Poznámky k přednášce Programování pro fyziky

Tomáš Ledvinka

Ústav teoretické fyziky
MFF UK Praha

2004-2016

Předmluva

Tento text představuje shrnuté poznámky k přednášce *Programování pro fyziky* pro studenty prvního ročníku fyziky na Matematickofyzikální fakultě Univerzity Karlovy. Rozsah přednášky postačuje jen k základnímu seznámení s technikou psaní počítačových programů, ať již by byl zvolen jakýkoli programovací jazyk. Volba jazyka PASCAL koncipovaného v roce 1970 se může zdát neaktuální. Domnívám se ale, že jde jediný jazyk, který byl vymýšlen s ohledem na výuku *programování* (včetně čitelného zápisu realizovaného algoritmu) a zároveň je pro něj dostupná volná implementace překladače a dovoluje pohodlně a poměrně efektivně využít výkon běžného počítače pro vědecko-technické výpočty.

Úvod

Jak zapsat program. Algoritmus. Je můj program správně?

Současné počítače představují nejpokročilejší technologii s níž se běžně můžeme setkat. Je proto rozumné většinu technických detailů počítačů vůbec neuvažovat. Místo technických detailů tak jen shrneme, jaké cesty jsou dnes v oblibě s tím, že ačkoli jsou ve většině případů ustálené již desítky let, zažijí asi dnešní studenti během své kariéry nějakou tu revoluční změnu.

Data musejí být nějak reprezentována. Informace lze sice skladovat a zpracovávat v analogové podobě (např. číslo reprezentované objemem vody v nádobě nebo nábojem uloženým v kondenzátoru, ...), ale zatím je nejefektivnější vše převést na (celá) čísla a tato čísla reprezentovat v pozičním zápisu. Až na (zajímavé) výjimky je pak základem číslo 2, tedy jediné dovolené číslice (bit) v jeho zápisu jsou 0 a 1. Užití dvojkového zápisu informace probublává nepřehlédnutelně do většiny počítačových jazyků.

Manipulaci s daty pak obstarává zařízení zvané procesor. Je konstruováno tak, aby dokázalo v daném čase provést co největší počet operací. Takovými operacemi nejsou jen ty aritmetické (jako sčítání, dělení či výpočet odmocniny), ale také všelijaké stěhování a porovnávání hodnot, větvení běhu kódu atp. Současné procesory stihnou takovýchto (záměrně elementárních) operací miliardy za sekundu (pokud data se kterými pracují jsou právě dostupná v některém z šuplíků (registrů) procesoru). Je třeba zmínit, že protože rychlost jednotlivého procesoru během posledního desetiletí začíná stagnovat, je u moderních výpočetních (a herních) stanic možné potkat až stovky procesorů, mezi které je potřeba práci dělit – to se ale učit nebudeme.

U imperativních jazyků je cílem programátora sdělit konkrétní přání oné součástce - procesoru, která sice umí pracovat rychle, ale rozumí jen několika výše zmíněným kategoriím příkazů. Tyto příkazy mají již dlouhou dobu podobu běžných dat (tedy bitů, bytů, slov, až na skryté detaily spravovaných stejně jako data s kterými program zachází). To m.j. umožňuje strojový kód programu vytvářet jiným programem (kompilátorem), který automatizuje únavné vytváření strojového kódu. I u nejjednodušších problémů je psaní programu v podobě instrukcí procesoru spíše zajímavou zábavou a s rostoucím rozsahem úlohy je prakticky nezbytné k zapsání návodu k výpočtu použít lidmi čitelný zápis ve vyšším programovacím jazyce.

Jazyk Pascal (stejně jako známé C a velké části i Fortran) patří mezi jazyky, které jen mírně zakrývají prováděné elementární operace procesoru, zkušený uživatel je v zápisu programu může stále tušit. I to je jeden z důvodů proč takový jazyk volit pro úvodní kurz programování.

Algoritmus

V počátcích informatiky se okolo počítačů motali nejen fyzici, ale také matematici. V téže době jako první počítače se objevil i matematický pojem algoritmus velmi podobný přirozené představě o návodu pro počítač.

Pro naše potřeby je algoritmus návod

- daný jako posloupnost elementárních kroků,
- jak získat ze nějakých (i prázdných) vstupních dat správný výsledek,
- v konečném počtu těchto kroků,
- a to pro stejná data vždy stejnou posloupností kroků stejný výsledek.

Pokud se elementárním krokem rozumí v konečném čase proveditelný výpočet, lze formou indukce postavit z jednodušších algoritmů algoritmus složitější a nebo třeba provádět důkazy sporem (opřené o konečný čas potřebný k realizaci algoritmu). Je zajímavé, že podoba jazyka Pascal v sobě ale nese i dědictví toho, jak se zapisovaly zkoumané algoritmy. Pro nás pojem algoritmu asi bude jen představovat matematicky posvěcený ideál, kterého bychom chtěli pro námi vytvářený návod k řešení problému, tj. kód programu, dosáhnout.

Euklidův algoritmus

Ano, již staří Řekové hledali algoritmy. Za významný lze považovat Euklidův návod (asi staršího data) jak hledat největší společný dělitel dvou čísel. Protože čísla v Euklidových úvahách měla spíše analogovou podobu délky úsečky, není původně celý algoritmus psán ani pro počítač, ani pro počtáře, ale pro geometra hledajícího nejdelší úsečku, z jejíchž celých násobků lze poskládat dvě zadané úsečky. I tento detail poskytuje vítanou možnost demonstrovat, jak nesamozřejmé je nalézt jazyk pro zápis algoritmů.

Velmi zjednodušený návod Euilkidův je:

- Nechť úsečky AB a CD představují dvě daná čísla, která mají nějakou největší společnou měrku.
- Pokud je CD stejně dlouhá AB jako je CD největší společnou měrkou AB a CD, protože nic většího to být nemůže.
- Pokud není CD stejně dlouhá jako AB, pak ta kratší z obou opakovaně odečítaná od delší ponechá zbytek, který bude mít stejnou měrku jako úsečka původní.
- Opakováním tedy zbyde dvojice stejných úseček, jinak by nebyl splněn předpoklad existence největší společné měrky z prvního bodu.

Především, dnes pro nás jsou délky úsečky modelem čísel reálných než celých, prakticky přesné odečítání úseček nejde ani provést. Navíc, i takto zjednodušený návod (prohlédněte si původní Euklidův text!) vyžaduje interpretaci inteligentním čtenářem prošlým lekcemi geometrie a dopředu chápajícím, co se má dosáhnout. Jazyk výše uvedeného popisu tedy není vhodný jako programovací jazyk. Pojďme zapsat tentýž algoritmus jinak NSD dvou hromádek kamenů zjistím tak, že dokud jsou obě hromádky různě velké, odebírám z větší hromádky takový počet kamenů, jaký je v hromádce menší.

Zde již máme přesnou reprezentaci čísel a možnost přesně určit jejich rozdíl. Je ale velmi neefektivní. Odečtení dvou hromádek s tisíci kamenů by procesora vyčerpalo. Slovní popis je opět nejednoznačný. Součástí programování v takovém jazyce by musela být praktická ukázka. (Mimo fyziku se takovéhoto způsobu zadávání požadavku na činnost počítače možná i dočkáme.) Další varianta NSD dvou čísel spočte otrok (jsme ještě ve starém Řecku) tak, že obdrží dvě čísla napsaná na dvou vedle sebe položených tabulkách a následuje návod:

1. Koukni jestli na obou tabulkách nejsou stejná čísla, pokud ano, jdi na bod 6.
2. Pokud je větší číslo napsané na pravé tabulce, prohodím je.
3. Vezmu novou prázdnou tabulku a položím ji napravo od obou.
4. Na ní pak napíši rozdíl čísel z levé a prostřední tabulky

5. Zahodím levou tabulku a pokračuji od bodu 1.

6. Levou tabulku pošlu zadavateli úlohy, pravou zahodím.

I zde je potíž s nejednoznačností slovního zápisu, tu lze odstranit zavedením "otrokódu", který navíc urychlí dekódování provádění operací. K jeho zápisu použiji stejné tabulky jako pro zápis čísel. A ještě vyřeším recyklaci tabulek z bodu 5 do bodu 3. Nejlépe tak, že místo malých tabulek budeme vše zapisovat na jednu větší tabuli.

Je na čase si ukázat, jak bude vypadat Euklidův algoritmus v "našem"jazyce Pacsal.

```
program Euk;
var a,b: integer;
begin
  a:=1998;
  b:=2516;
  while a <> b do
    if b>a then b:=b-a
           else a:=a-b;
  writeln(a);
end.
```

Shrňme pozorování, jaká lze na uvedeném programu o jazyce Pascal učinit

Text neobsahuje číslování kroků.

Jde o prostou posloupnost slov, čísel, a symbolů. Některá slova mají shora daný význam (program, begin, end, while,do, if, then, else); jedině var je zkratkou (z variable).

Shůry je dáno i slovo integer.

Symboly := očividně znamenají přiřazení.

Symboly > a <> znamenají porovnání.

Celý Euklidův algoritmus je zapsán na pouhých třech řádcích (while .. if .. then else)

Některé řádky jsou posunuté (uvidíme, že to není povinné).

Kód programu v Pascalu se skládá z deklarací a příkazů.

V době vzniku jazyka Pascal (N. Wirth, 1970) se teoretičtí informatici hodně zabývali možností dokázat správnost algoritmu. Jde o těžký problém, nejen proto, že se kromě otázky "Je můj algoritmus správně?"okamžitě nabízí i otázky "Je můj důkaz, že algoritmus X je správně správně?"atd. Produktem tohoto snažení ale byl návod, jak zapisovat algoritmy, aby bylo možno se o takové důkazy snažit. To se nakonec projevilo ve vybrané sadě a tvaru příkazů jazyka Pascal i tom, že respektuje tzv. doktrínu strukturovaného programování (detaily později). Ještě jeden důsledek dokazování správnosti algoritmů se přímo otiskl do jazyka Pascal – podoba komentářů ve složených závorkách pochází přímo odsud. Jednotlivé příkazy byly v takových důkazech obkládány předběžnou a následnou podmínkou – logickými výrazy zkonstruovanými v podobě předběžná podmínka příkaz následná podmínka tak, aby z co nejslabší podmínky předběžné provedením příkazu vyplývala (co nejsilnější) podmínka následná

$\{b \neq a\}$ `a:=a mod b` $\{\exists n \in Z$:nové-$a + bn$ =původní-$a\}$

Správný algoritmus tak z nějaké podmínky na počáteční hodnoty vstupu dal požadované tvrzení o výstupu.

```
{nechť a, b ∈ N, tj. a, b > 0 }
while a <> b do
{a≠ b a tedy mohou nastat dvě možnosti: b>a nebo a>b}
if b > a then b:=b-a
{nové-b >0 a má stejný NSD jako původní b}
else a:=a-b;
{nové-a >0 a má stejný NSD jako původní a}
{když jsem se dostal sem, musí být a=b}
{zároveň se ale nezměnil NSD a tedy a=b=NSD(původní-a,původní-b) }

{konečný počet kroků vyplývá z neustálé nenulovosti a,b, konečné číslo lze ostře zmenšovat
jen konečněkrát}
```

Píšeme nejjednodušší programy

Program, proměnná, příkaz. Podmínky a cykly. Výrazy.

Jedna z možností, jak naučit Váš počítač Pascal je v Dodatku A.

První program

Pro pohodlí můžeme okénka zvětšit a přesunout, jak nám to vyhovuje. Pak text programu poněkud rozšíříme.

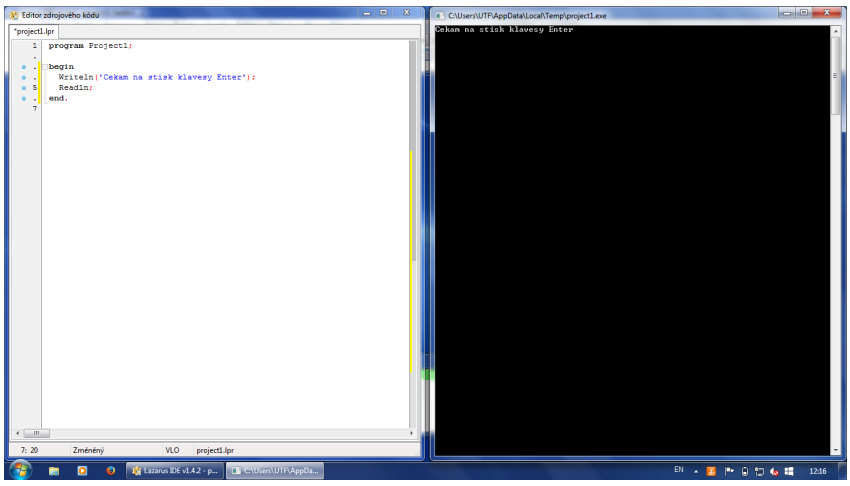
```
program Project1;

begin
 Writeln('Cekam na stick klavesy Enter');
 Readln;
end.
```

Význam kódu je zřejmý - program se skládá ze dvou příkazů

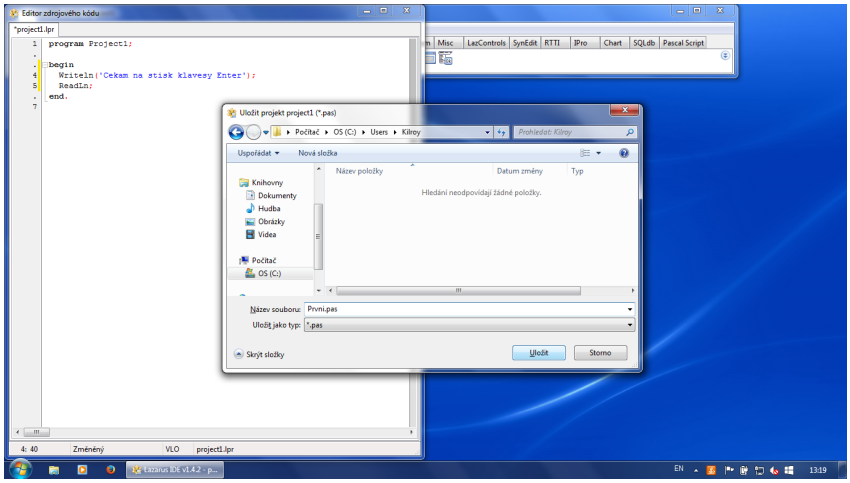
- první (Writeln) napíše onen text uzavřený v apostrofech,
- druhý (Readln) čeká než něco napíšeme a stiskneme Enter.

Pokud bychom Readln opomenuli, zmizelo by okno dříve, než bychom stačili přečíst, co se tam píše.

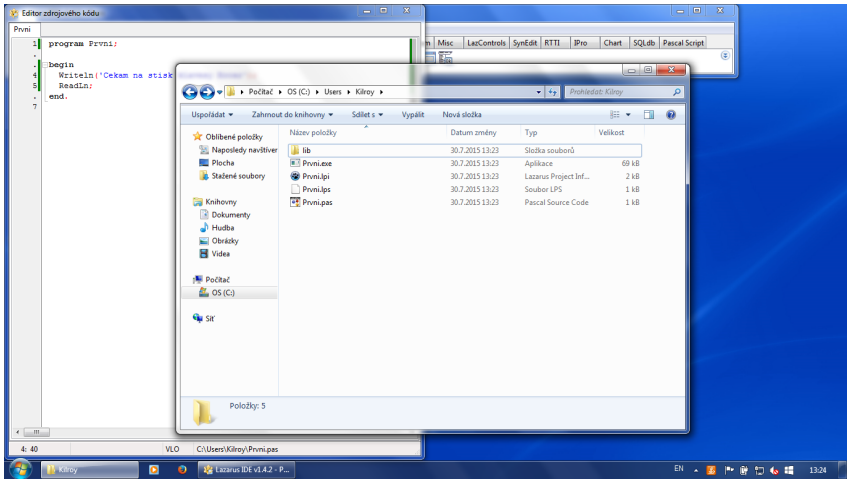
Poté stiskneme klávesu F9 a po chvíli se objeví okno, ve kterém "běží" náš program:



Jako obvykle je dobré svoji práci uložit (stačí obvyklé Ctrl-S). Je třeba vybrat nějaký adresář, kam smíme psát. Také je dobré aby v názvu cesty nebyly mezery a znaky s diakritikou, jinak nás mohou potkat záhadné potíže..



Po uložení (Ctrl-S) a spuštění (F9) se ve vybraném adresáři objeví mnoho souborů:

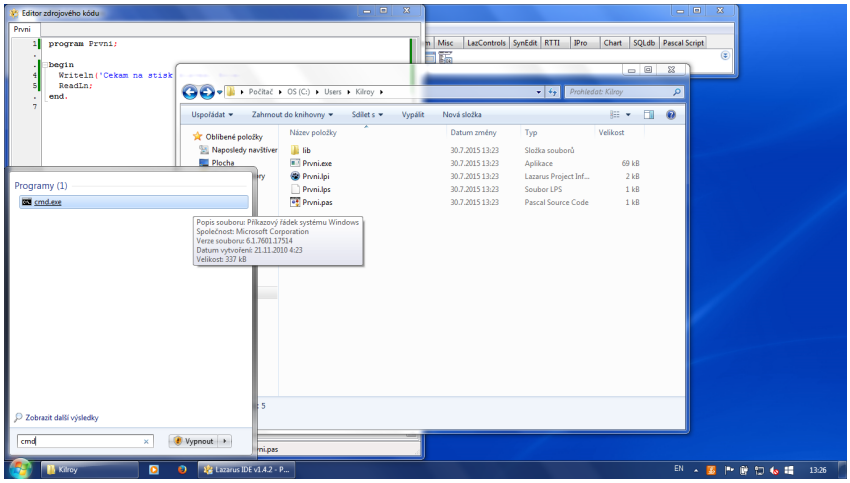


Aby se předešlo nedorozuměním, zde je něco o nich

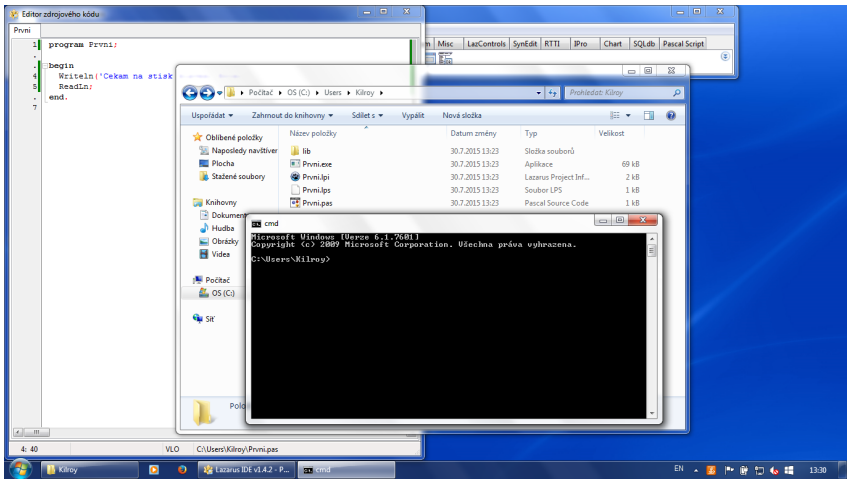
- Prvni **.pas** ... to je text programu v Pascalu. Ten posílejte vedoucímu vašich cvičení ke kontrole.
- Prvni **.exe** ... to je 'binární' program ke spuštění ve formátu, kterému rozumí váš počítač. Obvykle jej není dovoleno posílat kvůli virům emailem.
- Prvni **.lpi** ... pomocná informace (lazarus project info), u jednoduchých programů je prakticky nanic.
- Prvni **.lps** ... pomocná informace (lazarus project session) rozložení okének, až příště budete na programu pracovat. Úplně nanic

Příkazový řádek

Pro mnoho studentů zvyklých na současné moderní uživatelské rozhraní počítačů, telefonů a tabletů zde shrnuji několik poznámek o "příkazové řádce". Je dostupná po spuštění programu "cmd":

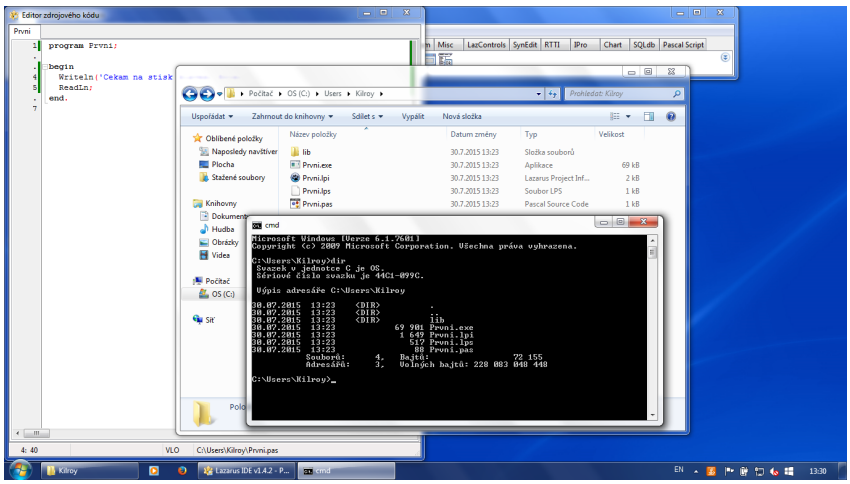


Obvykle má podobu černého okénka s textem:



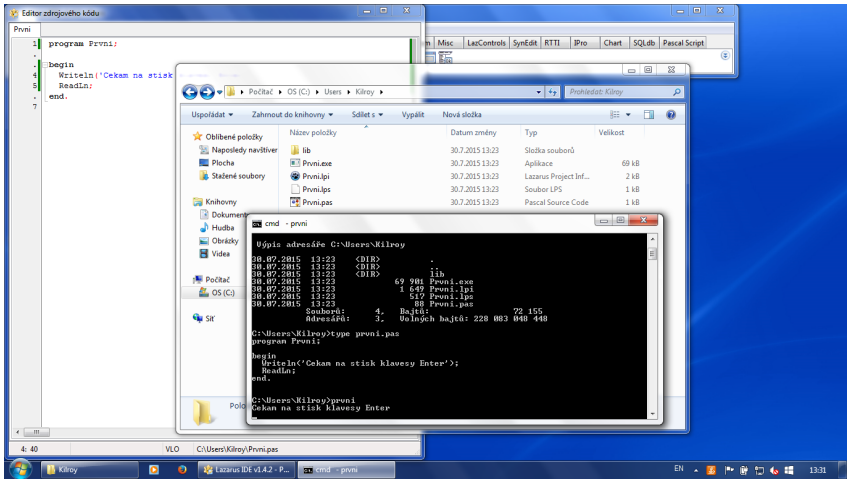
Optimálně lze z příkazové řádky dosáhnout téhož, co s programy s grafickým prostředím, jen poněkud jinak.

Například příkaz **dir** vypíše obsah adresáře:

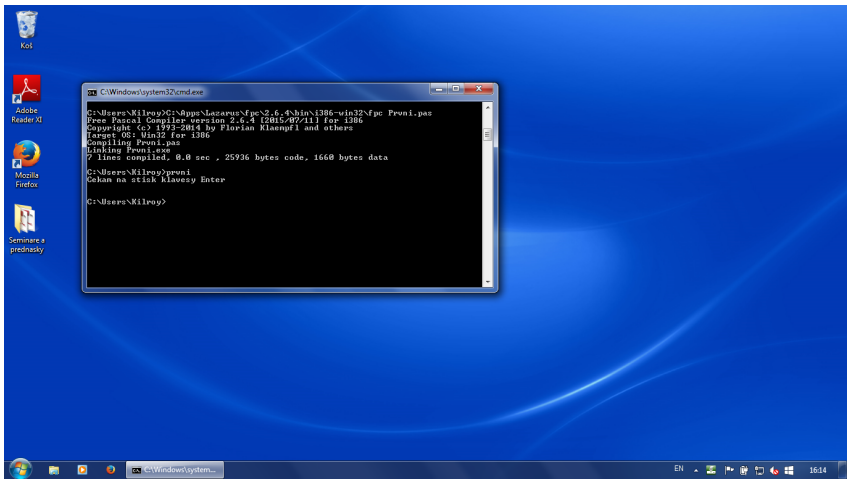


příkaz **TYPE** vypíše obsah souboru (zde námi dříve vytvořeného prvni.pas).

Protože jsme námi napsaný program již přeložili, (viz popis souboru prvni.exe výše), příkaz **prvni** se chová jako F9 v IDE:



Také lze z příkazové řádky spustit překladač:



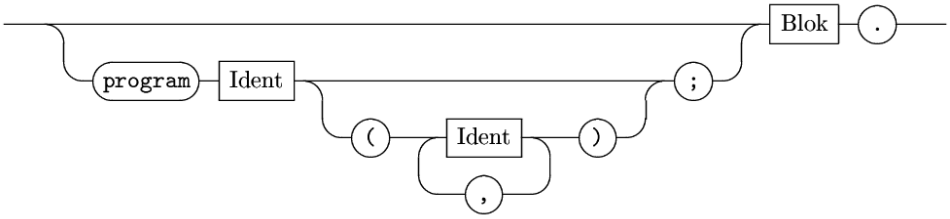
Na přednášce zmíníme:

- Adresář, soubor (cesta, textový a spustitelný soubor)
- Command prompt (shell)
- Příkazy na příkazové řádce pro Windows [Linux]
  - dir [ls -l]
  - mkdir
  - copy [cp]
  - move [mv]
  - del [rm]
  - fpc prvni.pas
  - notepad [pico]
- Textový editor

Počítačový jazyk – jak zapsat jeho pravidla

Začněme kolejištěm pro pascalovský program:

Program



Protože je uvozovací část nepovinná, je správně jak následující program

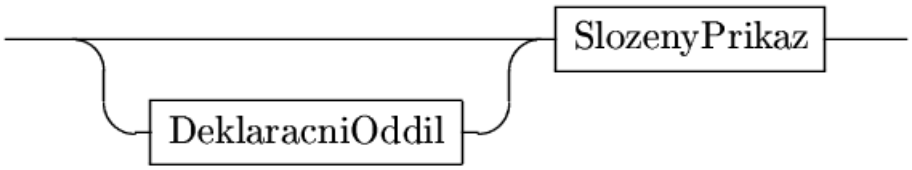
```
program  SHlavickou;
begin
  writeln('Jsem spravny program!');
end .
```

tak i tento kratký

```
begin  writeln('Jsem usporny program!')  end .
```

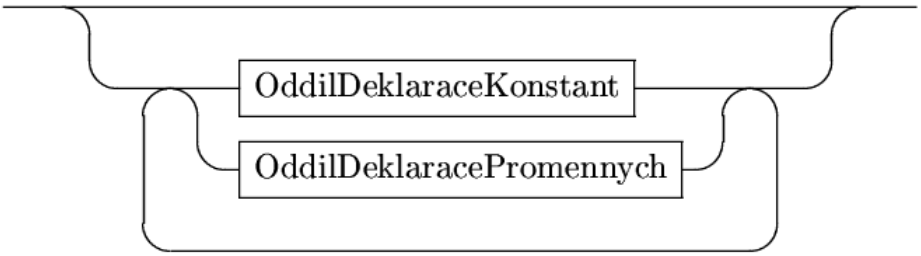
Předchozí programy byly těmi nejjednoduššími, jaké si lze představit. Neobsahují žádnou deklaraci a ten druhý se skládá pouze ze složeného příkazu. Tento složený příkaz ale bývá předcházen deklaračním oddílem, který jak později uvidíme, tvoří těžiště struktrurovaného programu.

Blok



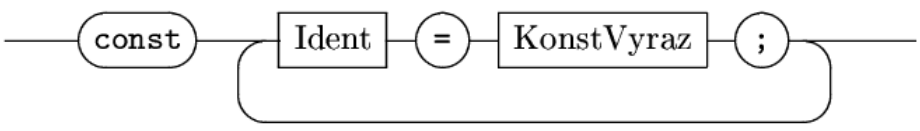
V naší zatím velmi zjednodušené verzi Pascalu budeme uvažovat pouze *proměnné* a *konstanty* a tak deklarační oddíl popisuje následující "kolejiště":

DeklaracniOddil



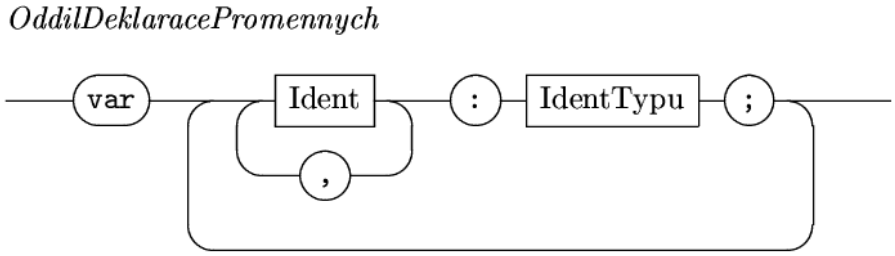
Konstanty jsou zkratky za konstantní výrazy. Existují dobré důvody proč používat konstanty:
Srozumitelnost, modifikovatelnost, pohodlí a bezpečí.

OddilDeklaraceKonstant




```
const  HorniMez      = 12;
       EulerovaKonst = 0.577215664901532861;
```

Proměnné představují místa pro uložení hodnoty, jak později uvidíme, ne nezbytně numerické povahy.

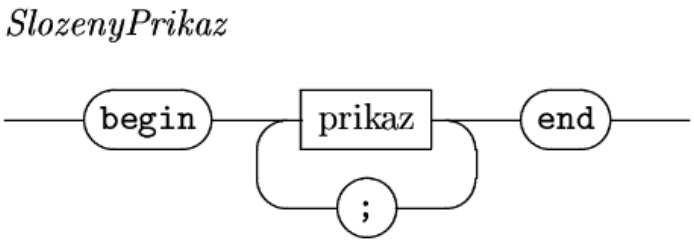


Identifikátory proměnných by měly stručně napovídat, co jsou proměnné zač. Při řešení jednoduchých úloh vystačíme ale s konvencí z hodin matematiky. Indentifikátory typu jsou prozatím shůry dány tyto:

- **Integer** ... proměnná tohoto tupu umí uložit celá čísla v rozsahu -2 147 483 648 .. +2 147 483 647
- **Real** ... reálné proměnné mají co nejlépe uložit reálné číslo. Poskytují přesnost zhruba 15 desetinných míst a pokrývají rozsah řádů zhruba 1E-300 .. 1E300
- **Boolean** ... V Pascalu je logická hodnota representována zvláštním typem který nabývá dvou hodnot

```
program  SKonstantouADvemaPromennymi;
const  N = 10;
var    i,s : integer;
begin
  i:=1;
  s:=0;
  while i <=N do
  begin
    s:=s+i;
    i:=i+1;
  end ;
  writeln('Soucet␣cisel␣od␣1␣do␣',N,'␣je␣',s);
end .
```

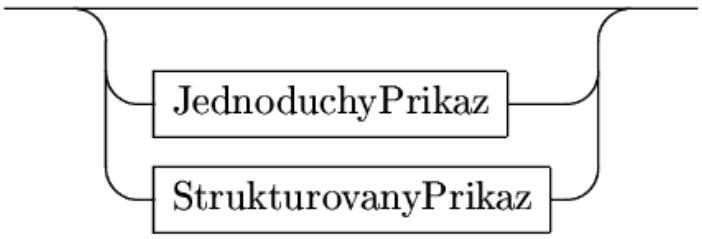
Tento velmi jednoduchý program nám kromě důvodů pro použití konstant ilustruje i použití složeného příkazu, který tvoří nejen závěrečnou (výkonnou) část pascalovského bloku, ale také umožňuje v cyklu **while** vykonávat více jak jeden příkaz:



Jak jsme viděli, program se skládá z hlavičky, deklarací a složeného příkazu, což je sekvence příkazů oddělená středníky a uzavřená mezi slova **begin** a **end** a tečky za programem.

A co je to ten **Příkaz** ?

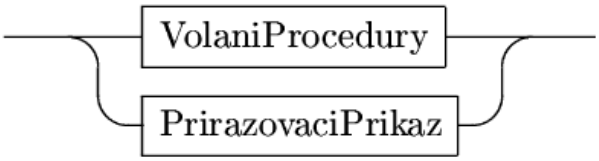
Prikaz



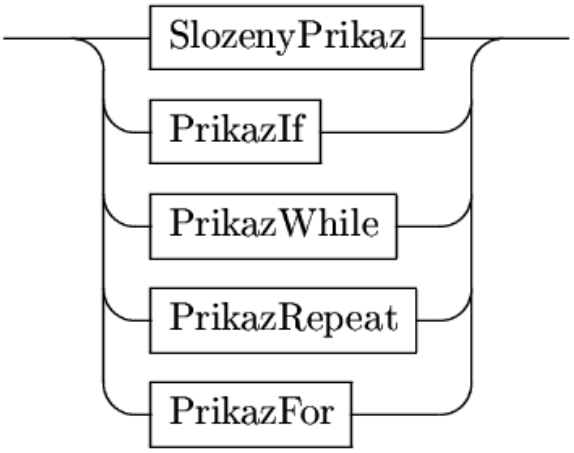
Za prvé, jak vidíme, nic (prázdný příkaz) je také příkaz.

Jednoduché příkazy jsou v podstatě dva, s *trukturovaných příkazů* je více, mezi ty základní patří samotný složený příkaz, podmíněný příkaz a příkazy cyklu.

JednoduchyPrikaz

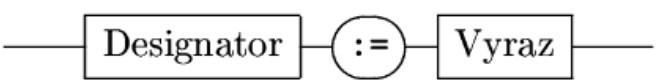


StrukturovanyPrikaz

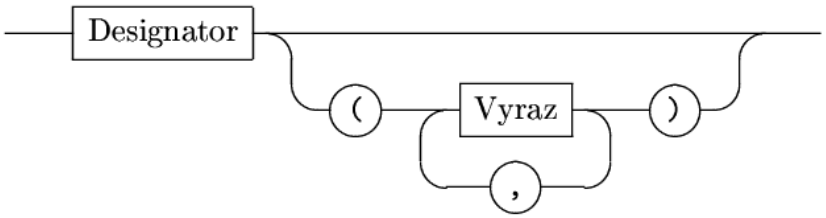


Jednoduché příkazy

PrirazovaciPrikaz



VolaniProcedury



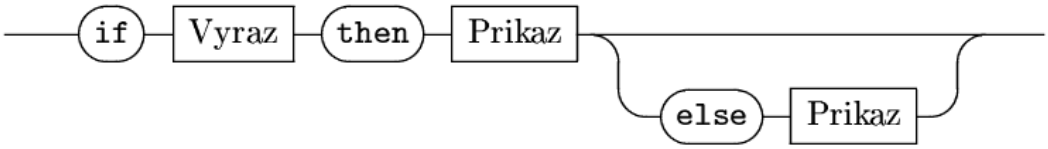
Ještě nejméně týden pro nás bude **designátor** totožný s **identifikátorem** , až se dozvíme, že jsou i další prvky jazyka, hodí se vědět, kde jazyk vyžaduje identifikátor, a kde můžeme použít "něco obecnějšího".

Obě uvedené formy jednoduchého příkazu ilustrují následjující dva řádky:

```
c := (a+b)/2;
Writeln( 'Prumer␣cisel␣',a,'␣a␣',b,'␣je␣',c);
```

Strukturované příkazy - Podmíněný příkaz

PrikazIf



má dvě varinaty. Krátkou (if ... then ...)

```
if a < b then a:=b;
```

a dlouhou (if ... then ... else ...)

```
if b*b-4*a*c>=0 then writeln('Rovnice␣ma␣reseni')
                else writeln('Rovnice␣nema␣reseni');
```

Problém je, že není úplně zřejmé, zda je správně tato

```
 if  a>0 then
           if  b>0  then  c:=1
         else  c:=2;
```

nebo naopak tato indentace

```
 if  a>0 then
          if  b>0  then  c:=1
                   else  c:=2;
```

Podle pravidla, že v syntaktických diagramech opuštíme dosaženou úroveň až když musíme, je správně druhá verse. Abychom dosáhli účinku zamýšleného indentací prvního příkladu, musíme psát

```
if  a>0  then
           begin
              if  b>0  then  c:=1
           end
         else  c:=2;
```

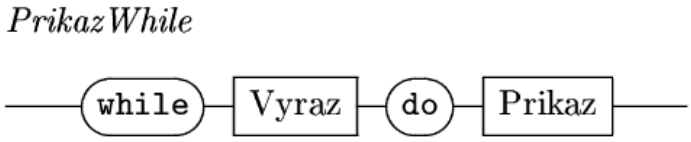
Případně můžeme použít prázdný příkaz (to je ono nic za prvním `else`)

```
if  a>0 then
          if  b>0  then  c:=1
                   else
       else  c:=2;
```

Strukturované příkazy - Cykly

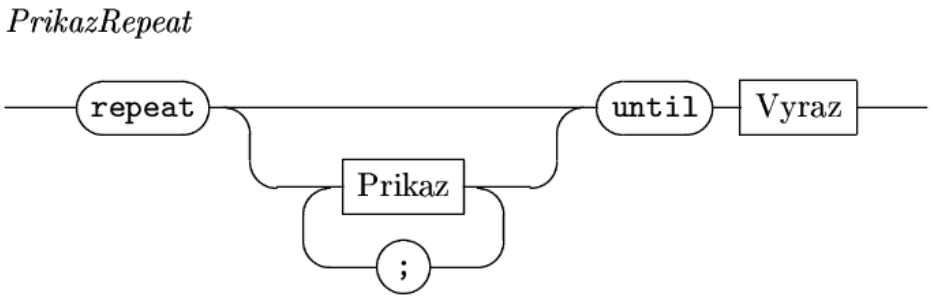
jsou důležitým prvkem jazyka. Představují opakování nějakého příkazu, které je ve správnou chvíli ukončeno. Strukturované příkazy rozlišují vlastní příkaz(y), které se mají opakovat a kontrolní část, která určuje za jakých podmínek se má příkaz provádět. Později uvidíme, že toto striktní oddělení, zvyšující "teoretické"kvality jazyka, bude povoleno porušit i jinak než pomocí **goto** .

Cyklus While



nejdříve se vyhodnotí podmínka a je-li splněna (hodnota **true**), provedede se příkaz, pak se znovu vyhodnotí podmínka atd. Nesplnění podmínky znamená konec provádění tohoto strukturovaného příkazu. V příkazu následujícím za přikazem **while** tak mohu předpokládat, že **Vyraz** má hodnotu nepravda (**false**).

Cyklus Repeat



nejdříve se vykonají všechny příkazy mezi klíčovými slovy **repeat** a **until** a pokud následně podmínka není splněna (**Vyraz** má hodnotu **false**), opakuje se provádění příkazů v cyklu atd. V příkazu následujícím za příkazem **repeat** tak mohu předpokládat, že **Vyraz** má hodnotu pravda (**true**).

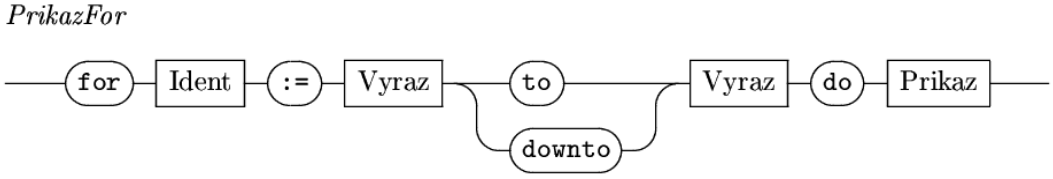
```
repeat
  writeln(i);
  i:=i+1;
until  i>10;
```

je tedy rovnocenný příkazům

```
writeln(i);
i:=i+1;
while  i <=10 do begin
   writeln(i);
   i:=i+1;
end ;
```

Cyklus For

Je určen jako zkratka cyklu **while** v případě, kdy potřebujeme projít všecha celá čísla v nějakém intervalu



a umožní nám zjednodušit náš sčítací program

```
program  SKonstantouDvemaPromennymiACyklemFor;
const  N = 10;
var    i,s : integer;
begin
  s:=0; {na tohle  nesmím  zapomenout}
  for  i:=1  to  N  do  s:=s+i;
 writeln('Soucet␣cisel␣od␣1␣do␣',N,'␣je␣',s);
end .
```

Příkaz **for** je zkratkou cyklu **while** a tak při nevhodné konstalaci mezí nemusí být příkaz ani jednou vykonán.

Příklad : následující program nám odhalí, kteráže trojciferná čísla jsou stejně jako třeba číslo 153 součtem třetích mocnin svých cifer.

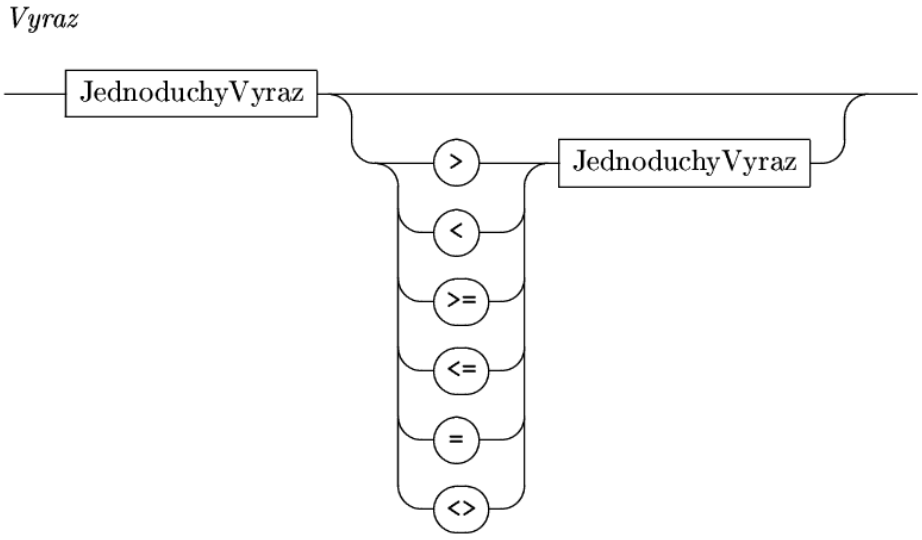
```
program  cisla;

var  a,i,j,k:integer;

begin
     for  i:=1  to  9  do  {např. 024 není  trojciferné,  tak  začínám od 1}
       for  j:=0  to  9  do
         for  k:=0  to  9 do
         begin
           a:=i*100+j*10+k;
           if  a=i*i*i+j*j*j+k*k*k  then  writeln(a);
         end ;
 end .
```

Pozn.: Algoritmus, který tento program popisuje, patří do třídy těch nejpotupnějších, neboť jej lze zapsat slovy: *zkus všechny možnosti* . (Slang: Brute force) Bohužel v diskrétních úlohách někdy ani lepší nenajdeme. Naštěstí není celých čísel "tak moc"jako těch reálných.

Výrazy

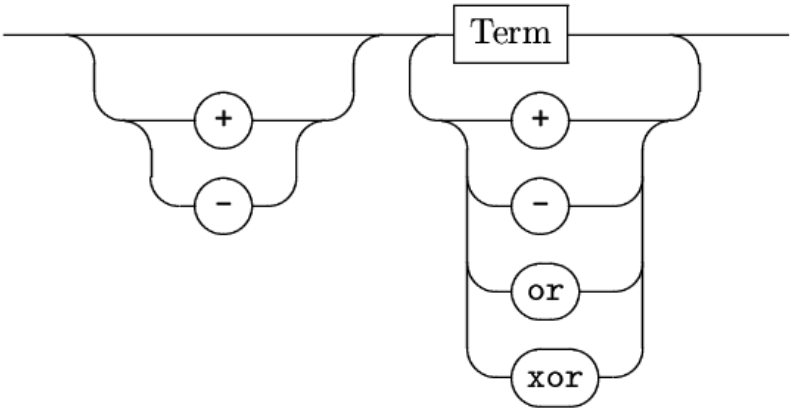
V předcházejících příkladech programů jsme používali m.j. přiřazovací příkaz a ten má na pravé straně **výraz** . Jde o přirozené zobecnění matematické notace do formy textu "vytisknutelného na dálnopise", zůstávají pojmy operand, operace, priorita.

Především, operace "porovnání"(>, <, <=, ...) jsou v Pascalu operátory s nějnižší prioritou. V syntaktickém diagramu to vypadá takhle:



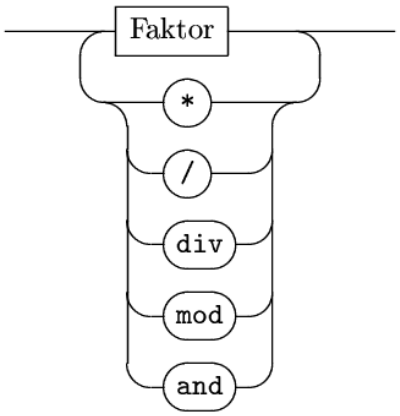
Teď přichází na řadu obvyklé sčítání, odčítání a analogické logické operace

JednoduchyVyraz



Termy, tedy sčítance mohou být součinem, podílem atp. jednotlivých faktorů.

