywołując program z linii poleceń podajemy jakieś parametry. Chyba każdy z nas uruchamiał polecenie `ping`:

```
ping 192.168.1.100
```

Napisany numer IP jest właśnie takim parametrem, które program `ping.exe` (tak się nazywa w Windows) pobiera sobie i używa przy uruchomieniu.

Poniżej przykład prawie najprostszego kodu obsługującego podane parametry.

```
program ParametryProgramu;
var i:integer;
BEGIN
  if ParamCount=0 then
  begin
    WriteLn( 'Program ',ParamStr(0), ' potrzebuje parametrów!' );
    halt( 1 ); { dostępne przez %errorlevel% }
  end;
  WriteLn( 'Program ',ParamStr(0),' ma ',ParamCount,' param.:' );
  For i:=1 to ParamCount do WriteLn( i:3, ' ', ParamStr(i) );
END.
```

Widzimy, że wyrażenia `ParamCount` i `ParamStr()` nie są nigdzie deklarowane w tekście programu. Ale mimo to program może w magiczny sposób z nich korzystać. Pierwsze oznacza liczbę podanych parametrów, drugie to funkcja, która zwraca i-ty parametr. `ParamStr(0)` zwraca nazwę wywoływanego programu.

```
D:\katalog>parametry pierwszy drugi trzeci
Program D:\katalog>parametry.exe ma 3 param.:
  1 pierwszy
  2 drugi
  3 trzeci
```

Prawie praktyczny przykład

Może się to wydawać dziwne, ale nawet w dzisiejszych czasach pisanie programów uruchamianych w trybie tekstowym ma sens. Chodzi o programy działające bez interakcji z użytkownikiem. Dobrym przykładem będą tu wszelkie prace wywoływane przez skrypty, np. uruchamiane w nocy backupy czy automatyczne konfiguracje. Administrator przygotowuje zestaw programów do wykonania, a skrypt wywołuje je jeden po drugim. Jeśli jednak wystąpi błąd i nie uda się zakończyć z sukcesem działania któregoś programu, to nie ma sensu uruchamiać kolejnych. Błędy wypisywane będą w poznanym na stronie 88 `StdErr`, który zwyczajowo przekierowuje się do pliku z logami. Po wykonaniu zadania, administrator sprawdzi w logach, co się nie udało.

Jeśli chodzi o tego typu programy, dobrze byłoby, gdyby program przed przystąpieniem do właściwej pracy, sprawdził czy wszystkie dane są w dobrym stanie. Jeśli coś nie gra, program wypisuje komunikat i kończy działanie. Poniżej jedna z kilku możliwości, jak sprawdzić poprawność wpisanych opcji programu. Oczywiście poniższemu programowi daleko do typowych programów uruchamianych przez administratorów – w końcu będzie on realizował tylko jedno – i to bardzo prościutkie – zadanie.

Program ma wypisywać kilka razy dane słowo. Zarówno słowo jak i liczba powtórzeń mają być podane przez użytkownika. Komenda:

```
ilerazy -n Adam -s 4
```

powinna zaowocować:

```
Adam
Adam
Adam
Adam
```

Od razu opiszemy tę funkcjonalność w poniższej instrukcji użytkownika:

```
procedure Info;
begin
  WriteLn( StdErr, 'Uruchomienie␣programu:' );
  Writeln( StdErr, NazwaProg(ParamStr(0)),'␣-n␣liczba␣-s␣slowo');
end;
```

Żeby nie epatować użytkownika całą ścieżką (bo taka jest podana w `Param(0)`), napisałem funkcję wycinającą samą nazwę programu:

```
function NazwaProg( sciezka : string ) : string;
var poz, i : integer;
begin
  poz := 0;
```

```
  for i:=1 to length(sciezka) do
    if sciezka[i]='\' then poz := i;
  NazwaProg := copy( sciezka, poz+1, length(sciezka) - poz );
end;
```

Program ma zadziałać prawidłowo gdy zostaną podane dwie opcje z parametrami. Po opcji `-s` ma być podane słowo, a po `-n` liczba. Rozwiązanie polega na napisaniu funkcji logicznej `CzyOpcjeOK()`, która w przypadku natrafienia na jakiś błąd w podanych opcjach, przerwie działanie, zwracając `false`. Jeśli wszystko się uda zwróci `true`. Funkcja musi udostępnić do programu odczytane opcje, należy więc przygotować odpowiednią paczkę danych:

```
type
TDane = record
  nr : integer;
  slowo : string
end;
```

i przekazać ją do funkcji przez zmienną:

```
function CzyOpcjeOK( var dane : TDane ) : boolean;
var i, err :integer;
  opcja : string;
begin
  dane.nr :=0;
  dane.slowo := '';
  For i:=1 to ParamCount do
  begin
    opcja := ParamStr( i );
    if opcja[1]='-' then
    begin
      if length(opcja)<>2 then exit( false );
      if i=ParamCount then  exit( false );
      case opcja[2] of
        's' : dane.slowo := ParamStr( i+1 );
        'n' : begin
                val( ParamStr( i+1 ), dane.nr, err );
                if err<>0 then  exit( false );
              end
      else
        exit( false );
      end;
    end;
  end;
  if ParamCount<>4 then  exit( false );
  if dane.nr<1 then  exit( false );
  if length( dane.slowo )<1 then  exit( false );
  CzyOpcjeOK := true;
end;
```

Może się wydawać, że case w tym przypadku użyte zostało nieco na wyrost, ale przy dłuższej liście opcji, okazuje się dość wygodnym narzędziem.

Powyższa postać funkcji – dodajmy: słabo udokumentowana – w przypadku niepowodzenia, nie przekazuje informacji, co było źle. Po prostu opcje zostały źle podane i tyle. Można temu zaradzić, pakując do paczki danych, jakie funkcja ma przygotować, treść komunikatu. Będzie on wykorzystywany, gdy `CzyOpcjeOK()` będzie miała wartość `false`. Wtedy każdy przypadek wystąpienia błędu skutkować będzie dwiema instrukcjami: utworzeniem komunikatu i wywołaniu `exit(false)`.