Term



Zde je třeba upozornit na operaci zbytku celočíselného dělení **mod**.

Tetno program najde všechna trojciferná čísla rovná součtu třetích mocnin svých cifer:

```
program  CislaII;

var  a,i,j,k:integer;

begin
 for a:=100 to 999 do begin
    i := a div 100;              { stovky }
    j := (a div 10) mod 10;      { desitky }
   {j := (a mod 100) div  10; }
    k := a mod 10;               { jednotky }
    if  a=i*i*i+j*j*j+k*k*k  then  writeln(a);
 end;
end.
```

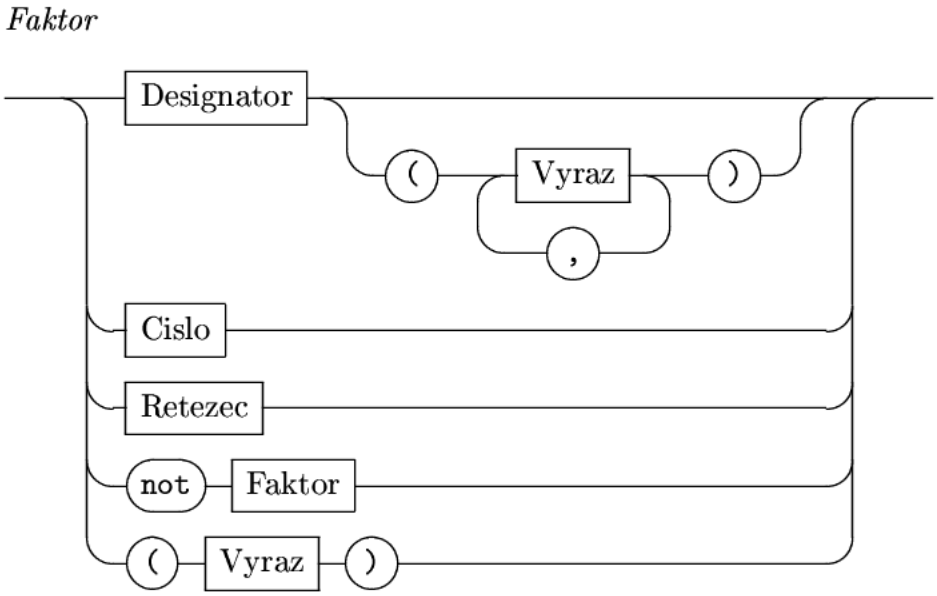
Euklidův algoritmus, kde se vyhneme opakovanému odečítání malého čísla od vetšího:

```
program  EuklidII;

var  a,b : integer ;

begin
  a := 11088;
  b := 17017;
  while  a <>b  do
    if  a>b  then  a:=(a-1)  mod  b + 1
           else  b:=(b-1)  mod  a + 1;
  writeln(a);
end .
```

Totéž ale jinak:

```
program  EuklidIII;

var  a,b,c : integer ;

begin
 a := 11088;
 b := 17017;
 repeat
   c := a  mod  b;
   a := b;
   b := c;   { význam: (a,b) := (b,a mod b) }
  until  b=0;
 writeln(a);
end.
```

No a konečně každý faktor může být identifikátor proměnné či konstanty, číslo atp.



Nejříve si připomeňme, že **designátor** je prozatím totéž, co identifikátor. První řádek (horní kolej) pak říká, že pravé strany následujících přiřazovacích příkazů jsou správné **faktory** , tedy i **termy** , tedy i **jednoduché výrazy** a konečně tedy i správné **výrazy** :

```
s:=a;
y:=sin(x);
b:=InRange(y,-1,1);
```

Jak víme ne každý syntakticky správný kus kódu nám překladač schválí, třeba

```
var  a:integer;
begin
  a:=a(1);
end .
```

není správný program. Identifikátor před závorkou totiž musí být **identifikátor funkce** . To že deklarujeme proměnnou či konstantu vlastně znamená, že uvedenému identifikátoru přiřadíme **význam** . Některé identifikátory jsou však definovány "samy od sebe". Mezi ně patří i tzv. **standardní funkce** . Zde je seznam některých z nich:

identifikátor funkce	*význam*	*typ výsledku*
ArcTan	arkustangens	Real
Cos	kosinus	Real
Sin	sinus	Real
Exp	exponenciála	Real
Ln	přirozený logaritmus	Real
Sqrt	druhá odmocnina	Real
Int	celá část čísla	Real
Round	nejbližší celé číslo	Int64
Trunc	celá část reálného čísla	Int64
Frac	destinná část	Real
Sqr	druhá mocina	Real nebo Integer*)
Abs	absolutní hodnota	Real nebo Integer*)
Odd	je argument liché číslo	Boolean

*) Protože druhá mocnina celého čísla je vždy celé číslo, je typ výsledku volání funkce **sqr** dán typem parametru. Podobně je tomu i s absolutní hodnotou čísla.

Aritmetické operátory a typy

Cím se liší následující výrazy?

```
1*1
1/1
1> 1
```

Především výraz (1>1) nemá hodnotu číselnou ale logickou.
Protože lomítko je symbolem pro reálné dělení, je výsledek podílu 1/1
"reálná jednička", zatímco u součinu je ta jednička celé číslo.
Operandy +-*/ mohou být čísla realná nebo celá, **div** a **mod** mají jako operandy pouze čísla celá.
Za předpokladu, že proměnné x,y,i a j jsou deklarovány takto:

```
var i,j : integer;
    x,y : real;
```

můžeme aritmetické oprace shrnout v následující tabulce

	význam	operand	výsledek	příklad	→	typ výsledku
+	sčítání	real, integer	real,integer	x+i	→	real
-	odčítání	real,integer	real,integer	i-j	→	integer
*	násobení	real,integer	real,integer	x*y	→	real
/	reálné dělení	real,integer	real	i/j	→	real
div	celočíselné dělení	integer	integer	i div j	→	integer
mod	zbytek při dělení	integer	integer	i mod j	→	integer

Pro oparace +-* platí, že výsledek je celé číslo pouze pokud jsou oba operandy celá čísla.
Hodnota celočíselného podílu

```
i div j
```

je rovna hdonotě reálného podílu zaokrouhlené směrem k nule, tedy

```
i div j = trunc(i/j)
```

Pro záporná i a/nebo j je tato definice kompatibilní se vztahy
(-i)/j = - (i/j), i/(-j) = - (i/j)
Operace **mod** splňuje vztah
i **mod** j = i ÷ (i **div** j) * j
Obvykle ji budme používat pouze pro j>0 a i>=0, kdy platí že
i **mod** j = 0 .. j-1
tedy jde o běžnou operaci zbytku po dělení a např.
17 **mod** 5 = 2 .
Pokud je j=0, způsobí operace
x / j
i **div** j
i **mod** j
krach programu.
Unární oprátory + - nemění typ.

Logické operátory

Typ boolean popisuje logický stav ano/ne, v řeči Pascalu true/false.

Jakkoli se interně reprezentují hodnota false jako 0 a hodnota true jako 1 jsou v jazyce Pascal logické hodnoty svým typem izolovány od celých čísel a běžné aritmetické oprece pro ně nejsou definovány. Proto nelze psát

```
k :=  (i=imax) + (j=jmax);
```

(umožní nám to v budoucnu operace/funkce *ord*).
Nad hodnotami *false* a *true* však pracují logické operátory:

Binární operátory: `and, or, xor`

and	false	true
false	false	false
true	false	true

or	false	true
false	false	true
true	true	true

xor	false	true
false	false	true
true	true	false

Unární operátor negace `not`

not	false	true
	true	false

Relační operátory (=, <>, <, <=, >, >=)

Porovnávají dvě hodnoty, přičemž podobně jako + či * umějí porovnat reálné a celé číslo (přesněji dva jednoduché výrazy těchto typů).
Navíc také umějí porovnat logické hodnoty ve smyslu

```
false <  true.
```

Ve všech případech je výsledkem porovnání logická (boolean) hodnota.

Celá čísla v počítači

V současné době je standardem používat k uložení celého čísla 32 bitů. Protože je to docela dlouhé dvojkové číslo, použijeme pro následující příklad pouze čtyři bity.
Ilustrace Do čtyřech bitů lze uložit následujících 16 kombinací 0 a 1:

```
3210   hex    unsigned     signed
----    -     --------     ------
0000    0        0            0
0001    1        1            1
0010    2        2            2
0011    3        3            3
0100    4        4            4
0101    5        5            5
0110    6        6            6
0111    7        7            7
1000    8        8           -8
1001    9        9           -7
1010    A       10           -6
1011    B       11           -5
1100    C       12           -4
1101    D       13           -3
1110    E       14           -2
1111    F       15           -1
```

Když potřebujeme do 4-bitového čísla uložit číslo s významem celého čísla se znaménkem máme několik možností. V posledním slupci tabulky je uveden dnes nejrozšířenější způsob - reprezentace celého čísla se znaménkem pomocí tzv. dvojkového doplňku. Je to technologicky nejméně nákladný způsob jak obvody počítače naučit pracovat zároveň s čísly se znaménkem i bez znaménka. To proto, že stroj nemusí rozlišovat, zda sčítá či odčítá číslo se znaménkem nebo bez:
Operace `1010+0010 = 1100` má podle okolností buď význam

```
10 + 2 = 12
```

a nebo

```
  -6 + 2 = -4.
```

Protože pomocí 4 bitů můžeme reprezentovat čísla 0..15 resp -8..7 vedou některé operace k tzv. **přetečení** .

Pro čísla **bez** znaménka je to např. operace 15+15, jejíž výsledek neleží v intervalu 0..15.

Pokud se natuto operaci díváme z pohledu operací se znaménkem, je vše O.K., neboť (-2)+(-2)=-4.

Pro čísla **se** znaménkem je nedovolená např. operace 4+4=8, neboť jako horní mez rozsahu čtyřbitových oznaménovaných čísel je 7. Z hlediska čísel bez naménka je to ovšem operace dovolená.

Pro nás znamená typ integer 32 bitové číslo se znaménkem povolující uložení celého čísla v rozsahu -2 147 483 648 **..** 2 147 483 647.

V případě, že nějaká oprace, např. v příkazu **k:=i*j** vede k přetečení, není obvykle spuštěn žádný poplach a program se posléze chová podivně bez zjevných příčin, protože výsledek přiřazovacího příkazu je jiný, než zamýšlený. Je ale možné donutit program, aby si tato možná přetečení ohlídal, stráví se tím nějaký čas navíc, ale ušetří to čas při hledání problémů. Až se budeme zabývat laděním programů bude zapnutí kontroly na přetečení jedním z bodů v návodu.

Ve **vyjímečných** případech budeme potřebovat vědět, že jsou k dispozici i jiné **celočíselné typy** :

Identif.Typu	*Rozsah*	*Formát uložení*
Shortint	-128..127	signed 8-bit
Smallint	-32768..32767	signed 16-bit
Longint	-2147483648..2147483647	signed 32-bit
Int64	$-2^{63}..2^{63-1}$	signed 64-bit
Byte	0..255	unsigned 8-bit
Word	0..65535	unsigned 16-bit
Longword	0..4294967295	unsigned 32-bit

Jak vidíme, typ Integer je v současné době totožný s typem Longint. Je pravděpodobné, že během několika let bude typ Integer odpovídat 64-bitovému číslu a neměli bychom spoléhat na to, že proměnné typu Integer jsou ukládány jako zrovna jako 32-bitové. Abychom však nemuseli hlídat každý součin **200*200** (nevejde se do SmallInt), budeme předpokládat, že těch bitů je nejméně 32, takže pozor na přetečení si budeme muset dávat až u součinů jako je **50000*50000** .

S výjimkou typu Int64 obecně neplatí, že operace s kratším formátem je rychlejší, takže důvody pro použití kratšího formátu čísla musí být v algoritmu samém, ne jeho optimalizaci. Jednou z výjimek je úspora paměti při uložení miliónů a miliónů celých čísel v poli (viz dále), převážným důvodem ale bude respektování formátu vstupních dat: např. komponenty RGB v bitmapě jsou typu Byte, zvuk v audiosouboru na kompaktním disku je zase posloupnost dvojic (L,R) oznaménkovaných 16-bitových čísel (typ SmallInt).

Logické operace nad celými čísly

Z technických důvodů se čísla v počítači uskladňují ve dvojkovém zápisu. Na jednotlivé bity lze pak aplikovat logické operátory and, or, xor a not ve stejném smyslu jako pro true a false.

Nechť

```
var x,y : Integer;

 ....

  x := 21;
  y := 12;
```

pak pro logický soiučet platí

```
    x  or  y = 29
```

neboť

```
       00000000 00000000 00000000 00010101   // 21 = 16+0+4+0+1
   or  00000000 00000000 00000000 00001100   // 12 =  0+8+4+0+0
  ------------------------------------------------------------
       00000000 00000000 00000000 00011101   // 29 = 16+8+4+0+1
```

pro logický součin

```
    x   and  y = 4
```

neboť

```
       00000000 00000000 00000000 00010101   // 21 = 16+0+4+0+1
 and   00000000 00000000 00000000 00001100   // 12 =  0+8+4+0+0
-------------------------------------------------------
       00000000 00000000 00000000 00000100   //  4 =  0+0+4+0+0
```

pro operaci exklusivní disjunkce (`xor`)

```
   x xor y = 25
```

neboť

```
       00000000 00000000 00000000 00010101   // 21 = 16+0+4+0+1
  xor  00000000 00000000 00000000 00001100   // 12 =  0+8+4+0+0
-------------------------------------------------------
       00000000 00000000 00000000 00011001   // 25 = 16+8+0+0+1
```

a pro logickou negaci

```
    not x = -22
```

neboť

```
 not   00000000 00000000 00000000 00010101   // 21 = 16+0+4+0+1
-------------------------------------------------------------
       11111111 11111111 11111111 11101010   //-22 = (-1)-16-0-4-0-1
```

Mimochodem, aby si programátoři šetřili klávesy 0 a 1, místo binárního zápisu používají zápis šestnáctkový (hexadecimální). čtveřice bitů se podle výše uvedené tabulky ozačí číslicí 0..9 nebo písmenem A-F. Proto nám následující příkaz *writeln* vypíše TRUE:

```
writeln( not $15 = $ffffffea ); // $ znamená hexadecimální číselnou konstantu
```

Protože Wirth nepoužil znak $ ve své versi Pascalu, mohl být později použit $ jako uvozovací znak při zápisu šestnáctkového čísla a výše uvedený kód je správně.

Protože jde jen o zápis čísla pro kompilátor, a $15 je naprosto totéž jako 21, zkuste uhodnout co udělá následující příkaz:

```
writeln( $15 );                    // $ znamená hexadecimální číselnou konstantu
```

Rady do života: Celá a reálná čísla

V jazyce Pascal se přes jeho přísnou kotrolu typů povoluje použít celé číslo na místě reálného. Proto do reálné proměnné **smíme** dosadit hodnotu s typem Integer, přesněji v *přiřazovacím příkazu*

idProm:=Vyraz

kde *idProm* je identifikátor reálné proměnné smí mít *Vyraz* nejen reálný typ ale i typ celočíselný.

Ačkoili tedy nepředstavuje současné použití celých a reálných čísel v Pascalu problém, je v případě podílu dvou celých čísel na místě naučit se dávat si pozor. V příkazu

```
   E := 1/2*m*v*v;
```

se podíl 1/2 vyhodnotí na konstantu 0.5 a příkaz provede, co jsme zamýšleli. Protože však např. v jazycích FORTRAN a C se podíl 1/2 vyhodnotí jako 0 a do E se dosadí nula, je pro budoucího fyzika vhodné zvyknout si psát např.

```
   V := 4/3.0*Pi*r*r*r;
```

a nebo 4.0/3 atp. Nejjednodušší je nezvyknout, abychom si pak nemuseli odvykat.

Podobně bychom si neměli zvykat jako celočíselné ukládat hodnoty typu **n!** nebo $\mathbf{2}^n$ neboť ani v 32-bitových celých číslech na ně není "dost místa".

Řešení problémů hrubou silou

Jako první při řešení nějakého problému uvažujeme algoritmus spočívající v použití **hrubé síly** .

Jako ilustraci tohot přístupu uvažujme následující program, který hledá všechny neuspořádané trojice kladných čísel, které leží na kouli o poloměru 2003, pokud je chápeme jako kartézské souřadnice bodu v prostoru a koule má střed v počátku.

Načtněme nejdříve obrysy takového postupu:

1. Pro všechny uspořádané trojice přirozených čísel které připadají v úvahu:
2. Zkontroluj, zda náhodou nesplňují zkoumanou rovnici
3. Pokud ano, vypiš výsledek.

Protože pro první bod nemáme k disposici odpovídající konstrukci jazyka Pascal, napíšeme jej podrobněji:

1.1 Pro všechna čísla **a** od **1** do **N-1**
1.2 Pro všechna čísla **b** od **1** do **a**
1.3 Pro všechna čísla **c** od **1** do **b**

Teď již vidíme, že po překladu do "angličtiny"získáme kostru kódu, třeba takovýto:

```
program Rozklady1;

var a,b,c,N :integer;
begin
   N:=2003;
  Writeln('Rozklady cisla ',N);
  {pro vsechny trojice N> a> =b> =c> 0 }
    for a:=1   to N-1   do
       for b:=1   to a   do
         for c:=1   to b   do
          begin
           if a*a+b*b+c*c=N*N then writeln(a,' ',b,' ',c);
          end ;

  Writeln('konec');
end .
```

Následující variantu je stále možné považovat za použití hrubé síly, i když je cca 60x rychlejší.

```
program Rozklady2;
var a,b,c,N :integer;
begin
 N:=2003;
Writeln('Rozklady cisla ',N);
{pro vhodne trojice N> a> =b> =c> 0 zkoumej zda plati}
  for a:=1   to N-1   do
     begin
     b:=1;
      while (b< =a)   and (N*N-a*a-b*b> 0)   do
         begin
            c:=trunc(0.5+sqrt(N*N-a*a-b*b));
            if (c< =b)   and (a*a+b*b+c*c=N*N)   then
                  writeln(a,' ',b,' ',c);
             b:=b+1;
         end ;
    end ;
```

```
  Writeln('konec');
end .
```

Navíc se následujícími příkazy můžeme přesvědčit, že oba programy dávají stejné výsledky. Zde použijeme trik s přesměrováním výstupu programu do souboru (To je to znaménko >mezi příkazem a názvem výstupního souboru). Necháme tak vytvořit dva soubory, jejichž názvy si samozřejmě můžeme zvolit libovolně, a posléze jejich obsah porovnáme. Práci s porovnáváním obsahu můžeme přenechat počítači, pokud si zjistíme, který program to za nás udělá. Seznam takovýchto užitečných programů dodám později, a pak se dozvíte, že v tomto případě je třeba použít program s názvem FC (pro příkazový řádek MS Windows).

```
C:\ Projects\ prog\ pokusy> Rozklady1 >  Vysledky1.txt

C:\ Projects\ prog\ pokusy> Rozklady2 >  Vysledky2.txt

C:\ Projects\ prog\ pokusy> fc Vysledky1.txt Vysledky2.txt
Comparing files Vysledky1.txt and Vysledky2.txt
FC: no differences encountered

C:\ Projects\ prog\ pokusy>
```

Příkládky

1. Předpokládejte deklarace

```
var i,j,k : integer;
    x,y,z : real;
       be : boolean;
```

Jaké jsou typy následujících výrazů? Jsou všechny zapsány správně?

```
    i+j
    i*j
    i/j

    i+y
    i*y
    i/y

    i   mod y

    be   and  i > j
    i > j   and  x > y
    be   or not  2*be
```

2. Zapište jako přiřazovací příkazy následující vzorečky (volbu identifikátorů a počet přiřazení je na vás)

- $w = \frac{x+y}{x-y}$
- $u = \frac{x+\sqrt{\frac{x+y}{x-y}}}{x-\sqrt{\frac{x+y}{x-y}}}$
- $z = \frac{1}{2}A^2 \sin^2 \vartheta$
- $\Phi = 2M \frac{\cos^3 \vartheta - \sin^3 \vartheta}{\cos^3 \vartheta + \sin^3 \vartheta}$

3. Všichni znáte součtové vzorce, zvažte následující kód:

```
program dvacetiuhelnik;

const N = 20;                // pocet vrcholu N-uhelnika nahrazuji-
ciho kruznici
      df = 2*Pi/N;           // odpovidajici uhel

var cos_df,sin_df : real;    // konstatntni hodnoty
    cos_f, sin_f  : real;    // promenne bezici po kruznici
                i  : integer; // ridici promenna cyklu ...

begin
  cos_df := cos(df);
  sin_df := sin(df);

  cos_f := 1;
  sin_f := 0;

  for i:= 0 to N do begin
    // vypis hodnoty
    writeln(cos_f, ' ', sin_f);
    // spocti nove hodnoty sin_f, cos_f
    cos_f := cos_f*cos_df-sin_f*sin_df;
    sin_f := cos_f*sin_df+sin_f*cos_df;
  end;
end.
```

Co je na tomhle kódu špatně? Přesněji, kde je v cyklu chyba, která způsobí, že nedostanu souřadnice vrcholů pravidellnécho 20-úhelníku?

Procedury a funkce

Parametry a jejich předávání. Lokální proměnné.

Čím je kód programu vzdálenější našemu jazyku, tím větší je pravděpodobnost, že při psaní kódu programu uděláme chybu. Platí to i u intelektuálně nenáročných kusů kódu: Při psaní

```
y := ln(x + sqrt(x*x + 1))
```

ještě nejspíš chybu neuděláme, i když zápis

```
y := ArcSinh(x)
```

je přehlednější a mnohem odolnější vůči chybě. Pokud budeme chtít spočíst

```
z := ArcSinh(sqrt((x-1)/(x+1)))
```

roste pravděpodobnost, že se při přepisu do tvaru

```
z := ln( sqrt( (x-1)/(x+1) ) + sqrt( 2*x/(x+1) ) )
```

dopustíme chyby.

Pododbně jako výpočet nějaké funkce může nějaká posloupnost příkazů tvořit jasný celek, který je pak pro přehlednost možné vyjmout z místa, kde jej chceme uplatnit a vytvořit z něj Proceduru.

Satrapa [Sa] píše:

> Kdykoli si řeknete "teď by se mi hodilo aby pascal měl příkaz (nebo funkci), který ....", vymyslete si vhodné jméno a obohaťte Pascal o nový příkaz (či funkci) - definujte podprogram.

Uvidíme, že budou případy, kdy to takto jednoduše nepůjde, ale jako motto je to výstižné.

I když jsme se vzdali představy, že o budeme dokazovat správnost programu, nepochybně chceme psát správné programy. Vytvořením vhodné hierarchie krátkých přehledných podprogramů, lze v ideálním případě dosáhnout toho, že na každé úrovni je podprogram očividně správně.

Ovšemže by měl být zmíněn také hlavní a každému zřejmý důvod zavední procedur a fukcí: V případě, kdy bych byl nucen opakovat již jednou napsaný kus kódu, nabízí se tu možnost ulehčit si práci se psaním. Protože jsme ze školy zvyklí myslet "strukturovaně", oba důvody pro použití funkcí (a procedur) se překrývají: Tentýž kód by se opakoval, protože representuje nějaký typický podproblém.

Euklidův algoritmus jako funkce

Jako příklad budiž zmíněn opět Euklidův algoritmus. Co kdybychom chtěli spočíst největší společný dělitel tří čísel? Jak? Co takhle spočíst NSD třetího čísla a NSD prvních dvou čísel? ..... Pak by zřejmě nebylo od věci moci zapsat celý postup třeba takto:

```
vysledek := GCD( c, GCD(a,b) );
```

Aby nám překladač rozumněl, musíme mu oznámit, že

1. GCD je identifikátor funkce
2. funkce GCD má dva celočíselné parametry
3. funkce GCD vrací jako výsledek celé číslo
4. Aby to fungovalo, musíme také říci jak se má ze vstupních hodnot vyrobit výsledek (jakýsi malý program).

V jazyce Pascal se to provede tak, že v deklaracích bloku, ve kterém chceme tuhle funkci použít, spolu s proměnnými a konstantami, deklarujeme ještě fukci.

```
    function  GCD(a,b : integer) : integer;
    var  c:integer;
    begin
       repeat
          c := amod b;
          a := b;
          b := c; { (a,b) := (b,a mod b); }
      until b=0;
       GCD := a;
    end ;  {function GCD}
```

Časem si nakreslíme syntaktický diagram, ale i bez něj rozpoznáváme jasnou strukturu Hlavička-Deklarace-Složený Příkaz, jakou má pascalský program. Z kódu je jasně vidět, že použití identifikátorů a,b se neodlišuje od použití identifikátoru proměnné c. Na rozdíl od proměnných mají ale a a b na začátku přiřazené hodnoty. Pokud bychom funkci GCD použili například takto:

```
   n := GCD(44,55) ;
```

bude na začátku provádění příkazů těla funkce mít a hodnotu 44 a b hodnotu 55. Proměnná c bude mít hodnotu nedefinovanou. Proto její hodnota nesmí být užita dříve, než jí bude nějaká přiřazena.

Takto pak vypadá celý kód programu:

```
 program  GCD3;

 const  a = 26112;
      b = 75548;
      c = 45288;

 var    n : integer;

 function  GCD(a,b : integer):integer;
   {Vrací NSD dvou kladných čí sel }
    var  c:integer;
    begin
       if  (a<=0) or (b<=0)  then  {oznam chybu a skonci}
       begin
         Writeln('Funkce␣GCD(', a, ',', b, ')␣:␣Neplatne␣parametry!');
        Halt;
      end ;

      repeat
         c := a mod b;
        a := b;
        b := c;
      until  b=0;
     GCD := a;
    end ;  {konec deklarce  funkce  GCD}

begin
   n := GCD( a, GCD(b,c) ) ;

   Writeln('Nejvetsi␣spolecny␣delitel␣cisel␣', a,'␣',b,'␣',c,'␣je␣',n,'␣nebot:');

   Writeln(a,'␣=␣',a div n,'*',n);
   Writeln(b,'␣=␣',b div n,'*',n);
   Writeln(c,'␣=␣',c div n,'*',n);

   Readln;
end .
```

Co se děje při použití procedury či funkce

Co se děje, když se provádí příkaz n := GCD(44,55) ? Je na překladači, zda to pochopí tak, že sem má "zkopírovat"náš kód pro NSD (tzv. makro/inline), nebo zda použije mechanismus volání podprogramu. Ten můžeme přirovnat k vyplnění žádanky byrokratem samotářem: Do kolonky a si napíše 44, do kolonky b 55. Pak si ještě vyplní kolonku s poznámkou, kde má pokračovat až se dozví výsledek. Poté nalistuje stranu v manuálu pro GCD a tam se přesně dozví co má s žádenkou dělat. Až nakonec nalezne výsledek, podívá se do kolonky kam se má vrátit, nalistuje příslušnou stránku a pokračuje. Takto se až na výjimky překládají funkce a procedury.

Vezměme jako příklad program P s funkcí f

```
program  P;
function  f(a:real):real;
begin
  f:=sin(a)
end ;

begin
  writeln(f(1));
  writeln(f(2));
end.
```

Otázka pak zní zda se program přeloží do kódu

```
   VezmiKonstantu 1.0
   SpočtiSinus
   VypišHodnotu
   VezmiKonstantu 2.0
   SpočtiSinus
   VypišHodnotu
   Skonči
```

nebo do kódu

```
  VezmiKonstantu 1.0
  ZavolejFunkci f
  VypišHodnotu
  VezmiKonstantu 2.0
  ZavolejFunkci f
  VypišHodnotu
  Skonči

TadyJeFunkce f:
  VyzvedniPředanouHodnotu
  SpočtiSinus
  Vrať Se
```

Vidíme, že druhá varianta je v tomto případě delší, ale tušíme, že pokud by funkce f byla komplikovanější, zabraly by její dvě(či ještě více) kopie více místa než vyžaduje varianta druhá. Tušíme, že první varianta (hantýrka: macro nebo též inline) je výhodná pouze pro extrémně krátké funkce. (Kompilátor FPC umožňuje o to požádat uvedením slova `inline` před středníkem na konci hlavičky procedury nebo funkce.) Měli bychom tedy vědět, že funkce a procedury se až na výjimky překládají jako samostatné kusy programu a jejich kód je uložen někdy i velmi daleko od místa, kde se používají.

Poznámka: ve starém hrozném BASICu to bylo jasné, protože jediný způsob jak volat proceduru se jmenoval GOSUB, jdi na podprogram, a každý tak viděl, že volání proceduru je jakýsi vylepšený příkaz skoku GOTO.

Procedury jako nástroj strukturovaného programování

Procedury na rozdíl od funkcí, nevracejí hodnotu, přesněji nepoužíváme je ke konstrukci výrazů, ale jsou to příkazy. Pomáhají nám, aby naše programy mohly být složeny z přehledných částí. Třeba takto:

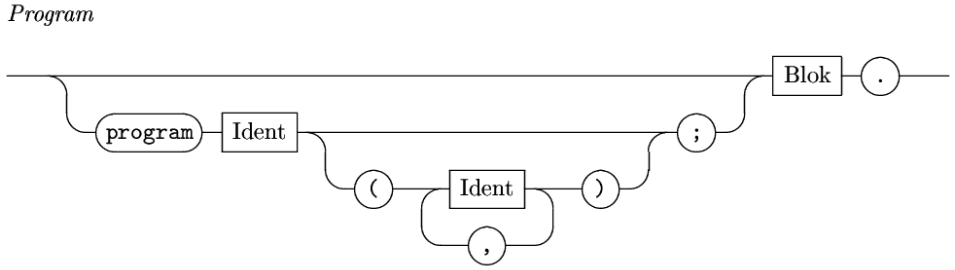
```
program  VylepsovacZvuku;
.....
begin
    NactiAudiosoubor;
    OpravPraskani;
    SnizSumeni;
    ZapisAudiosoubor;
end.
```

Teď už jen zbývá napsat ty čtyři procedury. Každou z nich napíšeme jako posloupnost dostatečně jednoduchých operací, a pokud nebudou v nabídce jazyka Pascal, vymyslíme vhodný identifikátor nové procedury, která tuto složitou operaci zařídí a tuto posléze stejným postupem rozepíšeme jako posloupnost ještě jednodušších příkazů. Takže psát programy je jednoduché, že.

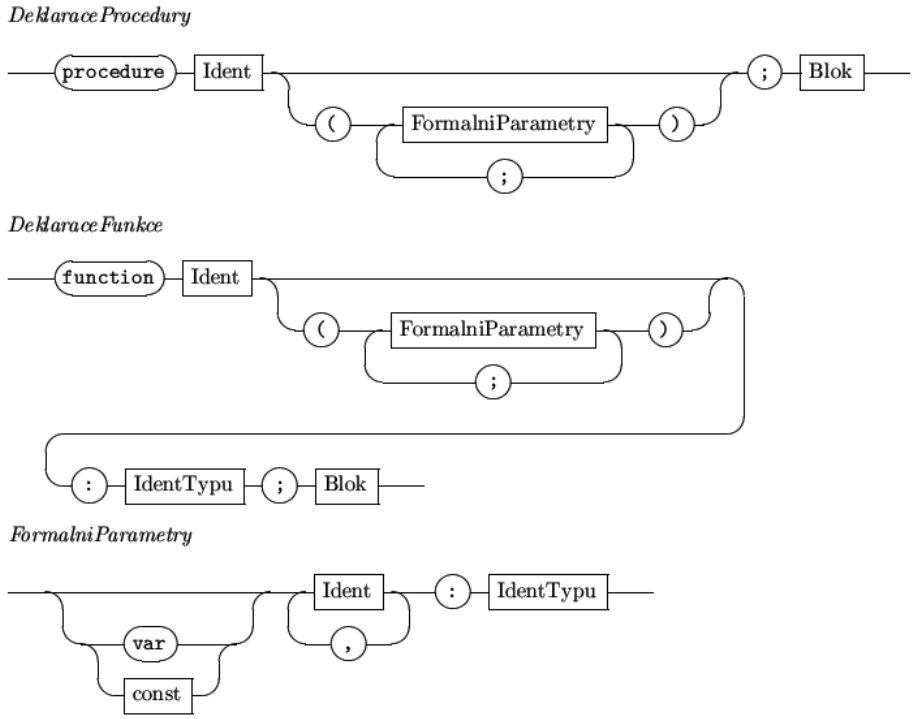
Toto je velmi zhruba idea psaní program shora dolů. Je dobré ji mít na paměti, když program píšeme, jakkoli nám nebude při psaní programu vždy pasovat na naši úlohu. V každém případě

stála u kolébky toho, jak se v jazyce Pascal program dělí na hlavní program, podporogramy a podprogramy podprogramů...

Připomeňme si nejprve syntaktický diagram pro program:



A takto vypadají diagramy pro proceduru a funkci



Jak vidíme jsou procedury složeny kromě hlavičky opět z bloku a v něm můžeme kromě proměnných a konstant deklarovat i opět další procedury a tak by náš program mohl vypadat takto:

```
Program  VylepsovacZvuku;
...
    Procedure  OpravPraskani;
   ...
       Procedure  OdectiPoruchu;
      ...
       begin
          ...
       end ;
   ...
    begin
      ...
     OdectiPoruchu;
      ...
    end ;
...
begin
  ...
  OpravPraskani;
  ...
end .
```

Procedura OdectiPoduchu se nachází uvnitř jiné procedury a ne programu, říkáme, že je to vnořená procedura (funkce). Upozornění: vnořené procedury (a funkce) nejsou v některých běžných programovacích jazycích podporovány, takže pokud si na ně příliš zvykneme hrozí nám dalším životě riziko, že si budeme muset odvykat.

Procedury a funkce mají své proměnné

Přesněji bychom měli mluvit o identifikátorech, protože v deklarační části procedury a funkce můžeme deklarovat cokoli, co můžeme deklarovat v bloku programu, proměnné jsou ale tím nejdůležitějším.

Abychom mohli používat podprogramy, je třeba vyjasnit které identifikátory platí uvnitř kterého bloku. Veměme třeba

```
program  KdoKdyKde;

var  N,i,s:integer;

function  f(a:integer):integer;
var  i,s:integer; {Co kdyz tenhle radek zakomentujeme ?}
begin
   s:=0;
   for  i:=1  to  N  do  s:=s+sqr(a+i);
   f:=s;
end ;

begin
   N:=10;
   s:=0;
   for i:=1 to N do s:=s+f(i);
   writeln(s);
end .
```

Když někde v bloku deklarujeme identifikátor, zůstává v platnosti až do konce tohoto bloku. Může však být zakryt deklarací uvnitř bloku nějaké funkce či procedury. To je případ identifikátorů i a s uvnitř funkce f v příkladu výše. Naopak, proměnná **N** je viditelná i uvnitř funkce **f** (není ničím zastíněna), čehož využíváme. Proměnné deklarované v bloku programu naýváme globální, ty deklarované v bloku procedury či funkce nazýváme lokální.

Identifikátor je definován počínaje nejbližším středníkem po jeho deklaraci a konče **end**-em složeného příkazu bloku v němž je deklarován.

Pozor, lokální proměnné nejenže nejsou vidět za **end**-em bloku kde byly deklarovány, ale ani místo v paměti pro ně není přiděleno, když kód procedury zrovna "neběží". Není tedy možné si v nich schovávat hodnoty mezi dvěma voláními téže funkce.

Parametry procedur

Především můžeme hodnoty předávat prostřednictvím společných (tedy většinou **globálních**) proměnných .

```
program  ProcSGlobProm;

var  i;

procedure  MojeProc;
begin
    i:=0;
end ;

begin
   i:=1;
   Writeln(i);
   MojeProc;
   Writeln(i);
end .
```

Pak můžeme použít parametry funkce. Nejjednodušším případem je tzv. **předání hodnotou.**

```
program  ProcSParHodnotou;

var  i;

  procedure  MojeProc(b:integer);
  begin
```

```
      b:=0;
   end ;

 begin
    i:=1;
    Writeln(i);
    MojeProc(i);
    Writeln(i);
 end .
```

V tomto případě mi program vypíše dvě jedničky. Procedura se dozví jakou hodnotu mělo i , ale dále pracuje jen s touto hodnotou. Proměnné **b** (na parametr předávaný hodnotou je vhodné nahlížet jako na proměnnou inicializovanou v okamžiku zavolání funkce) přísluší vlastní místo v paměti a jeho modifikací se nemění hodnota proměnné **i** .

Nyní jak vypadá **předání parametru odkazem**

```
program  ProcSParOdkazem;

var  i;

  procedure  MojeProc(var  b:integer);
  begin
    b:=0;
  end ;

begin
   i:=1;
   Writeln(i);
   MojeProc(i);
   Writeln(i);
end .
```

V tomto případě mi program vypíše jedničku a nulu. Procedura se dozví, kde je uskladněna proměnná **i** , a dále pracuje s tímto odkazem, tedy s proměnnou samou. Předání parametru odkazem zařídí, že místo v paměti opatřené názvem **i** se uvnitř procedury jmenuje též **b.**

Předávání odkazem je tak možné použít i k vrácení hodnoty tam, kde je použití funkce nevhodné: Především funkce neumí pohodlně vrátit dvě hodnoty zároveň, příkladem může být procedura z knihovny `math` s hlavičkou

```
procedure DivMod( Dividend, Divisor: LongInt; var Result, Remainder: LongInt);
```

která zabrání opakovanému dělení potřebujme-li znát současně podíl i zbytek po celočíselném dělení.

Často se také používají funkce jejichž **var**-parametry slouží pro návrat nalezených hodnot, zatímco funkce sama vrací jen logickou informaci, zda se to povedlo:

```
function  NactiSeznam( var  Sz : typSeznam; JakDlouhy : integer) : boolean;
begin
  ....

  ....
  NactiSenznam := NacenaDelka = JakDlouhy;
end ;
```

Použití takových funkcí totiž umočňuje pohodlnější řešení neobvyklých situací

```
function  JeVSeznamu( x: typPolozkaSeznamu ) : boolean;
  ...
  JeVSeznamu := false;
  if  not NactiSeznam(seznam, pocet) then exit;
  ...
```

Ještě se o tom zmíníme, ale je třeba uvést, že kromě použití **var** -parametrů pro návrat hodnoty, je dalším důvodem pro použití **var** -parametrů nezanedbatelná velikost předávaných dat. Náročnost předání odkazu totiž nezávisí na velikosti toho, na co odkazuji.

Pravidla a dokumentace Procedury a funkce tvoří nejdůležitější prvek, který používáme při členění programu. Je dobré, když při jejich psaní dodržujeme jistá pravidla. Především u funkce či procedury rozlišujeme:

1. Co od nás chce
2. Co nám vrátí

3. Co kromě toho udělá

Funkce sqrt od nás požaduje nezápornou hodnotu parametru, pak nám vrátí jeho odmocninu (s nějakou zaručenou přesností) a neudělá nic dalšího! Nepípá na nás, nevypisuje 'cekejte pocitram odmocninu', nemění přesnost s níž se provádějí výpočty (i to lze někde měnit) ani nic jiného, jen odmocňuje!

Jiný příklad: procedura `VypišSeznam` ... nemá žádné požadavky na seznam, nic nám nevrátí a neudělaá nic jiného, než že seznam vypíše. Nemění ho, nepřidává položku. Nemá vedlejší účinky.

Pokud to jde, píšeme takovéto čistokrevné procedury a funkce.

Často jsou ale naše procedry a funkce komplikované, pracují jen někdy (bod 1), vrací nám něco (bod 2) a zároveň ještě navíc něco provedou (bod 3). Protože je pak při větším rozsahu programu těžké pamatovat si všechny tyto informace, je vhodné všechny tři body dokumentovat. Jinak řečeno pokud od nás funkce něco chce, musíme si to poznamenat dokud si to pamatujeme. Pokud identifikátor funkce neříká jasně, co fuknce vrací či procedura dělá, přidáme komentář. A pokud funkce má ještě nějaké vedlejší účinky nesmíme zapomenout se o nich zmínit v komentáři. Podobně pokud procedura něco vrací, což od ní většinou nečekáme, nezapomeneme na komenář. Takové komenáře pak umístíme co nejblíže hlavičce procedury či funkce. Například:

```
// hledání kořene funkce (zkusí metodu sečen, neuspěje-li, pak půlení intervalu)
function KorenFunkce(
          f: tFunkce;              // funkce, jejíž kořen hledáme
          a,b : real;              // interval, musí platit a<b
          epsilon: real = 1E-10;   // chyba_x < epsilon*(b-a)
          priChybe: integer = priChybeZastav;
                   ): real;
```

V této deklaraci se již objevují věci, které teprve potkáme. Například `tFunkce` je identifikátor typu, jaký se časem naučíme deklarovat a rovnítko u parametrů funkce umožňuje stanovit jejich výchozí hodnotu tak, že se při použití (volání) funkce nemusíme obtěžovat s jejich vyplňováním.

Výpočty s reálnými čísly

Malujeme funkce. GNUPLOT. Počítáme funkce. Reálná čísla v počítači.

Malujeme funkci

Naše znalosti nám začínají umožňovat psát užitečné programy a díky možnosti dát programu strukturu můžeme jednou nalezená řešení znovu použít.

Nejdříve si ukažme, jak můžeme namalovat graf funkce.

```
program  MalujFci;

const  xa = 0;
     xb = 1;
     N  = 100;

function  MalovanaFce(x:real):real;
begin
  MalovanaFce := x*exp(x)
end ;

var  x,y : real;
     i : integer;

begin
   for  i := 0  to  N  do begin
    x := xa + (xb-xa)*i/N;
    y := MalovanaFce(x);
    writeln(x,'␣',y);
   end ;
end .
```

Především si všimněme, že zde je malovaná funkce izolovaná do zvláštní funkce a příkazy těla programu se výpočtem funkční hodnoty nezabývají. Sice jsme si tím program trochu zkomplikovali, ale až budeme chtít malovat jinou funkci, bude nám identifikátor funkce připomínat, kde je třeba program změnit.

I při letmém prohlédnutí kódu ovšem okamžite vidíme, ze program nic nemaluje, pouze vypíše tabulku skládající se ze dvou sloupečků oddělených mezerou. V prvním je hodnota nezávislé proměnné, ve druhém funkční hodnota. Proč se tedy tady mluví o malování fukce?

Pokud bychom měli "namalováním grafu funkce" na mysli zobrazení grafu na obrazovce počítače a posléze toho i dosáhli, brzy bychom došli k názoru, že si ten obrázek ještě chceme uložit. Po chvíli bychom pak pak zjistili, že nejmenší omezení pro budoucí práci s obrázkem dosáhneme, pokud si poznamenáme funkční hodnoty namalované funkce pro případ, že bychom si chtěli prohlédnout detailní chování funkce v okolí nějakého bodu nebo místo lineárního zvolit logaritmické škálování os, atp. Proto si necháváme otevřené všechny moznosti, a jednoduše vypisujeme pouze funkční hodnoty.

Aby data jentak "nezmizela" za horním okrajem okénka konsole, provedeme trik, který jsme jiz použili: přesměrujeme výstup programu do souboru. Možná si ještě pamatujete, že se to dělá tak, že na příkazové řádce kromě spuštení programu požádáme též o přesměrování uvedením znaku '>' a názvu souboru:

```
C:\Projects\prog\pokusy> MalujFci  >   graf1.dat
```

To ale znamená, že budme potřebovat něco, co nám z hodot uložených v souboru graf1.dat udělá obrázek. Na počítaci se takové něco nazývá obecně program a jak to s programem bývá není nad to, když uz někdo takový program napsal a dovolí nám ho používat.

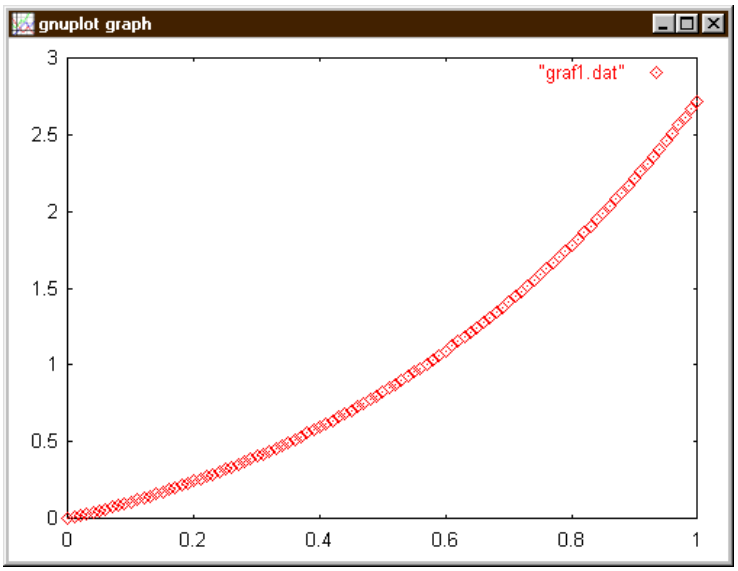
Malujeme funkci - GNUPLOT

Pro naše potřeby bude ideální program s názvem **gnuplot** (a jak nám napovídají první písmena názvu, nebudeme za něj muset nic platit). Uživatel ovládá gnuplot prostrednictvím príkazů.