Żeby osiągnąć ten cel powiększymy strukturę `TDane` o napis:

```
type
TDane = record
  nr : integer;
  slowo : string
end;
```

i zmienimy postać funkcji:

```
function CzyOpcjeOK( var dane : TDane ) : boolean;
var i, err :integer;
  opcja : string;
begin
  dane.nr :=0;
  dane.slowo := '';
  For i:=1 to ParamCount do
  begin
    opcja := ParamStr( i );
    if opcja[1]='-' then
    begin
      if length(opcja)<>2 then
      begin
        dane.komunikat:= 'Nieprawidłowe oznaczenie opcji '+opcja;
        exit( false )
      end;
      if i=ParamCount then
      begin
        dane.komunikat := 'Brak parametru dla opcji ' + opcja;
        exit( false )
      end;
      case opcja[2] of
        's' : dane.slowo := ParamStr( i+1 );
        'n' : begin
                val( ParamStr( i+1 ), dane.nr, err );
                if err<>0 then
                begin
                  dane.komunikat := 'Nieprawidłowa liczba';
                  exit( false );
                end
```

```
                end;
          'h' : begin
                  dane.komunikat := 'Pomoc dla programu';
                  exit( false );
                end
        else
          dane.komunikat := 'Nieprawidłowa opcja ' + opcja;
          exit( false );
        end;
      end;
  end;
  if ParamCount<>4 then
  begin
    dane.komunikat := 'Podano niewłaściwą liczbę opcji';
    exit( false );
  end;
  if dane.nr<1 then
  begin
    dane.komunikat := 'Brak liczby';
    exit( false );
  end;
  if length( dane.slowo )<1 then
  begin
    dane.komunikat := 'Brak słowa do wypisania';
    exit( false );
  end;
  CzyOpcjeOK := true;
end;
```

Funkcja nieco długa. Realizuje przy okazji „nieudokumentowaną” opcję `-h`, czyli „help”. W stosunku do poprzedniej wersji, łatwiej zorientować się w rodzaju błędu, bo kod zawiera treści komunikatów. Robi to jednak kosztem skomplikowania zapisu. Narzekałem, że konstrukcja `case of` może służyć do tworzenia potworkowatych konstrukcji i tu w zasadzie z taką mamy do czynienia.

Czy można jakoś poradzić sobie z takim skomplikowanym kodem? Tak. Zamiast napisu i wartości logicznej, funkcja może zwracać stałe typu wyliczeniowego . Unikamy wtedy blokowania dwóch instrukcji i z czterech linijek kodu, na powrót robi się jedna. Zmienimy więc `CzyOpcjeOK()`. W takim przypadku, trzeba też napisać funkcję, która „tłumaczyłaby” numer błędu na odpowiedni komunikat. Przerzucę wtedy funkcjonalność tworzenia komunikatów do osobnej funkcji, zgodnie z zasadą separacji funkcjonalności kodu.

Poniżej kompletna treść programu. Zobaczmy co się zmieniło w stosunku do poprzedniej, rozgadanej, wersji.

Skoro komunikat ma być tworzony niezależnie od funkcji sprawdzającej, zniknie on z rekordu `TDane`:

```
program ParametryProgramu;
```

```
type
TDane = record
  nr : integer;
  slowo : string
end;
```

ale pojawi się w specjalizowanej funkcji, która na podstawie wartości typu wyliczeniowego, podejmie decyzję, co wypisać:

```
StanOpcji = ( OpOK, OpFormat, OpBrakPar, OpZlaLiczba, OpZlaOpcja,
              OpCount, OpBrakLiczby, OpBrakSlowa, OpHelp );

function KomOpcji( stan : StanOpcji ) : string;
begin
  case stan of
    OpOK         : KomOpcji := '';
    OpFormat     : KomOpcji := 'Nieprawidłowe␣oznaczenie␣opcji';
    OpBrakPar    : KomOpcji := 'Brak␣parametru␣dla␣opcji';
    OpZlaLiczba  : KomOpcji := 'Nieprawidłowa␣liczba';
    OpZlaOpcja   : KomOpcji := 'Nieprawidłowa␣opcja';
    OpCount      : KomOpcji := 'Podano␣niewłaściwą␣liczbę␣opcji';
    OpBrakLiczby: KomOpcji := 'Brak␣liczby';
    OpBrakSlowa  : KomOpcji := 'Brak␣słowa␣do␣wypisania';
    OpHelp       : KomOpcji := 'Pomoc␣dla␣programu';
  end;
end;
```

Zaletą takiego rozwiązania jest zebranie napisów w jednym miejscu – gdy przyjdzie nam ochota je zmienić, nie trzeba ich będzie szukać rozrzuconych po kodzie długaśnej funkcji `CzyOpcjeOK()`.

Sama funkcja sprawdzająca nie zwraca już wartości logicznych, ale stałe typu `StanOpcji`. Przyznam, że takie rozwiązanie przekazuje mniej danych (brak informacji o którą opcję chodzi), ale jest zdecydowanie krótsze i czytelniejsze – pod warunkiem, że treściwie ponazywamy stałe typu `StanOpcji`.

```
function CzyOpcjeOK( var dane : TDane ) : StanOpcji;
var i, err :integer;
  opcja : string;
begin
  dane.nr :=0;
  dane.slowo := '';
  For i:=1 to ParamCount do
  begin
    opcja := ParamStr( i );
    if opcja[1]='-' then
    begin
      if length(opcja)<>2 then exit( OpFormat );
      if i=ParamCount then exit( OpBrakPar );
      case opcja[2] of
```

```
        's' : dane.slowo := ParamStr( i+1 );
        'n' : begin
                val( ParamStr( i+1 ), dane.nr, err );
                if err<>0 then exit( OpZlaLiczba );
              end;
        'h' : exit( OpHelp )
      else exit( OpZlaOpcja );
      end;
    end;
  end;
  if ParamCount<>4 then exit( OpCount );
  if dane.nr<1 then exit( OpBrakLiczby );
  if length( dane.slowo )<1 then exit( OpBrakSlowa );
  CzyOpcjeOK := OpOK;
end;
```

Nawiasem mówiąc, taki rodzaj zgłaszania błędów ma już dużo wspólnego z funkcjonalnością tzw. wyjątków, które pojawią się w kolejnych modułach.

Do tego dodajmy pomoc programu. Funkcja obcinająca, została potraktowana jako funkcja wewnętrzna, co minimalnie zmieniło sposób jej użycia.

```
procedure Info( komunikat: string );

  function NazwaProg : string;
  var poz, i : integer;
      sciezka: string;
  begin
    sciezka := ParamStr(0);
    poz := 0;
    for i:=1 to length(sciezka) do
      if sciezka[i]='\' then poz := i;
    NazwaProg := copy( sciezka, poz+1, length(sciezka) - poz );
  end;

begin
  WriteLn( StdErr, komunikat );
  WriteLn( StdErr, 'Uruchomienie programu:' );
  Writeln( StdErr, NazwaProg, ' -n liczba -s slowo' );
end;
```

Program główny wygląda zaś następująco:

```
var dane : Tdane;
    i : integer;
    stan : StanOpcji;
BEGIN
  stan := CzyOpcjeOK( dane );
  if stan<>OpOK then
  begin
    Info( KomOpcji( stan ) );
```

```
    halt( 1 );
  end;
  { właściwe wykonanie programu }
  for i:=1 to dane.nr do WriteLn( dane.slowo );
END.
```

* * *

W trakcie testowania znalazłem wywołanie, które uruchamia się poprawnie, choć chyba nie powinno:

```
ilerazy fakeparametr -s -n 4
```

Poprawa funkcji `CzyOpcjeOK()` niekoniecznie musi być prosta. Z drugiej strony, poprawne uruchomienie z punktu widzenia użytkownika:

```
ilerazy -s -+*+- -n 4
```

okazuje się być błędnym według `CzyOpcjeOK()`. Trudna sprawa ☺.