Například soubor, který se skládá ze dvou sloupců čísel vykreslíme jako graf funkce tak, že zadáme příkaz

```
plot "graf1.dat"
```

bouhužel takto ještě nezískáme, co chceme, protože nám vyleze graf složený z puntíků.

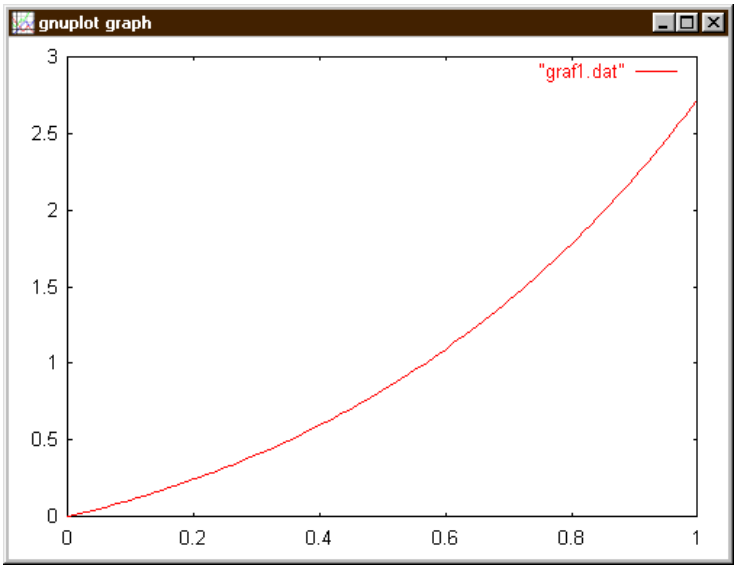


Proto budeme chtít program přesvedčit, aby maloval data ze souboru pomocí čar. Máme dvě moznosti. Buď k příkazu **plot** přidáme na konec „**with lines**“, tedy

```
plot "graf1.dat" with lines
```

nebo změníme nastavení pomocí příkazu **set** a pak uz můžeme jen poroučet **plot** , **plot** , ...

```
set style data lines
plot "graf1.dat"
```



K jednou zadanému příkazu se můžeme vracet pomocí kláves [nahoru] a [dolu], takze se nemusíme opakovaně obtěžovat s vypisováním názvu datového souboru.

Budeme-li chtít vykreslit do jednoho grafu data ze dvou souborů jendoduše názvy souboru v úvozovkách oddělíme čárkou

```
plot "graf1.dat", "graf2.dat"
```

Konečně, pokud má soubor tvar tří (nebo více) sloupečků čísel, můžeme nechat vymalovat nejdřív první versus druhý sloupec hodnot a poté první versus třetí sloupec následujícím příkazem:

```
plot "graf2.dat", "graf2.dat" using 1:3
```

jak je vidět, **using 1:2** jsme psát nemuseli, to se rozumí samo sebou.

Program gnuplot si podle rozsahu malovaných hodnot sám určí v jakém rozsahu se mají oškálovat osy. Pokud ale chceme některou z takto určených hodnot změnit, můžeme hned za slovo plot přidat meze v hranatých závorkách oddělené dvojtečkou. Vyzkoušejte, co udělají následující příkazy:

```
plot [0.2:0.5] "graf1.dat"
plot [:0.5] "graf1.dat"
plot [0.2:] "graf1.dat"
plot [][0:10] "graf1.dat"
plot [0.2:][:10] "graf1.dat"
```

Časem budeme potřebovat vědět, že pokud v datovém souboru vynecháme prázdný řádek, chápe gnuplot následující data jako novou křivku, pokud vynecháme dva prázdné řádky, chápe data jako rozdělená na sekce, přičemž můžeme s jednotlivými sekcemi pracovat zvlášť pomocí slova **index** .To použijeme později, tadˇ jen je dobré vědět, že když potřebujeme znát správné použití třeba slova index, napíšeme: **help index** . Dále je dobré vědět, že až budeme potřebovat obrázek začlenit do nějakého článku, nabídne nám gnuplot možnost vytvořit obrázek v moha ruzných formátech (mimo jiné jako postscriptový soubor pro úcely publikace v knize ci časopise a png-soubor (případně gif) pro zveřejnění grafu na "webu"). Samozřejmě, stačí napsat **help postscript** případně **help png** či **help gif** a dozvíme se jak na to **.**

Jak asi začíná být zřejmé, program gnuplot nám v přednášce postačí pro běžné malování křivek a pro svoji jednoduchost a pružnost se vám nejspíš stane uzitečným pomocníkem i v následujících letech . Až budeme ovšem v přednášce hovořit o 2D grafice a budeme místo průbehu funkcí třeba vybarvovat trojúhelníky, použijeme jinou metodu.

Matematické fukce

Podle toho, že v Pascalu máme zdarma tak málo matematických funkcí (abs, sqr, sqrt, sin, cos, arctan, exp a ln) by jeden mohl hádat, že Wirth měl aversi k matemtické analýze. Pravděpodobne ale jde o důsledné uplatnění principu jednoduchosti. Protože prehršle různých funkcí znamenala nezbytně prodloužení manuálu a právě nepřehlednost jazyků té éry, byla tím, co informatici generace autora Pascalu velmi kritizovali. Osud jazyka PL/I napovídá, že nejspíš měli v něčem pravdu.

Znamená to snad, že se tedy např. máme navždy smířit s absencí funkce ArcSin v Pascalu? Protože víme, že mužeme definovat nové funkce, je odpoved jasná: definujeme funkci ArcSin:

```
function ArcSin(sinus : real) : real;
{ spocte uhel, jehoz sinus zname }
var cosinus: real;
begin
  cosinus := sqrt(1-sqr(sinus)); {bereme vzdy znamenko +sqrt(...)}
  if cosinus=0 then if sinus>0 then ArcSin := Pi/2
                               else ArcSin := -Pi/2
               else ArcSin := ArcTan( sinus / cosinus );
end;
```

A je to! Za povšiknutí stojí, že výpočet se zhroutí pokud bychom chtěli počítat arcusinus arumentu v abolutní hodnotě větší než jedna.

Řady

Často jsou funkce dány ve formě mocninné řady. Vezměme třeba následující vzorec pro výpočet obvodu elipsy:

$$L = 2\pi a \left[1 - \sum_{k=1}^{\infty} \left(\frac{1.3.5...(2k-1)}{2.4.6...2k} \right)^2 \frac{\epsilon^{2k}}{2k-1} \right]$$

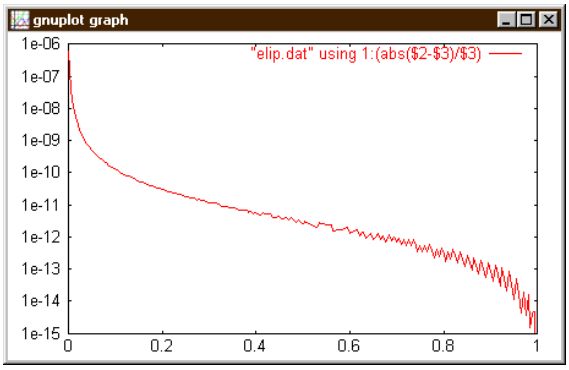
Tento vzoreček si říká o přímočarý přepis. Jedinou otázkou je jak dlouho řadu sčítat. Budeme předpokládat, že řada konverguje dostatečně rychle, takže ukončení rady členem nějaké velikosti znamená chybu ve výpoctu stejného řádu (viz dále). V okamžiku, kdy tento předpoklad neplatí, není řada vhodná pro sčítání a musíme nalézt jinou formulku [Jarník Integrání pocet II].

```
function ObvodElipsy(a,b : real) : real;

const posledni = 1E-12;

var epsilon,
    s,ds,f2: real;
    k      : integer;

begin
  epsilon := sqrt(a*a-b*b)/a;
  s  := 1;
  k  := 1;
  f2 := 1;  {zde budeme hromadit soucin  sqr   (1*3*5*.../(2*4*6*...)   )*eps^(2k)}
  repeat
    f2 := f2*sqr((k+k-1)/(k+k)*epsilon);
    ds := f2/(k+k-1);
    s  := s-ds;
    k  := k+1;
  until ds<posledni;

  ObvodElipsy := 2*Pi*a*s;
end;
```

Všimněte si postupu obvyklého u takovéhoto typu sčítání řad. Obecně se snažíme provádět uvnitř cyklu co nejméně operací, a protože lze většinou jednoduše k-tý koeficient odvodit z toho předchozího, není potřeba do cyklu sčítání vkládat ještě cyklus počítající všechny ty součiny. (Puntičkář by možná ještě odstranil dělení v přirazení ds := f2/(k+k-1)). Daleko důlezitejší než pár zbytecných operací je ovšem vědět, v jakémrozsahu parametrů se na funkci můžeme spolehnout. To je ovšem především záležitost matematické analýzy. Pokud má funkce jednoduchou formu závislosti na parametrech lze ale chování "naprogramované" funkce ověřit. Pro představu je na následujícím obrázku závislost relativní chyby výsledku volání funkce `ObvodElipsy(1,b)` na velikosti vedlejší poloosy b. Pripomeňme si, že jsme zvolili

const `posledni = 1E-12;`

a vzhledem k přesnosti zvolené metody bychom asi měli někam do kódu přidat varování, pokud podíl parametrů b/a není dostatečně velký.



Námět na přemýšlení: jak získat data potřebná pro určení chyby, když nemáme po ruce přesné hodnoty s výjimkou b=a a b=0, tedy kružnice a úsečky?

Funkce určené rekurentními vztahy

Nejjednodušším příkladem je samozřejmě funkce faktoriál:

$$n! = n(n-1)!$$

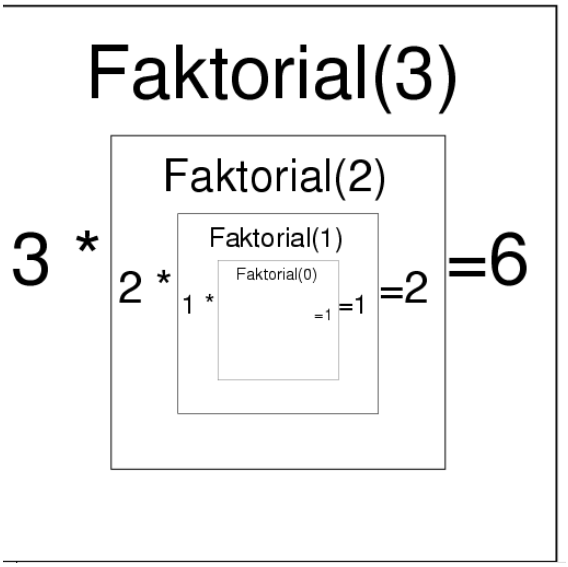
$$0! = 1$$

Jiným standardním příkladem je Fibonacciho posloupnost:

$$F(n) = F(n-2) + F(n-1)$$

$$F(0) = 0, F(1) = 1$$

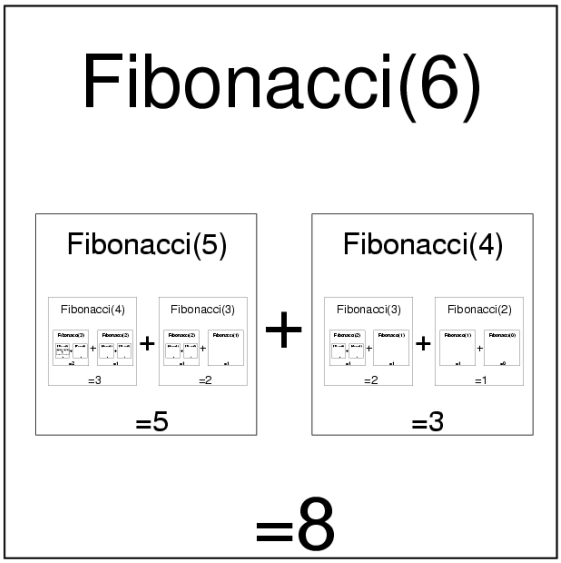
Jakkoli je nasnadě použít pro výpočet prostředky rekurse jazyka Pascal, nebudeme je v těchto případech raději používat. Výpočet faktoriálu založený na rekursi lze znázornit takto:



a odpovídá následujícímu kódu:

```
function  faktorial(a:integer):real;
begin
  if a<=1 then  faktorial := 1
          else  faktorial := a*faktorial(a-1);
end ;
```

Pro Fibonacciho posloupnost má rekuse podobu:



Neefektivnost kódu založeného na rekursi je tedy naprosto zřejmá.

Cvičení : Vyzkoušejte si napsat rekurentní versi funkce Fibonacci(n : integer).

Cvičení: Kolikrát se zavolá funkce Fibonacci, kdyz se takto pokoušíme spočíst Fibonacci(6)?

Jak tedy převedeme matematický rekurentní vztah na návod pro počítač? Nejjednodušší je samozrejme případ funkce faktorial

```
function  faktorial(a:integer):real;
var  i : integer;
     s : real
begin
  s:=1;
  for i:=2 to a do s := s*i;
  faktorial := s;
end ;
```

Takto zapsaný postup má jediný háček: vrací rozumnou hodntu i pro záporné hodnoty parametru a . Podobně jako by nebylo nejlepší, kdyby nám *sqrt* vracelo nulu pro záporné parametry (dáváme přednost krachu programu), měli bychom přidat test na záporné hodnoty a a program případně ukončit.

Cvičení: Zkuste na cyklus prevést výpocet funkce Fibonacci.

Zde si ukázeme, jak převést na cyklus složitější variantu Fibonacciho vztahu. Zadání, které nalezneme v učebnici fyziky zní:

$$P_n(x) = \frac{(2n-1)}{n}\, x\, P_{n-1}(x) - \frac{(n-1)}{n}\, P_{n-2}(x)$$
$$P_0(x) = 1$$
$$P_1(x) = x$$

kde Pn(x) je tzv. Legendreuv polynom a během studia jej mnohokrát potkáte. Pro nás je úloha omezena na nalezení hodnoty P n (x) pro dané celé $n>=0$ a reálné x . Jak je vidět, je potřeba rozlišit případy n=0, n=1 a zbytek, kdy použijeme uvedenou formulku opakovaně (n-1)-krát. Zde je jedna z variant:

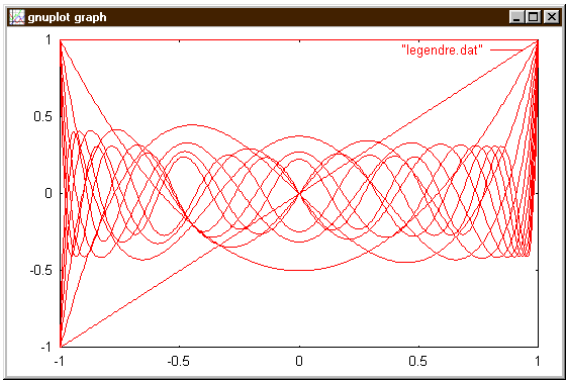
```
function  LegendreP( m : integer; x : real ) : real;
var  Pn2, Pn1,Pn : real;
             n : integer;
begin
   Pn := 1; {P 0 (x)}
   if  m = 1  then  Pn := x; {P 1 (x)}
   Pn2 := 1;
   Pn1 := x;
   for  n := 2  to  m  do begin
      Pn := ((n+n-1)*x*Pn1 - (n-1)*Pn2 )/n;
      Pn2:= Pn1;    { Pn -> Pn1 -> Pn2 }
      Pn1:= Pn;
   end ;
   LegendreP := Pn;
end ;
```

Příkaz cyklu **for** se nám postará o zvyšování n od 2 až do m , ale s každým zvýšením hodnoty n je potřeba také posunout hodnoty v proměnných, které si "pamatují"hodnoty P n-2 a P n-1 .

Pro vykreslení použijeme následující program

```
program MalujLegendreovyPolynomy;

const xa = -1;
      xb = 1;
      N  = 400;

function LegendreP( m : integer; x : real ) : real;
... // viz výše

var k,i : integer;
    x,y : real;

begin
  for k := 0 to 12 do begin
    for i := 0 to N do begin
      x := xa + (xb-xa)*i/N;
      writeln(x,' ',LegendreP(k,x));
    end;      { graf pro pevne x}
    writeln; { vypiš dva prázdné řádky}
    writeln;
  end;        { další k}
end.
```

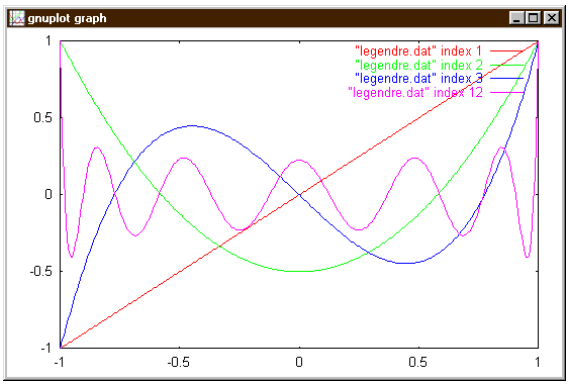
Ten vypíše tabulku hodnot Funkce $P_0(x)$, poté dva prázdné řádky, pak tabulku hodnot funkce $P_1(x)$, pak dva prázdné řádky atd. až nakonec tabulku funkčních hodnot $P_{12}(x)$. Zadáme-li programu guplot příkaz **plot 'legendre.dat'** , kde soubor *legndre.dat* obsahuje výstup našeho programu, dostaneme následující obrázek



Protože jsme ale oddělili jednotlivé tabulky hodnot dvěma prázdnými řádky, můžeme si z toho zmatku vybrat jen ty křivky, které nás zajímají a to pomocí slova **index** následujícího za názvem datového souboru. Např.

```
plot "legendre.dat" index 1, "" index 2,"" index 3, "" index 12
```

nám vymaluje tabulky hodnot polynomu P1, P2, P3 a P12. Ještě se k tomu vrátíme v povídání o polích, ale je dobré vědet, že první sekci dat odpovídá **index 0** a tak shodou okolností se index sekce shoduje s řádem Legendreova polynomu (název stejného souboru není nutno opakovat, proto prázdné úvozovky).



Pozor : někdy se může stát, že rekurentní vztah podobný výše uvedenému pro Legendrovy polynomy nefunguje, neboť s rostoucím n může do závratných výšek narůst úplně drobná počátecní nepřesnost. Diskuse tohoto jevu je ovšem mimo možnosti této prednášky.

Polynomy určené koeficienty (Hornerovo schéma)

Protože nás matematici učí, že spojité funkce můžeme aproximovat polynomy, často mají počítané funkce tvar polynomů. Následující funkce počítá s relativní chybou pod 1E-12 obvod elipsy. Hodnoty koeficientů polynomu samozřejmě spadly s nebe (od pana Čebyševa).

```
function ObvodElipsyP(a,b : real):real;
var x : real; {vzhledem k výrazu níže volíme krátký identifikátor}
begin
  x := sqrt(a*a-b*b)/a;   {epsilon tj. výstřednost elipsy }

  if x>0.5 then
  begin
    Writeln('Pouzivam␣aproximaci␣platnou␣do␣vystrednosti␣0.5!␣ale␣chce␣se␣po␣me:',x);
    Halt;
  end;

  x := ((((((((((-0.7447687857522e-1*x+0.1590001893248)*x-0.1740344498264)*x
      +0.1042163635809)*x-0.5263574825627e-1)*x+0.1129877259030e-1)*x
      -0.2157908636362e-1)*x+0.2458101342560e-3)*x-0.4689377871622e-1)*x
      +0.8483390673316e-6)*x-0.2500000198143)*x+0.1811318676024e-9)*x+0.9999999999997;

  ObvodElipsyP := 2*Pi*a*x;
end;
```

Za povšimnutí stojí jen způsob, jímž je polynom vyčíslován. Vidíme, že se nikde nemusí počítat bůhvíjaké mocniny x, vše vyřešil Mgr. Horner pomocí závorek a celé se to po něm jmenuje Hornerovo schéma.

Bohužel, v některých situacích není určování hodnoty polynomu z koeficientů vhodné, protože omezená přesnost, s níž jsou v počítači reálná čísla realizována vede k velké nepřesnosti výsledku.

Iterační algoritmy pro hledání kořenů

Tvoří velmi důležitou třídu algoritmů. Povětšinou spočívají v aplikaci zázračné formulky, která říká jak z méně přesného výsledku vyrobit výsledek přesnější a jejím opakování až do dosažení požadované přesnosti. Zde si ukážeme jak několik takových *formulek* a algoritmů na nich založených.

Přirozeně také patří do kapitoly o pasní funkcí, protože nalezení kořene funkce závislé na nějakém parametru znamená vyčíslení příslušné implicitně definované funkce.

Půlení intervalu

Nabývá-li spojitá funkce v bodě a zápornou a v bodě b kladnou hodnotu, musí se nejméně jeden kořen funkce nacházet někde mezi těmito dvěma hodnotami. Zkusíme-li uprostřed spočíst funkční hodnotu uprostřed intervalu v bodě *c=(a+b)/2* buď rovnou nalezneme kořen, nebo nalezneme kratší podinterval, ve kterém se kořen zaručeně nachází. Podle znaménka c je to buď <a,c>nebo <c,b>. Pokud je tento podinterval ještě stále moc velký, celý postup opakujeme. Délky intervalů tvoří geometrickou posloupnost s kvocientem 1/2. Proto každých deset iterací přidá tři desetinná místa a po 52 iteracích dosáhneme přesnosti s níž je uložena neznámá, pokud uvažujeme, že jsme začali s intervalem <1,2>. Pokud bychom se na reálnou proměnnou dívali ve dvojkovém zápise, dá se zhruba říci, že v každém kroku iterace přidáme jeden bit přesnosti. Toto je nejpomalejší metoda hledání kořene ale musíme ji ovládat. Pokud totiž máme dost času a nebo je funkce do té míry ošklivá, že rychlejší metody nemůžeme použít, je pro jeho spolehlivost rozumné užít právě půlení intervalu.

Newtonova metoda

Na rozdíl od půlení intervalu, vyžaduje newtonova metoda, aby poblíž kořene byla funkce dostatečne hladká. To proto, že metoda předpokládá, že funkci lze nahradit prvním diferenciálem a ten jako lineární rovnici použít k hledání kořene:

$$f(x_1) = f(x_0 + \delta x) = f(x_0) + f'(x_0)\delta x + ... = 0$$

a tedy, když vypočteme hodnotu x_1

$$x_1 = x_0 + \delta x = x_0 - \frac{f(x_0)}{f'(x_0)}$$

Obdobně jako u půlení intervalu tedy opakujeme jisto zázračnou formulku, tentokrát

$$\boxed{x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}},$$

dokud nedosáhneme požadované přesnosti. Je třeba uvést, že metoda tečen je mnohem rychlejší, ale může snadno selhat. Existují různě komplikovaná kritéria garantující konvergenci, ale v praxi je obtížné je ověřovat. Většinou stačí ukončit program s chybou, pokud počet kroků Newtonovy metody přesáhne, řekněme, 20.

Jako příklad použijeme snad nejjednodušší možný problém - výpočet převrácené hodnoty čísla. Představme si na chvíli, že Pascal spolu s umocňováním neovládá ani dělení. Co teď? Podíl a/b můžeme vyjádřit jako součin a s převrácenou hodnotou b. Jak ale spočíst převrácenou hodnotu? Zkusíme hledat kořen rovnice

$$f(x) = b - \frac{1}{x}$$

Newtonova metoda dokáže, že pokud vezmeme x_0 dostatečně blízko kořene, bude číslo

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = x_0 - \frac{b - \frac{1}{x_0}}{\frac{1}{x_0^2}} = x_0 - (bx_0^2 - x_0) = x_0 - x_0(1 - bx_0)$$

ještě mnohem blíže. Vidíme, že na vypočtení nám stačí pouze násobení a sčítání. (Že by bylo možné převést dělení na sérii násobení a sčítání? Ano, překvapivě, některé počítače právě takto počítají podíl reálných čísel.)

Nejdříve to zkusíme pro konkrétní hodnotu např. 0.9 a první přiblížení převrácené hodnoty odhadneme na 1.1. K jaké posloupnosti přiblížení převrácené hodnoty 1/0.9 povede Newtonova metoda?

x_0 =1
x_1 =1.1
x_2 =1.111
x_3 =1.1111111
x_4 =1.111111111111111

Chování chyby se dá studovat obecně, ale nám pro ilustraci vystačí tento příklad. Vidíme, že zatímco půlení intervalu by přidalo 0.3 cifry na iteraci, dosáhli jsme v každém kroku zdvojnásobení počtu platných cifer a po třech iteracích musíme skončit, protože již neumíme čísla ukládat přesněji.

Metoda regula falsi

Ne vždy ale dokážeme spočítat hodnotu derivace. Jistým vylepšením metody půlení intervalu je předpokládat, že funkce mezi body a a b vypadá téměř jako úsečka a zkoušet místo půlky intervalu vzít jako kandidáta na kořen prusečík této sečny s osou x , tedy

$$c = a\frac{f(b)}{f(b)-f(a)} - b\frac{f(a)}{f(b)-f(a)}.$$

Nyní se podobně jako u půlení intervalu rozhodneme podle znaménka *f(c)* tak aby kořen ležel ve zvoleném intervalu:

$$<a,b> \rightarrow \begin{cases} <c,b> & f(c)f(a) > 0 \\ <a,c> & f(c)f(a) < 0 \end{cases}.$$

Pro funkce, které nemění v blízkosti kořene znaménko druhé derivace tato metoda neustále upravuje jen jednu mez, jak uvidíme v našem příkladu hledání kořene $f(x) = b - 1/x$:

[a_0,b_0] = [1, 1.200000000000000]
[a_1,b_1] = [1, 1.120000000000000]
[a_2,b_2] = [1, 1.112000000000000]
[a_3,b_3] = [1, 1.111200000000000]
[a_4,b_4] = [1, 1.111120000000000]
[a_5,b_5] = [1, 1.111112000000000]
[a_6,b_6] = [1, 1.111111200000000]
[a_7,b_7] = [1, 1.111111120000000]
[a_8,b_8] = [1, 1.111111112000000]
[a_9,b_9] = [1, 1.111111111200000]
[a_{10},b_{10}]= [1, 1.111111111120000]
[a_{11},b_{11}]= [1, 1.111111111112000]
[a_{12},b_{12}]= [1, 1.111111111111200]
[a_{13},b_{13}]= [1, 1.111111111111120]
[a_{14},b_{14}]= [1, 1.111111111111112]
[a_{15},b_{15}]= [1, 1.111111111111111]

Zjednodušeně se dá říci, že v v blízkosti kořene se chyba snižuje geometrickou řadou, přičemž kvocient je dán mírou nelinerity. Jakkoli nemůže tato metoda kořen "ztratit", může se tedy stát, že se metoda regual falsi chová i hůře než půlení intervalu.

Metoda sečen

To že jeden z krajních bodů zůstával v minulé metodě trčet na místě, výrazně snižovalo rychlost konvergence. Pokud upustíme od požadavku různých znamének funkce f v bodech a a b , můžeme mít vždy po ruce poslední a předposlední aproximaci kořene a z nich počítat odhad polohy kořene podle stejného vzorce. Ten navíc upravíme na následující tvar:

$$c = a\frac{f(b)}{f(b)-f(a)} - b\frac{f(a)}{f(b)-f(a)} = b - \frac{f(b)}{\frac{f(b)-f(a)}{b-a}}$$

s tím, že vždy zahodíme nejstarší odhad kořene

$$x_{n-1} = a, \quad x_n = b, \quad x_{n+1} = c.$$

Všimněte si, že jmenovatel ve složeném zlomku je aproximací derivace funkce; v této podobě si lze snadno vztah pro polohu nového odhadu kořene u metody sečen zapamatovat. Protože se nyní obě hodnoty přibližují kořeni, nepřekvapí, že účinnost metody se blíží Newtově metodě.

```
[a0,b0] = [1.000000000000000, 1.200000000000000]
[a1,b1] = [1.200000000000000, 1.120000000000000]
[a2,b2] = [1.120000000000000, 1.110400000000000]
[a3,b3] = [1.110400000000000, 1.111116800000000]
[a4,b4] = [1.111116800000000, 1.111111114751999]
[a5,b5] = [1.111111114751999, 1.111111111111092]
[a6,b6] = [1.111111111111092, 1.111111111111111]
```

Povšimněte si, že v polovině případů je $a_n > b_n$.

Cvičení: Napište programy, které vypíší postup hledání kořene funkce $f(x) = 0.9 - 1/x$ u všech tří metod, a zkontrolují, zda jsem si výše uvedené numerické hodnoty nevycucal z prstu.

Neplánované chyby při běhu programu

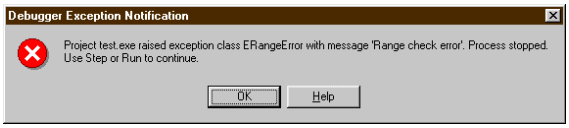
Následujíc program nám vypíše jako výsledek číslo 0. Je to pochopitelný, ale nejspíš nechtěný jev. Chyba spočívá v tom, že číslo 256 **nepatří do rozsahu** typu byte, který je 0..255.

```
program  test;
var  i : byte;
begin
  i:=255;
  i:=i+1;
  writeln(i);
  readln;
end .
```

Proto program trochu ozdobíme:

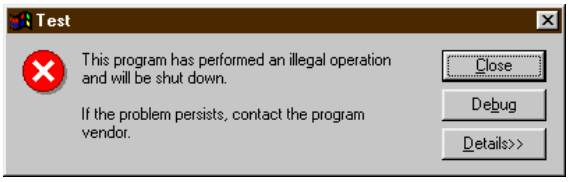
```
program  test;
  uses  SysUtils;
var  i : byte;
{$RANGECHECKS ON $}   { kamkoli nad radek i:=i+1 }
begin
  i:=255;
  i:=i+1;
  writeln(i);
  readln;
end .
```

Vložením `$RANGECHECKS ON`, tedy komentáře začínajícího zankem dolaru jsme překladač uplatili, aby do přeloženého kódu přidal kontrolu na meze při přiřazování a indexování polí. Pozor tedy na znak $ nacházející se za složenou závorkou nebo jejím dálnopisným ekvivalentem (* ktrerý také může omezovat komentář. Když tedy takto zapneme kontrolu rozsahu proměnných, vypsání nuly se nedočkáme, místo toho se po stisknutí klávesy F9 (zkratka pro spuštění programu) objeví varování:



a my po odklepnutí OK musíme v menu vývojového prosředí zvolit *Run/Program Reset* abychom se zbavili toho modrého zvýrazněného řádku s chybou.

Pokud místo práce ve vývojovém prostředí nastane chyba při běhu aplikace spuštěné z příkazové řádky, jsme upozorněni známým varováním, že program selhal:



Pokud by mezi importovanými kniovnami nebyla knihovna SysUtils, skončil by program jen vypsáním chybové hlášky, což nám většinou nestačí pro nalezení a odstranění chyby:

```
C:\ Projects\ prog\ pokusy> test
  Runtime error  201 at 00402570

C:\ Projects\ prog\ pokusy>
```

Podobně jako se výsledek nemusí vejít do proměnné, může se také stát, že se výsledek aritmetické oprace, řekněme násobení, vůbec nevejde do překladačem předpokládané délky slova 32 bitů, a operace vrací nesmysly. V tomto případě jde o jiný typ chyby, tzv **přetečení** a odpovídá jí jiný přepínač: $OVERFLOWCHECKS ON, v krátkém (turbopascalovém) znění $Q+.

```
program test;
 uses sysutils;
 var j : integer;
 {$OVERFLOWCHECKS ON $}
 begin
   j:=50000;
   j:=j*j;
   writeln(j);
   readln;
 end.
```

Používáme knihovny (Velmi úvodní poznámky)

Uvedení žádosti o použití knihoven, např.

```
uses SysUtils;
```

musí následovat před oddílem deklarací a má formu rezervovaného slova `uses` následovaného seznamem identifikátorů oddělených čárkami a ukončeného, ovšem, středníkem.

Import knihovny pomocí konstrukce

```
uses IdKnihovny1, IdKnihovny2, ... , IdKnihovnyN;
```

si můžeme představit jako zkratku za napsání deklarací všeho co uvedené knihovny exportují. Přečteme-li si v nápovědě:

Unit SysUtils

function `IsLeapYear(Year: Word): Boolean;`

Description
Call IsLeapYear to determine whether the year specified by the Year parameter is a leap year. Year specifies the calendar year.

znamená to, že v knihovnou SysUtils je kromě jiného poskytována fukce, která pro letopočty z rozsahu 0..65535 vrátí logickou hodnotu určující, zda jde o rok přestupný, či nikoli. Použijeme-li tedy spojení

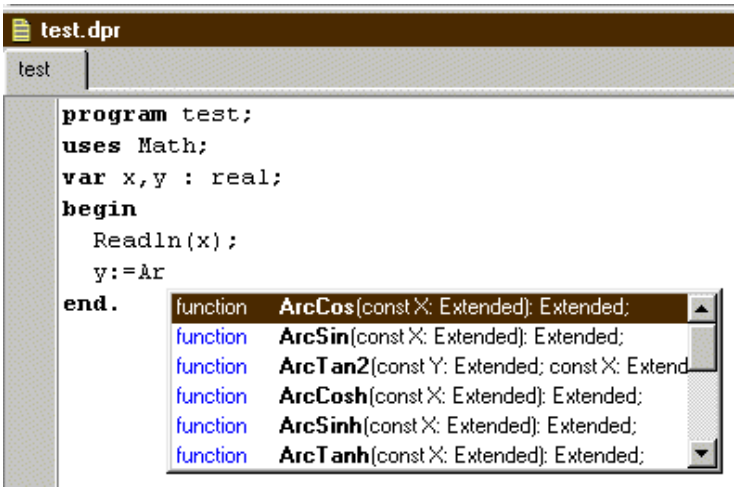
```
uses SysUtils;
```

můžeme funkci IsLeapYear používat, jako bychom si její deklaraci do programu napsali sami.

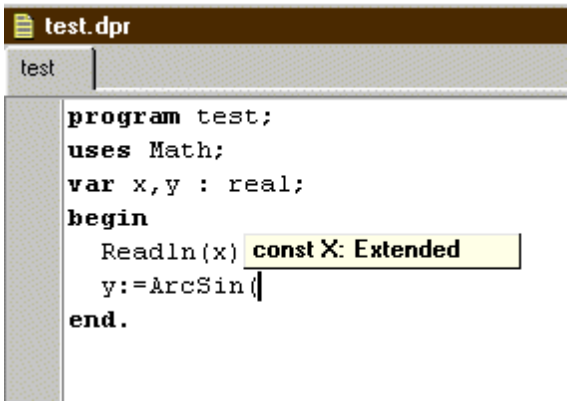
Jak jsme viděli na chování běhových chyb, použití knihovny může mít své důsledky, aniž vůbec použijeme jakoukoli funkci či proceduru z knihovny. Bohužel však podrobnější popis přesahuje rámec přednášky.

Z desítek knihoven dodávaných s překladači Delphi nebo FreePascal, bude pro nás asi zajímavější knihovna `Math`, sdružující mnoho užitečných matematických funkcí opatřených rozumnými identifikátory. Jsou mezi nimi např. goniometrické funkce (`ArcCos`), umocňování (`Power`), převodní funkce (`DegToRad`) atp.

Pokud si pamatujeme alespoň počáteční písmena identifikátoru, můžeme využít pomoci, kterou nám nabízí pracovní prostředí:



Pokud po napsání několika písmen použijeme klávesovou zkratku Ctrl-Mezera , objeví se nám výběr identifikátorů začínajících na daná písmena. Po napsání kompletního identifikátoru a otvírací závorky nám navíc editor nabídne nápovědu v podobě seznamu formálních parametrů a jejich typů:



V knihovně `Math` je definováno nepřehledné množství goniometrických a příbuzných funkcí

```
ArcCos,ArcCosh,ArcCot,ArcCotH,ArcCsc,ArcCscH,ArcSec,ArcSecH,ArcSin,ArcSinh
ArcTan2,ArcTanh,Cosh,Cosecant,Cotan,CotH,Csc,CscH,Sec,Secant,SecH,Sinh,Tan,Tanh
```

a dále např.

```
Sign(x)   // ... vrací −1,0,+1 tak aby x = Sign(x)∗Abs(x)
Ceil(x)   // ... nejmenší celé větší
Floor(x)  // ... největší celé menší
```

nebo logaritmy `Log10(x)`, `Log2(x)`, `LogN(10,x)` , funkce pro umocňování a zaokrouhlování

```
y := Power(10,x)
i := IntPower(3,k)
j := RoundTo(x,-2)   // ... zaokrouhlení na dvě desetinná místa
```

nebo porovnávání reálných čísel s tolerancí

```
if   SameValue(x,y,1E-10)    then ...
if   CompareValue(x,y,1E-12)  <= 0 then ....
```

Rozhodně tu nemůžeme očekávat Besselovy funkce a pod.

Reálná čísla v počítači

.. nejsou tak docela rálná. Nemohou, protože přesný zápis většiny reálných čísel vyžaduje nekonečný počet cifer. Navíc, základní typy, jež počítačové jazyky (i Pascal) nabízejí pro ukládání čísel odpovídají tomu, jakou podobu binární reprezentace čísel vyžaduje k tomu pověřený hardware (procesor), aby to zvládl co nejrychleji.

V proměnné typu `Real` uložené číslo tak obsahuje jen prvních 53 bitů dvojkového zápisu čísla. Jinak řečeno, jen racionální čísla se jmenovatelem rovným nějaké mocnině dvojky a čitatelem menším než 2^{53} lze v typu `Real` uložit přesně.

$$x = s \times \frac{m}{2^e},$$

kde $s = \pm 1$, $0 \leq m \leq 2^{53} - 1$ a $|e| < 2^{1024}$ představují složky zápisu čísla zvané znaménko, mantisa a exponent. Stroj pak sbalí všechnu tuto informaci do 64 bitů a pracuje s ní najednou jako jediným „reálným“ číslem.

```
          0011111111110000000000000000000000000000000000000000000000000000
  1.000000000000 = +1.0000000000000000000000000000000000000000000000000000 x 2^(0)

          1100000000100000000000000000000000000000000000000000000000000000
 -8.000000000000 = -1.0000000000000000000000000000000000000000000000000000 x 2^(3)

          0100000000100010000000000000000000000000000000000000000000000000
  9.000000000000 = +1.0010000000000000000000000000000000000000000000000000 x 2^(3)

          0100000000001001001000011111101101010100010001000010110100011000
  3.141592653590 = +1.1001001000011111101101010100010001000010110100011000 x 2^(1)

          0011111110111001100110011001100110011001100110011001100110011010
  0.100000000000 = +1.1001100110011001100110011001100110011001100110011010 x 2^(-4)
```

Například π je tedy nahrazeno aproximací

$$\pi \approx \frac{7074237752028440}{2251799813685248} = \frac{7074237752028440}{2^{51}}.$$

vzhledem k tomu, že čitatel může mít nejvýš 16 dekadických cifer, dá se říct, že reálná čísla a výpočty s nimi jsou k dispozici jen (necelými) 16 platnými ciframi. Protože mocniny desíti nejsou mocninami dvojky, racionální čísla $0.1 = 1/10, 0.99 = 99/100$, atp. mající konečný dekadický zápis, mají binární zápis nekonečný (podobně jako třeba číslo $1/3$) a tedy nejsou v proměnné typu `Real` uložena přesně.

Kvůli úspoře místa nebo času se v situacích, kdy vystačíme s šesti platnými ciframi někdy používají i reálná čísla s poloviční přesností zabírající 32 bitů – identifikátor typu je `Single`.

Skutečnost, že místo reálných čísel máme jen velmi malou podmnožinu čísel racionálních znamená, že výsledek každé operaci je většinou znám jen v zaokrouhlené podobě. Obvykle platí, že výsledek takové operace je zatížen podobnou chybou jako veličiny do operace vstupující. Důležitou výjimkou jsou operace sčítání, odečítání (a složitější, například trigonometrické funkce sčítání implicitně obsahující):

```
    1.0000000000123456
   -1.0000000000000000
  ---------------------
    0.0000000000123456  ->  1.23456E-11
```

Výsledek tedy najednou představuje číslo s pouhými šesti platnými ciframi. Přesně tento problém nastane při použití známého vzorečku pro kořen kvadratické rovnice. Stačí si prohlédnout tabulky integrálů nebo speciálních funkcí, aby člověk zjistil, jak často se rozdíl blízkých veličin počítá, např. $1 - \sqrt{1 - x^2}$ pro malá x. Existují situace, kdy ztráta přesnosti je extrémní:

```
x  := 0.005;
x3 := x*x*x;
y  := 105*((x3-15*x)*Cos(x) + (15-6*x*x)*Sin(x))/(x3*x3*x);
%
```

Potíž je v tom, že výsledná hodnota y by měla být 0.99999861111, ovšem při použití deklarace `var x,x3,y:Real;` nám vyjde $y \approx 14.007$. Nejde přitom o uměle zkonstruovaný problém, $y = 105 j_3(x)/x^3$, kde j_3 je tzv. sférická Besselova funkce hojně užívaná v mnoha částech fyziky.

Je několik postupů jak obejít tento problém. Vynecháme-li ty, co vyžadují hluboké znalosti reprezentace reálných čísel v počítači, je potřeba nalézt alternativní formu daného výrazu. Někdy to jsou hry s rozvoji, jindy, třeba u oné kvadratické rovnice, stačí úprava výrazu

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}},$$

kde při výpočtu volíme ten vztah, kdy oba sčítance mají stejná znaménka a tedy nedojde ke ztrátě platných cifer.

Příklad na vlastní reprezentaci čísel

Při práci s typ `Real` veštšinou nepřemýšlíme, jak je hodnota čísla uložena – dokud nejsme jeho omezeními nemile překvapeni. Jedním z těchto omezení je velikost čísla, která sice dobře stačí pro běžné výpočty, v následujícím příkladě ale budeme hledat velmi velká Fibonacciho čísla. Program bude mít za úkol nalézt Fibonacciho číslo začínající danými ciframi. Pokud bude cifer hodně, bude takové číslo a sice celé, ale natolik veliké, že (kvůli rychlosti výpočtu) budeme muset vystačit s jejich přibližnou hodnotou danou několika jeho počátečními číslicemi (tedy mantisou). Informace o velikost čísla a pak bude uložena zvlášť za pomoci dekadického exponentu e_a, tedy $a = m_a \times 10^{e_a}$. Tento postup dovolí zacházet i s čísly mnohem většími, než dovoluje rozsah typu `Real` ($\approx 10^{308}$). Pro uložení mantisy o hodnotě z intervalu $<1, 10)$ použijeme typ `Real`. Jeho přesnost nám dovolí nalézt po chvíli čekání Fibonacciho čísla začínající až zhruba 11 ciframi, pak bychom museli zkoumat dopady omezené přesnosti výpočtu. Použitý postup vyložíme přímo s odkazy na kód:

- Na ř. 5-6 definujeme interval mantisy čísla, jaké hledáme (zde číslo začínající čtyřmi šestkami).
- Pro jednoduchost na ř. 8 i exponent deklarujeme jako reálnou proměnnou, nemuseli bychom totiž vystačit s 32-bitovými celými čísly (se znaménkem).
- Algoritmus bude prohledávat Fibonacciho posloupnost počínaje F_1, proto ř. 11.
- Protože číslo reprezentujeme mantisou `a` a exponentem `e10`, je inicializace $F_1 := 1$ rozdělena do dvou příkazů na ř. 13 a 14.
- Páteř programu tvoří cyklus (ř. 18-34), který počítá další a další členy posloupnosti.
- Abychom mohli pohodlně sčítat (sčítání je pro Fibonacciho posloupnost jedinou potřebnou aritmetickou operací), použijeme pro následující Fibonacciho číslo tutéž hodnotu exponentu, proto jen jedno přiřazení pro inicializaci $F_2 := 1$ na ř. 16. Tedy hodnota $a = m_a \times 10^{e_a}$, ale při uložení následujícího členu posloupnosti volíme $b = m_b \times 10^{e_a}$. Hodnota m_b tedy může být větší jak 10. Potom se bude dobře počítat

$$a + b = m_a \times 10^{e_a} + m_b \times 10^{e_a} = (m_a + m_b) \times 10^{e_a}.$$

- Takto získáme výsledek $m_c = m_a + m_b$, hodnota se vztahuje ke společnému exponentu e_a, t.j. `e10`.
- Uvnitř cyklu nejprve zkontrolujeme, zda jsme hledané číslo nenašli. To testujeme na ř. 19 a pokud ano, výpočet ukončíme.
- Rekurentní pravidlo pro výpočet dalšího členu posloupnosti je na ř. 24. Ještě ale musíme posunout původní dvojici $(a, b) = (F_n, F_{n+1})$ na novou $(a, b) = (F_{n+1}, F_{n+2})$. Nesmíme zapomenout zvětšit `n`.
- Po přestěhování je někdy potřeba zvětšit exponent tak, aby mantisa čísla $a = F_n$ zůstala v rozsahu $m_a \in <1, 10)$. Tím se zabývá podmíněný příkaz na ř. 27-31.

```
1  program fibo_cislice;
2
3  // program hledá Fibonacciho číslo začínající danou sekvencí desetinných číslic
4
5  const p = 6.666;
6        q = 6.667;
7
8  var e10,a,b,c,n:real;
9
10 begin
11   n := 1;       // Začínáme s F_1
12
13   e10 := 0;
14   a := 1;       // F_1 = 1*10^0
15
16   b := 1;       // F_2 = 1*10^0
17
18   repeat
19     if (a>=p) and (a<q) then begin
20       Writeln(n:1:0,'. Fibonacciho cislo je cca ', a:-1:15 , 'e' , e10:-1:0 );
21       Halt;
22     end;
23
24     c := a+b;
25     a := b;
26     b := c;
27     if a>10 then begin
28       a:=0.1*a;
29       b:=0.1*b;
30       e10:=e10+1;
31     end;
32
33     n     := n+1;
34   until n>3E12;
35   Writeln('Tolik cislic uz nespoctu dost presne');
36 end.
```

Problémy:

1. Jak zařídít, aby program nalezl číslo začínající ciframi 1000 (např. $F_{662} \approx 1.00076669 \times 10^{138}$) a nezastavil se hned u čísla F_1?

2. Je nepohodlné zadávat zvlášť `p=1.23456789;` `q=1.23456790;` chceme-li nalézt číslo začínající ciframi 123456789. Rozšiřte program tak, aby obě hodnoty p, q odvodil z hodnoty konstanty `PocatecniCifry = 123456789;`.

3. Změňte program tak, aby nalezl Fibonacciho číslo F_m, jehož dekadický zápis končí danou sekvencí číslic. (Program nemusí počítat hodnotu F_m, stačí určit m.)

4. Vztah $m_a \times 10^{e_a} + m_b \times 10^{e_a} = (m_a + m_b) \times 10^{e_a}$ platí pro libovolné m_a, m_b. Kde v programu předpokládáme, že číslo $m_a \in <1, 10)$?

5. Ačkoli $F_{12} = 144$, pro `p=1.44;` `q=1.45` najde program až $F_{79} = 14472334024676221$? Proč?

I data mají svou strukturu

Jednoduché a strukturované typy, Pole, Záznamy.

Kromě deklarací proměnných, konstant, procedur a funkcí můžeme v jazyce Pascal deklarovat i nový typ. Prozatím jsme používali několik jednoduchých typů:

1. *Integer* (a příbuzné varianty *Byte*, ..., *Int64*)
2. *Real* (i ten má kratší variantu: *Single*)
3. *Boolean*

Typ Char

Dalším jednoduchým typem je typ *Char* jehož hodnoty jsou znaky (vlastně jistá malá, historií amerického dálnopisu definovaná podmnožina toho, co dnes chápeme jako znak).

Konstanta typu *Char* se zapisuje jako

1. jeden znak mezi apostrofy, např. ' ' (mezera) nebo '*' nebo '"' , což je jediný znak apostrof.

2. znak # následovaný číslem v rozsahu 0..255. Toto číslo může být též psáno pomocí znaku $ v hexadecimálním zápisu, např. #9 (znak tabulátor), #32 (mezera) je totéž jako #$20 nebo samozřejmě ' ', tedy

```
const mezera1 = '␣'; mezera2 = #32; mezera3 = #$20;
```

jsou všechno mezery. Souvislost mezi číselným zápisem a příslušným znakem je dána tabulou kódů, která se z historických důvodů jmenuje ASCII (americký standardní kód pro výměnu informací).

```
#$20=#32: '␣' '!' '"' '#' '$' '%' '&' ''' '(' ')' '*' '+' ',' '-' '.' '/'
#$30=#48: '0' '1' '2' '3' '4' '5' '6' '7'  '8' '9' ':' ';' '<' '=' '>' '?'
#$40=#64: '@' 'A' 'B' 'C' 'D' 'E' 'F' 'G'  'H' 'I' 'J' 'K' 'L' 'M' 'N' 'O'
#$50=#80: 'P' 'Q' 'R' 'S' 'T' 'U' 'V' 'W'  'X' 'Y' 'Z' '[' '\' ']' '^' '_'
#$60=#96: '`' 'a' 'b' 'c' 'd' 'e' 'f' 'g'  'h' 'i' 'j' 'k' 'l' 'm' 'n' 'o'
#$70=#112:'p' 'q' 'r' 's' 't' 'u' 'v' 'w'  'x' 'y' 'z' '{' '|' '}' '~' #127
```

Znaky #0..#31 nemají původně význam znaků ale kontrolích povelů (původně pro dálnopis).

Znak #9 se nazývá tabulátor, znaky #10 a #13 mají význam přechodu na nový řádek, případně jen "návratu vozíku", kdy se začne psát znovu od začátku ale stejného řádku.

Ostatní znaky z tohoto rozsahu mohou mít podle situace význam řídícího znaku nebo jen třeba jen znaku srdíčka. Pokud k tomu nemáme dobrý důvod (např. níže zvědavost) neposíláme do textového výstupu kontrolní znaky a pro řádkování používáme Writeln. Ten vypíše na každé platformě platnou sekvenci kontrolních zanků pro přechod na nový řádek (ano, ani v tom nepanuje shoda). S tabulátorem nebývají potíže.

Proměnná typu char se deklaruje podle očekávání jako

```
var  c : char;
```

Pro ujasnění významu znaků mezi 9 a 13 vyzkoušejte, co vypíše

```
writeln('abcd',#13,'12');
```

Případně též zkuste jednoduchý cyklus

```
var  c:char;
...
for c:=#9  to  #13  do
begin
  writeln(ord(c),':');
  writeln('abcd',c,'12');
end ;
```

Jak vidíme, proměnná typu char může vystupovat jako řídící proměnná cyklu **for** .

Pořadí znaku v tabulce (tedy číslo, kterým počítač znak representuje) získáme použitím funkce *ord*. Naopak znak z celého čísla vyrobíme použitím konverse typu tak, že idnetifikátor typu použijeme jako funci s jedním parametrem. Proto následující kód dá stejný výstup.

```
var  i:integer;
...
for i:=9  to  13  do
begin
  writeln(i,':');
  writeln('abcd',char(i),'12');
end ;
```

Protože moc nezáleží na tom, na jaký celočíselný znak převedeme, můžeme také místo *ord(c)* psát *integer(c)*.

Spolu s celočíselnými typy je typ char příkladem tzv ordinálního typu, tedy typu repsesentujícího množinu s bobře definovaným uspořádáním, 1:1 zobrazitelnou na nějaký (konečný-víc se nám do počítače nevejde) interval celých čísel. Pro takovou množinu je pak rozumné definovat funkce předchůdce (pred) a následník (succ).

```
pred('B') = 'A', pred(0) = -1, succ(7) = 8    // atp.
```

Deklarace typu

Následující řádek deklaruje typ int a to jako integer.

```
type int = integer;
```

tím získáme možost ušetřit si trochu psaní. Můžeme ale také napsat

```
type integer = int64
```

a naučit náš program počítat místo do 2 147 483 647 až do 9 223 372 036 854 775 807. Tím že takto počínaje touto deklarací zastíníme původní význam identifikátoru si ovšem můžeme přidělat řadu starostí, takže jde o trik nevhodný pro seriózní práci.

Je zřejmé, že takto bychom se daleko nedostali, pouze bychom mohli nazývat staré věci novým jménem. Pascal přináší řadu dalších způsobů, jak zkonstruovat nový typ.

Typ Interval

Nejjednodušší možností je prohlásit, že proměnná smí nabývat pouze hodnot z jistého intervalu ordinálního typu:

```
type  tCisloPoslance = 1..200;
      tMalePismeno   = 'a'..'z';
      tRocnikZS      = 1..9;
```

```
var   PredsedaPK : tCisloPoslance;
      Vychodil   : tRocnikZS;
```

Takto máme možnost přenechat kompilátoru starost o to, aby nám nehlasovalo příliš mnoho poslanců nebo zda vyplnili správně svoji povinnou školní docházku. Prostřednictvím chybových hlášení při kompilaci či běhu se tak můžeme dozvědět o nesrovnalostech.

Uvidíme později, že daleko nejdůležitějším užitím typu interval je určení mezí polí, které chápe Pascal jako zobrazení z podintervalu ordinálního typu do množiny dané typem prvku pole.

Výčtový typ

Je jakýmsi zobecněním typu interval, kde si můžeme prvky sami pojemnovat a nejde jen o podmnožinu výchozího typu.

```
type  Fukce = (Radovy, ClenVyboru, PredsedaKlubu);
```

deklaruje nejen nový typ ale též nové hodnoty v podobě identifikátorů. Kormě funkce ord, která nám dává `ord(Radovy) = 0`, atd máme opět k disposici také `succ` a `pred`, takze plati `succ(Radovy) = pred(PredsedaKlubu)`

Pro počítač je tahle deklarace podobná svým významem následující

```
const  Radovy = 0;
       ClenVyboru = 1;
       PredsedaKlubu = 2;
type   Funkce = Radovy..PresedaKlubu;
```

ovšem jde o izolovaný typ a nemůžeme tak do proměnné výčtového typu přiřadit celé číslo. To zvyšuje bezpečí při psaní programu.

Pozor identifikátory prvků výčtového typu nám mohlo něco zastínit, nebo vést ke kolizi, takže i zde musíme dávat pozor při volbě jmen. Situace je velmi podobná té při deklaraci **const** `ClenVyboru = 1;`.

Někdy má smysl tzv. "Maďarská notace"spočívající v přidání předpony k identifikátoru tak, aby byl hned jasný jeho význam:

```
type  tFukce = (eRadovy, eClenVyboru, ePredsedaKlubu);
```

Tedy, identifikátory typů se poznají podle toho, že začínají malým *t*, identifikátory hodnot výčtových typů začínají (např.) malým e, atp.

Zde je jiný příklad na výčový typ:

```
type  tKarty = (
    ZelenaSedma,  ZaludovaSedma,    KulovaSedma,    SrdcovaSedma,
     ZelenaOsma,   ZaludovaOsma,     KulovaOsma,     SrdcovaOsma,
  ZelenaDevitka,ZaludovaDevitka,  KulovaDevitka,  SrdcovaDevitka,
  ZelenaDesitka,ZaludovaDesitka,  KulovaDesitka,  SrdcovaDesitka,
   ZelenySpodek, ZaludovySpodek,   KulovySpodek,   SrdcovySpodek,
   ZelenySvrsek, ZaludovySvrsek,   KulovySvrsek,   SrdcovySvrsek,
     ZelenyKral,   ZaludovyKral,     KulovyKral,     SrdcovyKral,
      ZeleneEso,    ZaludoveEso,      KuloveEso,      SrdcoveEso
             );
type  tSedmy = ZelenaSedma..SrdcovaSedma;
     tOsmy  = ZelenaOsma..SrdcovaOsma;
     ...

var   SedmaKDalsimuPouziti : tSedmy;
     LibovolnaKarta    : tKarty;

begin
  LibovolnaKarta := ZaludovyKral;
  SedmaKDalsimuPouziti := SrdcovaOsma; // [Error] pokus.dpr(28): Constant expression violates
   subrange bounds
   ...
  LibovolnaKarta := ZaludovyKral;
  SedmaKDalsimuPouziti := LibovolnaKarta;

  writeln( ord(SedmaKDalsimuPouziti)); // Writeln neumi vypsat vyraz vyctoveho typu
  readln;
end .
```

Množiny

Z výše uvedených typů můžeme budovat typ množina. S množinami můžeme provádět obvyklé operace: sjednocení +, průnik *, rozdíl -. Logické operace nad množinami jsou jen podmnožina <=, nadmnožina >= , rovnost = a různost <>. Klíčové slovo `in` lze použít ke konstrukci dotazu na přítomnost prvku v množině, takže

```
LogProm := Znak in Mnozina;
```

je totéž jako

```
LogProm := [Znak] <= Mnozina;
```

Zde se využije, že množinu můžeme specifikovat jejími prvky tak, že seznam hodnot a intervalů oddělených čárkami uzavřeme do hranatých závorek. Zde je delší příklad

```
program  test;
type  tCharSet =  set of  char;

var   PouzitaPismena,RozbiteKlavesy : tCharSet;
     c : char;
begin
```

```
  RozbiteKlavesy := ['x','q','X','Q'];
  PouzitaPismena := []; //prázdná množina

   repeat
    read(c);
    PouzitaPismena := PouzitaPismena+[c];
   until  c='.';

   if  PouzitaPismena*RozbiteKlavesy = []  then  Writeln('Mohu␣Pouzit␣Tento␣Psaci␣Stroj
   ')
                                           else  Writeln('Musim␣Pouzit␣Jiny␣Psaci␣Stroj
   ');

end.
```

Z výše definovaného typu tKarty můžeme sestavit množiny

```
const  sSedmy = [ZelenaSedma..SrdcovaSedma];
       sOsmy  = [ZelenaOsma..SrdcovaOsma];
       ...
       sEsa   = [ZeleneEso..SrdcoveEso];
```

a testovat hodnotu karty vyrazem `(k in sSedmy)` misto `(k<=SrdcovaSedma) and (k>=ZelenaSedma)`.

Pole

(anglicky **array**) je již od počátku počítačového věku nejdůležitější složenou datovou strukturou. Píšeme

```
const   Dim = 3 ;
type   tVektor3 = array[1..Dim] of real;
var  a : tVektor3;
     x : real;
     i : integer;
     b : tVektor3;
```

a myslíme tím, že proměnná *a* je skupina tří reálných čísel. Při deklaraci strukturovaného typu pole musíme uvést typ prvku pole a typ indexu . Typ indexu musí být ordinální typ (s dostatečně malým rozsahem hodnot, aby nám stačila paměť na jejich uložení). Většinou píšeme místo typu indexu rovnou konkrétní interval, ale nikdo nám nebrání psát

```
type t3Dindex = 1..Dim;
     tVektor3 = array[t3Dindex] of real;
```

Operace s Poli: Přístup k prvku pole

S jednotlivými prvky pole můžeme pracovat zvlášť tak, že za identifikátor proměnné typu pole přidáme v hranatých závorkách index:

```
a[1] := 0;
a[2] := x+1;
a[3] := x-1;
```

Identifikátor pole následovaný výrazem v závorkách je první příklad toho, kdy designator není pouhý identifikátor. Vzpomeneme-li si na syntaktické diagramy z druhé přednášky, uvidíme, že přístup k prvku pole můžeme použít na levé straně přiřazovacího příkazu stejně jako ve výraze, pokud tam můžeme použít proměnnou typu z něhož je pole utvořeno.

```
x := a[1]+a[2]+a[3];
```

je tedy správně zapsaný přiřazovací příkaz. Kdybychom jako indexy používali pouze konstanty, vystačili bychom se třemi proměnnými *a1, a2, a3.* Důležité je, že jako index můžeme použít libovolný výraz kompatibilní s typem indexu udaným při deklaraci. Výše uvedený součet tedy můžeme zapsat cyklem.

```
x := 0;
for i := 1 to Dim do x := x + a[i];
```

Podobně jako v přiřazovacím příkazu může me použít prvek pole jako parametr.

```
for i := 1 to Dim do Writeln(i, '␣' , a[i]);
```

Vzhledem k deklaraci spadá ve všech krocích *i* do intervalu *1..Dim* a nedojde tedy běhové chybě. Již víme, že při deklaraci pole musíme uvést typ prvku pole a typ indexu . Při přístupu k prvku pole, ať již na některé ze stran přiřazovacího příkazu, nebo jako parametru volání procedury či funkce, teď kromě samozřejmé podmínky na typ prvku pole (ani nyní nemůžeme psát *CelePole[i]:=RealnePole[i]*), musíme navíc dbát o to aby index byl v povolených mezích.

Operace s Poli: Přiřazení

Pokud vytvoříme nový typ, neumí Pascal s proměnnými tohoto typu příliš zacházet. Kromě sestupu na nižší úroveň, což pro strukturovaný typ pole je použití konstrukce `Identifikator-Pole[index]` , umí jazyk Pascal přiřazovat mezi dvěma proměnnými stejného typu. Dva typy s odlišnými identifikátory jsou stejné pokud jsou navzájem svázány řadou rovnítek v deklaraci typů, kde na obou stranách rovnítka je jen a pouze identifikátor.

```
type   Typ1 = array [1..6] of integer;
       Typ2 = array [1..6] of integer;
       Typ3 = Typ1;
       Typ4 = Typ1; // Typ3 je stjený s  Typ2

var    Z1,Y1: Typ1;
       Z2,Y2: Typ2;
```

```
        Z3   : Typ3;
        Z4   : Typ4;
```

```
...
    Z1 := Y1;  // OK
    Z3 := Z1;  // OK
    Z2 := Y2;  // OK
    Z3 := Z4;  // OK
 ...
    Z1 := Z2;  // NE!
    Z2 := Z3;  // NE!
```

Chceme-li rozšířit schopnosti jazyka pracovat s nově definovaným typem, musíme použít **procedury** a **funkce** , které mají některý z parametrů tohoto typu.

Pole jako Parametry procedur a funkcí

Z předchozího vyplývá, že naše vektory nemůžeme sčítat, jak bychom chtěli:

```
type tVektor3 = array [1..3] of real;
var  a,b,c : tVektor3;
     x     : real;
...
   a := b+c; // NE!!!!
```

To je jistě škoda a je třeba hledat nějakou náhradu. Přeskočíme prozatím nejnovější možnosti např. překladače *FreePascal* a akceptujeme, že nelze rozšířit definici operace '+' tak, aby dávala smysl i pro pole. Proto pak musíme psát:

```
procedure  SectiV3( a,b : tVektor3; var c : tVektor3);
begin
   c[1] := a[1]+b[1];
   c[2] := a[2]+b[2];
   c[3] := a[3]+b[3];
end ;
...
   SectiV3(b,c, a);
```

nebo

```
function  SectiV3( a,b : tVektor3) : tVektor3;
begin
   SectiV3[1] := a[1]+b[1];
   SectiV3[2] := a[2]+b[2];
   SectiV3[3] := a[3]+b[3];
end ;
...
   a := SectiV3(b,c);
```

Tato druhá varianta vypadá přehledněji, ale jde spíše o výjimečné použití funkce vracející hodnotu strukturovaného typu. Jde o konstrukci, kterou připouštějí až současné překladače Pascalu. Měli bychom mít na paměti, že v tomto případě probíhá stěhování výsledku nadvakrát, nicméně, možnost zapsat formulky aspoň trochu čitelně je příjemná:

```
a := SectiV3(VektorovySoucin(b,c),VektorovySoucin(d,a));
```

Kdy předávat pole odkazem ?

Až na výjimky **pokaždé** ! Proto náš příklad

```
   a := SectiV3(b,c);
```

s vektorovou algebrou byl *špatně* hned dvakrát. nejen, že zbytečně kopíroval *jeden* výsledek funkce do cílové proměnné při přiřazení, ale především *dvakrát* kopíroval hodnotu obou parametrů předávaných hodnotou. Proto pro seriózní práci s poli nebudeme moci předání pole uskutečňovat **hodnotou**. Obecně musíme dbát o to aby *režie* při volání procedury a navracení výsledku nebyla příliš velká. Zde nejde ani tak o to, zda je únosné 5 nebo 50%, ale zejména o situace, kdy vlastní výpočet je dokonce mnohem kratší než předání hodnotou. Příkladem budiž výpočet stopy velké matice, kdy by vytváření její kopie pro předání hodnotou mohlo zabrat sto či tisícinásobek času potřebného pro výpočet součtu prvků na diagonále.

Konstantní parametry

Kromě předání odkazem existuje ještě jedna možnost, jak se u hodnotou předávaných parametrů vyhnout kopírování jejich hodnoty. Jde o tzv. konstantní parametry.

```
function  SectiV3(   const   a,b : tVektor3) : tVektor3;
begin
 SectiV3[1] := a[1]+b[1];
 SectiV3[2] := a[2]+b[2];
 SectiV3[3] := a[3]+b[3];
end ;
```

Tím překladači sdělíme, že hodnotu parametru nehodláme měnit, a on s ním bude zacházet šikovněji, především si nebude vytvářet kopii. Jak víme, hlavička procedury nebo funkce obsahuje nejen informaci pro kompilátor, ale vyjadřuje i záměry autora kódu. Identifikátor funkce by měl říkat co vrací. Identifikátor procedury, co dělá, a identifikátory parametrů by měly být také výmluvné. Pokud je naším záměrem, aby konkrétní parametr sloužil pro vstup hodnoty, je vhodné jej označit jako konstantní. Tím se vyvarujeme možné chyby, kdy se prostřednictvím parametru předávaného hodnotou pokusíme něco vrátit. Kompilátor si bude stěžovat. Vyzkoušejte!

Pole s více indexy

Z typu `tVektor3` bychom mohli deklarací

```
type  tMatice3 =  array  [1..Dim]  of  tVektor3;
```

vyrobit typ `tMatice3`. Poté bychom mohli psát

```
var  M: tMatice3;
    b: tVektor3;
....

M[1][1]:=1; M[1,2]:=0; M[1,3]:=0; ....

b := M[1];
```

Naproti tomu deklarace

```
type  tMatice3 =  array  [1..3]  of array  [1..3]  of  real;
```

nebo její zkrácená podoba

```
type  tMatice3 =  array  [1..3,1..3]  of  real;
```

by nám nedovolila přiřazovat vektory do M[1] atd.

Příklady použití polí

Pole mají pro nás nepřeberné množství užití. Pro představu pár příkladů:

Seznam (index je jen pořadím v seznamu a nemá sám o sobě význam)

```
var  TazenaCisla :  array  [1..6]  of  1..49;
```

Časové řady (index je stále pořadím v seznamu, ale jeho hodnota má reálný význam – lze z ní např. spočíst, kdy došlo k odečtení hodnoty)

```
var  Teplota :  array  [1..PocetVzorku]  of  real;
```

2D data

```
var  Teplota :  array  [1..PocetVzorkuX,1..PocetVzorkuY]  of  real;
```

2D obrázek (zatím stupně šedi)

```
type  tPixMap =  array  [1..PocetPixluX,1..PocetPixluY]  of  byte ;
var PixMap : tPixMap;
```

3D data

```
var  Teplota :  array  [1..PocetVzorkuX,1..PocetVzorkuY,1..PocetVzorkuZ]  of  real;
```

Tabulka funkčních hodnot

```
var Faktorial :  array  [0..170]  of  real; // 171! se do proměnné typu real nevejde
    Binomial  :  array  [0..1000,0..1000]  of  real; //zkuste urcit presnejsi meze !!!
```

Tabulka na překódovani

```
var  TajnyKod :  array [char]  of  char;
  ObrazkyKaret: array[tKarta] of tPixMap;
          s1 : array[integer] of char; //[Error] pokus.dpr(25): Data type too  large :  exceeds  2
    GB
          s2 : array[real] of char;     //[Error] pokus.dpr(26): Ordinal type  required
```

Poslední dva řádky nejsou správně a je u nich uvedena chyba, kterou nám překladač ohlásí.

Paměťové nároky polí

Pole mohou velmi snadno vyčerpat dostupnou paměť. U polí s jedním indexem to ještě není příliš aktuální. Pokud budeme ale psát program por odšumování audionahrávek a celou nahrávku se pokusíme nacpat do paměti najednou, můžeme narazit i tady. Pro představu si připomeňme známý fakt, že hodina stereofonní nahrávky v CD kvalitě nám zabere přes půl gigabytu (vzorek tvoří 2x16 bitů, rychlost vzorkování je 44 100 za sekundu). Daleko snazší je vyčerpat paměť při práci s poli se dvěma indexy. Takový RGB obrázek v rozlišení 10 000x10 000 zabere 300MB. Reálná matice se stejnými rozměry zabere skoro gigabyte. Jestliže se nám může číslo 10 000 velké a můžeme doufat, že tak velké matice potřebovat nebudeme, ve třech dimenzích se situace ještě zhorší. Budeme-li chtít nějakou fyzikální veličinu definovanou v prostoru studovat na počítači, často nám nezbyde, než si prostor redukovat na prostorovou mříž a pracovat s hodnotami v uzlech této mříže.

```
program  KrychloHydro;

const N  =  200 ;

type  tGridFunction  =  array   [1..N,1..N,1..N]

var   p,vx,vy,vz : tGridFunction;

....
```

Výše uvedený začátek programu pro naivní počítačovou hydrodynamiku na krychli předpokládá, že na krychli o hraně 200 bodů budeme definovat tři komponenty rychlosti a tlak. Těmito několika řádky jsme si vyžádali 256 MB paměti. Podle povahy problému se ukáže, jestli nám 200 bodů stačí. Pokud budeme potřebovat více, nesmíme zapomenout, že spotřeba paměti roste se třetí mocninou `N`.

Než budete psát diplomovou práci, pravděpodobně vzroste typická kapacita pamětí osobních počítačů 10x. To ovšem znamená, že dovolené `N` vzroste pouze dvakrát.

Algoritmy využívající pole

Eratosthenovo síto

je další klasický algoritmus, a navíc ilustruje, že pole potřebovali již antičtí informatici.

```
program  Sito;

const  N = 50000000;

var  MaDelitel :  array [0..N]  of  boolean;
i,p : integer;

begin
  p:=2; { prvni  prvocislo  }

  repeat
    {krok 1: oznacim  vsechny  nasobky  prvocisla  p}
    i:=p+p;
    while  i<=N  do begin
      MaDelitel[i]:=true;
      i:=i+p;
    end ;

    {krok 2: najdu  dalsi  prvocislo }
```

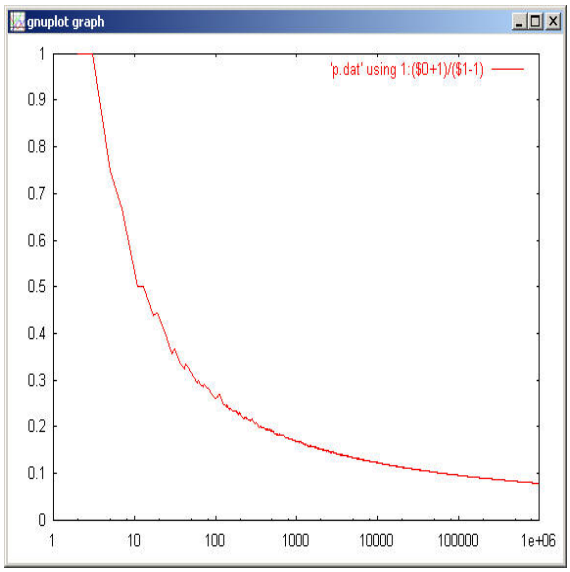
```
    p:=p+1;
    while  MaDelitel[p] {tedy neni to prvocislo}  do  p:=p+1;

  until  p*p> N; { } { a to cele  opakuji  ....  }

  // teď spočtu  počet  prvočí sel  od 2 do N
  p:=0;
  for  i :=2  to  N  do if not  MaDelitel[i]  then  p:=p+1;
  Writeln('Existuje␣',p,'␣prvocilsel␣<=␣',N);
  Readln;
end.
```

Důležitá poznámka: Pro přehlednost využívá algoritmus triku a to, že se spoléhá na vynulování globálních proměnných. (Je to oprávněný předpoklad, ale pokud bychom z hlavního programu učinili proceduru, přestane fungovat! Museli bychom přidat vynulování pole `MaDelitel[]`.) Po skončení hlavní smyčky repeat-until můžeme hodnoty v poli nějak využít. V příkladu se spočte, kolik prvočísel je od 2 do N.

Oba kroky nemusí být v programu tak výrazně odděleny, lze vše zkrátit

```
  while  p*p <= N do begin
    if not MaDelitel[p] then begin // je to prvočíslo, tak  vyškrtávej
      i:=p+p;
       while  i <= N  do begin
         MaDelitel[i]:=true;
         i:=i+p;
       end ;
    end ;

    p:=p+1; // postoupím dále
  end;
```

Samozřejmě můžeme hodnoty všech nalezených prvočísel vypsat, uložit do souboru a následně si například vykreslit obrázek. Takto vypadá relativní zastoupení prvočísel v intervalu 2..N:



Obrázek byl vykreslen následující řadou povelů pro program gnuplot:

```
gnuplot>  set style data lines
gnuplot>  set logscale x
gnuplot>  plot 'SeznamPrvocisel.dat' using 1:($0+1)/($1-1)
```

$1 zde znamená hodnotu čísla v prvním sloupečku (soubor obsahuje jen jeden sloupeček) a $0 je pořadové číslo hodnoty (ta první má samozřejmě pořadové číslo 0).Všimněte si, že v intervalu 2..3 je 100% prvočísel.

Typ záznam

Struktorovaný typ se skládá z menších kousků, podobně jako strukturovaný příkaz se skládá z "menších"příkazů, jednoduchých i strukturovaných (`while .. do for .. do if .. then`).

V případě pole byly jednotlivé kousky stejného typu a přístup k nim jsme měli pomocí indexace. V Pascalu máme ale ještě jednu možnost.

```
type  tVectorXYZ =  record
                      x,y,z : real;
                    end ;
var   a,x  : tVectorXYZ;
```

```
begin
  a.x := 1;
  a.y := -1;
  a.z := 0;

  x   := a;
  x.x := 2;
  ...
end.
```

Podobně jako u polí můžeme

- Přistupovat k menším kouskům, z nichž je strukturovaný typ složen pomocí konstrukce *IdentProm.IdentSlozky*
- Přiřazovat celou proměnnou do jiné stejného typu
- Předávat proměnnou (nebo výsledek volání funkce) jako parametr (hodnotou, odkazem, **const**)

Je libo pole záznamů nebo záznam s poli?

Samozřejmě můžeme kombinovat podle uvážení obě konstrukce a designátory nabyvaji na kráse:

```
type tComplex = record
                  Re,Im : real
                end;

       tCV3     = array [1..3] of tComplex;

       tRGB     =  packed record
            R,G,B : byte;
       end;
```

```
tKomlexniPuntik =  record
                     x     : tCV3;
                     barva :  tRGB;
                   end;

var   A : tKomlexniPuntik ;
      b : tRGB;

      ...

A.x[2].Im := 0;
A.x[1].Re := -1;
...
```

Strukturované příkazy With a Case

Účel příkazu `with` vysvětlí nejlépe příklad.

```
type  tComplex =  record
                  Re,Im : real
     end ;
var   z  :  tComplex;
      Re: Real;

      ...

Re := 1;
with z do
begin
```

```
  Re := 0;
  Im := 1;
end ;

Writeln (Re,'␣',x.Re);
```

Vypíše ' 1.00000000000000E+0000 0.00000000000000E+0000', protože unvitř příkazu **with** přestala být vnější proměnná *Re* vidět a byla zakryta identifikátorem složky záznamu *z*. Vhodným použitím `with` lze tak zestručnit práci s konkrétním záznamem, např.

```
var   U  :   array[1..N] of tComplex;
     ...

with U[k+i+1] do begin
    Re := k;
    Im := i;
end ;
```

místo

```
  U[k+i+1].Re := k;
  U[k+i+1].Im := i;
```

Příkaz Case dokáže nahradit řadu podmíněných příkazů tetsující rovnost jednoho výrazu na několik možností, např.

```
if  n=0  then  f:=1
else if  n=1  then  f:=x
else if  n=2  then  f:=x*x
else if  n=3  then  f:=x*x*x
else if  n=4  then begin  f:=x*x ;  f:= f*f; end;
else f:=exp(ln(x)*n);
```

lze ahradit jediným strukturovaným přikazem

```
case  n of
  0:   f:=1;
  1:   f:=x;
  2:   f:=x*x;
  3:   f:=x*x*x;
  4:   begin  f:=x*x ;  f:= f*f; end;
  else f:=exp(ln(x)*n);
end;
```

Část začínající **else** lze vynechat a také můžeme místo jednoho čísla psát seznam čísel a intervalů, které se ale nesmějí překrývat

```
case  c of
  '-':                    n:=-n;
  '␣a'..'z','_':          n:=n+1;
  '␣A'..'Z','0'..'9','@': n:=n-1;
end;              // kód nemá žádný význam
```

Inicializované proměnné a konstanty.

Dokud jsme pracovali jen s jednoduchými proměnnými, nebyl problém na začátku programu napsat pár přiřazení a inicializovat obsah proměnných. Pole nebo záznamy ale mohou být delší a jejich inicializace sérií přiřazovacích příkazů by byla nepohodlná. Mohou nastat dva případy

1. hodnoty je třeba inicializovat před výpočtem a ty se již nemění.
2. hodnoty je třeba inicializovat před výpočtem, a ty se poté budou dále měnit.

K tomu můžeme použít následující konstrukce se syntaxí:

DeklaraceIniKonstProm: (**var** | **const**) *Ident* **:** *Typ* = *IniKonstVyraz* ';'

IniKonstVyraz: *KonstVyraz* | '(' *SeznamIniKonstVyrazu* ')'

SeznamIniKonstVyrazu : *IniKonstVyraz* (',' *IniKonstVyraz*)*

a zde jsou příklady s inicializovanými poli :

```
const  iFaktorial : array [0..12] of integer =
   (1,1,2,6,24,120,720,5040,40320,362880,3628800,39916800,479001600);
var    ObjemNadob[1..PocetNadob] = (0,0,10);
       CisloKroku : integer = 0;
```

Konstrukce výrazu typu pole je dovolen jen v deklaraci inicializované proměnné nebo typované konstanty , do přiřazovacího příkazu nemůžeme použít

```
ObjemNadobi:=(10-x-y,x,y);          // tohle nejde
ObjemNadobi:=_objemy(10-x-y,x,y);   // OK při vhodně definované funkci _objemy
```

Další příklady:

```
type tSeznamAz10Cisel
         = record
             Pocet : 0..10;
             Cisla : array [1..10] of integer;
           end;

var Dlouhy : tSeznamAz10Cisel = (Pocet : 8; Cisla : (1,2,3,4,5,6,7,8,0,0) );
    Kratky : tSeznamAz10Cisel = (Pocet : 2; Cisla : (1,2,0,0,0,0,0,0,0,0) );
```

Samozřejmě můžeme inicializovat i jednoduché proměnné.

```
var    Oddelovac : char = ',';
       CisloKroku : integer = 0;
```

Proměnné z vícenásobná deklarace jako třeba

```
var   a,b : char ;
```

nemohou být inicilizovány. Neinicializované globální proměnné jsou při startu programu inicializovány na 0 (a to i tehdy, když 0 nepatří do jejich intervalu atp.), zatímco lokální proměnné obsahují před prvním použitím smetí. Typované konstanty smí být lokální (tedy deklarovány v bloku nějaké procedury či funkce), inicializované proměnné nikoli a jsou bohužel dovoleny jen na globální úrovni.

Variantní záznam

Někdy chceme ukládat jednotlivé položky přes sebe. To proto, že význam má jen jedna z nich a rezervovat místo na ty ostatní je zbytečné. Předpokládejme, že si chceme udělat pořádek v plechových geometrických součástkách na našem skladu. Taková plechová součástka je popsaná materiálem, tloušťkou plechu, tvarem a rozměry. Rozměry trojúhelníku se ale určují pomocí tří čísel zatímco u kruhu stačí jen poloměr.

```
program  CaseTest;

type
   tTvar = (tvObdelnik, tvTrojuhelnik, tvKruh);
   tMaterial = (mtHlinik,mtOcel);
   tPlechovyUtvar =  record
                     Material:tMaterial;
                     Tloustka:Real;
                      case  Druh:tTvar  of
                       tvObdelnik: (Vyska, Sirka: Real);
                       tvTrojuhelnik: (StranaA, StranaB, StranaC: Real);
                       tvKruh: (Polomer: Real);
                      end ; //zde končí jakcase takrecord!

function  ObvodIf(W:tPlechovyUtvar):real;
begin
   if  W.Druh=tvObdelnik     then  ObvodIf := 2*(W.Vyska+W.Sirka);
   if  W.Druh=tvTrojuhelnik  then  ObvodIf := W.StranaA+W.StranaB+W.StranaC;
   if  W.Druh=tvKruh         then  ObvodIf := W.Polomer*2*Pi;
end ;

function  ObvodCase(W:tPlechovyUtvar):real;
begin
```

```
    case  W.Druh of
       tvObdelnik:    ObvodCase := 2*(W.Vyska+W.Sirka);
       tvTrojuhelnik: ObvodCase := W.StranaA+W.StranaB+W.StranaC;
       tvKruh:        ObvodCase := W.Polomer*2*Pi;
    end ;
 end ;
```

V ukázce jsou dvě funkce, používající k rozlišení tvaru součástek buď příkaz *if* a nebo *case*.

Cvičení: dodělejte funkce pro plochu, objem a hmotnost součástek.

Alignment a packed record (array)

V současných počítačích je vyzvednutí hodnoty proměnné z paměti, které musí předcházet libovolné operaci s proměnnou, velmi náročnou operací, např. ve srovnání s časem potřebným pro sečtení dvou celých čísel. Aby se urychlila práce počítače, vyzvedává více bytů najednou, řekněme že 8. Pro optimální běh programu je pak vhodné, aby procesor dokázal načíst např. celé číslo (4 byty) na jedinou operaci přístupu do paměti. Nejjednodušším řešením tohoto problému je, že každá položka záznamu o velikosti 4 byty začíná na adrese dělitelné 4 a podobně pro proměnné velikosti 8 bytů. To znamená, že se v paměťovém prostoru pro uložení záznamu nacházejí nevyužitá místa. Konkrétní způsob uložení je nejlépe vyzkoumat experimentálně, protože je dán použitým překladačem. Někdy může být plýtvání opravdu velké:

```
type  rec_brb =  record
                   a:byte; //tady nasleduji 7 prázných bytů
                   j:real;
                   b:byte; //tady nasleduji 7 prázných bytů
                 end ;

    rec_bb =  record
                a:byte;
                b:byte;
              end ;
...
Writeln( sizeof(rec_brb) );  //  ... 24
Writeln( sizeof(rec_bb) );   //  ... 2
```

Tedy pro uložení 10 bytů informace, použil překladač 24 bytů paměťového prostoru. To jsme zjistili použitím universální funkce `sizeof` , která jako parametr akceptuje identifikátor typu nebo proměnnou libovolného typu (jde jakoby o předávání odkazem, takže ne libovolný výraz) . Vrátí pak počet bytů, který proměnná či typ zabírá.

Zabránit takovémuto zarovnávání (angl.: alignment) můžeme použitím klíčového slova **packed** v deklaraci typu.

```
 type  rec_brb =  packed record
         a:byte;
         j:real;
         b:byte;
       end ;
```

Tentokrát bychom dostali velikost záznamu 10 byte.

I naskládání položek v poli může být ze stejných důvodů plýtvavé. Proto i před slovem **array** se může nacházet slovo **packed** , pokud chceme zařídit aby se šetřilo místem.

Endiáni: (J. Swift)

Cvičení: Pomocí variantního záznamu prozkoumejte proměnnou typu integer jako pole 4 bytů. Je zřejmé, že pokud do celého čísla typu integer uložíte hodnotu 1, pak ve třech bytech bude 0 a v jednom 1, o tom říkáme, že je nejméně významný. Zjistěte jestli váš počítač ukládá nejméně významný byte na nejnižší nebo naopak nejvyšší adrese. (Předpokládmáme ovšem, že pole se ukládá vždy tak, že s rostoucím index roste i adresa uložení prvku.) Použijte např. deklaraci typu

```
type tRozkladCisla =  record
                        case  of
                          CeleCislo:  Integer;
                          CtyriBajty: array [0..3] of byte;
                      end;
```

Vlastně jsme k tomu ani nepotřebovali variantní záznam, potomci TurboPascalu umožňují umístit dvě proměnné na stejné místo paměti:

```
var MojeCeleCislo :       integer;
    RozkladNaCtyriBajty : array [0..3] of byte absolute MojeCeleCislo;
```

Používáme pole

Pole s proměnným počtem prvků, dynamická pole. Řetězce. Textové soubory.

Při psaní programů se nevyhme situaci, kdy počet prvků pole bude znám až za běhu programu. To vylučuje možnost stanovit jeho rozměry při kompilaci. Jde o natolik častý a důležitý problém, že se mu budeme podrobně věnovat. Není bez zajímavosti, že po dlouhou dobu stály chyby v řešení tohoto problému za mnoha "dírami" jimiž byly napadány různé počítačové systémy a dokonce motivoval vznik počítačových jazyků odolných vůči takovým chybám.

Pole s volným koncem - zarážka

V mnoha případech neznáme v okamžiku kdy píšeme program, kolik hodnot bude v poli potřeba uskladnit. Běžným řešením je odhadnout shora maximální rozumný počet a pole dimenzovat na tuto maximální zátěž. Jde o velmi běžný postup a i některé solidní programy (TeX) vám někdy nahlásí, že jste vyčerpal kapacitu a musíte si program překompilovat s větší konstantou určující kapacitu polí pro uskladnění.

Jak ale do takovéhoto pole naskládat seznam s proměnnou délkou? Prvním řešením je použití *zarážky*. Jde o to, že za posledním uloženým prvkem nastavíme ten další na nějakou dohodnutou hodnotu, jež nás upozorní, že již nejde o data ale o oznámení konce. V kódu pak procházíme pole dokud nenarazíme na zarážku. Pokud je procházení pole součástí zamýšleného kódu, nepřidá kontrola na zarážku příliš práce a není ani příliš náročnější než si pamatovat rovnou počet údajů v poli. Technologie zarážky se běžně používá u jednoho typů polí: řetězců. Zatímco Pascal s implementací řetězců nejdříve otálel, jiný jazyk jeho éry, C, který musel od počátku sloužit pro psaní *skutečných programů*, rozhodl, že dnes je nejčastějším způsobem uložení řetězců takzvaný *zero-terminated string*. Nejčastějším proto, že dnešní OS mají kdesi na počátku právě operační systém, kvůli jehož psaní si K&R vymysleli jazyk C.

```
program szTest;

const maxDelkaRetezce = 1234;
type tRetezec = array[0..maxDelkaRetezce] of char;

function delkaRetezce( s : tRetezec ) : integer;
var i:integer;
begin
  i:=0;
  While (i<=high(s)) and (s[i]<>#0) do i:=i+1;
  delkaRetezce := i;
end;

var s1: tRetezec;

begin
  s1 := '123456789';
  Writeln( 'delka(''',s1,''')=',delkaRetezce(s1) );
  s1 := 'AbCdEf';
  Writeln( 'delka(''',s1,''')=',delkaRetezce(s1) );
end.
```

Poznámky k příkladu:

- Ve funkci `delkaRetezce` je vidět jednoduchý příklad na operaci s každým znakem nulou ukončeného řetězce – procházíme pole dokude nenarazíme na znak `#0`.
- Příkaz

  ```
  s1 := 'AbCdEf';
  ```

  ukazuje další z výjimek v kompatibilitě typů: potřeba chápat pole znaků jako řetězec je natolik běžná, že překladač neprotestuje.

- To, že při běhu najde správně délku obou řetězců znamená, že přiřazení nezapomene na konec řetězce přidat i `#0`.
- Funkce `high` vrací horní mez pole. I když víme, že to má být `maxDelkaRetezce`, je její použití pohodlnější a odolnější vůči chybám (např. změně horní meze v deklaraci `tRetezec`).

Pozor, zarážka je jen jedna. Když ji přehlédnete, mohou za ní následovat běžné znaky a na další zarážku už v poli vůbec nemusíte narazit. Pokud pak překročíte hranice pole, mohou se začít dít věci....

Cvičení: Napište vnitřnosti následujících procedur:

```
type tStrZ = array[0..1024]  of Char;

procedure SpojRetezce(const A,B: tStrZ;
                      var AaZaNimB: tStrZ);

procedure KusRetezce(const A: tStrZ;
                     OdKterehoZnaku, KolikZnaku: integer;
                     var VybranyKusA: tStrZ);
```

Pole s volným koncem - proměnná pro délku

Je zřejmé, že místo abychom na určení délky řetězce měli zvláštní funkci, která jen počítá počet znaků před zarážkou, můžeme přidat ještě proměnnou, která bude **vždy** říkat, kolik hodnot máme v poli uloženo.