3.8 Debugger we Free Pascalu

Kiedy programy są coraz bardziej skomplikowane, tracimy kontrolę nad tym co się dzieje w programie. Zasady programowania proceduralnego, a właściwie zasada „dziel i zwyciężaj”, czyli dzielenie zadania na kilka pomniejszych, pomagają tworzyć poprawny kod. Każdą z części piszemy i testujemy osobno od innych. Przy małych projektach taka technika wystarcza, przy większych potrzebne jest dodatkowe narzędzie do wyszukiwania błędów.

> Polska terminologia nie dopracowała się rodzimego odpowiednika debuggera. Były próby wprowadzenia pojęcia odpluskwiania, ale chyba się nie przyjęły. W każdym razie używam obu terminów zdając sobie sprawę, że oba nie wyglądają naturalnie w języku polskim.

Potrzeba stosowania debuggera pojawia się szczególnie wtedy, gdy uruchomiliśmy program (według kompilatora poprawny), ale wyniki nie są takie jak oczekiwaliśmy. Takim przypadkiem może być całkowite milczenie programu. Można wtedy podejrzeć, co się dzieje w jego wnętrzu. Poniżej krótka instrukcja.

Odpluskwianie polega głównie na podglądaniu stanu zmiennych. W tym celu wybieramy kilka z nich i dopisujemy do listy oglądanych (**Watches**): Po kliknięciu pozycji menu **Debug** → **Add Watch**, pojawi się okienko, w którym wpisujemy nazwę zmiennej do obserwacji. Pojawi się okno **Watches** z listą zmiennych (przed uruchomieniem programu, będą miały opis `<Unknown value>`

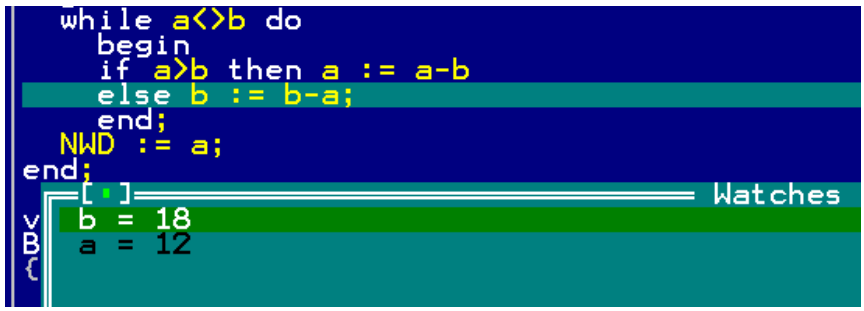
Mamy teraz dwie możliwości:

- Uruchamiamy debugowanie od samego początku programu klawiszem **F8** (menu **Run** → **Step over**), żeby przejść do następnej linii kodu znów wciskamy **F8** i tak dalej.
- Wstawiamy gdzieś w kodzie tzw. breakpoint czyli punkt kontrolny – punkt do którego program będzie się wykonywał normalnie – po uruchomieniu **Ctr+F9** (menu **Run** → **Run**). Po napotkaniu takiego punktu, program się zatrzyma i trzeba będzie uruchamiać jego wykonanie klawiszem **F8** linia po linii, jak w punkcie poprzednim. Punkt kontrolny wstawiamy poleceniem menu **Debug** → **Breakpoint**. Objawi się on podświetleniem linii na czerwono.

Przy rozpoczynaniu procesu, otworzy się osobne okno do wykonywania programu a ściślej do operacji wejścia i wyjścia do/z konsoli.

Jeśli linia kodu zawiera wywołanie procedury czy funkcji, to wciśnięcie **F8**, będzie skutkować wywołaniem jej w jednym kroku. Żeby móc nadzorować wykonanie podprogramu linia po linii, trzeba użyć klawisza **F7** (menu **Run** → **Trace into**), proces odpluskwiania przeskoczy wtedy do jego kodu.

Jeśli w trakcie działania wypisane zmienne znajdą się w używanym zakresie, pojawią się ich wartości. Odczytane dane stanowią zwykle istotne informacje podpowiadające nam, w którym miejscu programu zaczyna dziać się coś niedobrego.



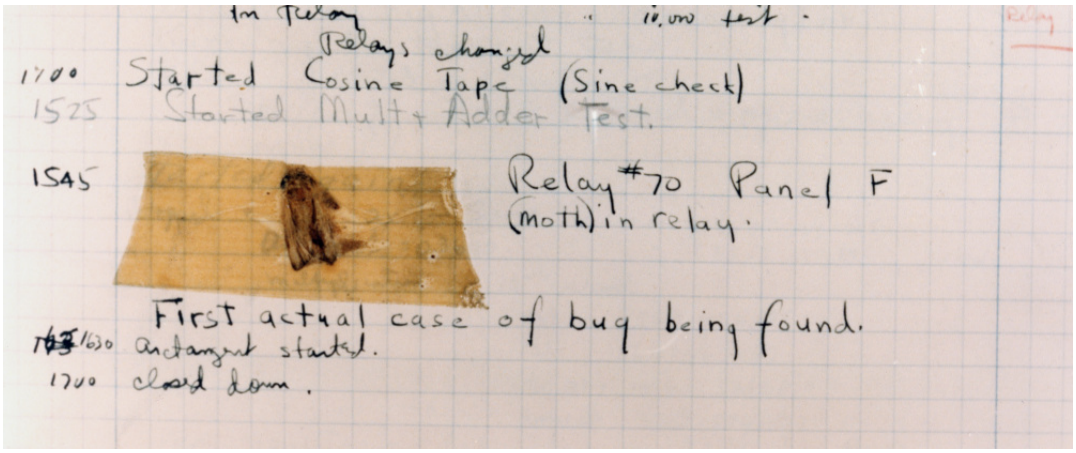
Debugowanie kończymy wybraniem z menu **Run** → **Program reset**, albo dojściem programu do końca.