Typické použití je v následujícím kódu: Když už umíme ukládat řadu hodnot do pole, můžeme si ukázat jak tato data načteme z klávesnice.

```
{$RANGECHECKS ON $}
const  Max = 1000;
var  PocetHodnot : integer = 0;    // inicializace na nulu není nutná
     Hodnoty     : array[1..Max] of real;
     x : real;

begin
  Writeln('Zadejte␣až␣',Max ,'␣kladných␣reálných␣čísel');
  Writeln('Zadávání␣ukončete␣záporným␣číslem␣nebo␣nulou.');
  ReadLn(x);
  while  x>0  do begin
     PocetHodnot :=  PocetHodnot + 1 ;
     Hodnoty[PocetHodnot] := x;
     ReadLn(x);
  end;

  Writeln( 'Načetl␣jsem␣␣', PocetHodnot, '␣čísel.␣Příště␣s␣nimi␣i␣něco␣udělám');
end.
```

Takto obvykle vypadají programy z učebnic zpracovávající vstup dat. Jde o to načíst řadu čísel, něco s ní udělat (v našem případě jsme je jen naskládali do pole), přičemž konec vstupu je indikován nějakým číslem mimo rozsah povolených hodnot. Jakkoli jsme tedy odstranili zarážku z našeho popisu dat uvnitř programu, zůstala tu stále zarážka jako konečník vstupních dat.

Poprvé je zde použita funkce *ReadLn* k něčemu lepšímu než jen k čekání na stisk klávesy Enter. Jde o vylepšenou versi procedury *Read* , která umí načíst ze vstupu (nebo souboru, to uvidíme později) data několika základních typů: Celé číslo, reálné číslo, jeden znak a řetězec znaků (uvidíme dále). *ReadLn* pak ještě přečte a zahodí všechny znaky do konce řádku včetně. Naproti tomu *Read* pokračuje tam, co skončila.

Pole s volným koncem - proměnná pro délku a pole sbalené do záznamu

Proměnné Hodnoty a PocetHodnot jsou úzce svázány a zvláště ta první bez druhé není použitelná. Mohlo by nás napadnout spojit je do jednoho záznamu a chápat je jako jednu (strukturovanou) proměnnou.

```
type tHodnoty = record
           Pocet : integer;
           Data  :  array[1..Max] of real;
end;
```

Víme, že Pascal s nově definovaným typem příliš zacházet neumí, nezbude nám, než si pro něj napsat sadu procedur a funkcí a nebo jednoduše psát `Hodnoty.Data[Hodnoty.Pocet]`. V situaci, kdy náš program musí zacházet s několika různě dlouhými seznamy dat může spojení k sobě příslušných dat a metadat do jedné struktury zvýšit pohodlí a bezpečí.

Pole s otevřeným koncem (dynamická pole)

Námi používaná verse Pascalu nám umožňuje ještě lepší řešení:

```
program DynArrTest;

var Hodnoty : array of real;
     x : real;
     kam : integer;
begin
  Writeln('Zadejte␣libovolný␣počet␣kladných␣reálných␣čísel');
  Writeln('Zadávání␣ukončete␣záporným␣číslem␣nebo␣nulou.');

  ReadLn(x);
  while x>0  do begin
     Kam:=  High(hodnoty)+1;      // Low(Hodnoty) je 0
     SetLength(Hodnoty,Kam+1);
     Hodnoty[Kam] := x;
     ReadLn(x);
  end;

  Writeln('Načetl␣jsem␣', High(hodnoty)+1, '␣čísel.␣Tady␣jsou␣jejich␣druhe␣mocniny:');

  for  kam := 0  to High(hodnoty) do
     Writeln(Hodnoty[Kam],sqr(Hodnoty[Kam]));

  readln;
end.
```

Od předešlého se program hlavně liší tím, že pole *Hodnoty* má jako spodní mez intervalu indexů nulu. Počet prvků pole můžeme měnit procedurou *SetLength(Pole,JakDlouhe).* Aktuální horní mez zjistíme pomocí funkce *High.* Dolní mez si můžeme zjistit s pomocí funkce *Low* , ale bude to vždy nula a nelze ji měnit. Funkce *Low* a *High* můžeme použít i pro běžná pole, ale tam nám vrátí konstantu danou příslušnou mezí intervalu indexu, a může to být třeba i znak, když je má příslušné pole typ indexu char.

- **První položka pole s volným koncem má index nula.**
- **Na počátku je High(Pole) = -1 , takže ani ta není k dispozici**
- **Po provedení SetLength(Pole,N) je poslední prvek pole Pole[N-1]**
- **Protože SetLength musí najít dostatečně velký kus volné paměti na souvislé uložení pole, může se při změně délky provádět kopírování starých hodnot na nové místo.**
- **Nové prvky vzniklé po SetLength mají obecně nedefinované hodnoty, výjimkou jsou pole řetězců.**
- **Původní místo zůstává nevyužito až dokud se pro něj nenajde použití při nějaké další operaci SetLength**

Následující program má být varováním, že při volání funkce SetLength mohou přestat platit odkazy na původní prvky pole.

```
program  PozorNaNe;

type  tCA =  array of  Char;
var  A,B : tCA ;


Procedure  UdelejDveVeci( var  X : tCA;  var  c:char; N : integer);
begin
   SetLength(X,N);
   X[0]:='A';
```

```
   c:='X';  // tady je ten prusvih, c uz nemusi odkazovat na spravne misto.
end;

begin
   SetLength(A,10);
   SetLength(B,10);

   UdelejDveVeci(A,A[0],10);
   Writeln(A[0]);

   UdelejDveVeci(A,A[0],100); // Nejspis staci 13
   Writeln(A[0]);             // Uz to pise 'A'

   readln;
end.
```

Pokud bychom kromě SetLength(X,N) měnili ještě velikost jiných polí, mohlo by se stát, že parametr *c* nebude odkazovat do prázdna jako nyní, ale přiřazení c:='X' naruší některé z těchto polí, které po změně velikosti skočilo na paměťovém místě původního pole A.

Pole s otevřeným koncem jako formální parametr

pokud deklarujeme formální parametr typu **array of** real tak jako v následujícím příkladu, můžeme při volání předat jako hodnotu

- libovolné pole reálných čísel
- dynamické pole reálných čísel
- pole zkonstruované přímo na místě aktuálního parametru a zapsané jako [Vyraz,Vyraz,...,Vyraz]

```
program test;

type tDP = array of real;
var    b : array [char] of real;
       c : tDP;

function Soucet( const V: array of real) :real;
var  i : integer;
     s : real;
begin
  s:=0;
  for i := low(V) to high(V) do s:=s+V[i];
  Soucet:=s;
end;

begin
  Writeln(Soucet(b));        {vypíše 0 neboť statické pole je inicializov áno na 0}
  Writeln(Soucet(c));        {vypíše 0 (dynamické pole nemá na počátku ani jeden prvek)}
  Writeln(Soucet([0,1,2])); {vypíše 3}
  Readln;
end.
```

Je třeba rozlišit jeden malý rozdíl mezi předchozími dvěma příklady. V prvním byl parametr typu tCA a šlo tak o dynamické pole. Pro něj je definována operace *SetLength* . Naproti tomu v druhém případě byl typ formálního parametru **array of** ... . Z minula víme, že formální parametr typu **array** *[interval]* **of** *NejakyTyp* , není povolen, protože by nebyl kompatibilní s žádnou proměnnou a proceduru by nebylo možno vůbec použít. Formální parametr typu **array of** je tedy moderní konstrukcí určenou k tomu aby bylo možno funkci předat pole s nějakým základním typem ale libovolnými mezemi. S dynamickými poli se shoduje v tom, že *Low* vrací vždy 0, i když definiční interval byl třeba 1..3 nebo 'a'..'z'. High je tak rovno (DélkaPole - 1).

Řetězce

Zatím jsme používali řetězce jen na komentování a oddělování výsledků v příkazech `Writeln`. Řetězce, tedy pole znaků, mohou ale sloužit k lecčemu a některé z úloh formulovaných zcela přirozeně pro řetězce jsou, nahlíženo okem informatiků, docela složité. Příklad velmi užitečné oblasti práce s řetězci jsou tzv. regulární výrazy. Jako rekreaci pak doporučuji si prohlédnout třeba

nějakou literaturu k hledání nejdelšího společného podřetězce dvou řetězců. (Podobné problémy se objevují u analýzy DNA.) Právě pro rozsáhlost problému budeme řetězce studovat jen velmi utilitárně a vyhneme se většině detailů a problémů.

Řetězcové konstanty

Již víme vše o znacích a řetězce jsou speciální pole znaků. Protože by bylo nepříjemné psát něco jako

```
var X:tSomeCharArray = ('A','h','o','j',' ','l','i','d','i','!');
```

lze v Pascalu zapisovat řetězcové konstanty jako text uzav5en7 v apsotrofech.

```
Writeln( 'Ahoj lidi!' );
Writeln( '''Apostrof uvnitr retezce  ('') se pise jako dva (''')''' );
```

Vypíše:

```
Ahoj lidi!
'Apostrof uvnitr retezce (') se pise jako dva (')'
```

Typ String

Naneštěstí přes všechnu snahu o pravidelnost v Pascalu se ukázalo, že užitečné věci jsou nepravidelné. Příkladem jsou třeba řetězce. Jde o složitý problém. Již jsme viděli, že kvůli spolupráci s OS mají speciální význam pole znaků se spodní mezí intervalu indexů 0. Z historických důvodů je jazyce Object Pascal připraveno několik variant řetězcových typů. Povíme jen něco málo o typu *string* .

```
var  s,t :  string ;
     i,j : integer;

begin
  s := 'ABCD1234';  {Přiřazení konstanty do řetězce}
  t := s;           {Přiřazení hodnoty s do t}
  Writeln(t);       {Vypíše 'ABCD1234'}

  t := copy(s,2,3);  {Přiřazení výsledku funkce do proměnné,
                      kus řetězce s počínaje druhým znamkem dlouhý tři znaky\}
  t[1] := 'B';
  Writeln(t);       {Vypíše 'BCD'}

  Delete(s,1,4);    {Vypustí 'ABCD'}
  Writeln(s);       {Vypíše '1234'}

  t := '..'+s;      {Složení řetězce z kousků}
  Writeln(t);       {Vypíše '..1234'}

  Insert('xx',t,2);
  Writeln(t);       {Vypíše '.xx.1234'}

  Writeln('Retezec '''+t+''' ma delku ',Length(t));    {délka řetězce ve znacích}
  Writeln('Podretezec ''23'' se nachazi na indexu ',
           Pos('23',t),    {nalezení polohy podřetězce jinak 0}
           ' retezce '+t);

  Setlength(t,20);
  t[20]:='6';
  Writeln(t);       {Vypíše '.xx.1234           6'
                     ale ty mezery nejsou mezery, jak uvidíme:}

  for i:=1 to length(t) do
        Write('#',ord(t[i]));
  {Vypíše '#46#120#120#46#49#50#51#52#0#0#0#0#0#0#0#0#0#0#0#54'); }

  str(Pi:20:10,s);              {Nacpi Pi do retezce}
  Writeln(#10#10#10#10+s);      {čtyři novéřádky a Pi}
  val(copy(s,11,10),i,j);       {převeď řetězec na číslo}
  Writeln(i);                   {Vypíše 1415926536}
  Readln;
end.
```

Ještě máme po ruce další spoustu procedur a funkcí např. *UpperCase*
Velmi důležitá je někdy funkce

```
procedure Val(S; var V; var Code: Integer)
```

která umí do celočíselné nebo reálné proměnné V dosadit hodnotu zapsanou v řetězci. Code je 0 pokud nenastanou problémy, jinak je to index problematického znaku v řetězci. Pozn.: Vidíte, že když má funkce vracet dvě hodnoty, je použití dvou proměnných předávaných odkazem populární.

Interně je typ string podobný poli znaků s otevřeným koncem, první znak řetězce (pokud má délku >0) se ale nachází na indexu 1. Proto **SetLength(Retezec,Delka)** znamena výjimečně, že použití **Retezec[Delka]** je OK.

Proceduru `SetLength(Retezec,Delka)` použijeme především tehdy, pokud chceme pomocí indexů pracovat s jednotlivými znaky řetězce. V rámci přiřazovacího příkazu a při volání systémových funkcí pracujících s řetězci tuto funkci volat smozřejmě nemusíme.

Ve výjimečných případech se může hodit vědět, že řetězec s danou maximální délkou se deklaruje jako

```
type  tJmenaReckychPismen = string [10];     //zadne neni delsi
```

Jde ale o přežitek z minulosti, třeba jen tím, že v hranatých závorkách smíme uvést jen číslo do 255.

Práce s textovými soubory – typ Text

Wirthův Pascal odrážel ještě nerozvinutý obor skladování dat té doby, proto se původní funkce pro práci se soubory učit nebudeme. Pro praktické použití souborového systému, který nám operační systém (OS) nabízí, můžeme samozřejmě použít přímé volání služeb OS. Pro představu tady je zjednodušená hlavička knihovnou Windows exportované funkce pro zápis dat:

```
function  WriteFile(hFile: integer; // uchopitko (cislo) souboru
                    const Buffer;   // data
                    nNumberOfBytesToWrite: integer; // kolik zapsat
                    var lpNumberOfBytesWritten: integer): boolean;
```

Nad podobnými funkcemi, které v podstatě nabízejí jen různé formy stěhování binárních dat mezi soubory a proměnnými, máme již od ranných verzí jazyka TurboPascal k dispozici také prostředky pro práci s datovými a textovými soubory na úrovni jazyka Pascal. Stále ale když pracujeme se souborem, musíme respektovat, že je to objekt spadající do kompetence OS a v jazyce máme jen vrátka, skrze která nám je dovoleno provádět se soubory užitečné operace pohodlněji, omezení plynoucí z vlastností souborů však zůstávají.

Nejdříve budeme uvažovat vstup z a výstup do textového souboru. Dnes již je jakýkoli textový soubor posloupností bytů, nikoli štítků či bloků na pásce nebo magnetickém bubnu. Textový soubor je soubor, ve kterém, co byte to znak nebo řídící znak. (Pozn. byte ve smyslu nejmenšího kvanta zapsatelného do souboru, 8 bitů, nikoli jako předdefinovaný typ ObjectPascalu pro krátké neoznaménkované číslo. Ovšemže pomíjíme Unicode, tedy textový soubor obsahující např. čísk é znaky. Pozor: Soubor vytvořený „textovým editorem“ jako je třeba MS Word, není textový soubor ve smyslu naší přednášky.) Pokud neplatí, co byte to znak, mluvíme souboru binárním (opět, opomeňme pro jednoduchost UTF8 kódování textových souborů). Příkladem může být třeba soubor nějakého obrázku: Nehomogenní skládačka z binární hlavičky souboru následované bloky, každý se svojí hlavičkou a komprimovanými daty .... Práci s takovým souborem si necháme na jindy.

V Pascalu je textový soubor (tedy klíč od oněch dveří vedoucích k datům v souboru) zastoupen proměnnou typu `Text`. Takto vytvoříme (nebo, pokud soubor již existuje, zkrátíme na nulovou délku) textový soubor s názvem 'Soubor.txt' v běžném adresáři a zapíšeme do něj krátkou zprávu:

```
var T: Text;

begin
  Assign(T,'Soubor.txt');
  Rewrite(T);
  Writeln(T,'Toto␣je␣prni␣radek␣souboru,␣druhy␣zustane␣prazdny!');
  Close(T);
end.
```

Co se souborem můžeme dělat nám velmi určuje OS, on je za něj zodpovědný. Proto i proměnné reprezentující soubory zdědí jistá omezení. Především, je zakázáno přiřazení do proměnné typu

Text . Navíc nemá pro nás přístupnou vnitřní strukturu, takže nemůžeme psát něco jako `T.Name := 'Soubor.txt'`. Zbývá tak jen třetí způsob práce s proměnnou typu `Text` , a to použití této proměnné jako parametr procedury nebo funkce. I zde jsme omezeni, nesmíme předávat soubor hodnotou. Proto všechny operace se soubory budou mít formu volání procedur, kde jako první parametr bude proměnná typu `Text` . Následujícími procedurami, které podobně jako třeba `ReadLn` umí překladač aniž se musíme doprošovat nějaké knihovny, můžeme ovládat práci s textovými soubory.

`Assign(TextVar, Retezec)` ... Přiřazení jména. Musíme zadat platné jméno na daném stroji.

`Rewrite(TextVar)` ... Vytvoř soubor nulové délky a otevři pro zápis. Pokud existuje zahoď co je v něm.

`Append(TextVar)` ... Otevři soubor pro zápis na jeho konci. Připisuje se za původní obsah. Musí existovat.

`Reset(TextVar)` ... Otevři soubor pro čtení. Musí před tím existovat.

`Close(TextVar)` ... Uzavři soubor.

Mezi voláním `Rewrite` a `Close` (případně `Append` a `Close`) můžeme psát do soubor pomocí příkazů `Write` a `Writeln` , viz výše uvedený příklad. Mezi voláním Reset a Close můžeme ze souboru číst pomocí `Read` a `ReadLn` .

Pokud se vyskytne problém dostaneme buď

Runtime error 2 at 0040432E

nebo o něco hezčí okénko s oznámením, že se nám počítač a tvůrci programu omlouvají....

Jak víme chování se dá v případě chyby ovlivňovat pomocí direktiv. Jako obvykle, máme na výběr mezi krátkou a dlouhou formou:

Samo od sebe (default) je hlídání nastveno na $I+ t.j. {$IOCHECKS ON} a dostaneme výše uvedené chování.

V případě, že je hlídání "vypneme"$I- t.j. {$IOCHECKS OFF}, pozastaví se vykonávání operací vstupu a výstupu až do doby, než se na zeptáme funkce

```
function IOResult : integer; // je k dispozici vždy, nepotřebujeme žádnou knihovnu
```

Ta nám vrátí 0, pokud nenastaly problémy. Pokud vrátí něco jiného, znamená to že nastaly problémy a podle hodnoty se můžeme dohadovat, co se stalo. Především jsou ale od okamžiku volání *IOResult* opět povoleny operace se soubory.

Pokud tedy chceme např. vědět, zda nějaký soubor existuje, můžeme výše uvedeného chování využít a psát

```
function SouborExistuje( const Jmeno) : boolean;
var T:text;
begin
   {$IOCHECKS OFF}
   Assign(T,Jmeno);
   Reset(T); // pokud není, vznikne chyba a pozastaví se další operace
   Close(T); // nesmíme zapomenout, kdyby existoval
   SouborExistuje := IOResult=0;
   {$IOCHECKS ON}
end;
```

Jak tušíme, důvody proč nám OS nedovolí ze souboru číst můžou být různé a výše uvedená funkce opravdu jen testuje jestli je operace *Reset* pro daný soubor povolena. Pokud potřebujeme detailnější informace o souboru, nezbyde než se zeptat OS přímo.

Příkazy Write a WriteLn

Obecně se vyskytují ve dvou variantách:

```
Write(PromTypuSoubor, Vyraz, ...)
```

nebo

```
Write(Polozka, ...)
```

V prvním případě je první parametr proměnná typu Text a výstup probíhá do příslušného souboru, jeho obsah je tentýž jaký v druhém případě skončí na standardním výstupu (konsoli, přesměrovaný někam...).

Jak vidíme Write a Writeln nejsou obyčejné procedury a nemají ani obyčejné parametry. Především jich mohou mít libovolný počet výrazů mnoha typů (celočíselné, reálné, řetězce, znaky,

logické). Dále za výrazem mohou následovat i dva další celočíselné výrazy oddělené od toho prvního dvojtečkami, které určují formát výstupu, tedy způsob, jímž se hodnota převede to textové podoby. Za první dvojtečkou se nachází šířka pole, do níž je třeba hodnotu vypsat. Pro reálné výrazy může za druhou dvojtečkou následovat počet desetinných míst.

Vyzkoušejte jak se chovají následující příkazy textového výstupu.

```
Writeln(Pi);
Writeln(Pi:5);
Writeln(Pi:10);
Writeln(Pi:15);
Writeln(Pi:20);
Writeln(Pi:25);
```

```
Writeln(Pi);
Writeln(Pi:30);
Writeln(Pi:30:0);
Writeln(Pi:30:4);
Writeln(Pi:30:8);
Writeln(Pi:30:12);
Writeln(Pi:30:16);
Writeln(Pi:30:20);
Writeln(Pi:30:24);
```

Podobně můžeme psát

```
Write('':i);
```

a vypsat tak i -krát mezeru.

`Writeln` je znám dobrou vůlí vypsat výstup i když se hodnota nevejde do předepsané šířky. V tom případě se vypíše v minimální nutné šířce. Bohužel to většinou příliš nepomůže:

```
for i := 995 to 1005 do Writeln(i:4,i*i:7,i*i*i:10);
```

vypíše

```
 995 990025 985074875
 996 992016 988047936
 997 994009 991026973
 998 996004 994011992
 999 998001 997002999
1000100000001000000000
1001100020011003003001
1002100400041006012008
1003100600091009027027
1004100801610120480064
1005101002510150751125
```

Pro další strojové zpracování jsou asi data stejně nepoužitelná i když se `Writeln` snažil.

Příkazy Read a ReadLn

Funkce slouží (opět ve dvou variantách) ke čtení ze standardního vstupu nebo z textového souboru. Ve druhém případě musí být první parametr typu text.

Protože procedura `Readln` vždy po načtení všech parametrů přeskočí na začátek nového řádku,

vyzkoumáme, co z následujícího vstupu skočí ve kterých proměnných dále uvedených příkladů.

```
1 2 3 4 5
10 20 30
```

Budeme uažovat dva kousky kódu.

```
Read(a,b); // a=1 b=2
Read(c);   // c=3 zbytek se nepouž ije
```

a

```
ReadLn(a,b); // a=1 b=2 zbytek řádku se přeskočí
ReadLn(c);   // c=10
```

Naproti tomu pro vstupní data ve formě

```
1
2
3
4
```

se výsledky obou variant kódu lišit nebudou.

Samozřejmě podobně je to s načítáním hodnot reálných proměnných, formát zatím popíšeme příklady

```
1
+1
-1
1.23
1E10
1E+10
1.003E-10
```

atp. dle libosti, ovšem

```
1.000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000E10
```

si ale vyloží jako 1.0 takže všeho s mírou.

Read/Readln ale umí načíst i řetězce a například následující kód načte do pole celý textový soubor

```
program ReadText;

var T : array of string;

 i,s  : integer;
begin
  SetLength(T,100);

  i:=-1;
   while not   eof   do begin
    i:=i+1;
     if  (i>High(T))  then  SetLength(T, 2*i );
    Readln(T[i]);
   end ;
  SetLength(T,i+1);

  // ted spoctu pocet znaku, abych uz s tim nactenym souborem neco udelal
  s := 0;
   for  i := 0  to  High(T)  do  s:=s+length(T[i]);

  Writeln('Soubor␣obsahuje␣',s,'␣znaku␣na␣',High(T)+1,'␣radcich');
  Readln;
end .
```

Pokud jej spustíme a jako vstup mu přesměrujeme nějaký delší textový soubor, dozvíme se, že

```
C:\Projects\prog\pokusy> test < "C:\Program Files\Borland\Delphi6\Source\Rtl\Win\Windows.pas"
 Soubor obsahuje 1113550 znaku na 30848 radcich
```

Poznámka: Volba `SetLength(T,2*i)` místo `SetLength(T,i+1)` souvisí s tím, že zvětšování pole zabere nějaký čas. Navíc by mohly nastat velké potíže s fragmentací *haldy* a program by mohl být velmi neefektivní z hlediska správy paměti. Jakkoli jsme zatím tomu tak neříkali, program vytváří *dynamické datové struktury* a my narážíme na nejvlastnější problémy dynamických datových struktur, tzv. potřebu sběru odpadků. [...Pár slov na přednášce.. Čtenáře poznámek odkazuji na pokračování přednášky.]

Pro základní vstup číselných hodnot jsou použitelné přímo procedury `Read/ReadLn`. Pro složitěji formátovaný vstupní text je nejjednodušší si vstup načíst do řetězce, v něm nalézt polohu číselného podřetězce a ten převést na číslo pomocí funkce `val` . Viz velký příklad na řetězce výše. Za zmínku stojí, že narozdíl od běhové chyby nebo mechanismu oznamování chyb přes IOResult, umí `val` vrátit přímo index znaku, který mu zabránil v převedení řetězce na číslo a ošetření případné chyby tak může být méně drsné.

Pole plná čísel

Polynomy a polynomiální interpolace. Vektory, matice. Gaussova - Jordanova eliminace

Reálná funkce v tabulce

Budeme se zbývat situací, kdy z nějakého důvodu musíme mít funkci v počítači uloženou jako tabulku hodnot. Bohužel v nejbližší budoucnosti nebude možné psát

```
var funceF: array[real] of   real; // Opravdu to takto nejde!!!
```

a tak budeme graf funkce parametrizovat *celočíselným* parametrem z nějakého intervalu. To už v Pascalu jde

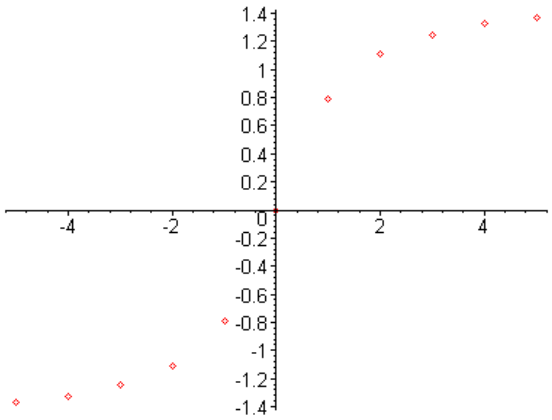
```
type  tIndexTabulky =  0 .. 12;
      tPolicko = array[tIndexTabulky] of real;
const HodnotyX : tPolicko = (-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5);
      HodnotyY : tPolicko = (-1.3734008, -1.3258177, -1.2490458, -1.1071487,
   -0.78539816,
                             0, 0.78539816, 1.1071487, 1.2490458, 1.3258177, 1.3734008)
   ;
```

Pokud bychom jedno z polí nahradili explicitní funkcí indexu, stále bychom mluvili o určení funkce pomocí tabulky.

Již víme, že pokud vytvoříme přesměrováním standardního výstupu (kam píše běžně Writeln) soubor se dvěma sloupečky čísel, můžeme program *gnuplot* požádat, aby za nás namaloval pěkné obrázky.

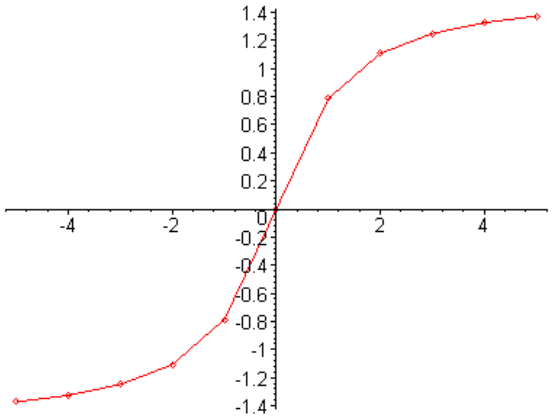
Interpolace

Ve výše uvedeném případě je rozložení hodnot nezávislé proměnné x rovnoměrné, tzv. ekvidistantní. Velmi často se s hodnotami takto rovnoměrně poskládanými setkáme u veličin měřených v závislosti na čase (např. vzorky zvuku na CD, teplota odečítaná každé poledne, ...) Data v proměnných HonotyX a HodnotyY můžeme znázornit grafem



Jak máme ale určit funkční hodnotu mezi jednotlivými body grafu?

V závislosti na charakteru dat může a nemusí dávat smysl jejich rekonstrukce z dat obsažených v tabulce. Co třeba proložit sousedními body úsečky? Pokud jde o zmiňované polední teploty, ze zkušeností víme, že to není správně. Jindy to mít smysl může.



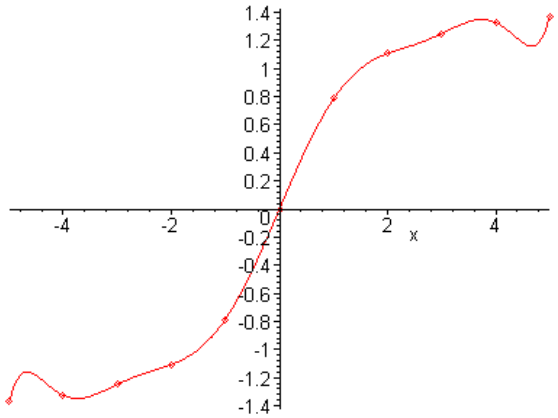
Vidíme, že výsledek není příliš dokonalý, ale přesto

Cvičení: (důležité) napište funkci, která mi pro reálné $x \in <-3, 3>$ z hodnot v proměnných `HodnotaX, HodnotaY` spočte a vrátí po částech lineárně interpolovanou hodnotu a bude mít hlavičku např.

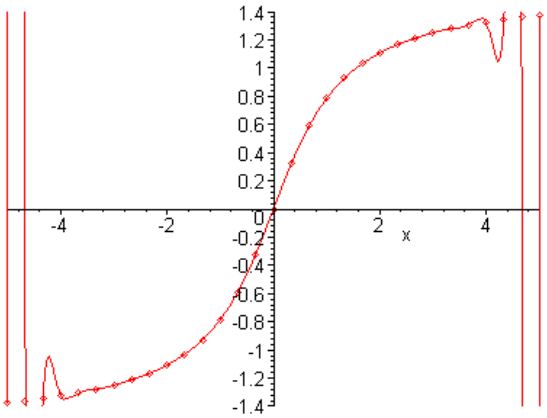
```
function FzTabulky(x : real) : real;
```

a namalujte její graf. Pokud se ke cvičení dostanete později, nezapomeňte, že v uspořádaném poli se dá hledat velmi rychle.

Pokud se nám tato hranatá funkce nezdá dost dobrá, můžeme zkusit lepší. Víme, že N+1 body můžeme proložit právě jeden polynom N-tého stupně, takže máme hned kandidát. A kolika body budeme polynom prokládat? No přece všemi, když už je máme. Tady je výsledek:



Že nevypadá stále dost dobře? No tak zkusíme někde sehnat víc bodů a to musí pomoci

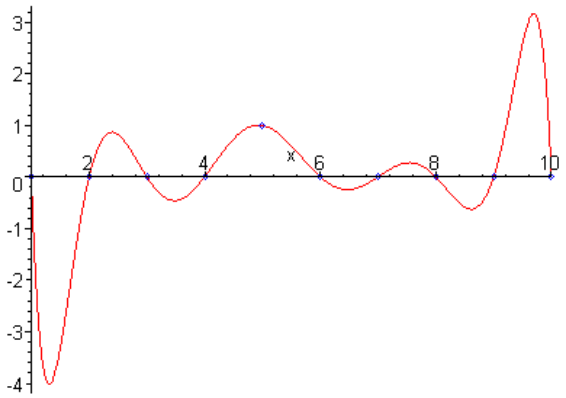


Nepomohlo...

Přes toto počáteční varování jsou polynomiální interpolace velmi užitečné, a tak si o nich povíme více.

V příkladu byla ale použit lehce zákeřná funkce *arcustangens* a navíc aspoň uprostřed intervalu vypadá výsledek hezky.

Především problém je lineární a tak stačí zjistit jak vypadá funkce proložená implusem a tu pak oškálujeme a sečteme přes všechny vzorky. Takto vypadá impuls v x=5:



Funkce proložená vyznačenými puntíky bude určitě úměrná polynomu (nepřehlédněte, chybí v něm $x-5$)

$$Y_5(x) = (x-1)(x-2)(x-3)(x-4)(x-6)(x-7)(x-8)(x-9)(x-10)$$

Pokud ji vydělíme $Y_5(5)$ bude v $x=5$ nabývat hodnotu 1 jako funkce na obrázku. Takovouto úvahou dostáváme Lagrangeův interpolační vzorec

$$f(x) = \frac{(x-x_2)(x-x_3)\ldots(x-x_n)}{(x_1-x_2)(x_1-x_3)\ldots(x_1-x_n)}y_1 + \frac{(x-x_1)(x-x_3)\ldots(x-x_n)}{(x_2-x_1)(x_2-x_3)\ldots(x_2-x_n)}y_2 + \ldots + \frac{(x-x_1)(x-x_2)\ldots(x-x_{n-1})}{(x_n-x_1)(x_n-x_2)\ldots(x_n-x_{n-1})}y_n$$

Pro obecné rozložení bodů $\{x_i\}$ budeme muset provést $n(n-1)$ násobení. Protože ale není důvod předpokládat, že n bude nabývat nějakých závratných výšek, nemluvíme zde ani tak o náročnosti výpočtu jako o faktu, že kód bude obsahovat dva cykly. Samozřejmě existují i jiná schémata pro výpočet interpolace.

Cvičení: Kromě obecné procedury pro interpolaci polynomem obecného stupně, si zkuste napsat některou z řady užitečných interpolačních funkcí jako např:

```
function  Interp2(x, x1,x2,y1,y2 : real) : real;
function  Interp3(x, x1,x2,x3,y1,y2,y3 : real) : real;
```

atp. které určitě někde využijeme.

Příklad: toto je funkce pro výpočet interpolované hodnoty přímo z Lagrangova vzorečku:

```
function LInterp(t:real;  const  X,Y : array of real):real;
var  i,j,n : integer;
 s,f : real;
begin
   n := High(X);
   assert( n = High(Y) ); { kontrola rozměrů pole }

   s := 0;
   for i := 0  to  n do begin
     f:=Y[i];
     for j := 0  to  n  do if i<>j  then  f := f*(t-X[j])/(X[i]-X[j]);
     s := s+f;
   end;
  LInterp := s;
end;
```

přičemž jde o přímý přepis vzorečku, bez jakýchkoli triků.

Polynomy jako pole koeficientů

Protože teď umíme spočíst funkční hodnotu polynomu proloženého body x_i,y_i můžeme chápat tuto dvojici vektorů také jako zápis polynomu. Obvyklé ale polynomem rozumíme spíše pole jeho koeficientů.

Následující program ukazuje jak definujeme pro takový polynom základní operace a jak je můžeme použít ke konstrukci koeficientů interpolačního polynomu. Jde o rozdílné obou přístupy a nakonec vykreslíme jejich rozdíl. Jak je vidět po spuštění programu, spočíst nejdříve koeficienty

interpolovaného polynomu a pak používat Hornerovo schéma k výpočtu hodnot interpolované funkce není vůbec přesné. Přesto program stojí za prostudování, je v něm použito mnoho z toho, co jsme dosud probrali:

- Pole pro uložení sady hodnot.
- Inicializace proměnné typu pole.
- Aserce.
- Proměnná délka pole.
- Předávání polí s různou délkou téže funkci.
- Vytváření polí jako parametrů funkcí.

```
Program Lagr2;

type tPolynom=array of real;

 //  Pozor formalni parametry funkci jsou typu array of real
 //  a nikoli tPolynom, abych mohl psat
 //  W := SoucinPolynomu(A ,[-1,0,1]) ; pro W:=(x^2-1)*A


Function HodnotaPolynomu(const P:array of real;x:real): real;
var s : real;
    i : integer;
{Spocte funkcni hodnotu polynomu pomoci Hornerova schematu}
begin
  s := P[High(P)];
  for i := High(P)-1 downto 0 do s:=s*x+P[i];
  HodnotaPolynomu := s;
end;


Function KopiePolynomu(const P:array of real): tPolynom;
var Q : tPolynom;
    i : integer;
begin
  SetLength(Q,High(P)+1);
  for i := 0 to High(P) do Q[i]:=P[i];
  KopiePolynomu := Q;
end;

Function SoucetPolynomu(const A,B:array of real): tPolynom;
var Q : tPolynom;
    i,n : integer;
    s : real;
begin
  n:=High(A);
  if n<High(B) then n:=High(B);     SetLength(Q,n+1);
  for i := 0 to n do begin
    s:=0;
    if i <=High(A) then s:=s+A[i];
    if i <=High(B) then s:=s+B[i];
    Q[i]:=s;
  end;
  SoucetPolynomu := Q;
end;

Function SoucinPolynomu(const A,B:array of real): tPolynom;
var Q : tPolynom;
    iq,ia,ib,n : integer;
    s : real;
begin
  n:=High(A) + High(B);
  SetLength(Q,n+1);
  for iq := 0 to High(Q) do Q[iq]:=0;
  for ia := 0 to High(A) do
    for ib := 0 to High(B) do
      Q[ia+ib] := Q[ia+ib] + A[ia]*B[ib];

  SoucinPolynomu := Q;
```

```
end;

function InterpolacniPolynom(const X,Y : array of real):tPolynom;
var i,j,n : integer;
    s,f   : tPolynom;
begin
  n := High(X);
  assert( n = High(Y) );

  s := KopiePolynomu([0]);
  for i := 0 to n do begin
    f:=KopiePolynomu([Y[i]]);
    for j := 0 to n do
        if i <>j then
            f := SoucinPolynomu(f,[-X[j]/(X[i]-X[j]),1/(X[i]-X[j])]);
    s := SoucetPolynomu(s,f);
  end;
  InterpolacniPolynom := s;
end;

function LInterp(t:real; const X,Y : array of real):real;
var i,j,n : integer;
    s,f   : real;
begin
  n := High(X);
  assert( n = High(Y) );

  s := 0;
  for i := 0 to n do begin
    f:=Y[i];
    for j := 0 to n do if i <>j then f := f*(t-X[j])/(X[i]-X[j]);
    s := s+f;
  end;
  LInterp := s;
end;



const N = 12;
type tIndexTabulky = 0 .. N;
    tPolicko = array[tIndexTabulky] of real;
const HodnotyX : tPolicko = (-3,-2.5,-2.0,-1.5,-1.0,-0.5, 0,0.5,1.0,1.5,2.0,2.5,3.0);
var   HodnotyY : tPolicko ;

const M = 1000;
var iX,iY,iDelta : array [0..M] of real;
    var i : integer;
    var xa,xb:real;
    IP : tPolynom;
begin

  HodnotyY[6]:=1;

  // nyni spoctu nejdriv interpolacni polynom
  IP := InterpolacniPolynom(HodnotyX,HodnotyY);

  // Nasledujici prikaz nas muze presvedcit , ze problem je ve vypoctu polynomu:

  Writeln(HodnotaPolynomu(IP,-3),'␣',LInterp(-3,HodnotyX,HodnotyY));
end.
```

Cvičení : Nakreslete grafy obou funckí i jejich rozílu. Měňte počet bodů a prokládané funkce a pozorujte co se stane.

Cvičení: V jedné z minulých přednášek jsme zavedli Legendrovy polynomy. Zkuste je spočíst pomocí funkcí uvedených v programu, tedy nalezněte koeficienty $P_n(x)$.

Cvičení: Jak je to s přesností pro polynomy vyššího řádu. Porovnejte podobně jako ve výše uvedeném programu pro interpolace.

Matice

Doposud jsme uvažovali p ředevším pole s jedním indexem. V P ascalu, jak víme, ale můžeme psát

```
type tMatice3x3 = array [1..3,1..3] of real;
```

```
var  A : tMatice3x3
...
A[3,1] := ...
```

Protože jsme si definovali nový typ, musíme pro praktické použití definovat procedury a funkce.

```
function  Det3x3( const  A:tMatice3x3) : real;
begin
Det3x3 := A[1,1]*A[2,2]*A[3,3] + A[1,2]*A[2,3]*A[3,1] + A[1,3]*A[2,1]*A[3,2]
- A[1,1]*A[2,3]*A[3,2]  {- } A[1,2]*A[2,1]*A[3,3]  {- } A[1,3]*A[2,2]*A[3,1] ;
end ;
```

Často se ale stane, že rozměry matic v programu jsou různé. Dokonce se může stát, že jeden program pracuje chvíli s maticemi 3x3, za chvíli 14x14 a za další chvíli 100x100 atd. V takovém případě bychom museli mít 3 různé funkce pro determinant, tři pro sčítání matic, tři pro násobení atd.

Naštěstí můžeme matici chápat jako pole řádků a tak použít deklaraci

```
type  tVektor = array of real;
      tMatice = array of tVektor;
```

Oproti přednášce z algebry nazýváme vektory řádky matice, takže pozor. To proto, že chceme aby matice z Algebry

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

přešly na prvky matice v následující tabulce

```
A[0,0] A[0,1] A[0,2]
A[1,0] A[1,1] A[1,2]
A[2,0] A[2,1] A[2,2]
```

a proto např. `A[1]` je typu `tVektor` a odpovídá prostřednímu řádku této matice. Pokud by se někomu chtělo, může obětovat nulté prvky polí a nulté řádky matic a mít indexy počínající jedničkou.

Pro velká N budeme matici konstruovat přímo programem, ale pro nižší hodnoty bychom mohli chtít zapsat matici takto:

```
A := [ [0  ,1  ,1-x],
       [1+x,y  ,1  ],
       [1  ,x  ,y  ] ];
```

Bohužel toto zatím Pascal neumí, ale pokud si definujeme vhodné funkce, následující zápis je v pořádku:

```
A := _m([_v([ 0 , 4 , 0 ]),
         _v([1-x, 0 ,1+x]),
         _v([ 0 , y , 8 ])] );
```

Podle nálady, může jít o přijatelnější variantu série přiřazení:

```
A[0,0]:=0;  A[0,1]:= 4;  A[0,2]:= 0;
A[1,0]:=1-x;A[1,1]:= 0;  A[1,2]:= 1+x;
A[2,0]:=0;  A[2,1]:= y;  A[2,2]:= 8;
```

ve které je ale velmi snadné poplést indexy.

Už s vektory se daly dělat věci (Cvičení: zkuste si napsat proceduru pro Gramm-Schmidtovu ortogonalizaci) a v případě matic nemáme šanci ani povrchně probrat ty nejzákladnější operace (a to ani z pohledu potřeb budoucího uživatele nějaké knihovny pro práci s maticemi).

Cvičení: Po prostudování použití matic v níže uvedeném programu pro Gauss-Jordanovu eliminaci zkuste napsat operace násobení a sčítání matic.

Řešení soustavy lineárních rovnic (Gaussova - Jordanova eliminace)

Mějme soustavu 4 rovnic pro 4 neznámé:

```
       9y + 6z + 4w = 3
  2x + 3y + 4z      = 3
   x      - 3z + 2w = 0
```

```
   x + 4y + 4z -  w = 1
```

Tato soustava se běžně representuje maticí

$$\begin{bmatrix} 0 & 9 & 6 & 4 & 3 \\ 2 & 3 & 4 & 0 & 3 \\ 1 & 0 & -3 & 2 & 0 \\ 1 & 4 & 4 & 1 & 1 \end{bmatrix}$$

Protože matici budme chtít upravit na diagonální tvar, kde nenulové prvky budou jen na diagonále, musíme nejdříve **prohodit** první dva řádky (tedy první dvě rovnice, to jistě smíme):

$$\begin{bmatrix} 2 & 3 & 4 & 0 & 3 \\ 0 & 9 & 6 & 4 & 3 \\ 1 & 0 & -3 & 2 & 0 \\ 1 & 4 & 4 & 1 & 1 \end{bmatrix}$$

Nyní první řádek (tedy rovnici) vydělíme tak aby na diagonále zbyla jednička – **normalizace**

$$\begin{bmatrix} 1 & \frac{3}{2} & 2 & 0 & \frac{3}{2} \\ 0 & 9 & 6 & 4 & 3 \\ 1 & 0 & -3 & 2 & 0 \\ 1 & 4 & 4 & 1 & 1 \end{bmatrix}$$

Nyní od druhého až čtvrtého řádku **odečteme vhodný násobek** prvního řádku tak aby v prvním sloupci mimo diagonálu zbyly nuly (lineární kombinace rovnic je OK):

$$\begin{bmatrix} 1 & \frac{3}{2} & 2 & 0 & \frac{3}{2} \\ 0 & 9 & 6 & 4 & 3 \\ 0 & \frac{-3}{2} & -5 & 2 & \frac{-3}{2} \\ 0 & \frac{5}{2} & 2 & 1 & \frac{-1}{2} \end{bmatrix}$$

Teď již stručně - normalizace 2. řádku (nic nemusíme prohazovat)

$$\begin{bmatrix} 1 & \frac{3}{2} & 2 & 0 & \frac{3}{2} \\ 0 & 1 & \frac{2}{3} & \frac{4}{9} & \frac{1}{3} \\ 0 & \frac{-3}{2} & -5 & 2 & \frac{-3}{2} \\ 0 & \frac{5}{2} & 2 & 1 & \frac{-1}{2} \end{bmatrix}$$

odečtení násobků druhého řádku

$$\begin{bmatrix} 1 & 0 & 1 & \frac{-2}{3} & 1 \\ 0 & 1 & \frac{2}{3} & \frac{4}{9} & \frac{1}{3} \\ 0 & 0 & -4 & \frac{8}{3} & -1 \\ 0 & 0 & \frac{1}{3} & \frac{-1}{9} & \frac{-4}{3} \end{bmatrix}$$

Normalizace 3.řádku

$$\begin{bmatrix} 1 & 0 & 1 & \frac{-2}{3} & 1 \\ 0 & 1 & \frac{2}{3} & \frac{4}{9} & \frac{1}{3} \\ 0 & 0 & 1 & \frac{-2}{3} & \frac{1}{4} \\ 0 & 0 & \frac{1}{3} & \frac{-1}{9} & \frac{-4}{3} \end{bmatrix}$$

Odečtení násobků 3. řádku od ostatních

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \frac{3}{4} \\ 0 & 1 & 0 & \frac{8}{9} & \frac{1}{6} \\ 0 & 0 & 1 & \frac{-2}{3} & \frac{1}{4} \\ 0 & 0 & 0 & \frac{1}{9} & \frac{-17}{12} \end{bmatrix}$$

Normalizace 4. řádku

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \frac{3}{4} \\ 0 & 1 & 0 & \frac{8}{9} & \frac{1}{6} \\ 0 & 0 & 1 & \frac{-2}{3} & \frac{1}{4} \\ 0 & 0 & 0 & 1 & \frac{-51}{4} \end{bmatrix}$$

Konečně, odečtení násobků čtvrtého řádku

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \frac{3}{4} \\ 0 & 1 & 0 & 0 & \frac{23}{2} \\ 0 & 0 & 1 & 0 & \frac{-33}{4} \\ 0 & 0 & 0 & 1 & \frac{-51}{4} \end{bmatrix}$$

Připoměňme, že tato matice representuje lineární systém

```
x = 3/4
y = 23/2
z = -33/4
w = -51/4
```

Celá metoda tak postupně opakuje tři kroky: **podmíněné prohození řádků, normalizaci řádku** a **eliminaci nediagonálních elementů** daného sloupce.