* * *

Wśród kilku legend od lat powtarzanych w świecie komputerowym, dwie są na pewno nieprawdziwe. W pierwszą wierzą chyba wszyscy wyznawcy IT – opowiada ona jak to firma Microsoft sprzedała DOSa firmie IBM. Powszechna akceptacja tej

wyretuszowanej historii jest jednym z dowodów, wydającym się potwierdzać negatywny stereotyp komputerowców. Ale zostawmy ów przypadek, zajrzyjmy do czasów, gdy świat IT był zupełnie inny – przynależał nie do korporacji, ale do ośrodków akademickich, z dużym udziałem kobiet, ciężko było tam znaleźć nieprzystających do świata quasi-aspergerowców o niskim poziomie empatii. Tam właśnie narodziła się druga legenda…

Otóż w roku 1947 jedna z harwardzkich programistek, Grace Hopper znalazła w trzewiach uczelnianego komputera Mark II ćmę. Ponieważ termin debugowanie, czyli wyszukiwanie robaków, był rozpowszechniony w inżynierskich kręgach, ona i jej koledzy zażartowali, że było to pierwsze debugowanie w dziedzinie komputerów. Albowiem debugowanie oznaczało wyszukiwanie drobnych usterek sprzętowych – mówiono tak już w laboratoriach Edisona.



„Pierwszy faktyczny przypadek znalezienia buga” - wyjątek z dziennika. Kiedyś był taki zwyczaj, że każdy komputer miał prowadzony dziennik, w którym zapisywano wszelkie wykonane czynności.

Cała historia popadłaby w zapomnienie, ale Grace Hopper lubiła ją opowiadać i dzięki temu nadała jej drugie życie. Inni programiści zaadaptowali ten termin do swoich działań w latach 50-tych. W latach 60-tych był już powszechnie stosowany. Oczywiście w mikroskopijnym, z dzisiejszego punktu widzenia, środowisku.

A gdzie w tym wszystkim legenda? Ano jakoś się utarło, że to ta ćma (bug) była źródłem neologizmu oznaczającego wyszukiwanie błędu. Czasami można przeczytać nawet, że ów robak powodował zwarcia i generował błędy wykonania programu. Więc dlatego od-robaczenie poprawiło program.

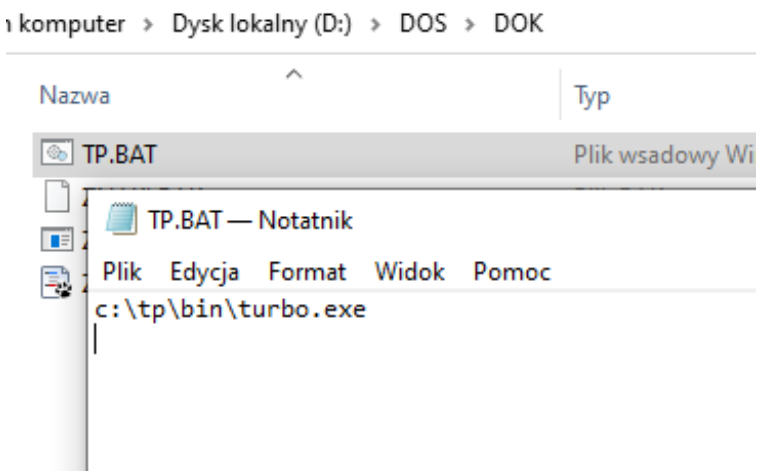
Dziś Wikipedia poinformuje nas, jak było naprawdę. Wiem, że chodzi tu tylko o kolejność, co było pierwsze: termin debugging czy znalezienie ćmy; dlatego szkodliwość mitu o robakach nie jest wielka. A jak się o nim wie, historia informatyki nabiera nieco cieplejszych barw.

Więcej szczegółów znajdziecie choćby tu: `https://www.webdevelopersnotes.com/the-first-computer-bug-was-a-moth` (zdjęcie pochodzi z Wikipedii)

3.9 Rzut oka na Turbo Pascala w DOSie

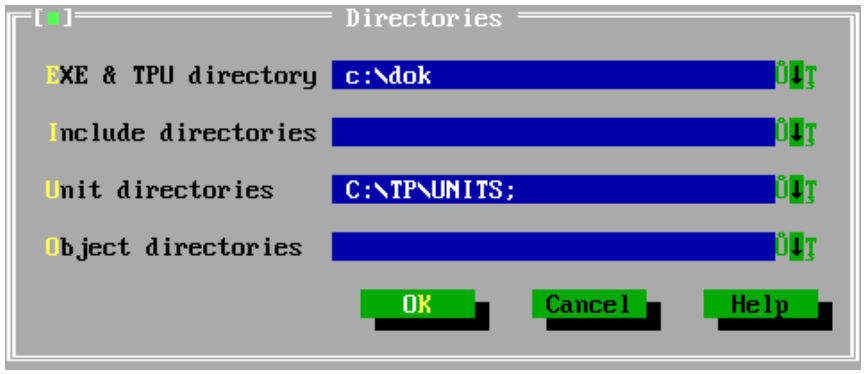
Jak przygoda, to przygoda. Cofamy się o 30 lat, by zobaczyć, jak to kiedyś bywało. Żeby wykonać podróż w czasie do krainy 16-bitowych systemów w dobie systemów 64-bitowych, należy zainstalować emulator DOS. Chyba najbardziej popularny to DOSBox – to bardzo fajny program, bo dla Turbo Pascala umożliwia nie tylko widok konsoli ale i tryb graficzny. Warto to docenić.

Emulator jest dostępny na `https://www.dosbox.com/download.php?main=1`. Dobrze jest utworzyć sobie katalog na nasze dzieła (dalej będzie to wirtualna ścieżka `C:\DOK`) i tam umieścić skrypt do uruchamiania Turbo Pascala:

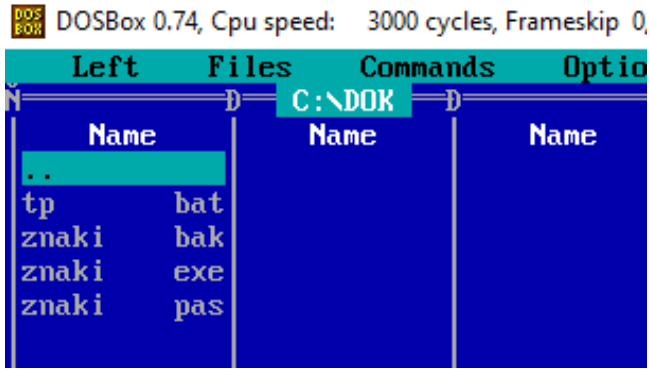


Alternatywą jest zmiana katalogu bieżącego przy każdym uruchomieniu: **File** → **Change dir…**

Jeśli chcemy mieć dostęp do skompilowanego pliku, trzeba wskazać ścieżkę (może być to katalog z naszymi plikami źródłowymi) w **Options** → **Directories…**:



oraz wybrać opcję **Disk** w menu **Compile** → **Destination**. Wtedy IDE do pliku PAS i BAK (backup), dołoży jeszcze EXE.



Na koniec zapisujemy opcje środowiska w **Options** → **Save … TURBO.TP**.

Gdyby ktoś chciał rzeczywiście bawić się ze starożytnym Turbo Pascalem – skompilowany program uruchamiać trzeba z menu, bo kombinacja **Ctr+F9** wyłącza DOSBoxa.

Baza danych

Szumnie powiedziane: baza danych. Dwie tabelki zapisane jako dwa pliki – tekstowe lub jednorodne, jak kto wolał. Menu tekstowe. Żonglerka procedurami `GotoXY()` i rysowanie ramek. Odczytywanie wciśnięcia klawiszy z ominięciem `ReadLn()`, za pomocą `ReadKey`. Ci lepsi znali wstawki assemblerowe żeby „zniknąć" kursor. Kiedy już każdy nauczył się, jak w miarę ładnie to wszystko połączyć, przyszło windowsowe GUI i nagle cała ta wiedza stała się nieaktualna. Nie ma więc sensu pokazywanie całej aplikacji typu „baza danych", ale niektóre jej elementy mogą być ciekawe.

Taką ciekawostką wydaje się rysowanie ramki. W stronie kodowej IBMa (popularnie zwanej DOSową) znajdują się znaki mogące tworzyć semigrafikę – pojedyncze i podwójne ramki. Są poniżej wypisane jako stałe znakowe. Można z nich składać coś w rodzaju okienek:

```
const
  ramLG=#218; ramPG=#191;
  ramPoz=#196;
  ramPio=#179;
  ramLD=#192; ramPD=#217;

procedure Ramka( pozx, pozy, szer, wys : integer );
var  i : integer;
begin
  GotoXY( pozx, pozy );
  Write( ramLG );
  for i:=1 to szer do Write( ramPoz );
  Write( ramPG );
  for i:=1 to wys do
  begin
    GotoXY( pozX, pozY+i );
```

```
    Write( ramPio );
    GotoXY( pozX+szer+1, pozY+i );
    Write( ramPio )
  end;
  GotoXY( pozX, pozY+wys+1 );
  Write( ramLD );
  for i:=1 to szer do Write( ramPoz );
  Write( ramPD );
end;
```

Stosowało się też znaki ramek w kodzie programu całkiem jawnie. W tym celu trzeba było zapisać szereg kolejnych kodów ASCII do pliku (np. od 127 do 255), otworzyć ten plik w edytorze Turbo Pascala i przekopiować potrzebne znaki do kodu programu.

Jak już się ma pustą ramkę, to można pokusić się o wpisanie w nią dowolnego tekstu:

```
procedure RamkaNapis( pozx, pozy : integer; napis : string );
var wys, szer : integer;
begin
  wys := 1;
  szer := 2 + length( napis );
  Ramka( pozx, pozy, szer, wys );
  GotoXY( pozx+2, pozy+1);
  Write( napis );
end;
```

Przydaje się jeszcze funkcja do czytania napisu znak po znaku:

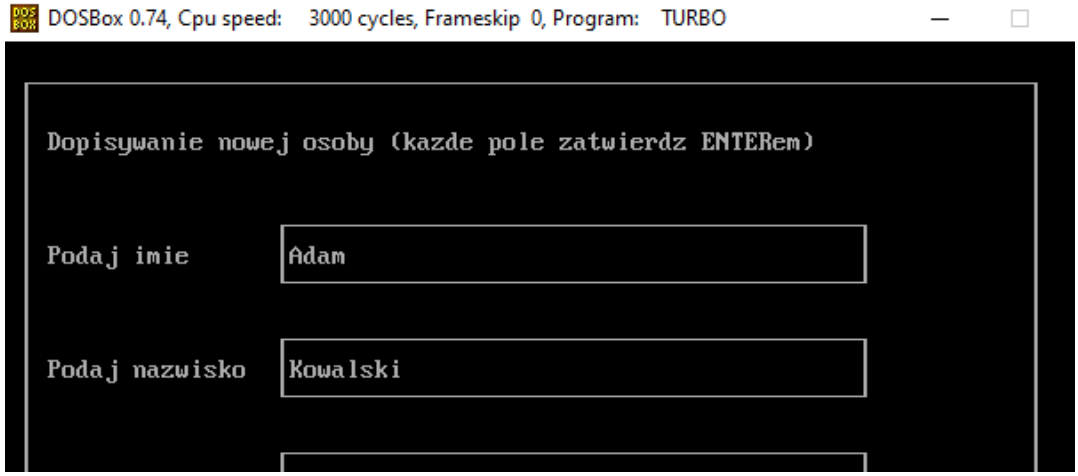
```
function CzytNapis : string;
var wyn : string;
  znak :char;
begin
  wyn := ''; {zerowanie napisu}
  repeat
    znak := ReadKey;
    if ( znak<'A') or (znak>'z') then continue;
    Write( znak );
    wyn := wyn + znak;
  until znak=#13;
  CzytNapis := wyn;
end;
```

Zauważmy, że przeczytane zostaną tylko litery – niestety bez polskich ogonków, ale w tym czasie w ogóle były one trudno dostępne.

Przy testowaniu tej funkcji okazało się, że Turbo Pascal tworzy napis (zmienna `wyn`), który na początku niekoniecznie musi być pusty. Dlatego wymusiłem `{zerowanie napisu}` – właśnie tam gdzie wpisałem ten komentarz. Ciekawostką

jest to, że powyższa procedura omija wszelkie kombinacje klawiszy i zatrzymuje się dopiero po wciśnięciu ENTERa. To dobry kandydat na pętlę nieskończoną!

Z tych podprogramów złożyć można np. taką planszę do dopisywania pojedynczego rekordu danych:



Nie namawiam nikogo do kopiowania tych rozwiązań, bo nawigacja jest szczątkowa, a pomylić się łatwo. Tym bardziej, że techniki wymagane przy tworzeniu okienkowego GUI są zupełnie inne.

Kolorowe obrazki

Uzyskanie ekranu który nie ma tekstowej rozdzielczości 80×25, ale większą pikselową, było dość trudnym zadaniem, jeśli robiło się to po raz pierwszy. Potem używało się raz sprawdzonego kodu i nie było kłopotów.

Z moich przygód pamiętam dwa rodzaje kart graficznych: Helcules – miała większą rozdzielczość ale tylko dwa kolory – i VGA ze swoimi 16 kolorami. Jak widać, w zależności od komputera, mieliśmy różną liczbę kolorów i wielkość obszaru graficznego do dyspozycji. Na ekranie można było rysować po uruchomieniu trybu graficznego. Potrzebne funkcje, procedury i stałe mieściły się w module `graph`:

```
uses graph;
```

Samo uruchomienie sprowadzało się do detekcji odpowiednich parametrów (numer sterownika, numer trybu graficznego) i zainicjowania za ich pomocą grafiki:

```
DetectGraph( gdriv, gmode );
InitGraph( gdriv, gmode, '' );
```

Trzecim argumentem `InitGraph()` jest ścieżka pod którą będą widoczne pliki sterowników do karty. W przypadku uruchamiania programów przez środowisko

można było zostawić ścieżkę pustą (IDE sobie radziło). Jeśli zabieraliśmy plik EXE ze sobą, wskazane było dołączyć te pliki do katalogu bieżącego i wtedy ścieżka też mogła być pustym napisem.