Technický detail: je zvykem **prohazovat řádky** tak, aby se na diagonále objevil v absolutní hodnotě nejvyšší použitelný (řádek>=sloupec) člen daného sloupce. To že nestačí jen kontrola na 0 tak aby šel řádek normalizovat je zřejmé, i číslo 10^{-15} by nám vadilo (vyzkoušejte). To že se vyplatí najít právě v absolutní hodnotě největší prvek je už *numerická matematika* .

Následuje program, ve kterém je také procedura pro GJ eliminaci. Současným výpočtem pro N pravých stran se spočte inverzní matice (taková aby M $M^{(-1)}$ = JednotkovaMatice).

```
Program Maticni;
uses Sysutils;
{Pouzivame assert a ten v Delphi-IDE neni videt bez SysUtils }

type tVektor = array of real;
     tMatice = array of tVektor;

function _v( const a : array of real) : tVektor;
{Udela z pole realnych cisel vektor }
var b : tVektor;
    i : integer;
begin
    SetLength(b,High(a)+1);
```

```
     for i :=0 to High(a) do b[i] := a[i];
    _v:=b;
end ;

function _m( const a : array of tVektor) : tMatice;
{udela z pole vektoru matici}
var b : tmatice;
    i : integer;
begin
    SetLength(b,High(a)+1);
     for i :=0 to High(a) do begin
      Assert(High(a[i])=High(A[0]),'_m:␣Matice␣musi␣byt␣obdelnikova!');
      b[i] := a[i];
     end ;
    _m:=b;
end ;

Procedure WriteM( const M : tMatice;W:integer=9;D:integer=5);
var r,s : integer;
begin
   for r:=0 to High(M) do begin
     for s:=0 to High(M[r]) do Write(M[r,s]:W:D);
    Writeln;
   end ;
end ;

Function IdentM(N:integer):tMatice;
var Q : tMatice;
    r,s: integer;
begin
  SetLength(Q,N,N);
  for r:=0 to N-1 do
    for s:=0 to N-1 do
      if r=s then Q[r,s]:=1 else Q[r,s]:=0;
  IdentM:=Q;
end ;

Function TranspM( const A:tMatice):tMatice;
var Q : tMatice;
    N,M,r,s: integer;
begin
  N := High(A);
  M := High(A[0]);
  SetLength(Q,M+1,N+1);
   for r:=0 to M do
     for s:=0 to N do
      Q[r,s]:=A[s,r];
  TranspM:=Q;
end ;


function GJElim( const A,B: tMatice): tMatice;
{Samozrejme resi A.x_i=b_i pro vice pravych stran}
{Cviceni: zkuste funkci pretizit a pridat dalsi pro jedinou pravou stranu}
var M,Z : tMatice;
    t : tVektor;
    r1,r,s,N,L,rmax : integer;
    u : real;

begin
    N := High(A);  // rozmery ctvercove matice A
    L := High(A)+High(B)+1;

    Assert(N = High(A[0]),'GJElim:␣Matice␣musi␣byt␣ctvercova!');
    Assert(N = High(B[0]),'GJElim:␣Vektory␣v␣B␣musi␣byt␣spravne␣dlouhe!' );

    {1. A a B spojime do jedine matice}
    SetLength(M, N+1,L+1);

   for r := 0 to N do {radek}
       for s := 0 to N do {sloupec}
        M[r,s] := A[r,s];

   for r := 0 to N do
       for s := 0 to High(B) do
```

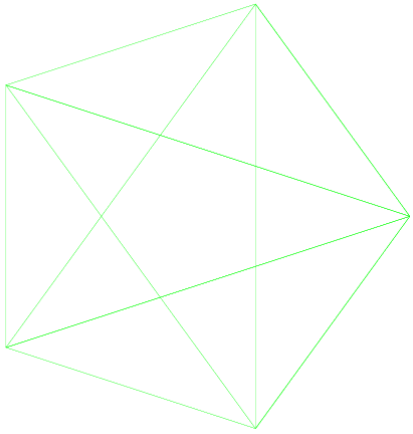
```
        M[r,s+N+1] := B[s,r];

    {2. Gauss-Jordanova eliminace}
   for  r := 0  to  N  do begin {pro vsechny radky}
      {2.1 Najdi  nejvetsi  |prvek|  v  r-tem slopecku  na  radcich  r .. N}
      rmax := r;
      s := r;      {v r-tem sloupecku, ze}
     for  r1 := r+1  to  N  do
        if  abs(M[r1,s]) > abs(M[rmax,s])  then  rmax:=r1;

     {2.2 Prehod r-ty a rmax-ty radek}
     if  r <> rmax  then  begin
        t := M[r];
        M[r] := M[rmax];
        M[rmax] := t;
     end ;
      {2.3 Normalizuj  radek  r}
      u := M[r,r];
      Assert(u <> 0,'GJElim:␣Matice␣neni␣regularni');
     for  s := 0  to  L  do  begin
        M[r,s] := M[r,s]/u;
     end ;

      {2.4 odecti  od  ostatnich  radku  r-ty*A[r,s]}
      for  r1:=0  to  N  do  if  r <> r1  then  begin
        u := M[r1,r];
        M[r1,r]:=0;
       for  s := r+1  to  L  do
           M[r1,s]:=M[r1,s]-u*M[r,s];
       end ;
      end ;

    {3. Vratit  vysledek}
    SetLength(Z,High(B)+1,N+1);
     for  r := 0  to  N  do
       for  s := 0  to  High(B)  do
        Z[s,r]:=M[r,s+N+1];

    GJElim := Z;
end ;

Function  InvM( const  M:tMatice):tMatice;
begin
   InvM:=TranspM(GJElim(M,IdentM(High(M)+1)))
end ;

var  M   : tMatice;


begin

   M := _m([_v([ 0,  4,  0]),
            _v([ 2,  0,  0]),
            _v([ 0,  8,  8])] );

   WriteM(M);
   Writeln;
   WriteM(InvM(M));

   Readln;
end .

{ A takto  si  můžeme vyzkoušet, zda  funguje  transpozice
   WriteM(_m([_v([  11,  12,  13]),
              _v([  21,  22,  23])] ) );
   Writeln;
   WriteM(TranspM( _m([_v([  11,  12,  13]),
                       _v([  21,  22,  23])] ) ));
   Writeln;
}
```

Vybrané nenumerické algoritmy

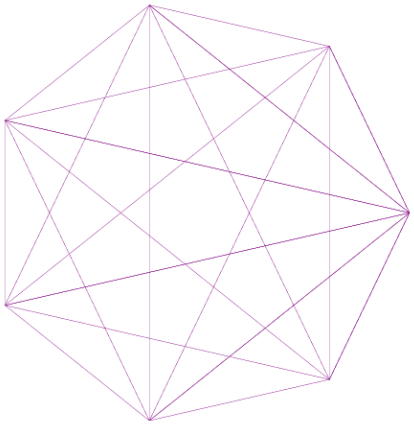
Časová a paměťová náročnost algoritmu, lineární datové struktury, třídění.

Časová a paměťová náročnost algoritmu

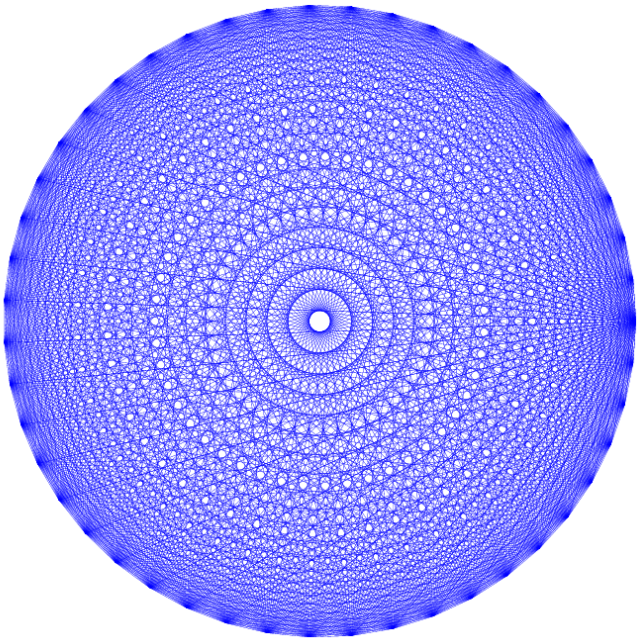
Vezměme jednoduchý obrázek

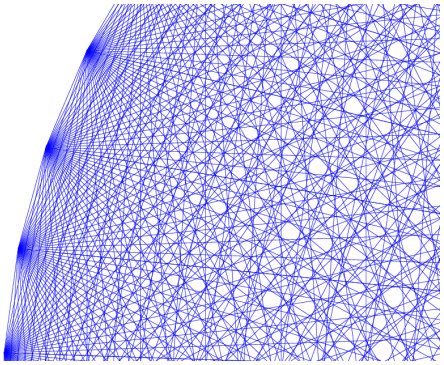


tušíme, že věci se zesložití, pokud vezmeme větší počet vrcholů



Co můžeme říci o chování pro velký počet vrcholů (47, v pravo je detail)?





Nechť počet vrcholů je N. Pak se můžeme například ptát

- kolik je potřeba čar k nakreslení takového obrázku ?
- kolik různých průsečíků je v takovém grafu ?

Podobně se můžeme ptát třeba

- jak dlouho poběží program, který bude v takovém grafu hledat nejkratší vytnutou úsečku?

Stejně tak nás může zajímat kolik paměti bude program potřebovat.

Pro každý problém zkusíme najít charakteristické N číslo udávající jeho velikost. Ne vždy je to možné ale pro představu pár příkladů:

Počet položek na skladu v programu pro skladové hospodářství.

Počet neznámých v soustavě lineárních rovnic.

Počet jednání které má vyřídit obchodní cestující.

Počet souborů, které chceme efektivně vypálit na cédéčka.

Nyní se můžeme ptát jak se bude program zabývající se výše uvedenými problémy chovat při růstu onoho charakteristického čísla N . Samozřejmě, pokud neplánujeme růst skladu, zvyšování počtu neznámých atp., nemusíme se tím zabývat. I ve fyzice se ale projevuje tendence počítat složitější a složitější problémy (třeba ty jednodušší už někdo vyřešil) a tak na problém velkého N můžeme narazit.

Při porovnávání různých algoritmů máme možnost porovnat, jak se chovají pro různé velikosti vstupních dat přímo

Někdy závisí doba řešení problému na konkrétních vstupních hodnotách, často ale, jak jsme viděli, u soustav lineárních rovnic, závisí pouze na velikosti problému.

Vzpomeneme-li si na definici algoritmu, víme že jde o posloupnost elementárních operací. Předpokládejme nejdříve, že elementárními operacemi rozumíme základní operce jako jsou sčítání, násobení, větvení, přístup k jednoduché proměnné, přístup k prvku pole volání či návrat z podprogramu atp. Změříme-li dobu, kterou potřebuje počítač k provedení každé z těchto operací a budeme-li předpokládat, že celý program se vykoná za dobu odpovídající součtu těchto jednotlivých operací, víme jak spočíst dobu potřebnou k provedení programu. Řekněme třeba, že podprogram pro skalární součin dvou vektorů

```
Function  SkalSoucin( const  A,B:tVektor):real;
var  i:integer;
 begin
     s:=0;
     for i:=0 to High(A) do
         s:=s+A[i]*B[i];
   SkalSoucin := 0;
 end;
```

provede následující operace (časy jsou z dobrých důvodů jen přibližné a konkrétní hodnoty nedůležité, mysleme si ale, že časy jsou v nanosekundách nebo, ještě lépe, v tiknutích hodin procesoru počítače).

Operace	**Čas**	**Počet**	**Celkem**
Předávání parametrů	4	1	4
Vynulování proměnné	2	1	2
Zjištení `High(A)`	30	1	30
Příprava cyklu	6	1	6
Načetní A[i]	3	N	3N
Přinásobení B[i]	6	N	6N
Přičtení s	6	N	6N
Uložení do s	4	N	4N
Zvětšení i	2	N	2N
Skok na začátek cyklu	6	N-1	6N-6
Přiřazení do výsledku	4	1	4
Návrat z funkce	10	1	10

a tedy celkový čas T = 27 N + 50.

Podobně bychom pro násobení matice vektorem dostali třeba

T = 36 N^2 + 44 N +26

Pokud nás ale zajímá chování pro větší N, je rozhodující první člen. Píšeme

T ≈ 36 N^2

Určit koeficient přesně je ale velmi obtížné a asi jedinou možností je až analýza měření závislosti času výpočtu na N. Pokud chceme mluvit o výkonu algoritmu v okamžiku jeho návrhu ještě před jeho kódováním a nákupem počítače, ukazuje se, že nejjednoduuší je prostě psát

T = O($N^{2)}$

Velké $O(f)$ je označení pro libovolnou funkci g, která splňuje vztah

$$0 < \lim_{N\to\infty} \left|\frac{g}{f}\right| < \infty$$

Následující tabulka má ilustrovat, že opravdu rozhodující je právě řád, nikoli konkrétní hodnota koeficientu u vedoucího členu.

N	N.log 2 N	N^2	N^3	2^N	N!
3	6	9	27	8	6
10	30	100	1 000	1 024	3 628 800
30	150	900	27 000	1.1×10^{9}	2.65×10^{32}
100	700	10 000	1 000 000	1.27×10^{30}	9.33×10^{157}
1 000	10 000	1 000 000	1×10^{9}	1.1×10^{300}	4.02×10^{2567}
10 000	140 000	100 000 000	1×10^{15}	2.0×10^{3010}	2.84×10^{35659}

Mimochodem, od velkého třesku uplynulo asi $4x10^{26}$ nanosekund.

Vztah *O(f) . O(g) = O(f . g)* nám pak umožňuje místo rozkladu algoritmu na elementární kroky O(1), uvažovat strukturovaně.

Třeba Gauss-Jordanova eliminace (za předpokladu, že počet pravých stran je <=O(N)) pro každý řádek (tedy *O(N)* krát) provádí

- hledání největšího prvku na/pod diagonálou *O(N)*
- normalizace *O(N)*
- odečtení násobku řádku od všech ostatních *O($N^{2)}$*
- A na začátku a na konci ještě nějaké kopírování *O($N^{2)}$*

Odsud máme

$$O(N^2) + O(N) * (O(N) + O(N) + O(N^2)) = O(N^3)$$

V tabulce si pak snadno najdeme, pro jaká N jde o zelený, oranžový či hnědý problém.

Podobně jako se s rostoucím N nějak chová čas potřebný k provedení výpočtu, nějak rostou i **paměťové nároky** . Protože nad velikostí datových struktur máme trochu lepší kontrolu, lze v praxi velmi dobře odhadnout co se vejde a co ne. V rozvahách o schůdnosti různých algoritmů ale také často vystačíme s notací velkého O.

Seznamy a jiné lineární datové struktury

Mimo jiné si zde ukážeme, že má smysl mluvit o typickém a nejhorším možném případě a jeho časové náročnosti

Netříděný seznam

Jmeno='Pavel' Teplota=36.5	Jmeno='Martin' Teplota=37.5	Jmeno='Jan' Teplota=36.9	Jmeno='Hugo' Teplota=39.5	Jmeno='Igorl' Teplota=37.2	Jmeno='Ivo' Teplota=37.1

Přístup přes index.....	O(1)
Přidání položky	O(1)
Ubrání položky	O(N)
Nalezení položky	O(N)

Seřazený seznam

Jmeno='Hugo' Teplota=39.5	Jmeno='Igorl' Teplota=37.2	Jmeno='Ivo' Teplota=37.1	Jmeno='Jan' Teplota=36.9	Jmeno='Martin' Teplota=37.5	Jmeno='Pavel' Teplota=36.5

V setříděném seznamu se dá rychleji vyhledávat. Platíme za to pomalejším vkládáním i odstraňováním položek:

Přístup přes index	O(1)
Přidání položky	O(N)
Ubrání položky	O(N)
Nalezení položky podle klíče....	O(log(N))
Nalezení položky obecně	O(N)

Komentář: Nejlepší, nejhorší a typický případ pro operaci přidání.

Cvičení : napište proceduru, která v seřazeném seznamu hledá pomocí půlení intervalu a ukažte, že časová náročnost je O(log(N)).

Poznámka: toto je důležitý návod jak hledat v tabulkách funkčních hodnot pokud nejsou hodnoty nezávislé proměnné rovnou ekvidistantní, kdy se dostáváme na O(1).

Asociativní pole

Jde o velmi zajímavou a důležitou datovou strukturu. Základní idea po uživatelské stránce je mít možnost jako index použít místo pořadí rovnou klíč, třeba řetězec.

Bohužel není možné psát přímo

```
TeplotaPacienta := Pacineti['Pavel'].Teplota
```

a) protože to Object Pascal neumí b) protože není jasné jestli tam položka s klíčem 'Pavel' vůbec je. Proto se přístup (vlastně vyhledání) podle klíče rozdělí na dva kroky

1. Nalezení indexu k položce s daným klíčem a ověření její existence
2. Vlastní přístup přes index

Trik spočívá v tom, že první operaci lze uskutečnit v čase O(1).

1. Nejprve se spočte hodnota tzv. matlací (hash) funkce, která přiřadí klíči celé číslo. Tato funkce musí mít velmi divokou závislost své hodnoty na klíči, rozhodně nestačí součet hodnot znaků řetězce nebo něco jiného jednoduchého, ale zároveň nesmí být výpočetně příliš náročná.

   ```
   hash('Pavel') → 1249765812
   ```

2. Poté se spočte, kde by podle matlací hodnoty měla v poli ležet hledaná hodnota

   ```
   1249765812 mod VelikostPole → 7
   ```

3. Na indexu 7 se buď

- nenachází nic
- je tam hledaný prvek
- je tam nějaký jiný prvek se stejnou hodnotou MatlaFce **mod** VelikostPole.

  Pak se dějí věci, např. se prohlížejí všechny položky až do první prázdné, jestli není tam, když už bylo správné místo obsazené.

Důležité je, že pokud Matlafunkce funguje správně, je nepravděpodobné, že se v jednom políčku sejdou dvě položky a bude třeba řešit kolizi, pokud máme pole dimenzováno, řekněme, třeba na dvojnásobek požadované kapacity.

Podobně se přidává prvek, hned poté, co zjistíme jestli tam náhodou už není a nejde tedy jen o přepsání. (Pozn. Hotovou máme v Delphi bohužel jen strukturu (objekt) THashedStringList)

Více o asociativních polích se do přednášky nevejde, je ale důležité vědět, že informatici tuto strukturu promysleli a v případě potřeby máme k dipozici seznam následujících parametrů:

Přístup přes index	O(1)
Přidání položky	O(1)
Ubrání položky	O(1)
Nalezení položky podle klíče....	O(1)
Nalezení položky obecně	O(N)

Pro fyzika nabízí tato struktura možnost "kešovat"výsledky volání funkce.

Pokud výpočet funkce několika diskrétních parametrů trvá dlouho, může se vyplatit schovat si již jednou vypočtené hodnoty. Na rozdíl od funkce s jedním parametrem jako je faktoriál, kde si hodnoty schováme do pole a je to, musíme v tomto případě sáhnout po složitějším řešení. Parametry dohromady tvoří klíč. Ten zamatláme a podle matlacího indexu výsledek spolu s klíčem uložíme v poli. Pokud pak potřebujeme znovu již jednou spočtený výsledek, můžeme okamžitě zjistit, zda je ještě k dispozici, nebo byla kolonka "keše"přepsána. Obvykle totiž kolize řešíme zahozením původního obsahu. To jsme u seznamu nemohli. Pokud víme, že se stejné parametry opakují 10x na každých 10 000 volání, víme, že hotovostní paměť na 1000 výsledků nás vytrhne i když výsledky v okamžiku kolize v keši zahazujeme.

Vnitřní třídění seznamu

Nejprve vysvětlení názvu kapitoly: *Třídění* = řazení. *Vnitřní* proto, že se odehrává uvnitř počítající části počítače a ne třeba na magnetické pásce.

Potřebujeme mít definovanou funkci $\leq$ dvou parametrů typu položky pole k setřídění. Část položky, která obsahuje informaci potřebnou pro porovnání se nazývá klíč. Obvykle z klíče nevyplývá přímo poloha v setříděném seznamu a má význam jen při porovnání s jiným klíčem.

Seznam považujeme za setříděný, platí-li

$$A_1 \leq A_2 \leq A_3 \leq ... \leq= A_N$$

Uvažujme tři příklady algoritmů pro třídění seznamu.

Třídění výběrem největšího prvku

Nejdříve najdeme mezi položkami 1..N tu největší a tu pak přehodíme s tou poslední. Poté mezi položkami 1..N-1 najdeme tu největší a tu dáme na N-1 místo. Atd.

Algoritmus je viditelně $O(N^2)$.

Třídění probubláváním

Spočívá v likvidaci všech míst, kde nejsou výše uvedené nerovnosti splněny. Seznam procházíme zdola nahoru a kdykoli narazíme na pár sousedních prvků seznamu, který nesplňuje požadované řazení, oba prvky prohodíme. Když na žádnou inverzi nenarazíme, je hotovo. Všeobecně je považován za příklad *špatného* algoritmu.

Quicksort

je název algoritmu (Hoare cca 1960), který většinou dokáže seřadit seznam v čase $O(N \log N)$. Využívá toho, že do přesné polohy položky mají nejvíce co mluvit ty sousední. Proto:

- Rozdělí celý seznam na dvě skupiny: tu, kde jsou položky menší než nějaká zvolená a tu druhou, rozdělení probíhá přehazováním položek, které do příslušných částí nepatří.
- Poté obě části předá sám sobě k rekurentnímu přeřazení. Pokud nám minulý krok rozdělí pole na zhruba dvě stejně velké části, dostáváme, podobně jako metody u půlení intervalu, jen logaritmický počet kroků, kterých je zapotřebí, abychom došli k seznamu délky 1, který již není třeba třídit.

Potíž spočívá v tom, že položku, podle které rozdělujeme seznam na dvě části, vezmeme aniž víme jestli je blízko "prostředku". Proto se může stát, že pro nevhodně uspořádaný vstup vezmeme vždy tu nejmenší/největší a skočíme u časové náročnosti $O(N^{2)}$.

Cvičení: Vyzkoušejte si to napsat pro jednoduché pole čísel a) pomocí rekurze b) pomocí zásobníku.

Motivace: Otevřete si stránku `http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html` http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html s prohlédněte si animace a porovnejte rychlost výše uvedených algoritmů.Ty tam najdete pod názvy Selection Sort, Bubble Sort a Quick Sort.

Kód programu testujícího quicksort by mohl vypadat takto:

```
Program  Sort;

procedure  Quicksort(var A:  array of  real; l, r: Integer);
var   i, j : Integer;
     swp, rozhod: real;
begin
  rozhod := A[(l + r) div 2];

  i := l; j := r;
   while  i < j  do begin
     while  (A[i] <  rozhod)   do  i:=i+1;
     while  (rozhod <  A[j])   do  j:=j-1;
     if  i <= j  then begin
      swp := A[i]; A[i] := A[j]; A[j] := swp;
      i:=i+1; j:=j-1;
     end ;
   end ;
   if  l < j  then  Quicksort(A, l, j);
   if  i < r  then  Quicksort(A, i, r);
end ;

var  data : array [0..220000 ] of real;

       i : integer;
begin
   for  i :=0  to  High(data)  do  data[i]:=random;
  Quicksort(data,0,High(data));
   for  i :=1  to  High(data)  do if  data[i-1] > data[i]  then  writeln('NESETRIDEN0')
   ;
  Writeln('OK');
  readln;
end.
```

Potíž se tříděním dnes nespočívá v neznalosti algoritmu, ale v tom, jak naučit knihovní funkci, která třídění za nás obstará třídit data jejichž strukturu sama nezná. O tom ale příště.

Binární soubory

Často se stává, že dohodnutý formát pro uložení toho, či onoho druhu dat má podobu řady bytů, a nikoli textu. Takovému souboru se říká binární i když přesnější by bylo ne-textový.

Jako příklad nám poslouží nekomprimovaný obrázek. Ten má povahu matice hodnot, které popisují barvu toho kterého bodu matice. Bohužel nestačí do souboru např. zapsat jen 1200 hodnot barvy, protože je ještě potřeba dodat, zda je to obrázek 30x40 nebo 40x30 nebo 20x60 atp. Proto vlastním datům předchází hlavička, kde je uvedeno vše, co je potřeba vědět pro správné zobrazení obrázku.

Na následujících stránkách je program pro malování jednoho takového druhu obrázků známých pod označením Mandelbrotův fraktál (viz dále).

Protože zápis programu v jazyce Pascal začíná „odspoda", i zde začneme popisem funkcí pro binární zápis grafického soboru ve formátu `bmp`. Nejprve kus, který ve výkladu přeskočíme.

```
program fraktal;
```

```
uses ucomplex;

// https://en.wikipedia.org/wiki/BMP_file_format
type tBMP_Header = packed record
    AcsiiB        : Char    ; // B
    AcsiiM        : Char    ; // M
    Size          : LongInt ; // 54 + DataSize
    Unused1       : Word    ;
    Unused2       : Word    ;
    DataOfs       : LongInt ; // 54
    HdrSize       : LongInt ; // 40 bytes
    Width         : LongInt ;
    Height        : LongInt ;
    Planes        : Word    ; // 1
    BitPerPixel   : Word    ; // Bits per pixel t.j. 24
    Compression   : LongInt ; // RGB = 0
    DataSize      : LongInt ;
    PPMx          : LongInt ; // pixesl per meter
    PPMy          : LongInt ;
    PalletteColors: LongInt ;
    ThoseImportant: LongInt ;
end ;

procedure mkBMPhdr(w,h : integer; var hdr:tBMP_Header;var pad4:integer);
begin
  hdr.AcsiiB:='B';
  hdr.AcsiiM:='M';
  hdr.Size:=54+w*h*3;
  hdr.Unused1:=0;
  hdr.Unused2:=0;
  hdr.DataOfs:=54;
  hdr.HdrSize:=40;
  hdr.Width:=w;
  hdr.Height:=h;
  hdr.Planes:=1;
  hdr.BitPerPixel:=24;
  hdr.Compression:=0;
  pad4 := w mod 4;  // delka dat radku se zarovnava na nasobek 4 byte
  hdr.DataSize:=(w*3+pad4)*h;
  hdr.PPMx:=72*10000 div 254;  // 72 dpi -> dpm
  hdr.PPMy:=72*10000 div 254;
  hdr.PalletteColors:=0;
  hdr.ThoseImportant:=0;
end;
```

V tomto kuse kódu se definuje typ (záznam) `tBMP_Header` který odpovídá tomu, jak jsou v hlavičce uloženy informace o velikosti obrázku a druhu uložení grafické informace. Procedura `mkBMPhdr` pak proměnnou tohoto typu inicializuje tak, aby potřebné informace obsahovala. Šlo by to provést i přes inicializovanou globální proměnnou, ale takto lze o velikosti bitmapy rozhodnout až za běhu.

Nyní se podívejme, jak se vytváří „binární“ soubor.

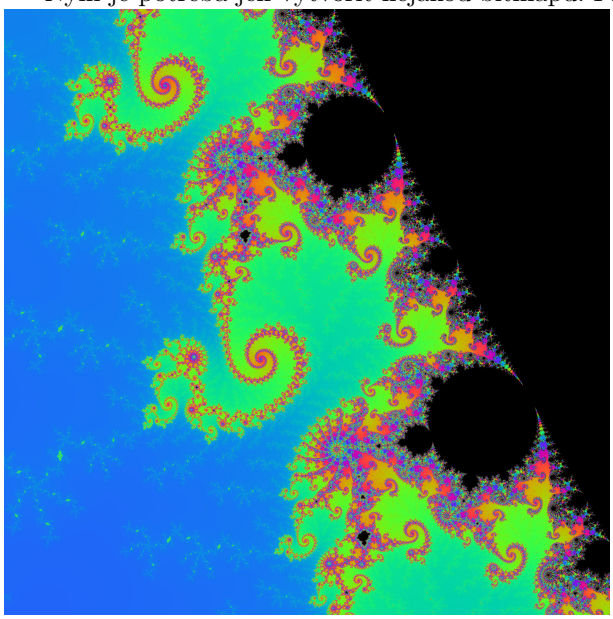
```
type tRGBval = packed record
    B,G,R:  byte;
end;

type tRGBmatrix = array of array of tRGBval;

procedure wrBitmap(const img:tRGBmatrix; const fn:string);
const nula:integer=0;
var  hdr: tBMP_Header;
     h,w,r,pad4: integer;
bmp_file: file;
begin
  h:=high(img)+1;
  Assert(h>0,'Chybi radky bitmpay');
  w:=high(img[0])+1;
  Assert(h>0,'Chybi sloupce bitmapy');

  mkBMPhdr(w,h,hdr,pad4);
  assign(bmp_file,fn);
  Rewrite(bmp_file,1);
  BlockWrite(bmp_file,hdr,sizeof(hdr) );
```

```
  for r:=0 to h-1 do begin
     Assert(high(img[r])=w-1,'Vsechny␣radky␣bitmapy␣musi␣byt␣stejne␣dlouhe');
     BlockWrite(bmp_file,img[r][0], 3*w );
     if pad4>0 then BlockWrite(bmp_file,nula,pad4);
  end;
  close(bmp_file);
end;
```

Vidíme, že místo identifikátru `text` je nyní použito slovo `file`. Operace `Assign` je stejná, ovšem `Rewrite` má nyní druhý parametr určující velikost elementárního záznamu v souboru na jeden byte. Dále se ale místo `Write` atp. používá `BlockWrite`. Jeho parametry jsou jednak soubor, kam se píše, a dále proměnná, která obsahuje data, která mají skončit v souboru a konečně jejich počet (v bytech).

V uvedeném kódu se takto zapisují data ze dvou proměnných – `hdr` po zavolání `mkBMPhdr` obsahuje hlavičku bitmapy, `img` pak obsahuje matici tvořenou trojicemi hodnot RGB. Formát bmp vyžaduje, aby každý řádek bitmapy byl zarovnán (align) na násobek čtyř bajtů (proto proměnná `pad4`).

Nyní je potřeba jen vytvořit nějakou bitmapu. Příkladem může být tato:



Souřadnice na obrázku jsou reálná a imaginární část komplexního čísla v okolí čísla $a = -0.52886 - 0.66867i$. Barva znázorňuje, zda posloupnost $0, f(0), f(f(0)), f(f(f(0))), f(f(f(f(0)))), ...$, kde $f(x) = x^2 + a$ je omezená (černá) nebo po kolika krocích přesáhne jistou velikost. Kód který počítá kolik takových kroků je potřeba je

```
function kolikrat(a:Complex; nmax:integer):integer;
var n:integer;
    z:Complex;
begin
    n:=nmax;
    z:=0;
    repeat
      z:=z*z+a;
      n:=n-1;
    until (z.Re*z.Re+z.Im*z.Im>8) or (n=0);
    kolikratSlow:=n;
end;
```

Zde je využito komplexní aritmetiky dostupné v jazyce `freepascal` začleněním modulu `uses ucomplex`. Protože tato funkce vrací počet kroků po nichž posloupnost přesáhne $|z|^2 > 8$ je potřeba toto celé číslo ještě převést na barvu. Jedna z možností je

```
const Black: tRGBval = (B:0;G:0;R:0);

function paletteF(x:real):tRGBval;
begin
   x:=2*Pi*x;
   paletteF.r := round( 127+ 17*cos(x)+126*sin(x) );
   paletteF.g := round( 127-116*cos(x)-048*sin(x) );
```

```
   paletteF.b := round( 127+100*cos(x)- 78*sin(x) );
end;

function barva(a:Complex):tRGBval;
var n:integer;
const cyklu= 4;
      nmax = 256;

begin
    n:=kolikrat(a,cyklu*nmax);
    if n=0 then barva:=Black
           else barva:=paletteF(n/nmax);
end;
```

Funkce `paletteF` přepočítává reáné číslo mezi 0 a 1 na barvu (je vidět, že jde o kružnici v třírozměrném barevném prostoru RGB). Funkce `barva` pak pro dané komplexní a spočte jak brzy posloupnost opustí kruh o poloměru $\sqrt{8}$ a vrátí podle toho barvu. Speciální hodnota `Black` je rezervována pro taková a, kde to trvá moc dlouho.

Zbytek programu pak jednak definuje seznam zajímavých míst, kde stojí se na obrázek podívat velmi zblízka

```
type tVyrez = record
   x,y,w:real;
end;

var vyrezy:array[1..7] of tVyrez = (
( X:-0.8;                Y:0;                      w:1.43 ),
( X:-1.47606062296183;   Y:-0.00354445233736481; w:1e-10),
( X:-0.528858517633617;  Y:-0.668674232801171;   w:3e-05),
( X:-1.18740781386776;   Y:-0.304146550365096;   w:9e-07),
( X:-0.0747837256554886; Y:-0.970653564199432;   w:1e-08),
( X:-1.25586912146251;   Y:-0.382814298251953;   w:5e-07),
( X:-1.74652309431234;   Y:1.06246014559333e-06; w:2e-08));

const KteryVyrez = 3;

function vyrez(fx,fy:real):Complex;
begin
  with vyrezy[KteryVyrez] do begin
     vyrez.re := x+(2*fx-1)*w;
     vyrez.im := y+(2*fy-1)*w;
  end;
end;
```

Zde se též definuje funkce přepočítávající relativní souřadnice v obrázku do komplexních čísel.

Vlastní hlavní program za použití několika proměnných projde všech $720x720$ pixelů bitmapy, nastaví jejich barvu a výsledek zapíše do souboru „abc.bmp“.

```
var   img : tRGBmatrix;
       i,j: integer;

const hImg= 720;
      wImg= 720;

begin
    SetLength(img,hImg,wImg);
    for i:=1 to hImg do
      for j:=1 to wImg do
        img[i-1,j-1]:=barva(vyrez(j/wImg,i/hImg));

    wrBitmap(img,'abc.bmp');
end.
```

Z hlediska přednášky je těžiště programu v proceduře `wrBitmap`, která používá procedury pro čtení z a zápis do binárních souborů. Jak je vidět, operace s binárními soubory se v několika ohledech odlišují od textových i otypovaných souborů.

1. U operací `Reset` a `Rewrite` musíme dodat jak velká je základní nedělitelná jednotka dat. Pokud tento údaj zapomeneme uvést, volí ObjectPascal automaticky prehistorickou jednotku o velikosti 128 byte, místo aby si stěžoval !

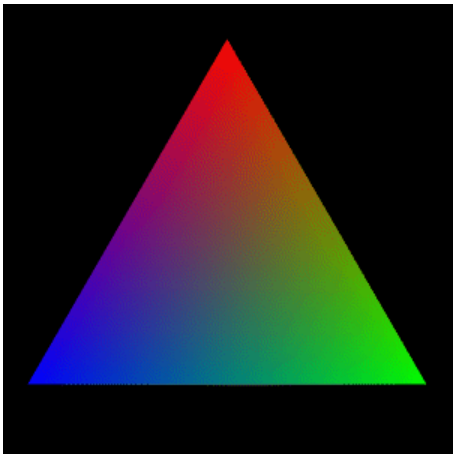
2. Operace pro čtení se teď jmenuje BlockRead

BlockRead(**var** BinarniSoubor : **file** ; **var** PromennaDoNizChcemeNacistData; KolikJednotek : integer; **var** KolikSePovedloNacist : integer)

3. Operace pro psaní se teď jmenuje BlockWrite

BlockWrite(**var** BinarniSoubor : **file** ; **var** PromennaZNizChcemeZapsatData; KolikJednotek : integer; **var** KolikSePovedloZapsat : integer)

Cvičení: Nakreslete rovnostranný trojúhelník barev pro které platí r+g+b=1 (využívá vlastností rovnostranného trojúhelníka, že součet všech třech výšek je konstantní, pro ty, co nemají rádi analytickou geometrii r := y;g := (sqrt(3)*x-y)/2;b := (2-sqrt(3)*x-y)/2, ale ješte je třeba zkontroovat, že jsou všechny kladné.)



Lineární datové struktury II.

Fronta

Tato datová struktur má jako prototyp službu či zařízení, které se stará o komunikaci pratiskárny s prapočítačem. Prapočítač dokáže vmžiku vygenerovat požadavek na vytisknutí tisíce znaků, ale pratiskárna jich dokáže vytisknout jen 50 za sekundu. Prapočítač stojí tisíc dolarů za hodinu provozu, takže se nemůže zdržovat čekáním na pratiskárnu. Proto je tu něco, co vmžiku spolyká výstupní data a pak je ve vhodném tempu předává dál.

Jde tedy o datovou strukturu, která má řešit problém, kdy jedna část programu chrlí data rychleji, než je jiná dokáže zpracovávat, často je pak potřeba při komunikaci se světem, třeba v případě, že uživatel mačká klávesy rychleji, než je stačí program zpracovávat.

Máme k dipozici tři operace: **Vložení** , **Výběr** a **Test na prázdnou frontu.**

```
[]        znázorňuje úplně prázdnou frontu
Ulož(A)
[A]       znázorňuje frontu, níž je jen jeden prvek – A
Ulož(B)
[AB]
Ulož(C)
[ABC]
Vyber →  A
[BC]
Ulož(D)
[BCD]
Vyber →  B
[CD]
Ulož(E)
[CDE]
```

atd.

Interně si fronta musí pamatovat, kde má hledat první a kam má přidat poslední prvek. Při realizaci v poli je z důvodů efektivity, tedy aby operace byly O(1), potřeba realizovat frontu tzv. kruhovým uložením. Při dosažení konce pole se další položka přidává na začátek, pokud odtud již někdo zapsaná data vyzvedl. Abychom snadno rozlišili plnou a prázdnou frontu, je v tomto případě lepší jako jednu ze stavových proměnných fronty používat počet položek ve frontě. Pokud bychom použili index začátku a index konce, nebudeme moci přímočaře využít celé pole pro frontu. Pokud má ploe délku 2^n stačí místo operace MOD jen laciný AND.

X	X	X	X
A	X	X	X
A	B	X	X
A	B	C	X
X	B	C	X
X	B	C	D
X	X	C	D
E	X	C	D

Pozn. V učebnicích programování často najdete úlohy, na procházení všech stavů nějakého diskrétního systému. Třeba mnoho variací známé úlohy na přelévání vody je snadno řešitlených, pokud si zavedeme podatelnu, kde uložíme všechny stavy, které můžeme z aktuálního dostat jedním přelitím a odkud si vždy vyzvedneme jeden stav k dalšímu přelévání. Podatelna je příkladem realizace fronty. Viz sbírky úloh z programování o tzv. *prohledávání do šířky* , kde se také dozvíte různé možnosti, jak si zapamatovat, že už jsme v daném stavu byli, což nám zaručí, že program skončí [Příklady z programování].

Jiným příkladem je nalezení nějakého souboru. Pokud by šlo jen o jméno, je to natolik obvyklá oprace, že ji dokážeme realizovat prostředky příkazové řádky:

dir /S/B — grep -i jmeno

(tedy požádáme příkazem **dir** o vypsání všech souborů v daném adresáři a jeho podadresářích a poté jeho výstup převezme program grep a ten se podívá jestli na nějakém řádku není řetězec "jmeno". Alespoň základní popis důležitého programu *grep* snad stihneme na poslední přednášce.

Takovýto postup ale nepůjde rozšířit na případ, kdy soubor, který hledáme nezle nalézt jen podle jména. Proto může nastat situace, kdy se budeme muset uchýlit k psaní programu. Ten by

pak mohl vypadat takto (hledá se soubor délky 8274, na což je asi taky zbytečné psát program, ale ...) :

```
program  Soubornik;
{Chcete-li program pouzit, vyhlednete si soubor dostatecne skryty
 hluboko v nejakem podadresari a nastavte konstantu velikost }
uses  Sysutils;

const  dirsep = '\';
      Velikost =  8274;


const  VelikostPoleProFrontu = 1024;
var  DelkaFronty : integer = 0;
    PoleProFrontu : array[0..VelikostPoleProFrontu-1] of string;
    KdeZacinaFronta :  integer = 0;

procedure  VlozDoFronty( const  A:string);
var  KamPsat : integer;
begin
   assert(DelkaFronty < VelikostPoleProFrontu);

......
......
......
end ;

function  VyzvedniZFronty : string;
begin
   assert(DelkaFronty > 0);

......
......
......
end ;

procedure  SkonciJestliJeToOn( const  Adresar: string; const  F:TSearchRec);
begin
    if  F.Size = Velikost  then begin
     Writeln('Nasel␣jsem␣ho␣v␣adresari␣',Adresar,'␣pod␣jmenem␣',F.Name);
     Readln;
     Halt;
    end ;
end;

var  F : TSearchRec;
    Adresar : string;


begin
  VlozDoFronty('C:\windows');

   while  DelkaFronty > 0  do begin
      Adresar := VyzvedniZFronty;
       if  FindFirst(Adresar+DirSep+'*',faAnyFile,F)=0  then
         repeat
           if  F.Name[1] <> '.'  then begin  // . a .. nechci
              if  (F.Attr and faDirectory) <> 0  then  VlozDoFronty(Adresar+dirsep+F.
   Name)
                                              else  SkonciJestliJeToOn(Adresar,F);
           end ;
         until  FindNext(F) <>  0;
      FindClose(F);
   end ;

  Writeln('Nenasel␣jsem␣ho');
  readln;
end.
```

Rozmyslíme-li si funkci programu vidíme, že adresářový strom porhledáváme do šířky.

Cvičení: Dopište vytečkované vnitřnosti procedur pro frontu řetězců.

Zásobník

Jedničkou mezi lineárními datovými strukturami je zásobník. Od prvopočátku počítačové doby se totiž používá pro řešení otázky kam se má vrátit běh programu po ukončení podprogramu a v

posledních desetiletích se také stará o postor a život lokálních proměnných procedur.

Poznámka : (Rozpor idejí a reality): V matematice se zavedla dvě značení pro operace s dvěma operandy:

1. Plus(A,B)
2. A Plus B

Protože ale operaci nemůžeme obecně uskutečnit dokud nemáme k dispozici oba operandy, měl by být preferovaným zápisem tento:

A B Plus

Např. 3+4*2 bychom přece měli psát jako

```
4 2 * 3 +
```

Podobně sice v Pascalu píšeme Secti(x,Soucin(y,z)), ale je zřejmé že vše probíhá jak je psáno výše:

Ilustrace (na přednášce): Zásobník volání a vůbec jak v principu probíhá předávání parametrů, volání funkcí a alokace lokálních proměnných (bez konstrukce ebp-leseni /tzv. stack frame/)

Ilustrace na přednášce: Vyhodnocení o něco složitějšího reálného aritmetického výrazu na zásobníkovém stroji (bez FPU registrů) ln(-x+sqrt(x**2++1))

Cvičení: *Zásobník a prohledávání do hloubky* - Změňte program pro hledání souboru tak aby prováděl hledání do hloubky. Napište dvě verze: jednu s explicitním zásobníkem, druhou jen s použitíme zásobníku volání tj. pomocí rekurse. Ze způsobu prohledávání adresářů je zřejmý název *prohledávání do hloubky.*

U datové struktury zásobník máme opět k dipozici tři operace: **Vložení** , **Výběr** a **Test na prázdný zásobník.**

```
[]
Vlož(A)
[A]
Vlož(B)
[AB]
Vlož(C)
[ABC]
Vyber →  C
[AB]
Vlož(D)
[ABD]
Vyber →  D
[AB]
Vlož(E)
[ABE]
```

Při realizaci v poli nám stačí jediná stavová proměnná a snad to ani nejde napsat jinak než O(1).

Ilustace (na přednášce): prohledávání do hloubky, viz [Příklady z programování]

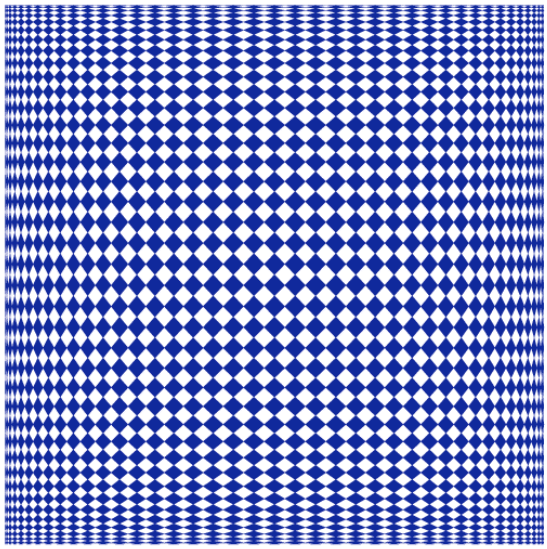
Pozn. Zásobník v dynamickém poli a jak často alokovat a proč ne pokaždé SetLength.

Cvičení: Změňte minulý program aby porhledával adresářový strom do hloubky.

Malujeme obrázek (v Postscriptu)

Letos jsme tohle nestihli, ale jako *kulturní* zajímavost ponechávám ve studijním textu.

Pro malování grafů funkcí používáme již několik týdnů program gnuplot. Vstupem pro tento program jsou tabulky čísel a výstupem hotový obrázek s osami, stupnicemi, barevnými křivkami atd.. Co když budeme chtít namalovat něco jiného než graf funkce, třeba následující obrázek:



Tušíme, že bychom mohli zkusit najít vhodnou knihovnu (modul) a pak psát program který nám na obrazovku kreslí pomocí volání vhodných procedur, třeba

```
program  Malovaci;
uses  KnihovnaProMalovani;
```

```
begin
  NamalujCaru(...);
  NamalujTrojuhelnik(...);
  ...
end.
```

Opět bychom ale museli řešit otázku, jak schovat jednou namalovaný graf, jak jej začelnit do publikace či jen tak vytisknout na papír.

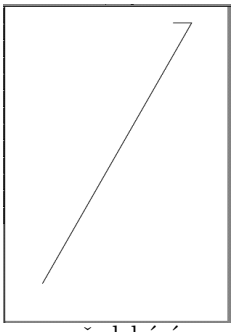
Nejjednodušší by mohlo být poznamenat si, jaké procedury, s jakými parametry a v jakém pořadí, voláme, a v případě potřeby, pak tato volání zopakovat. Nemůžeme se ale spolehnout, že příště bude naše výstupní medium mít stejné rozlišení. To pro obrazovku činí zhruba jeden megapixel, zatímco při tisku na papír A4 jde o desítky megapixelů. Náš záznam o malování by měl být vůči takovým změnám imunní. Právě proto místo povelů "obarvi puntík [241,1104] na modro"si musíme poznamenat něco jako na souřadnice 241,1104 namaluj kolečko o průměru 1. Takový příkaz lze splnit na obrazovce i na papíře, v jendom případě vystačíme s jedním pixlíkem, v druhém půje o stovky kapiček inkoustu. Terminologie: mluvíme v tomoto případě o vektorové grafice. Její základní vlastností je, že na kvalitnějším zařízení obdržíme lepší výsledky, jemnější čáry, kulatější křivky. Oproti tomu grafika založená na matici bodů nám na lepším zařízení dá jen lépe propracované čtverečky, pokud nezvětšíme rozměry matice, ale pak půjde již o jiná data. U vektorové grafiky půje dpokaždé o tentýž příkaz NamalujKřivku.

Jaký formát vektorové grafiky přesně zvolit za nás už vyřešili a to tentokrát v komerční sféře. A protože to udělali velmi dobře, vznikl standard grafického jazyka Postscript (R). Tak, jako jsme si zvykli, že program je nejlépe zapsat v textové podobě v nějakém jazyce a před jeho provedením na nějakém počítači si jej necháme přeložit, i u obrázku budeme mít jednu universální textovou podobu (postscriptový soubor). Pokud budeme chtít získat výsledek na papíře, můžeme se spolehnout, že každá lepší tiskárna soubor správně pochopí, pokud si jej budme chtít prohlédnout na obrazovce počítače, na každé myslitelné platformě (prozatím snad s výjimkou herních konzolí a mobilních telefonů) najdeme zdarma dostupný prohlížeč obvykle s názvem odvozemým ze slova GhostScript.

Čáry, plošky, barvy

Především, postscriptový obrázek je program. Na rozíl od Pascalu tento jazyk nerozlišuje deklarační výkonné části (protože, jak z časových důvodů neuvidíme, deklarace je v něm také příkazem), a tak ve své nejjednoušší podobě (jaká nám bude muset stačit) máme jen sérii volání procedur:

```
%priklad 1
newpath
 100 100 moveto
 500 800 lineto
 450 800 lineto
stroke
showpage
```

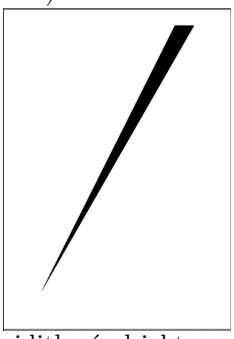


Zde vidíme základní principy. Především identifikátor procedury nepředchází parametry, jak je tomu v Pascalu. Naopak, všechna čísla, která napíšeme, uloží vykladač jazyka Poscscript na zásobník a odtud si je procedura (řekněmě *lineto*) vyzvedne.

Vzhledem ke způsobu předávání parametrů na zásobníku jsou totožné následující kusy kódu:
A: "100 100 moveto 500 800 lineto"
B: "500 800 100 100 moveto lineto"

Z hlediska grafických operací vidíme, že za účelem co největší universálnosti je proces malování trochu neobvyklý. Zkonstruujeme cestu (path) a to tak, že někde začne (moveto) pak pokračuje čarami (lineto) a, když jsme s cestou hotovi, cestu obtáhneme (stroke) nebo

```
%priklad 2
 newpath
  100 100 moveto
  500 800 lineto
  450 800 lineto
 fill
 showpage
```



vybarvíme (fill). Cesta je před obtažením nebo vybarvením neviditlený objekt.

Zde vidíme, že z nějakých důvodů zvolili autoři jednotky tak, že na palec máme 72 bodů (formát A4 na výšku tak poskytuje k malování plochu [0-595]x[0-842] "bodů"). Slovo bod je zde názvem jednotky, dvaasedmdesátiny palce, nikoli jednotkou rozlišení. U běžných tiskových zařízení se totiž fyzické rozlišení pohybuje v desítkách pixelů na bod a proto samozřejmě můžeme používat přesnější polohu zadáním desetinných míst u čísel.

```
100.001 200.002 moveto
```

Druhou možností bude (viz dále) změnit měřítko a vystačit si s celočíselnými souřadnicemi, řekněme, v mikrometrech.

Barvy

Na počátku se předpokládá, že budeme malovat ve stupních šedi. To znamená, že u každého malovaného objektu můžeme nastavovat jeho jas a to voláním procedury *setcolor* s jedním reálným parametrem (0 = černá, 1 = bílá).

```
0.45 setcolor
```

Pokud chceme vytvářet barevný obrázek, musíme to oznámit zavoláním procedury *setcolorspace* se speciálním parametrem

```
/DeviceRGB setcolorspace
```

a od té chvíle nastavujeme barvu zavoláním procedury **setcolor** se třemi parametry

```
%priklad 3
 /DeviceRGB setcolorspace

 40 setlinewidth
 1 0.5 0 setcolor
 newpath
 400 100 moveto
 100 400 lineto
 100 100 lineto
 400 100 lineto
 stroke

 0 0.5 0 setcolor
 newpath
 500 100 moveto
 100 500 lineto
 500 500 lineto
 500 100 lineto
 closepath
 stroke

 showpage
```

Všiměte si na obrázku, že *uzavření cesty* procedurou *closepath* změní způsob, jímž jsou úsečky pospojovány. Proto je rozdíl, jestli namalujeme tři úsečky nebo lomenou cestu za tří úseček. Kromě toho, pokud jsou různé, *closepath* spojí konec a počátek aktuální cesty úsečkou.

Grafický stav

Na výše uvedených příkladech vidíme, že

- čáry mohou být různé
- systém interpretující postscriptový obrázek si udržuje informaci o *grafickém stavu*
- grafický stav měníme voláním specializovaných procedur
- příkazy, např. *stroke,* pak malují podle toho, jaký je zrovna grafický stav (*stroke* namaluje jednou spodní oranžový, podruhé horní zelený trojúhelník)

Mimo jiné je součástí grafického stavu

- okažitá poloha (nastaví *moveto* , mění mj. *lineto*)
- cesta (*newpath* , *moveto* , *lineto, curveto* , *closepath*)
- barva (mění procedura *setcolor*)
- měřítko, poloha počátku a natočení os x a y (*scale* , *translate a rotate* , viz dále)
- font (viz *Poznámky pro život a ne zkoušku* dále)
- způsob čárkování čáry (*setdash,* viz cvičení pro zvědavé.)

Princip grafického stavu má svůj původ již u souřadnicových zapisovačů, kde se barva měnila výměnou malovacího pera, a běžná poloha byla opravdu polohou pera nad papírem. Pro nás znamená existence grafického stavu, že vystačíme s jedinou procedurou *stroke* pro malování čáry, ať už je barevná, čárkovaná, rovná nebo křivá.

Cvičení: Měňte barvy

Cvičení: Vynechte přepnutí do barevného módu a změňte počet parametrů u všech volání procedury *setcolor* tak aby měly jeden parametr (0.0 = černá,1.0 = bílá).

Cvičení pro zvídavé: Přidejte do příkladů 1 a 3 před newpath příkaz

[30 30 160 30] 0 setdash

a pozorujte co se stane (první parametr je typu pole, proto ty hranaté závorky, druhý parametr je reálné číslo).

Příkazem *setcolor* se mění nenávratně barva jíž malujeme, podobně *newpath* nenávratně ničí dosavadní cestu. Protože barva i aktuální cesta jsou součástí grafického stavu, můžeme si je schovat pomocí procedury *gsave* , která na vnitřní zásobník (jiný, než ten pro předávání paramtrů procedurám) uloží kompletní grafický stav. Nyní můžeme dle potřeby měnit barvu, měřítka, aktuální polohu atd.a až skončíme, obnovíme původní grafický stav zavoláním procedury *grestore* .

Transformace souřadnic

Prozatím jsme respektovali, že počátek souřadnic se nachází vlevo dole, a že jednotkou jsou "body". Následujícími příkazy změníme nejprve měřítko z bodů na milimetry a poté si posuneme počátek do prostřed strany A4 (72 bodu na palec / 25.4 milimetrů na palec= 2.8346...bodu na milimetr):

2.8346 2.8346 scale

105 148.5 translate

Obecně příkaz

x y translate

posune počátek na uvedenou polohu [x,y], podobně

uhel rotate

pootočí osy o daný úhel, a

meritko_x meritko_y scale

změní měřítka, ne nezbytně na obou osách stejně.

Následující příklad je ilustrací výše uvedených operací, tedy schovánání grafického stavu, rotací, posunů a škálování.

```
%priklad 4
 2.8346 2.8346 scale
 105 148.5 translate

 newpath
 0 0 moveto
 gsave
   20 0 lineto
   stroke
 grestore
 gsave
   30 rotate
   2 2 scale
   20 0 lineto
   stroke
 grestore
 gsave
   60 rotate
   3 3 scale
   20 0 lineto
   stroke
 grestore
 gsave
   90 rotate
   3 3 scale
   20 0 lineto
   stroke
 grestore
 showpage
```

Cvičení: Napište program v Pascalu, který namalujte (tedy vytvoří postscripový soubor, který se vykreslí jako) ony kompletní n-úhelníky z minulé přednášky.

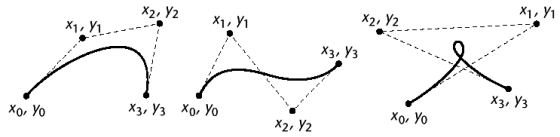
Poznámka: Operacemi moveto lineto lineto moveto linet

Křivky

V praxi bychom neměli používat cestu složenou ze stovek kousků. Pokud jde o malování plných čar vystačíme s jejím rozložením na více kratších cest (až na artefakty při napojování, které jsou vidět u tlustých čar, viz Příklad 3). Pokud je cesta obzvlášť křivá a na její konstrukci bychom potřebovali příliš mnoho úseček tak, aby nebyla viditelně polámaná, máme k dispozici proceduru

na malování křivky **curveto** . Ta maluje parametrickou křivku [x(t),y(t)], kde funkce x(t) a y(t) jsou jsou kubické polynomy. (Mimochodem, úsečka je také takovou parametrickou křivkou, ale polynomy jsou jen prvního stupně.)

$$x(t) = x\,0 + 3\,(\,x\,1 - x\,0\,)\,t + 3(\,x\,2 - 2\,x\,1 + x\,0\,)\,t^2 + (\,x\,3 - x\,0 + 3\,x\,1 - 3\,x\,2\,)\,t^3 \quad y(t) = y\,0 + 3\,(\,y\,1 - y$$



Třetí stupeň byl zvolen proto, aby si člověk mohl zvolit nejen počáteční a koncový bod křivky (na to stačí úsečka - první stupeň), ale také tečny v obou koncových bodech (viz obrázek). To že tečný vektor[dx(t)/dt,dy(t)/dt] v bodě [x0,y0] (tedy v t=0) míři do bodu [x1,y1] se z derivace výše uvedených polynomů v t=0 pozná snadno. O něco méně je vidět, že v hodnotě parametru t=1, je [x(t),y(t)]=[x3,y3], a ještě skrytější je fakt, že tečna míří z [x3,y3] do bodu [x2,y2].

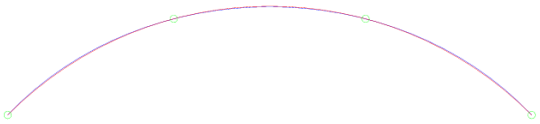
Pro nás nejjednodušší použití **curveto** je prosté proložení křivky čtyřmi body, A,B,C a D. Budeme požadovat aby v hodnotě parametru t=0 křivka procházela bodem A, v hodnotě t=1/3 bodem B, v hodnotě t=2/3 bodem C, v hodnotě t=1 bodem D. Tak dostaneme soustavu čtyř lineárních rovnic

x(0/3) = A x
x(1/3) = B x
x(2/3) = C x
x(3/3) = D x

pro čtyři neznámé x0,x1,x2,x3 (nacházející se ve funkci x(t)) a další, nezávislou soustavu čtyř rovnic pro čtyři neznámé y0,y1,y2,y3. Její řešení je

$$x\,0 = A\,x \quad x\,1 = (18*B\,x - 5*A\,x + 2*D\,x - 9*C\,x\,)/6\,x\,2 = (2*A\,x - 5*D\,x + 18*C\,x - 9*B\,x\,)/6\,x\,3 = D\,x$$

Následující obrázek ilustruje, jak dobrou aproximací skutečné křivky mohou tyto kubické křivky být.



Červená křivka je čtvtkružnice. Modrá je výsledek *curveto* s parametry podle výše uvedených vzorečků. Zelené kroužky vyznačují polohu boudů A.B,C a D a odpovídají středovému úhlu kruhového oblouku 0,30,60 a 90 stupňů.

Příklad

Když už mluíme o parametrických křivkách, nelze vynechat zmínku o těch nejznámějších, Lissajousových obrazcích. Zde je program, který je za nás namaluje

```
program  LissaPS;
{
  Maluje  Lissajousovy  obrazce
  Umí je  malovat  čarou  nebo  vyplnit
   Postscriptov ý  obrázek  vypíše  na  standardn í  výstup
}
const  Vybarvit = true;

var  AktualniPoloha : record
           x,y : real;
            OK : boolean;
    end;

procedure  KusKrivky(Ax,Ay, Bx,By, Cx,Cy, Dx,Dy : real);
{Používá výše uvedené vzorečky a proloží body ABCD kubický oblouk}
var  x1,y1,x2,y2 : real;
begin
            { nejdriv  spocist  polohu  ridicich  bodu}
       x1 := (18*Bx-5*Ax+2*Dx-9*Cx)/6.0;
       x2 := (2*Ax-5*Dx+18*Cx-9*Bx)/6.0;
       y1 := (18*By-5*Ay+2*Dy-9*Cy)/6.0;
       y2 := (2*Ay-5*Dy+18*Cy-9*By)/6.0;
            { na pocatku cesty  musi  byt  newpath & moveto}
```

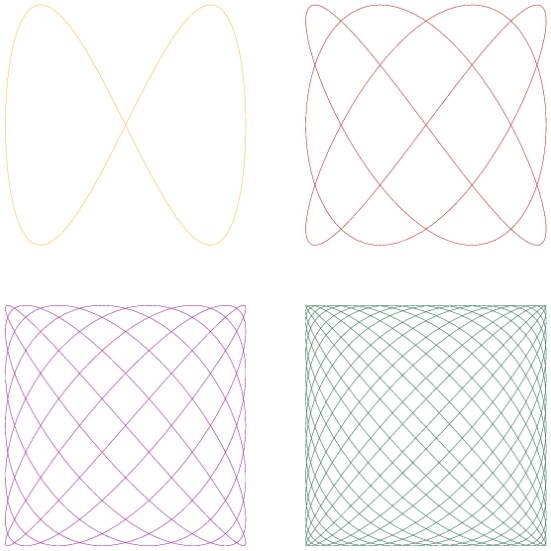
```
       if  ( not  AktualniPoloha.OK)  then  Writeln('␣newpath␣');
       if  ( not  AktualniPoloha.OK)  or  (AktualniPoloha.x<>Ax)  or  (AktualniPoloha.y
   <>Ay)
                                    then  Writeln(Ax:4:2,'␣',Ay:4:2,'␣␣moveto␣');
            { pokazde pak curveto }
       Writeln(x1:4:2,'␣',y1:4:2,'␣',x2:4:2,'␣',y2:4:2,'␣',Dx:4:2,'␣',Dy:4:2,'␣','␣␣
   curveto␣');
       AktualniPoloha.x := Dx;
       AktualniPoloha.y := Dy;
       AktualniPoloha.OK:= true;
end ;

procedure  Obtahni;
begin
  Writeln('␣␣stroke␣');
  AktualniPoloha.OK:= false;
end ;

procedure  Vybarvi;
begin
  Writeln('␣␣fill␣');
  AktualniPoloha.OK:= false;
end ;

procedure  Lissajous(sx, sy, Polomer, fazex,fazey : real; kx,ky:integer);
{[sx,sy] je stred, faze* jsou faze obou harm. oscilatací ve stupních}
var  i : integer;
    N : integer;
    Ax,Ay,Bx,By,Cx,Cy,Dx,Dy : real;

    function  x(m:integer) :real;  begin  x:=sx+Polomer*sin(fazex+2*Pi/N*kx*(i+m/3.0));
     end ;
    function  y(m:integer) :real;  begin  y:=sy+Polomer*sin(fazey+2*Pi/N*ky*(i+m/3.0));
     end ;

begin
    N := kx;  if  N<ky  then  N:=ky;
    N := N*8; {videli jsme, ze 4 staci na kruznici }

    fazex := fazex*Pi/180.0/kx;
    fazey := fazey*Pi/180.0/ky; {uz jsou v radianech}
     for  i := 0  to  N-1  do begin
       KusKrivky(
             x(0),y(0),   {pocatecni bod krivky}
             x(1), y(1), {t = 1/3}
             x(2), y(2), {t = 2/3}
             x(3), y(3)   {koncovy bod oblouku krivky}
                );
     end ;
     if  Vybarvit  then  Vybarvi  else  Obtahni;
    Writeln;
end ;

Procedure  SetColor(r,g,b:byte);  {R G B v rozsahu 0..255}
begin
  Writeln(r/255.0:6:3,g/255.0:6:3,b/255.0:6:3,'␣␣setcolor␣');
end ;

const  R = 50;  {polomer v milimetrech}

begin
    Writeln('␣␣/DeviceRGB␣setcolorspace␣');

    Writeln('␣␣2.8346␣2.8346␣scale␣');  {milimetry}
    Writeln('␣␣105␣148.5␣translate␣');  {do stredu A4}
    Writeln('␣␣0.1␣setlinewidth␣');     {tenke cary}

    SetColor(255,174,17);
    Lissajous(-R,R, 40, 0,0, 1,2);

    SetColor(155,0,0);
    Lissajous(R,R, 40, 0,0, 3,4);

    SetColor(130,0,130);
    Lissajous(-R,-R, 40, 0,0, 7,8);
```

```
    SetColor(11,80,50);
    Lissajous(R,-R, 40, 10,0, 15,16);

    Writeln('␣␣showpage␣');
end.
```

Výstup tohoto programu je potřeba přesměrovat do souboru, a ten si poté můžeme prohlédnout, vytisknout či poslat emailem.

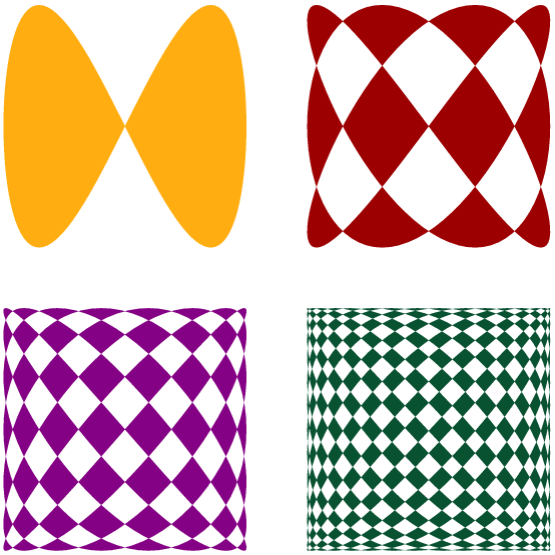
```
C:\Adresar> LissaPS > obr.ps
C:\Adresar> start obr.ps
C:\Adresar>
```

Příkaz **start** nám spustí ten program, který má v počítači na starosti postscriptové obrázkya pak uvidíme:



.

a nebo



Cvičení: Upravte program tak, aby místo křivek používal lomenou čáru. Použít můžete následující proceduru

```
procedure  Lomenice(Ax,Ay, Bx,By, Cx,Cy, Dx,Dy : real);
begin
```

```
            { na pocatku cesty musi byt newpath & moveto }
          if ( not AktualniPoloha.OK) then Writeln('newpath');
          if ( not AktualniPoloha.OK) or (AktualniPoloha.x<>Ax) or (AktualniPoloha.
    y <> Ay)
                                      then Writeln(Ax:4:2,'␣',Ay:4:2,'␣moveto');
            { pokazde pak 3x lineto }
         Writeln(Bx:4:2,'␣',By:4:2,'␣lineto␣',Cx:4:2,'␣',Cy:4:2,'␣lineto␣',Dx:4:2,'␣',Dy
    :4:2,'␣','␣lineto');
         AktualniPoloha.x := Dx;
         AktualniPoloha.y := Dy;
         AktualniPoloha.OK:= true;
end ;
```

Výsledek by pak měl vypadat takto:

Poznámky pro život a ne zkoušku:

Jazyk Postscript vziknul jako jazyk pro ovládání počítačových tiskáren a proto je největším odborníkem na písmenka. Protože může být někdy užitečné doplnit do obrázku pár písmenek, je v následujícím příkladě shrnuto několik nejpotřebnějších procedur pro práci s textem.

```
%priklad 5

(Helvetica) 40 selectfont

300 420 moveto

(ABC) show
(123) show

showpage
```

ABC123

Vidíme, že

- Řetězce jsou, jak vidíme uzavřeny v závorkách
- procedura *show* má jediný parametr a to řetězec
- *show* píše se na aktuální polohu a tu pak změní o šířku vypsanécho textu
- Druh a velikost písma se nastavuje procedurou *selectfont*

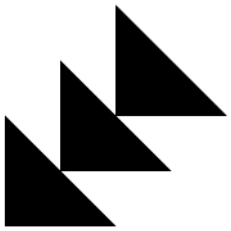
Procedura selectfont má za první parametr název písma, druhý je jeho velikost.
Vždy k dispozici by (mimo jiné) měla být následující písma:

(Helvetica)	ABC pqr 123
(Helvetica-Oblique)	*ABC pqr 123*
(Times-Roman)	ABC pqr 123
(Times-Italic)	*ABC pqr 123*
(Times-Bold)	**ABC pqr 123**
(Courier)	ABC pqr 123
(Σψμβολ)	ABX πθρ 123

Vypadá to, že bychom už měli o jazyce postscript vědět to nejpodstatnější. Třeba bychom si mohli myslet, že když si budme prohlížet postscriptový soubor narazíme na série příkazů 213 321 *moveto 545 545 lineto 45 544 moveto 577 889 lineto 54 889 lineto ...* .

Když ale nějaký otevřeme v textovém editou, zjistíme, že na počátku nejsou skoro žádná čísla, takže se tam dost dobře nemohou malovat příslušné křivky, na konci zase narazíme na spousti čísel a skoro žádná písmena. Vysvětlení je jednoduché. Postscritpt je programovací jazyk. Proto na začátku narazíme na oblast definic procedur a funkcí, tzv. prolog, na konci se pak tyto funkce používají. Proto program který má v úmyslu malovat spoustu trojúhelníků nejprve definuje proceduru pro malování trojúhelníků, a pak už používá ji místo neustálého *newpath, moveto ...*

```
 %priklad 6
/t
 newpath
 moveto
 lineto
 lineto
 fill
 def

100 100 300 100 100 300 t
200 200 400 200 200 400 t
300 300 500 300 300 500 t

showpage
```



Programu nejlépe porozumíme, přeložíme-li si začátek podle tabulky

/t → **procedure** t; **begin**
def → **end;**

a uvědomíme-li si, že procedury *moveto* , *lineto* a ještě jednou *lineto* vyzvednou ze zásobníku každá dva parametry. Pak je zřejmé, že procedura *t* jich potřebuje šest.

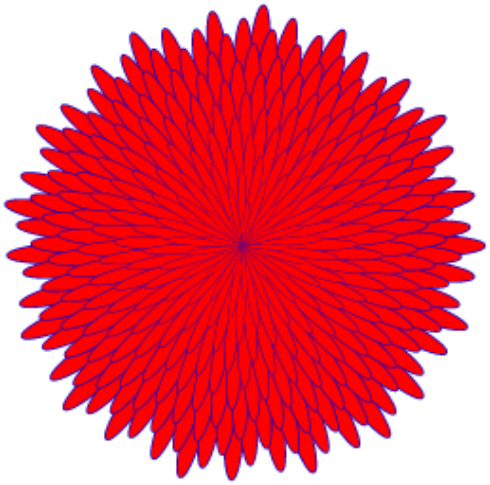
Zájemce o další informace o jazyce Postscript může použít dokumenty, které firma Adobe dává volně k dipozici, především pak učebnici, kterou lze vygooglovat dotazem "postscript bluebook pdf"(240 stran), případně referenční příručku ("postscript PLRM pdf", 912 stran).

Konec poznámek pro život a ne zkoušku.

Cvičení z postscriptu

Pro zájemce jsou zde příklady, jak v postscriptu namalovat v různých měřítcích se opakující vzory.

Detaily jazyka nad rámec tohoto příkladu je možno nastudovat z `http://www.adobe.com/products/postscript/pdfs/PLRM.pdf` manuálu [adobe.com] .



PrProgFyz/11/kytka002.ps Obycejna cervena synteticka kytka

Výše uvedený obrázek se skládá z 300 lístků geometricky podobných tomuto:

PrProgFyz/11/kytka003.ps Pocet listku se meni zmenou parametru N, ten urcuje pocet opakovani cyklu (zde N=1)

Jeden lístek nakreslíme pomocí Bezierova splinu takto:

```
newpath                              %  nova cesta
   0 0 moveto                        %  pocatek
   300 40 300 -40 0 0 curveto        %  bezieruv listek (konci opet v pocatku)
closepath
```

Abychom ale mohli snadno měnit velikost a tvar lístků definueme si symboly `w` a `D`pomocí příkazu `def`. Lomítko před symbolem v postscriptu říká, že symbol se má chápat jménem a ne hodnotou.

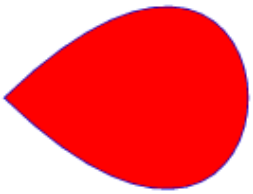
```
/D 350 def                           %  prumer kyticky
/w  40 def                           %  sirka listku
newpath                              %  nova cesta
0 0 moveto                           %  pocatek
D w D w neg 0 0 curveto              %  bezieruv listek (konci opet v pocatku)
closepath
```

Číslo `-w` jsme z `w` vyrobili zavoláním funkce pro otočení znaménka, tedy `w neg`.

Poté je třeba cestu obtáhnout a tutéž cestu znova vybarvit, proto použijeme `gsave, grestore`.

```
  gsave                                  %  schovej cestu a barvu
    0 0 0.8 setcolor                     %  modre
    stroke                               %  obtahni
  grestore
  fill                                   %  a vypln
```

Číslo w můžeme změnit a nakreslit širší lístek.



PrProgFyz/11/kytka004.ps Sirka listku se meni zmenou parametru w

Vykreslení lístku zapíšeme jako proceduru folia :

```
/ folia                                    % procedure folia
   newpath                                 %  nova cesta
  0 0  moveto  D w D w  neg  0 0  curveto   %  bezieruv listek
   closepath
   gsave                                   %  schovej stav
     0 0 0.8  setcolor                     %  modre
      stroke                               %  obtahni
   grestore                                %  vrat stav, tj. priprav cestu
   fill                                    %  vypln aktualni barvou
  def
```

Nyní je potřeba lístky vhodně otáčet a škálovat. Jak a proč se dočtete třeba zde https://en.wikipedia.org/wiki/Golden_spiral.

Nám stačí vědět, že úhel mezi lístky má být o delta = (sqrt(5)+1) *180 stupnu a plocha lístku má růst lineárně s úhlem, jeho velikost tedy roste jako odmocnina. Lístků budeme kreslit N .

```
/D 350   def                               % prumer kyticky
/w  40  def                                % sirka listku kyticky
/N 300  def                                % pocet listku v kvetu
/delta 5  sqrt  1  add  180  mul def        % zlaty rez je nejiraciola-
nejsi cislo
```

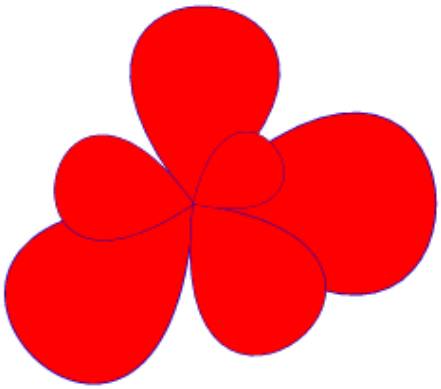
Pro snadné zapsání budeme potřebovat provádět cyklus a to od větších lístků k menším. Cyklus for se zapíše jako

spodnimez krok hornimez prikazy tela cyklu for .

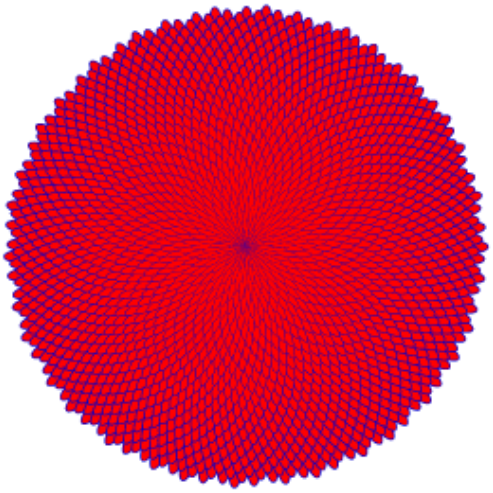
Hodnota řídící proměnné cyklu je dostupná příkazům těla cyklu na vrchu zásobníku.

```
0 1 N 1   sub                                  % for i := 0 to N-1 do  (step 1)
  1 0 0  setcolor
  N  sub    neg  N  div                          %  t := (N-i)/N
   gsave
    sqrt dup scale                             %  Scale(sqrt(t),sqrt(t))
   folia                                        %  zavolej proceduru folia
   grestore                                     %  navrat k puvodnimu meritku
  delta  rotate                                 %  otoc o uhel delta
  for
```

Výsledný kód nemusíte slepovat z výše uvedených kousků, u každého obrázku je uveden v odkazu pod ním.



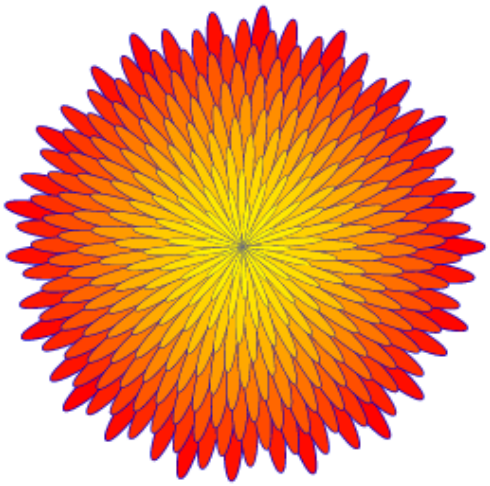
`PrProgFyz/11/kytka005.ps` Malo listku (N=6). Je dobre videt poradi



`PrProgFyz/11/kytka006.ps` Hodne uzkych listku (N=2000).

Chceme-li měnit barvu použijeme řídící proměnnou k výpočtu barvy, nejdříve jako obvykle spočteme hodnotu (N-i)/N, tu si pak na zásobníku zduplikujeme příkazem dup a použijeme ji jako zelenou komponentu pro poceduru setcolor . Protože ale ta očekává pořadí R G B setcolor, musíme k výchozímu G přidat R a pak jejich pořadí G R prohodit příkazem exch , čímž získáme R G. Stačí přidat B=0 a můžeme zavolat setcolor.

```
  N sub neg N div                       %  t := (N-i)/N
  dup 1 sub neg  1 exch 0 setcolor      %  SetRGBColor(1,1-t,0) ... pre-
chod z cervene do zlute
```

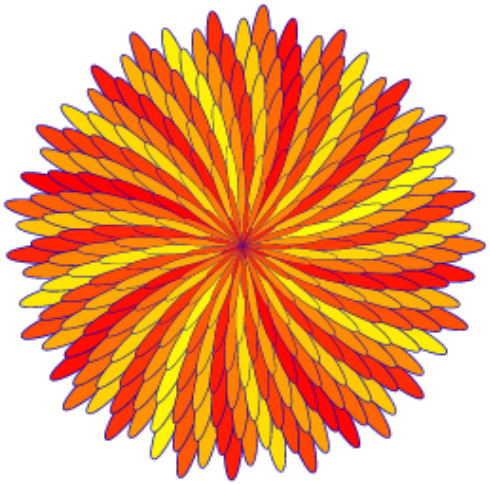


`PrProgFyz/11/kytka007.ps` Barvit listky lze treba takto: SetRGBColor(1,1-t,0)

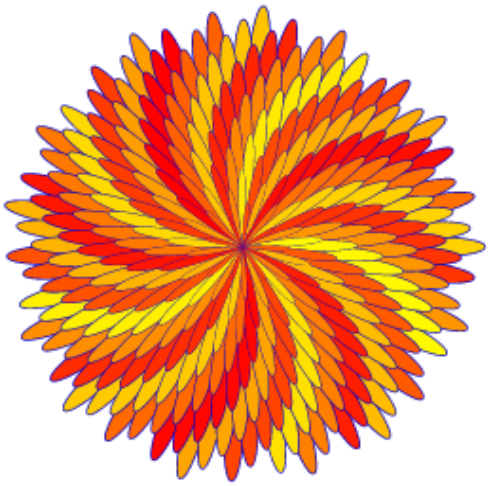
V manuálu si můžeme přečíst jak funguje fukce rand pro náhodná celá čísla a použít ji.

`PrProgFyz/11/kytka008.ps` Nahodna barva

Barvu také můžeme vytvořit podle vztahu SetColor(1, (i mod K)/K,0) a pro vhodně zvolené K nám vyjde:

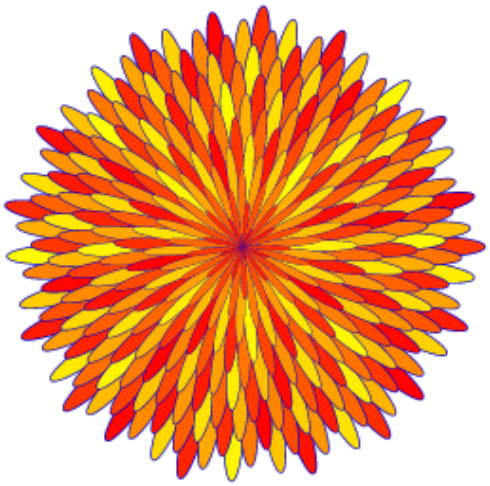


`PrProgFyz/11/kytka009.ps` Modulární barvení: SetColor(1,i mod 55,0)



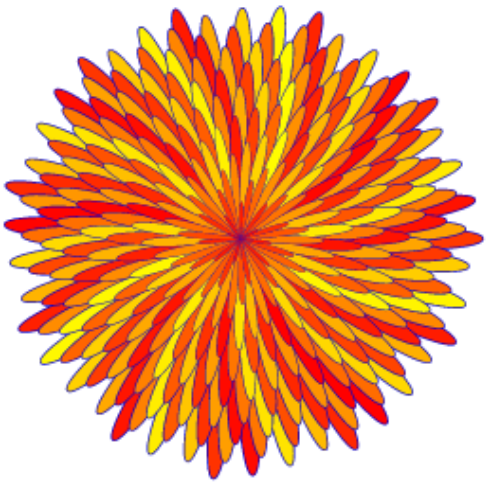
`PrProgFyz/11/kytka010.ps` Modulární barvení: SetColor(1,i mod 34,0)

nebo třeba



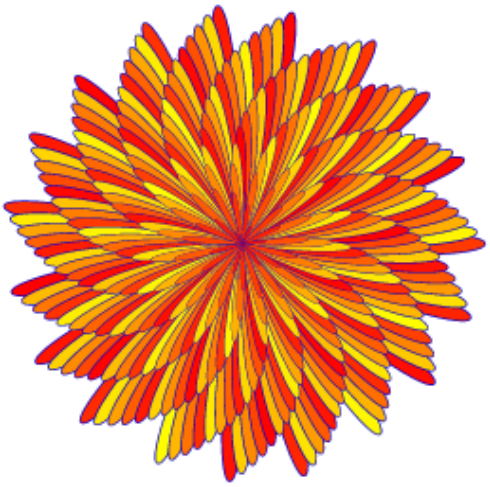
`PrProgFyz/11/kytka011.ps` Barvení SetColor(1,i mod 71,0)

Taktéž můžeme měnit hodnotu úhlu poootočení, třeba na delta = (sqrt(3)+1) *180 :



`PrProgFyz/11/kytka012.ps` delta = (1+sqrt(3))*180 & Barva modulo 71

nebo delta = (sqrt(7)+1) *180 :



`PrProgFyz/11/kytka013.ps` delta = (1+sqrt(7))*180 & Barva modulo 71

Příjemnou zábavu.

Ještě k vyhodnocování výrazů

Když napíšeme a+b*c je zřejmé, kolik má být výsledek. Pokud ale `a` , `b` nebo `c` jsou funkce, může být kromě výsledku důležité i pořadí, v němž se funkce volají. Pokud potřebujeme zaručit dodržení nějakého konkrétního pořadí, je nejjednodušší nějdříve provést sérii přiřazovacích příkazů a pak teprve spočíst výraz:

```
a1 := a;
b1 := b;
c1 := c;

X := a1+b1*c1;
```

Shrnutí: pozor na postranní efekty funkcí volaných ve výrazech.

Neúplné vyhodnocování logických výrazů

U logických výrazů nastává zajímavý jev: počítáme-li hodnotu logického výrazu, řekněme

```
(a >  0) and (b-a >n)
```

tak pro `a<=0` rovnou víme, že výsledek je `false` ať už je hodnota `b` jakákoli. *Neúplné vyhodnocování logických výrazů* je metoda *překladu* logických výrazů, kdy jakmile je jasný výsledek logického výrazu *při jeho vyhodnocování zleva doprava* , vyhodnocování se ukončí. To má několik použití:

```
if  (a <> 0)  and (b/a-c > 0) then  ...
```

takto předřazením testu na dělení nulou zabráníme vlastnímu dělení, protože *a=0* znamená, že výsledek je FALSE ať už je *b* a *c* jakékoli.

Podobně

```
if  JeToZena(C) or MaVPoradkuOhryzek(C) then ...
```

může v programu pro zdravodtní pojištovny kontrolovat osoby C, aniž se dopustíme nedovoleného dotazu na zdravotní stav neexistující části pacientek, obzváště je-li příslušná informace uložena např. ve variantním záznamu a tak není vůbec definována.

Pozn.: Pokud se najde opravdu dobrý důvod, lze si úplné vyhodnocování vynutit zapnutím

```
{$BOOLEVAL ON}
```

Skoky

Strukturované programování mělo za cíl učinit příkaz skoku až na výjimky zbytečným. Tento svůj cíl splnilo, protože jsme jej doposud na přednášce nepotřebovali. Přesto existují situace, kdy je jejich použití na místě. K dipozici máme následující příkazy skoku:

break	ukončí průběh cyklů for, repeat a while
continue	vrací na začátek dalšího cyklu
exit	ukončí běh procedury nebo funkce
halt	ukončí běh programu
goto	skočí na určené návěští (pouze v daném bloku)

Příklady:

```
program  pLabel;
label  L;
begin
 L:  goto  L;
end .
```

Dále

```
for  i :=Low(data)+1  to  High(data)  do
   if  data[i-1]>data[i] then  writeln('NESETRIDENO␣!');
```

z příkladu na třídění bychom spíš měli nahradit

```
for  i :=Low(data)+1  to  High(data)  do
   if  data[i-1] > data[i]  then begin
     writeln('NESETRIDENO␣!');
     break;
   end;
```

aby se varování vypsalo jej jednou. Podobně procedura pro nalezení položky v seznamu může využít exit .

```
program  Test;

function  JeVSeznamu( const  S: array of  integer; n : integer) : boolean;
var    i: integer;
begin
  JeVSeznamu := true;
   for  i := low(S)  to  High(S)  do
     if  S[i] = n  then  exit;

  JeVSeznamu := false;
end ;

begin
 Writeln( JeVSeznamu([12,5,44,69,2,5,3,44,58,56,4],3) );
 Readln;
end .
```

Ladění

Základním postupem při ladění je vložení ladících výpisů do programu. Program tak vypisuje, co zrovna dělá (kde je) a jaké hodnoty mají klíčové proměnné. Do místa, kde se výpisy rozcházejí s očekáváním vložíme další podrobnější výpisy, až lokalizujeme zdroj problému.

Moderní vývojové prostředky umožňují sledovat činnost programu krok za krokem. Přesněji můžeme pozastavit běh programu

- na dalším řádku se vstupem do procedur (F7)
- na dalším řádku bez vstupu do procedur (F8)
- na určeném řádku (breakpoint trvalý F5 a dočasný F4)
- po návratu z procedury (Shif-F8)

Při pozastavení programu můžeme kontrolovat hodnotu výrazů a to buď jednorázově (Ctrl-F7) nebo je můžeme zařadit do seznmu sledovaných výrazů (Ctrl-F5), hodnotu proměnných můžeme dokoce měnit a třeba zkusit, zda tak napravíme co nějaká chyba poškodila....

Ilustrace: hledáme nějakou chybu

Pozor: Když to nechodí zkusíme nejdříve zapnout {`$R+,Q+`} a přidat `uses sysutils`.