Po poprawnym wykonaniu inicjalizacji, znaki konsoli znikały, ekran robił się jednolicie czarny i można było rysować i wypisywać najróżniejsze rzeczy – tu możliwości były niesamowite. Poniżej mały przegląd kilku procedur graficznych – współrzędne pikselowe liczy się od lewego górnego rogu w dół i na prawo, wszystkie argumenty są w miarę małymi liczbami całkowitymi.

```
{ wypisywanie tekstu (bez polskich ogonków) }
OutTextXY( x, y, 'BARDZO␣WAZNY␣NAPIS' );
{ jeden piksel }
PutPixel( x, y, kolor );
{ ustalenie koloru pisaka }
SetColor( kolor );
{ linia }
Line( xp, yp, xk, yk );
{ prostokąt }
Rectangle( xp, yp, xk, yk );
{ ustalenie wypełnienia kształtu }
SetFillStyle( wzor, kolor );
{ owal z wypełnieniem }
FillEllipse( Xsrodek, Ysrodek, PolOsX, PolOsY );
{ owal bez wypełnienia }
Ellipse( Xsrodek, Ysrodek, PolOsX, PolOsY );
{ prostokąt z wypełnieniem }
Bar( xp, yp, xk, yk );
{ i wiele innych... }
```

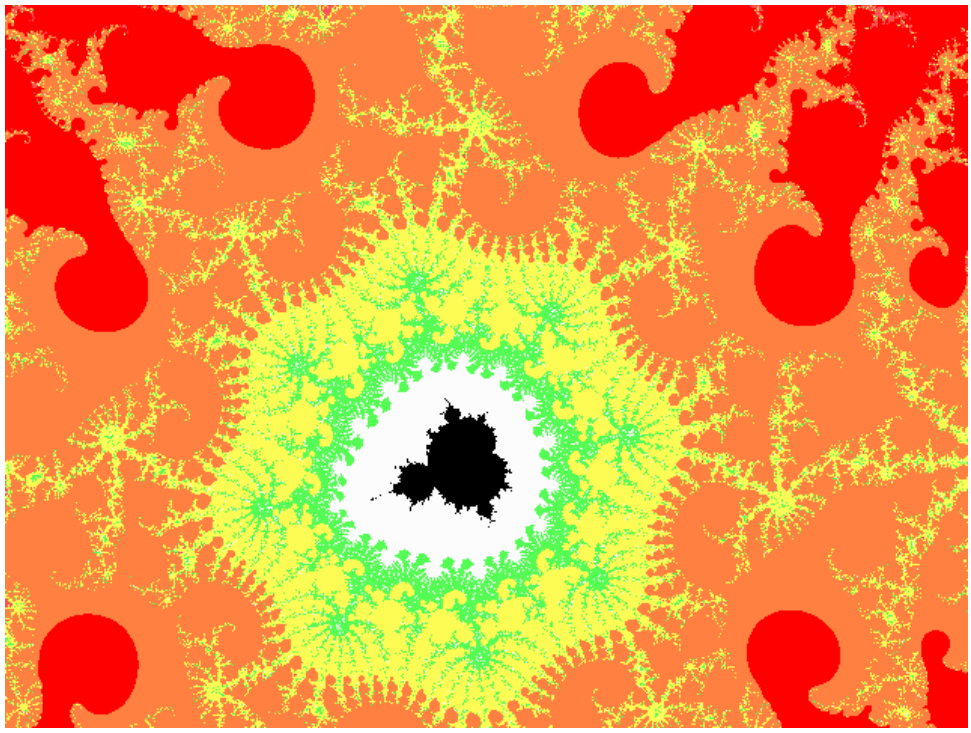
Na koniec zamykało się tryb graficzny

```
CloseGraph;
```

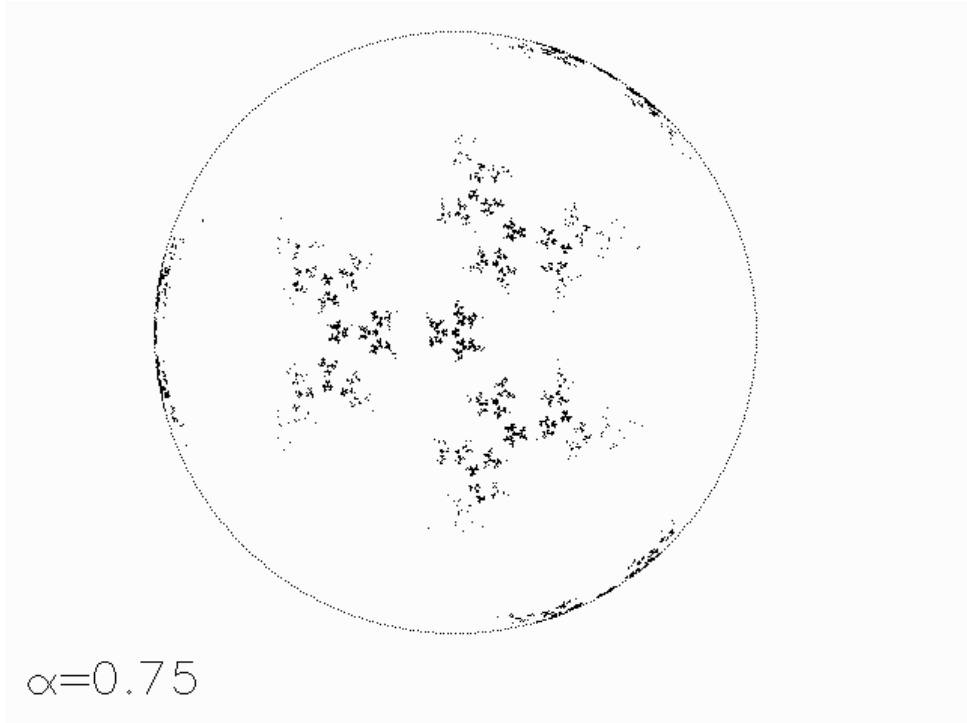
i znów na ekranie widać było znak zachęty konsoli.

* * *

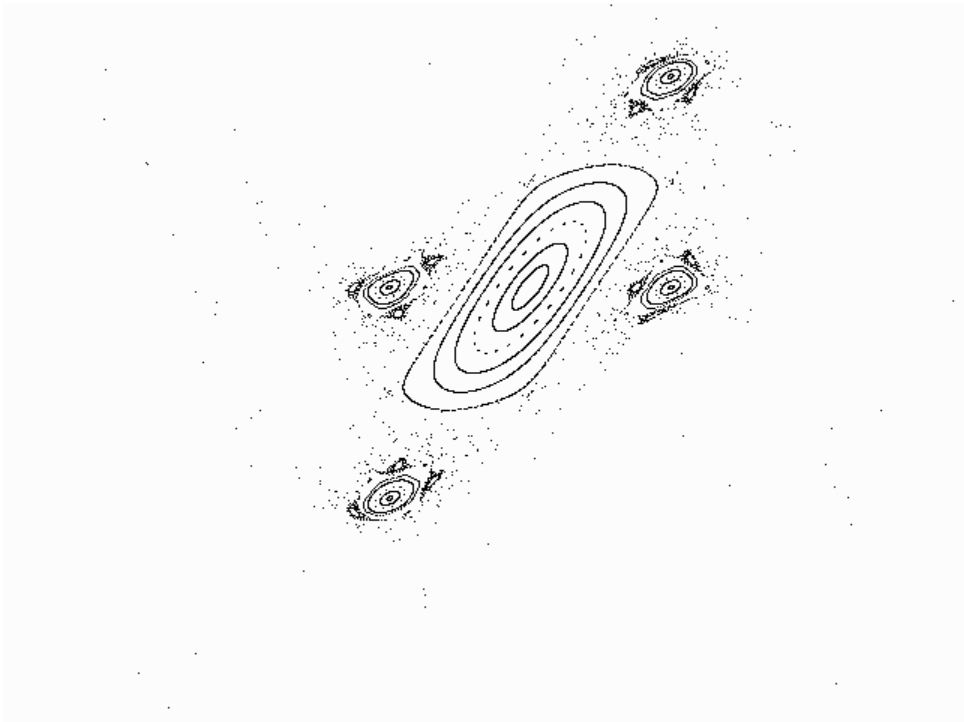
Podzielę się teraz kilkoma starożytnymi obrazkami na różne tematy, jeszcze raz narzekając, że kiedyś ich uzyskanie było bezwstydnie proste. To znaczy: główna trudność tkwiła w matematyce obiektów, a nie w technologii ich wyświetlania.



Żuk Mandelbrota (jakieś powiększenie)



Pewien specyficzny przykład IFS



Orbity Chirikova

3.10 Kilka słów na koniec

Pomijając ostatni, wspomnieniowy rozdział o czasach DOSa, przedstawiony materiał stanowi sobą wystarczającą porcję wiedzy, potrzebnej do pisania w miarę poprawnych i użytecznych programów działających w trybie tekstowym. Może się to wydawać niemożliwe, ale wiedza ta na przełomie lat 80/90 była w miarę powszechna. Programy w pascalu pisali informatycy, fizycy, matematycy a nawet studenci nauk przyrodniczych. Pascala uczyło się w szkołach średnich na lekcjach informatyki.

To kiedyś, a co dziś?

Jeśli z jakiegoś powodu zajrzałeś tu i nie masz kłopotów z trzema pierwszymi modułami, to zachęcam do zapoznania się z trzema kolejnymi – na razie w przygotowaniu. Zaglądaj więc na stronę `https://amatorkod.wordpress.com/pascal/` by sprawdzić, czy faza przygotowania nie zmieniła się w fazę publikacji ☺. Będzie tam napisane, jak utworzyć program okienkowy, co właściwie do dziś nie jest sprawą prostą.

Czeka na nas jeszcze opanowanie kilku technik. Chodzi przede wszystkim o elementy programowania obiektowego i generycznego, wyjątki czy sposoby podziału projektu na osobne pliki. Czy są potrzebne? O obiektówce być może coś słyszałeś, więc podam skrótową motywację co do zasad programowania generycznego. Przypomnijmy sobie mianowicie szybkie sortowanie opisane na stronie 144. W tamtym przykładzie sortowałem liczby zmiennoprzecinkowe. Dla innego typu

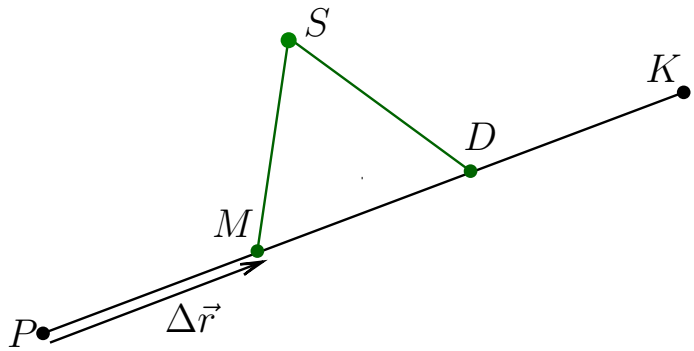
trzeba było skorygować kod, wprowadzając funkcję porównującą. Stąd już blisko do wizji posiadania wielu kopii prawie tego samego kodu, który robi to samo dla różnych typów danych. A może warto napisać jedną procedurę, którą da się uruchomić dla dowolnego typu, bez zmiany napisanego kodu? Tym bardziej, że stosowanie metody **Ctr+C** $\rightarrow$ **Ctr+V** jest bardzo ryzykowne i wielce niezalecane.

> Wielu zagadnień nie poruszyłem i nie poruszę: Nie znajdziemy w kursie niskopoziomowej dłubaniny – choćby wstawek assemblerowych. Niestety nie znam się na tym i pewnie w swym życiu już się tego nie nauczę.

Fraktal Kocha jako bonus

Bonus, a w zasadzie powtórzenie materiału, bo żeby taki program napisać, trzeba coś niecoś wiedzieć o rekurencji, zapisywaniu danych do pliku, tworzeniu podprogramów. Dlatego możesz odnieść wrażenie, że niektóre rzeczy już raz były gdzieś omawiane…

Komputery świetnie się nadają do generowania fraktali. Jednym z przykładów jest krzywa Kocha, której kolejne iteracje mogą dać jakieś wyobrażenie, jakimi są one zbiorami. Tworzenie tego akurat fraktala polega na dzieleniu odcinka na trzy części, wyrzucaniu środkowej części i dorysowywaniu dwóch boków trójkąta równobocznego, którego trzecim bokiem jest wyrzucany odcinek. W kolejnym kroku powtarzamy tę procedurę dla każdego z czterech powstałych w ten sposób odcinków.



Prześledźmy jakie działania matematyczne są przeprowadzane, żeby mając punkty P (początek) i K (koniec) wyliczyć pozostałe: M (mały), S (środkowy) i D (duży). Na początek wyliczymy pomocniczy wektor:

$$\Delta\vec{r} = (\vec{K} - \vec{P})/3$$

Dzięki niemu znajdziemy:

$$\vec{M} = \vec{P} + \Delta\vec{r}$$

$$\vec{D} = \vec{M} + \Delta\vec{r}$$

Wyliczenie środkowego punktu jest nieco bardziej skomplikowane: najpierw trzeba obrócić $\Delta\vec{r}$ o 60°

$$r'_x = (1/2) \cdot r_x - (\sqrt{3}/2) \cdot r_y$$

$$r'_y = (\sqrt{3}/2) \cdot r_x + (1/2) \cdot r_y$$

i dodać do M:

$$\vec{S} = \Delta\vec{r'} + \vec{M}$$

Teraz należy ten algorytm zapisać w programie. Skutkiem takiego programu (nie napisałem tego jeszcze!?) będzie zestaw współrzędnych X i Y, które zapisane w pliku CSV stanowić będą punkt wyjściowy dla Excela.

W programie wygodnie będzie posługiwać się osobnym typem przechowującym współrzędne punktu (zauważmy nieśmiało do tej pory stosowaną konwencję, że nowe typy oznacza się literą T):

```
type
  TPunkt = record
    x, y : real
  end;
```

Wspomniane operacje sumy i obrotu to funkcje:

```
function obroc( wekt : TPunkt ) : TPunkt;
const
  WartSin = 0.5*sqrt(3);
  WartCos = 0.5;
begin
  obroc.x := WartCos*wekt.x - WartSin*wekt.y;
  obroc.y := WartSin*wekt.x + WartCos*wekt.y;
end;

function dodaj( punkt , przes : TPunkt ) : TPunkt;
begin
  dodaj.x := punkt.x + przes.x;
  dodaj.y := punkt.y + przes.y
end;
```

Zatrzymajmy się jeszcze przy procedurze zapisującej współrzędne do pliku. Z grubsza mamy trzy możliwości obsługi pliku (przy aktualnym stanie wiedzy). Pierwszy: Każda próba zapisu będzie otwierała plik w trybie do dopisywania i zaraz po zapisie, zamykała go. Sposób bardzo nieefektywny, bo przy kilkuset punktach, program będzie głównie otwierał i zamykał plik. Drugi: Plik otwieramy na początku i przekazujemy go do procedury wypisującej:

```
procedure wypisz( var plik : Text; punkt : TPunkt );
{ uwaga! zmienne plikowe przekazujemy w argumencie poprzez var }
begin
  WriteLn( plik, punkt.x, ';', punkt.y );
end;
```

Zmienna plikowa musiałaby być przekazywana do każdej procedury, która wywołuje `wypisz()`. Trzeci: Deklarujemy plik jako zmienną globalną, inicjując ją na początku programu głównego – i tę drogę wybiorę. Złamanie ważnej zasady, czyli wprowadzenie zmiennej globalnej, powinno się uzasadnić. Podstawowym powodem jest oczywiście łatwość napisania kodu (słaby powód). W programie używamy tylko jednego pliku (nie pomyli się), którego będzie dotyczyło każde odwołanie do zmiennej typu `Text`. Co prawda oprócz `wypisz()` do zmiennej globalnej będą miały dostęp wszystkie inne, ale w drugim podejściu i tak procedury wywołujące `wypisz()` musiały by mieć do niej dostęp.

Teraz spróbujmy napisać procedurę rekurencyjną, która oprócz wyliczania trzech punktów, będzie zapisywać współrzędne. Zauważmy, że realizuje ona wyżej opisany algorytm matematyczny.

```
procedure trojka( n : integer; pocz, kon : TPunkt );
var
  maly, duzy, srodek, przes : TPunkt;
begin
  if n=0 then exit;
  { wyliczenie współrzędnych trzech punktów }
  przes.x := (kon.x - pocz.x)/3;
  przes.y := (kon.y - pocz.y)/3;
  maly := dodaj( pocz, przes );
  duzy := dodaj( maly, przes );
  przes := obroc( przes );
  srodek := dodaj( maly, przes );
  { zapisanie współrzędnych i wywołania rekurencyjne }
  trojka( n-1, pocz, maly );
  wypisz( maly );
  trojka( n-1, maly, srodek );
  wypisz( srodek );
  trojka( n-1, srodek, duzy );
  wypisz( duzy );
  trojka( n-1, duzy, kon );
end;
```

Program główny będzie zawierał: nadanie wartości dwóm punktom początkowym – obrazek jest wygenerowany dla punktów $P = (0, 0)$ i $K = (1, 0)$ – inicjalizację pliku, uruchomienie procedury rekurencyjnej i zamknięcie pliku:

```
BEGIN
  pocz.x := 0;
```

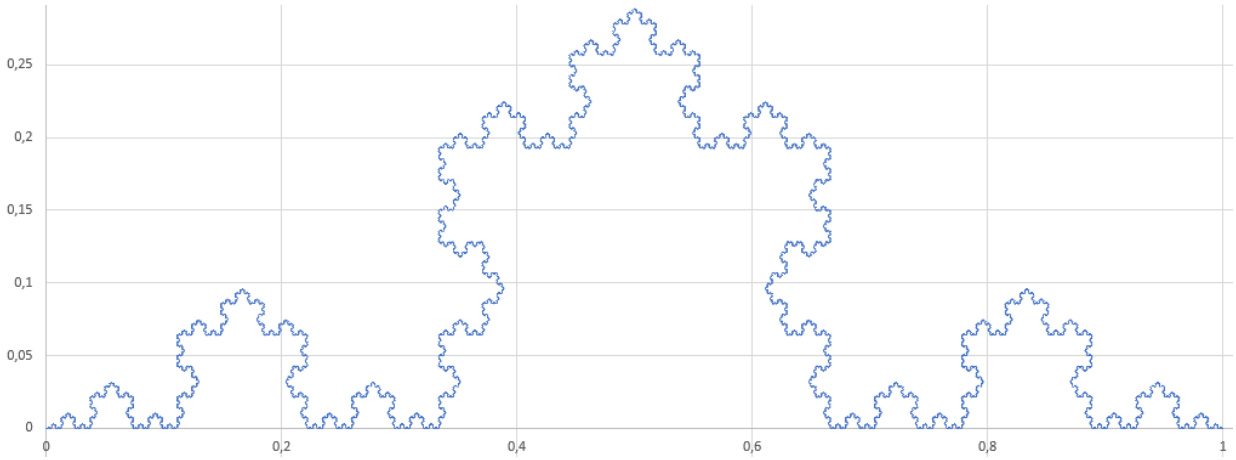
```
  pocz.y := 0;
  kon.x := 1;
  kon.y := 0;
  AsSign( plik, 'punkty.csv' );
  ReWrite( plik );
  Wypisz( pocz );
  trojka( 7, pocz, kon );
  Wypisz( kon );
  Close( plik );
END.
```

Przypomnę jeszcze o możliwości wpisania liczb do pliku CSV z przecinkami a nie kropkami. Korzystamy z funkcji `FormatFloat()` z modułu `sysutils`.

```
program gwiazdkaKocha;
uses sysutils;

...

procedure wypisz( punkt : TPunkt );
begin
  WriteLn( plik, FormatFloat( '', punkt.x ),
           ';',  FormatFloat( '', punkt.y ));
end;
```



Gotowa gwiazdka