Parametr typu procedura a funkce

Předpokládejme, že píšeme modul pro tříděni. Řekněme, že již víme, jaký algoritmus použít i jak psát modul. Když ale začneme psát, zjistíme, že nám něco chybí - možnost napsat modul tak, aby nemusel vědět jaká data vlastně třídí, případně aby mohla tatáž procedura třídit seznamy různých typů. To proto, že když chce procedura pracovat s nějakými daty, musí v Pascalu znát jejich typ.

Jedním z řešení je předpokládat, že ten, kdo bude chtít třídit, místo dat pošle proceduře pro třídění jako parametry něco jiného než samotná data. Třeba jen odkaz na procedury, jednu která porovná j-tý a k-tý prvek a druhou, co je na požádání přehodí.

Zde je příslušný modul:

```
unit  Tridicka;

interface

type  tPorovnaciFunkce = function( j,k : integer ) : integer;
     tPrehazovaciProcedura = procedure( j,k : integer );

procedure  Setrid(Porovnej:tPorovnaciFunkce; Prehod:tPrehazovaciProcedura;l,r:integer);

implementation

procedure  Setrid(Porovnej:tPorovnaciFunkce; Prehod:tPrehazovaciProcedura;l,r:integer);
var   i, j, k_rozhod : Integer;
begin
  k_rozhod := (l + r) div 2;

  i := l; j := r;
  while  i < j  do begin
     while  Porovnej(i,k_rozhod) < 0  do  i:=i+1;
     while  Porovnej(k_rozhod,j) < 0  do  j:=j-1;
     if  i <= j  then begin
      Prehod(i,j);
       if  i=k_rozhod  then  k_rozhod:=j
       else if  j=k_rozhod then k_rozhod:=i;
```

```
      i:=i+1; j:=j-1;
     end ;
   end ;
   if  l < j  then  Setrid(Porovnej, Prehod, l, j);
   if  i < r  then  Setrid(Porovnej, Prehod, i, r);
end ;

end .
```

A zde je program, který modul používá k setřídění pole reálných čísel.

```
program  tridtest;
uses  Tridicka;


var  Data :  array  [0..220000]  of real ;

function  PorovnejData( i,j : integer ) : integer;
{musi se shodovat s  type  tPorovnaciFunkce =  function  ( j,k :  integer  ) :  integer ;}
begin
   if  Data[i] < Data[j]       then  PorovnejData := -1
   else if  Data[i] = Data[j]  then  PorovnejData := 0
                               else  PorovnejData := +1;
end ;

procedure  PrehodData( i,j : integer );
{musi se shodovat s  type  tPrehazovaciProcedura  =  procedure  ( j,k :  integer  );}
var  s : real;
begin
  s       := Data[i];
  Data[i] := Data[j];
  Data[j] := s;
end ;

var  i : integer;

begin
   for  i :=Low(data)  to  High(data)  do  data[i]:=random;
  Setrid(PorovnejData, PrehodData, Low(Data) , High(data) );
   for  i :=Low(data)+1  to  High(data)  do if  data[i-1] > data[i]  then  writeln('
   NESETRIDENO␣!');
  Writeln('OK');
  readln;
end .
```

Těžiště příkladu je na řádcích obsahujících slovo `Setrid`

```
procedure  Setrid(Porovnej:tPorovnaciFunkce; Prehod:tPrehazovaciProcedura;l,r:integer);
...
Setrid(PorovnejData, PrehodData, Low(Data) , High(data) );
```

a v deklaraci typů

```
type  tPorovnaciFunkce = function( j,k : integer ) : integer;
      tPrehazovaciProcedura = procedure( j,k : integer );
```

kde překladači sdělujeme, že se má naučit tyto dva typy, neboť je použijeme jako typy formálního argumnetu. Všimnětě si, že typy se liší od deklarace procedury či funkce jen vynecháním jejícho identifikátoru. Protože je formální parametr *Porovnej* deklarován s typem tProvnavaciFunkce, ví překladač, co má udělat, když narazí na výraz

```
Porovnej(i,k_rozhod)
```

Ilustrace (na přednášce) : krokování programem, ladění

Cvičení: Změňte výše uvedený program tak aby generoval pole 20 náhodných komplexních čísel a pak jej vytiskněnte seřazené podle

a) hodnoty reálné části

b) hodnoty imaginární části

c) velikosti

d) argumentu

Modul Tridicka samozrejme nemente!

Soubory stejných záznamů

Již jste si asi všimli, že v posledních letech umožnil růst výkonu počítačů uchovávat v textové podobě kdejaká data, např. elektronickou poštou (tedy síťovým protokolem pro přenos textového dokumentu) dokážete přenést (a poté uchovat) nejen "Ahoj Honzo"ale i obrázky, archivy, viry...

Mnohé soubory ve vašem počítači ale nejsou textové.

Někdy se používají pro uložení mnoha stejných položek soubory vzniklé záznamem jednotlivých položek tak, jak jsou uloženy v paměti, za sebou do souboru. Budeme-li chtít realizovat velmi dlouhou strukturu typu fronta, můžeme ji kromě paměti uložit realizovat i na disku. K tomu použijeme soubor složený z libovolného počtu stejných záznamů. To Pascalu oznámíme deklarací typu

```
type tSouborPolozek = file of Typ_polozky;
```

s proměnnými, tedy vlastně soubory, tohoto typu pak pracujeme pomocí procedur

Assign	přiřazení jména souboru
Rewrite	Vytvoření nebo zkrácení na nulovou délku + otevření souboru pro četní i zápis
Reset	Otevření existujícího + otevření souboru pro četní i zápis
Read	Přečtení položky
Write	Zápis položky
Seek	Přesunutí polohy pro čtení a zápis na danou položku

S těmito informacemi můžeme již pochopit následující program:

```
program  FrontaVSouboru;


type  tZaznam =  record
                  Jmeno : string[20];{!!! nesmi tam byt jen string  !!! }
                  PolozkaA,
                  PolozkaB,
                  PolozkaC : integer;
                end ;

{Fronta v souboru:}
var  frSoubor :  file of  tZaznam;
    frZacatek,
    frKonec  : integer;

{Operace:}

Procedure frZacni;
begin
  Assign(frSoubor,'Fronta.tmp');
  Rewrite(frSoubor);
end ;

procedure  frVloz(const X: tZaznam);
begin
  Seek(frSoubor,frKonec);
  frKonec:=frKonec+1;
  Write(frSoubor,X);
end ;

function  frVyber(var X: tZaznam): boolean;
begin
  frVyber := false;
  if  (frZacatek > frKonec)  then begin
    Seek(frSoubor,frZacatek);
    frZacatek:=frZacatek+1;
    Read(frSoubor,X);
    frVyber := true;
  end ;
end ;

Procedure  frSkonci;
begin
  Close(frSoubor);
end ;

var  X : tZaznam;

begin
  frZacni;
  X.Jmeno := 'Polozka␣1';  frVloz(X);
```

```
  X.Jmeno := 'Polozka␣2';  frVloz(X);

  if frVyber(X) then Writeln(X.Jmeno);

  X.Jmeno := 'Polozka␣3';  frVloz(X);

  if frVyber(X) then Writeln(X.Jmeno);

  X.Jmeno := 'Polozka␣4';  frVloz(X);
  X.Jmeno := 'Polozka␣5';  frVloz(X);

  while frVyber(X) do Writeln(X.Jmeno);

  frSkonci;
  readln;
end.
```

Velký pozor ! V případě, že používáme tento druh práce se soubory, musíme si být jisti, že v položce jsou opravdu uložena data, která chceme zapsat na disk. Uvažujme následující program:

```
program  Zrada;

type  tZaznam =  record
                  Jmeno : string;
                end ;
var  X : tZaznam;
begin
   X.Jmeno := 'Velmi␣dlouhe␣jmeno␣(mozna␣jeste␣delsi)';

   Writeln(X.Jmeno);
   Writeln(sizeof(X));

   readln;
end .
```

Velmi nás překvapí, že velikost proměnné X jsou pouhé 4 byte. Důvod je prostý, řetězce typu string jsou jistou variantou dynamických polí a vlastní proměná je jen odkazem, kde má program hledat m.j. délku řetězce (délku pole), a znaky řetězce (prvky pole). Uložením tohoto údaje do souboru bychom si uložili pomíjivou (meta-) informaci, ale nikoli potřebná data.

Proto musí být součástí položek záznamů, ze kterých vytvážíme (otypovaný) soubor konstrukcí file of tPolozka, jen typy, jejichž velikost je neměnná, např:

Jednoduché typy (`integer, real, boolean, ...`)

Pole s uvedenými mezemi (`array [1..N] of ...`)

Řetězce s danou délkou (`string[255] ...`)

I když budeme dodržovat tato pravidla, mohou nastat potíže, pokud přenášíme soubory a program, který s nimi pracuje, mezi počítači odlišných architektur (pozor na endiány) a nebo používáme různé kompilátory jazyka Pascal (ve starších verzích byl integer je 16-ti bitové číslo, v příštích nás nemine 64 bitů).

Pozn.: Myslím, že použití otypovaných souborů je velmi omezené.

Dynamické datové struktury (poznámky povrchní)

Prozatím jsme se omezili jen na pole s proměnnou délkou, seznamy, fronty a zásobníky. Ve všech případech jde jen o velmi primitivní dynamické datové struktury.

Viděli jsme, na příkladu procházení souborového systému, že důležitou datovou strukturou jsou stromy, jako speciální případy tzv. grafů. (Pozn. při výkladu: Vrcholy a hrany grafu). V následujícím programu zavedeme takové typy a proměnné, že v nich dokážeme uložit libovolný strom. (Pozn. při výkladu: Graf v matici

```
var  VedeHrana :  array  [tCisloVrcholu, tCisloVrcholu] of boolean;
```

a proč ne (obzvlášť) u stromu.)

Zavedeme rodokmen organismů rozmnožujících se dělením. Každý bude mít Jméno a seznam svých potomků. Pomocí vhodně definované funkce umožníme zapsat strukturu stromu jedním příkazem programu a ukážeme si práci se stromem pomocí dvou funkcí, první *Jmeno2COP* převede jméno na identifikační číslo organismu (index v seznmau t.j. pra-ukazatel) a druhá mi pro každého spočte počet potomků jako součet počtu potomků + počtu jejich potomků+....

```
program  DynDa;

type  tCOP = integer;
```

```
type  tUzel =  record
              Jmeno : string;
              Deti  :  array of  tCOP;
              end ;

var   Pamet :  record
              Prvky :  array of  tUzel;
              Pocet : tCOP;
              end ;

function  X( const  NoveJmeno : string;  const  NoveDeti :  array of  tCOP): tCOP;
var k : integer;
    i : tCOP;
begin
  // Sileny  trik : necham nulty  prvek  nastaveny  na zadne jmeno, zadne  deti
  {Nejdříve musím vyrobit  NOVOU POLOŽKU i}
  i := Pamet.Pocet+1;
  Pamet.Pocet := i;
  if  i > High(Pamet.Prvky)  then begin  SetLength(Pamet.Prvky,10+2*High(Pamet.Prvky));
     end ;
  {Nikdy  nezvetsovat  po jedne , vzdy  geometrickou  radou  !!! }
  X := i;
  with  Pamet.Prvky[i]  do   begin
    Jmeno := NoveJmeno;
    SetLength(Deti,High(NoveDeti)+1);
    for  k:=Low(NoveDeti)  to  High(NoveDeti)  do  Deti[k] := NoveDeti[k];
  end ;
end ;

function  Jmeno2COP( const  Jmeno:string) : tCOP;
var  i : tCOP;
begin
  Jmeno2COP := 0;
    for  i := Low(Pamet.Prvky)  to  High(Pamet.Prvky)  do
      if  Pamet.Prvky[i].Jmeno = Jmeno  then begin
      Jmeno2COP := i;
      exit;
      end ;
end ;

function  PocetPotomku( const  COP:tCOP) : integer;
var  i,s : integer;
begin
  s := 0;
    with  Pamet.Prvky[COP]  do
      for  i := Low(Deti)  to  High(Deti)  do  s:=s+1+PocetPotomku(Deti[i]);
  PocetPotomku := s;
end ;


var  Otec : tCOP;
begin

  Otec := X('Petr',[
                   X('Karel',[
                             X('Mirek',[]),
                             X('Zdenek',[])
                             ]),
                   X('Ivan',[
                             X('Hugo',[
                                       X('Rudolf',[]),
                                       X('Gustav',[
                                                   X('Cecil',[])
                                                   ]),
                                       X('Klement',[])
                                       ])
                            ])
                ]);

  Writeln(PocetPotomku( Jmeno2COP('Ivan') ) );

  Readln;
end .
```

Příklady problémů, které vedou na uložení dat v podobě stromu.

- aritmetický výraz
- setříděná data

Cvičení: Napište proceduru, která vytiskne rodokmen. Nejjednodušší bude rekursivní varianta s parametry COP, a pořadím generace (= počet mezer).

```
Petr
   Karel
      Mirek
      Zdenek
   Ivan
      Hugo
         Rudolf
         Gustav
             Cecil
         Klement
```

Případně zksute i složitější podobu s čárkami

```
Petr
 + Ivan
 |   +  Hugo
 |        + Rudolf
 |        + Gustav
 |        |    -  Cecil
 |        - Klement
 - Karel
     +  Mirek
     -  Zdenek
```

Novinky v jazyce Pascal

Přetěžování funkcí a operátorů.Něco o objektech.

Objektů se nelekejte (na tečky nehleďte)

Tendence a paradigmata v programování jsou v mnohém podobná evoluci biologických druhů. Ve velkých množstvích je to jednoduché, přežije *the fittest* . Pokud ovšem mluvíme o dostatečně malém rybníku, hraje velkou roli i náhoda.

Nelze dokázat, že dnes používané metody programování jsou ty nejlepší, prostě se vyvinuly z těch předchozích a kdejaký jazyk má své příznivce a odpůrce. Kromě jednotliých jazyků se vyvíjí i způsoby jak psát programy (jakési vzory správně psaných programů, tzv. paradigmata). Některé koncepce vyhynou (třeba dnes by žádného tvůrce nového počítačového jazyka nenapadlo označovat povinně řádky rostoucí posloupností čísel - BASIC) jiné jsou do té míry úspěšné, že se s nimi musejí v konkurečním boji o přežití vyrovnat všichni. Dnes je takovým vítězným paradigmatem objektově orientované programování (OOP) a v posledním desetiletí OOP dozrálo pro nasazení v rozličných oblastech programování. Výjimkou je bohužel právě oblast, kterou se snaží pokrývat tento úvodní kurs programování, a kde jakkoli žádná z inkarnací OOP nebyla, myslím, vhodná. Jenže jste si určitě všimli, že v Delphi (tedy sytému s překladačem, vývojovým prostředíma, knihovnami, dokumentací ....) je použitá verse Pascalu nazývána ObjectPascal. Objekty nezahrnuli tvůrci do jazyka z idealistických důvodů, nýbrž proto, že právě OOP umožnilo běžnému programátoru zvládnout tvorbu Opravdu Užitečných Programů v prostředí pokročilých grafických, komunikačních a databázových API dnešní doby (definice příkladem: Assign, Reset, Rewrite, Read, Write, Close... tvoří v Pascalu API pro práci se soubory).

Existuje několik pokusů o Objektově Orientované Vědecké Výpoočty, ale bohužel současné počítačové jazyky neposkytují zdaleka vše, co by bylo potřeba. (Není divu, počítačové jazyky jsou méně a méně vyvíjeny s ohledem na naše potřeby, typický zákazník není student ani učitel fyziky. Evoluční boj se dnes odehrává v pro nás vzdálených oblastech C#, Javy a servletů - abych vás praštil žargonem).

Připoměňme si základní události v dávné historii počítačových jazyků (zamlčuji COBOL, PL1, ...)

1945 - stroj. kód

1957 - překlad výrazů – (FORTRAN nebo později BASIC)

1961 - strukturované příkazy (třeba ALGOL 60)

1971 - strukturovaná data (typ record v Pascalu, struct v C)

Jenže to jsme zhruba u roku 1971 a zřejmě se ještě něco převratného muselo na poli poč. jazyků urodit ...

Především, jak že se má pracovat s proměnnou typu záznam?

Aby mělo smysl pakovat víc různých věcí dohromady musí se to taky dohromady používat. A to, jak jsme viděli, jde kromě přiřazovacího příkazu (nuda) jen předáváním záznamu jako paramteru proceduře či funkci. Jako příklad vezměma následující program:

```
program  Maticka;

type  tVektor = array of real;
      tMatice = record
                  M , N : integer;
                  a : array of tVektor;
                end;

procedure  VytvorJednotkovou( var  A : tMatice; N : integer);
var  i,j : integer;
begin
  SetLength(A.a,N,N);
  A.M := N;
  A.N := N;
   for  i := 0 to N-1  do for  j := 0 to N-1 do
     if  i=j  then  A.a[i,j]:=1  else  A.a[i,j]:=0;
end ;
```

```
function  JeCtvercova( var  A : tMatice): boolean;
begin
  JeCtvercova := A.M = A.N;
end;

function  Stopa( var  A : tMatice): real;
begin
 ...
end;

procedure  UvolniPamet( var  A : tMatice);
begin
  SetLength(A.a,0,0);
  A.M := 0;
  A.N := 0;
end ;

var  S : tMatice;

begin
  VytvorJednotkovou(S,3);
  Writeln(JeCtvercova(S));
  Readln;
end .
```

V programu je záměrně použito předání proměnné typu tMatice odkazem a vždy je to první parametr, takže všechny procedury a funkce se volají:

UdelejNeco(PromTypuMatice, ostatní parametry);

tedy vlastně (vzpomeneme-li si, že příkazy jazyka Pascal se skoro mohly/měly číst jako věty)

UdělejNeco s Čím Tak a Tak.

První změna, se kterou přichází OOP je obrácení slovosledu na

S Tímhle UdělejNeco Tak a Tak.

což psáno v Pascalu bude vypadat

PromTypuMatice.UdelejNeco(pripadne parametry)

Proto se výše uvedený program změní takto:

```
program  MatickO;

type  tVektor =  array of  real;
      tMatice =  object
                  M , N : integer;
                  a :  array of  tVektor;
                   constructor  VytvorJednotkovou(k : integer);
                   function  JeCtvercova: boolean;
                   function  Stopa: real;
                end ;


  constructor  tMatice. VytvorJednotkovou(k : integer);
var  i,j : integer;
begin
  SetLength(  a ,k,k);
    M  := k;
    N  := k;
   for  i := 0  to  N-1  do for  j := 0  to  N-1  do
     if  i=j  then   a [i,j]:=1  else   a [i,j]:=0
end ;

function    tMatice. JeCtvercova: boolean;
begin
 JeCtvercova :=   M = N ;
end ;

function    tMatice. Stopa: real;
begin
 ...
end ;

var  S:tMatice;

begin
    S. VytvorJednotkovou(3);
  Writeln(  S. JeCtvercova);
```

```
  Readln;
end .
```

V deklarační části oznámíme, že procedura VytvorJednotkovou a dvě funkce JeCtvercova a Stopa jsou součástí součástí záznamu, který se teď jmenuje objekt. K prvkům záznamu tak přibyly tzv.metody, které s prvky záznamu pracují. Jejich vlastní deklarace vypadá jako běžná deklarace funkce, s tím, že prvky záznamu/objektu jsou přístupné, jako by to byly lokální proměnné a identifikátor metody předchází určení pro který typ objektu danou metodu vlastně deklarujeme, protože nic nebrání tomu aby dva objekty mohly mít stejně se jmenující metodu. Proto také typ objekt nemůže být beze jména:

```
var  x,y: object  // nelze!!
           ...
            Procedure  MetodaX;  // pod jakým jménem bych asi pak MetoduX deklaroval
          end ;
```

Navíc musí být typy objekt deklarovány jako globální (tady si tvůrci jen ulehčili práci, když to stejně nikdo nechce).

Důležité jsou pro nás objekty hlavně proto, že i když sami nebudme chtít vlastní objekty tvořit, může nějaký užitečný modul exportovat (místo typu a sady procedur pro práci s ním) právě objekt. Proto musíme vědět, že pri použití tohoto modulu budeme muset proměnnou daného typu deklarovat a používat právě způsobem, jímž se objekty používají, tedy PromennaTypuObjekt.NejakaMetoda(parametry) .

Cvičení: Doplňte v obou příkladech výše tělo procedury/metody stopa....

Druhou podstatnou vlastností objektů je dědičnost a opět si ji ukážeme na příkladě, který by měl připomínat situaci ze života. Řekněme, že máme (z webu) k dispozici následující modul pro *quicksort* třídění:

```
unit  ObjTridic;

interface

type  tTridic =  object
                  function   Porovnej( j,k : integer ) : integer;  virtual; abstract ;
                  procedure  Prehod( j,k : integer );  virtual; abstract ;

                 procedure  Setrid(l,r:integer);
                end ;

implementation

procedure  tTridic.Setrid(l,r:integer);
var   i, j, k_rozhod : Integer;
begin
  k_rozhod := (l + r) div 2;

  i := l; j := r;
   while  i < j  do begin
     while  Porovnej(i,k_rozhod) < 0   do  i:=i+1;
     while  Porovnej(k_rozhod,j) < 0   do  j:=j-1;
     if  i <= j  then begin
      Prehod(i,j);
       if  i=k_rozhod  then  k_rozhod:=j
       else  if j=k_rozhod  then  k_rozhod:=i;

      i:=i+1; j:=j-1;
     end ;
   end ;
   if  l < j  then  Setrid(l, j);
   if  i < r  then  Setrid(i, r);
end ;

end .
```

Procedury pro porovnání a přehozní z minulé verse s procedurálními parametry byly tentokrát nahrazeny metodami. Tento objekt je ale nehotový, umí sice třídit ale přitom nemá co a tady ani tedy neví jak to nic porovnávat a přehazovat. Metody Porovnej a Prehdo jsou sice tedy deklarovány, aby mohly být použity v metodě Setrid, ale jejich kód se odkládá do budoucna. Tentokrát ale nejde jako u předběžné (*forward)* deklarace jen o odložení na pozdější místo v

daném modulu, fukce jsou označeny jako abstraktní a pokud je použít skončí behovou chybou. Deklarce funkcí je odložena až do doby, kdy bude co třídit.

Vezmeme tedy data (pole reálných čísel) a přidáme k nim metody pro porovnácní dvou prvků a jejich přehození a vytvoříme z nich potomka objektu typu tTridic jak je tomu v následujícím programu. Navíc přidáme inicializační metodu, která tam musí být z technických důvodů (a ještě navíc má místo slova procedure psáno constructor) a využijeme ji k obsazení pole náhodnými čísly. Navíc i z kontroly správnosti setřídění učiníme metodu v souladu s principy OOP. Tak dostaneme:

```
program  ObjTridTest;
uses  ObjTridic;

type  tSeznamRCisel =  object (tTridic) // je to potomek tTridic
                        Data :  array  [0..220000]  of  real;

                         constructor  Init; // naprosto nezbytný kvůli virtuálním metodám

                         function   Porovnej( j,k : integer ) : integer;  virtual ;
                         procedure  Prehod( j,k : integer );  virtual ;

                         function   Zkontroluj : boolean;
                       end ;

function  tSeznamRCisel.Porovnej( j,k : integer ) : integer;
begin
   if  Data[j] < Data[k]       then  Porovnej := -1
   else   if  Data[j]=Data[k]  then  Porovnej := 0
                          else  Porovnej := +1;
end ;

procedure  tSeznamRCisel.Prehod( j,k : integer );
var  s : real;
begin
  s       := Data[k];
  Data[k] := Data[j];
  Data[j] := s;
end ;

constructor  tSeznamRCisel.Init;
var  i : integer;
begin
   for  i := Low(Data)  to  High(Data) do Data[i]:=random;
end ;

function   tSeznamRCisel.Zkontroluj : boolean;
var  i : integer;
begin
  Zkontroluj := false;
   for  i :=1  to  High(Data)  do if  Data[i-1] > Data[i]  then  exit;
  Zkontroluj := true;
end ;

var   Seznam:tSeznamRCisel;

begin
  Seznam.Init;
  Seznam.Setrid(0 , High(Seznam.data) );
   if  Seznam.Zkontroluj  then  Writeln('OK')  else  Writeln('Prusvih');
  readln;
end .
```

Výklad (3 minuty): Dědičnost, virtuální metody, kostruktor.

V případě náhrady var parametrů tečkou šlo jen o jakýsi přepis, který sice obrážel změnu pohledu na operace s daty (procedury →metody), který ale např. ve výsledném strojovém kódu nemusí být vůbec vidět. (Nevypadá ale kód X.Init; X.Setrid; X.Zkontroluj nějak hezčeji...) Výše uvedený kód ale využívá také druhé klíčové vlastnosti zvané **dědičnost** . Ta představuje opravdovou změnu na všech úrovních.

Tak jako v minulém příkladě bylo možno jednou provždy vyřešit quicksort a už jen kostruovat seznamy různých druhů, našly dnes objekty (hlavně kvůli dědičnosti) svoje velké využití v oblasti grafického rozhraní programů a toto hlavní použití ovlivnilo zpětně jazyk.(Na přednášce za 10 sekund říct proč...) Pro užití objektů ve vědeckých výpočtech ale chybí v ObjectPascalu některé důležité možnosti a tak s objekty skončíme výše uvedeným ilsutračním příkladem na třídění, který

už tak používá dost prvků OOP aby k jejich úplnému vyložení bylo potřeba několik přednášek.

Cvičení: Opět uvažujte seznam náhodných komplexních čísel, modifikujte výše uvedený typ tSeznamRCisel na tSeznamCCisel a pridejte do nej metody SetridPodleRealCasti, SetridPodleImagCasti a SetridPodleAbsHodnoty a asi uvazujte i tri testy ZkontrolujPodleRealCasti atd... Zkompilujte. Vyzkoušejte.

Vybrané numerické algoritmy

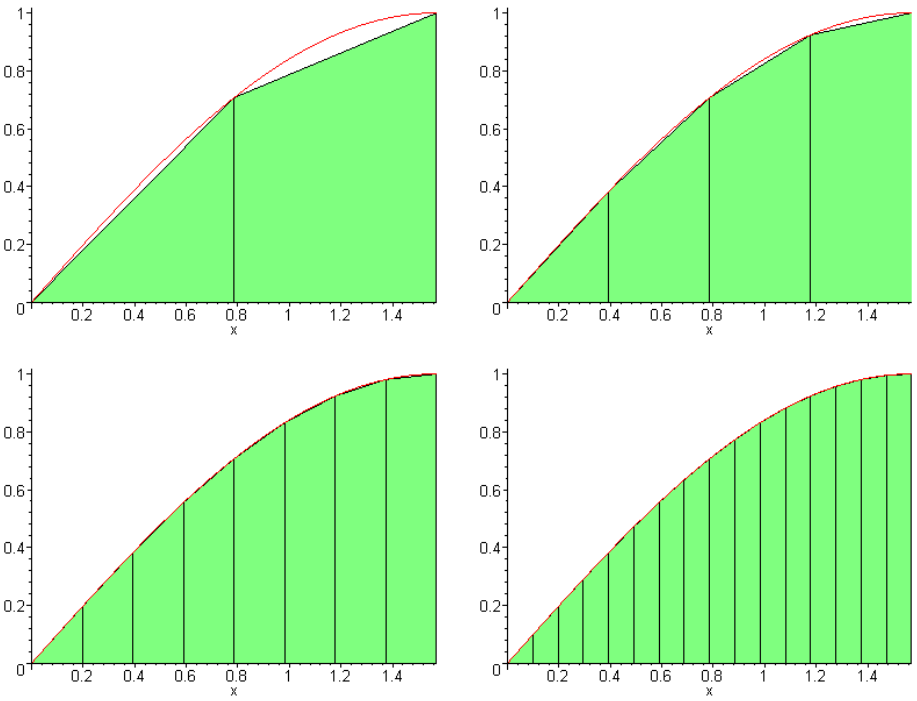
Numerická kvadratura, integrace diferenciálních rovnic. Náhodná čísla v počítači.

Numerická kvadratura (výpočet určitého integrálu funkce jedné proměnné)

Nejdříve si ukážeme jak spočítat tzv. lichoběžníkovým pravidlem přibližnou hodnotu určitého integrálu. Řekněme, že počítáme

$$\int_0^{\pi/2} \sin(x)\, dx$$

Pro rozdělení integračního intervalu na 2, 4, 8 a 16 podintervalů (kde funkci nahradíme lineární interpolací) vypadá lichoběžníkové pravidlo takto:



Protože určitý integrál je lineární zobrazení z prostoru funkcí na intervalu <a,b>do R, nepřekvapí, že lichoběžníkové pravidlo se redukuje na lineární kombinaci funkčních hodnot, konkrétně

$$\int_a^b f(x)\,dx = dx\left(\frac{1}{2}f(x_0) + f(x_1) + f(x_2) + f(x_3) + \ldots + f(x_{n-1}) + \frac{1}{2}f(x_n)\right)$$

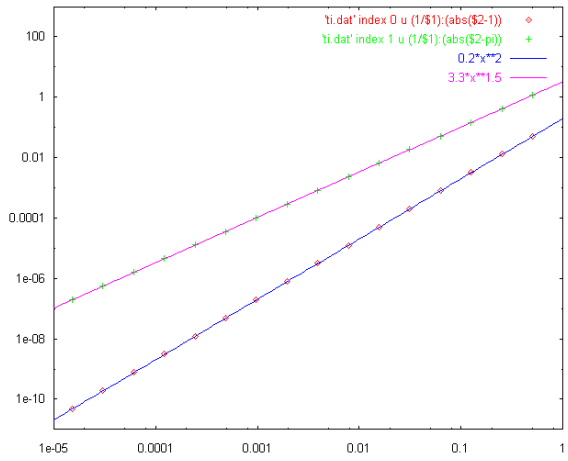
$$dx = \frac{b-a}{n} \qquad x_k = a + k\,dx \qquad x_0 = a \qquad x_n = b$$

Následující program spočítá podle tohoto schématu jak výše uvedený intergrál tak dvojnásobek plochy pod půlkružnicí sqrt(1-x$^{2)}$.

```
program  TrapInt;

type  tRFunkce =  function  (x:real) : real;

function  LichobeznikovePravidlo(f: tRFunkce; a,b : real; N:integer):real;
var  dx : real;
    s_kraj,s_vnitr : real;
    i : integer;
begin
  dx := (b-a)/N;
  s_kraj  := f(a) + f(b);
```

```
  s_vnitr := 0;
   for  i := 1  to  N-1  do  s_vnitr := s_vnitr + f(a+dx*i);

  LichobeznikovePravidlo:=(s_vnitr+s_kraj/2.0)*dx;
end ;

function  fce(x:real):real;
begin
  fce := sin(x);
end ;

function  gce(x:real):real;
begin
  gce := sqrt(1-x*x);
end ;

const  nmax=200000;
var  n: integer;

begin
  n:=2;
   while  n < nmax  do begin
    Writeln(n, LichobeznikovePravidlo(fce, 0,Pi/2, n):18:14);
    n:=n*2;
   end ;
  Writeln;Writeln;

  n:=2;
   while  n < nmax  do begin
    Writeln(n, 2*LichobeznikovePravidlo(gce, -1,1, n):18:14);
    n:=n*2;
   end ;
  Readln;
end .
```

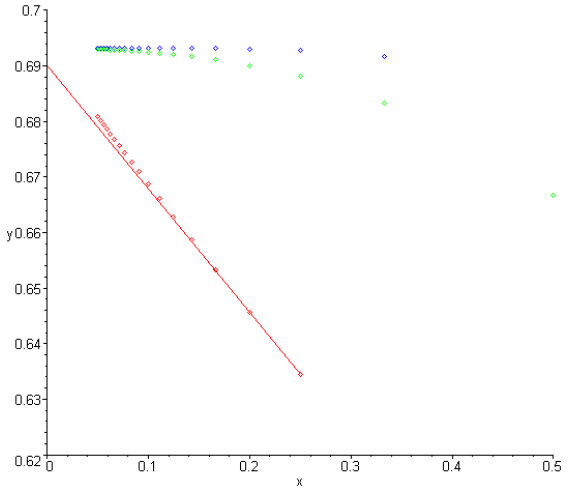
Vhodně zpracován gnuplotem, dá výstup programu tento graf závislosti chyb na velikosti intervalu dx .



Na přednášce:

- Komentář o hezkých a ošklivých funkcích.
- Koměntář o sčítání $(-1)^{k-1}/k$, aneb jak co nejsložitěji spočíst $\ln(2)$.

  K tomu potřeba následující obrázek, kde se má ilustrovat, že sečny konvergují k tečně v $x = 0$ a má-li funkce Taylorův rozvoj v $x = 0$ musí zelené body jít k limitě kvadraticky atd.



Lichoběžníkové pravidlo splňuje tři základní valstnosti integrálů, lineariatu při násobení konstantou, translační nezávislost a škálování při lineární transformaci měnící šířku intervalu. To znamená, že můžeme zkoumat, jak umí integrovat funkce x^k v intervalu $(0, 1)$ (jde totiž i o varianty vzorce pro součet aritmetrické řady) a výsledky pak zobecnit pro všechny polynomy a intervaly. Při rozdělení na N stejných intervalů jsou hodnoty spočtené lichoběžníkovým pravidlem uvedeny v tabulce:

f	Lichob$(f, 0, 1)$
1	1
x	$\frac{1}{2}$
x^2	$\frac{1}{3} + \frac{1}{6N^2}$
x^3	$\frac{1}{4} + \frac{1}{4N^2}$
x^4	$\frac{1}{5} + \frac{1}{3N^2} - \frac{1}{30N^4}$
x^5	$\frac{1}{6} + \frac{5}{12N^2} - \frac{1}{12N^4}$
x^6	$\frac{1}{7} + \frac{1}{2N^2} - \frac{1}{6N^4} + \frac{1}{42N^6}$
x^7	$\frac{1}{8} + \frac{7}{12N^2} - \frac{7}{24N^4} + \frac{1}{12N^6}$
x^8	$\frac{1}{9} + \frac{2}{3N^2} - \frac{7}{15N^4} + \frac{2}{9N^6} - \frac{1}{30N^8}$

Proto tzv. inženýrskou indukcí dostáváme, že pro každý polynom f bude platit, že

$$I(f, dx) = \text{SpravnaHodnotaIntegralu} + a * q + b * q^2 + ...$$

kde $q = 1/N^2$ případně $q = dx^2$(na konstantě nesejde)

Protože umíme prokládat několika body polynom, lze sou4et pro lichoběžníkové pravidlo spočíst pro několik q a poté skusit extrapolaci na hdonotu $q = 0$. Klíčová část může za použití dříve napsané procedury pro Lagrangeovu intepolaci třeba takto:

```
function  RTQ3(f: tRFunkce; a,b : real; N:integer):real;
begin
  RTQ3:=LInterp(0,                  // extrapolace do q = 0
     [1,sqr(0.5),sqr(0.25)],        // z hodnot q = 1/N^2, 1/(2N)^2, 1/(4N)^2
     [LichobeznikovePravidlo(f,a,b,N),
      LichobeznikovePravidlo(f,a,b,2*N),
      LichobeznikovePravidlo(f,a,b,4*N)]);
end ;
```

Poznámka: Klíčová slova jsou Richarsonova extrapolace, Rombergova integrace. Výše uvedená funkce je natolik jednoduchá, že nestojí za to vykládat tzv. Simsonovo pravidlo, které odpovídá extrapolaci prokládáním dvěma body.

Kromě ekvidistatních vzorků funkčncích hodnot se můžeme spolu s Gaussem ptát jestli by nebylo lepší brát funkční hodnoty v takových bodech, aby jejich vhodně vážený součet uměl přesně integrovat co nejvíce polynomů. Pro zajímavost je zde uveden program používající osmibodový Gausův kvadraturní vzorec, kdybyste chtěli experimentovat...

```
program  GaussInt;
```

```
type  tRFunkce = function (x:real) : real;

function  Gauss8(f: tRFunkce; a,b : real):real;
const  t : array [1..4] of real = (0.183434642495649805,0.525532409916328986,
                                   0.796666477413626740,0.960289856497536232);
       w : array [1..4] of real = (0.181341891689180991,0.156853322938943644,
                                   0.111190517226687235,0.050614268145188l296);
var  s,K,L : real;
     i : integer;
begin
  s := 0;
  K := (b+a)/2.0;
  L := (b-a)/2.0;
  for i:=low(t) to high(t) do s := s + ( f(K+L*t[i]) + f(K-L*t[i]) )*w[i];
  Gauss8:=s*(b-a);
end ;

function  fce(x:real):real;
begin
  fce := sin(x);
end ;

function  gce(x:real):real;
begin
  gce := sqrt(1-x*x);
end ;

begin
  Writeln(Gauss8(fce, 0,Pi/2):1:16);
  Writeln(2*Gauss8(gce,-1,1 ):1:16);
  Readln;
end .
```

Jeho výstup, tedy čísla

```
1.00000000000000
3.14431044825165
```

jasně ukazují, že pro hezkou funkci, jakou je sinus, lze z osmi fukčních hodnot spočíst výsledek integrace přesně na 16 míst, zatímco pro funkce ošklivé máme potíže na čtvrtém desetinném místě.

Protože existuje mnoho různých způsobů, jímž může být funkce ošklivá, jedinou obecnou radou může být, že před integrací rodělíme v problematických bodech interval na více dílů a v každém podintervalu pak substitucí převedeme integrand na hezkou fukci. (Samozřejmě, i tuto operaci můžeme více či méně převést na kvadraturní vzorce, pokud k tomu budmeme mít důvod.)

Náhodná čísla v počítači V Pascalu je k dispozici funkce **function** `random: real`, která vrátí (pseudo) náhodné číslo v [0,1) přičemž platí, že pravděpodobnost, že padne do daného podintervalu je rovna jeho délce.

Na přednášce si ukážeme, že s jejich pomocí lze počítat integrály. Konkrétně spočteme težiště polokoule, přičemž teorii nahradíme pozorováním, že náhodně voleneé body odpovídají polohám atomů ideálního plynu a těžiště polokoule se tak s rostoucí hustoto plynu musí přibližovat těžišti homogenního tělesa.

```
program teziste;
var i:integer;
    deset:integer=10;

    Tx,Ty,Tz, x,y,z:real;  // implicitní inicializce na 0

begin
  for i:=1 to 1000000000 do begin
    repeat
      x:=2*random-1;        // nejprve vezmi rovnoměrně
      y:=2*random-1;        // rozdělené body v opsaném kvádru
      z:=random;
    until x*x+y*y+z*z<1;    // a pak zkontroluj, že bod lezi v polokouli

    Tx := Tx + x;
    Ty := Ty + y;
    Tz := Tz + z;           // připočti příspěvek k težišti stejně hmotných bodů
```

```
    if i=deset then begin  // a nevypisuj miliardu hodnot
      writeln( i:11,Tx/i:11:7,Ty/i:11:7,Tz/i:11:7);
      deset:=deset*10;
    end;
  end;
end.
```

Jeho výstup ukazuje, jak se přibližujeme přesné poloze ($\vec{x}_T = \{0, 0, 3/8\}$):

```
          10  0.0650535  0.2024963  0.5772875
         100  0.0195484  0.0103283  0.3972573
        1000  0.0012102  0.0061291  0.3735217
       10000  0.0065276  0.0013014  0.3741451
      100000 -0.0014587 -0.0017565  0.3744419
     1000000  0.0000842  0.0004729  0.3753041
    10000000 -0.0000549 -0.0000908  0.3750039
   100000000 -0.0000461  0.0000162  0.3750245
  1000000000 -0.0000138 -0.0000292  0.3750123
```

Numerické řešení diferenciálních rovnic Diferenciální rovnicí budeme rozumět rovnici prvního řádu

$$\frac{dy}{dt} = F(t, y)$$

Na přednášce si ukážeme následující kód. Realizuje tzv. Eulerovu metodu založenou na jednoduché nakhrazení derivace

$$\frac{dy}{dt} \doteq \frac{\Delta y}{\Delta t} = \frac{y(t + \Delta t) - y(t)}{\Delta t},$$

tedy konkrétně

$$y(t + \Delta t) = y(t) + \Delta t\ F(t, y(t)).$$

Program, který řeší rovnici $dy/dy = -y$ s počáteční podmínkou $y(0) = 1$ je:

```
program Priklad1;

const subdiv=100;
      dt=0.1/subdiv;
var   n : integer;
t,y,dydt: real;

begin
  n =0;

  t = 0;  // jako obvykle nezapomenout na  inicializaci , tady ma navic
  y = 1;  // vyznam pocatecnich podminek

  while (t<20) do begin
    if n mod subdiv = 0 then writeln(t,'␣',y);
    dydt:= -y;
    y   := y + dydt * dt;
    t   := t + dt;
    n   := n+1;
  end;

end.
```

Snadno se lze přesvědčit, že s jemnějším dělením ma kratší a kratší krok Δt pomocí konstanty `subdiv` se více a více přibližujeme přesnému řešení.

Dále si ukážeme, že Newtonovy pohybové rovnice $m\ddot{\vec{x}} = \vec{F}/m$ lze také zapsat jako diferenciální rovnici, ale pro vektorovou veličinu, jejíž polovina složek jsou polohy a druhá polovina rychlosti

$$\frac{d}{dt}\begin{pmatrix} \vec{x} \\ \vec{v} \end{pmatrix} = \begin{pmatrix} \vec{v} \\ \vec{F}(t, \vec{x}, \vec{v})/m \end{pmatrix}.$$

Protože chceme mít po ruce kód, který řeší rovnice pro libovolnou podobu síly, kód přeuspořádáme.

```
program odr4;

type tIndex = (ix, iy, ivx, ivy);

type tVektor = array[tIndex] of real;
tPohybovaRovnice = function (t:real; const U:tVektor): tVektor;

procedure Krok_Euler(var U:tVektor; fce_dUdt:tPohybovaRovnice; var t:real; dt:real);
var i:tIndex;
    prava_strana:tVektor;
begin
  prava_strana:=fce_dUdt(t,U);
  for i:=low(tIndex) to high(tIndex) do U[i] := U[i]+dt*prava_strana[i];
  t:=t+dt;
end;

function pohybova_rovnice_planety(t:real; const U:tVektor): tVektor;
const GM = 4*Pi*Pi;
var   r2,r_3 :real;
begin
  pohybova_rovnice_planety[ix] := U[ivx];
  pohybova_rovnice_planety[iy] := U[ivy];
  r2 := sqr(U[ix])+sqr(U[iy]);
  r_3 := 1/(r2*sqrt(r2));
  pohybova_rovnice_planety[ivx] := -GM*U[ix]*r_3;
  pohybova_rovnice_planety[ivy] := -GM*U[iy]*r_3;
end;

var Y: tVektor;
    t:real;
    n:integer=0;

const dt=1/365/24; // jedna hodina (ani tak nebude s Eulerovou metodou vysledek prilis presny,
     zkuste)
      tmax=2;
      vypis_po=24; // lze naredit vystup

begin
  t := 0;
  Y[ix] := 0.5;      // jednotka: AU
  Y[iy] := 0;
  Y[ivx] := 0;
  Y[ivy] := 2*Pi;    // Au/rok, kruhova rychlost pro r=1AU

  while t<tmax do begin
    if n mod vypis_po = 0 then writeln( t:7:4, '␣',Y[ix]:9:6,'␣',  Y[iy]:9:6);
    Krok_Euler( Y , @pohybova_rovnice_planety, t, dt);
    n:=n+1;
  end;

  // Readln; // neni potreba pri vystupu do souboru
end.
```

Toto rozdělení na rovnici a numerickou metodu je užitečné proto, že nyní můžeme (i bez pochopení detailů) nahradit Eulerovu metodu metodou lepší (u pohybu planety se to projeví potlačením nefyzikálního stáčení její eliptické orbity). Například

```
procedure Krok_Midpoint(var U:tVektor; fce_dUdt:tPohybovaRovnice; var t:real; dt:real);
var i:tIndex;
    prava_strana, Upul:tVektor;
begin
  prava_strana:=fce_dUdt(t,U);
  for i:=low(tIndex) to high(tIndex) do Upul[i] := U[i]+0.5*dt*prava_strana[i];

  prava_strana:=fce_dUdt(t+0.5*dt,Upul);
  for i:=low(tIndex) to high(tIndex) do U[i] := U[i]+dt*prava_strana[i];

  t:=t+dt;
end;
```

Na webu cvičení najdete i lepší metodu (zvanou RK4).

Přetěžování funkcí, procedur a operátorů (overloading) I když dnes už nemusí být identifikátory dlouhé nanejvýš šest či osm znaků, stále je problém, když dvě procedury dělají

totéž, ale ne docela. V následujícím programu jsou definovány čtyři funkce označené stejným identifikátorem *Soucet* , místo abychom použili čtyři různé idnetifikátory, řekněme *SoucetIntPole, SoucetRealPole, SoucetRRPoli, SoucetRIPoli* . Pravidlo pro tvorbu přetížených funkcí a procedur je snadné: žádné dvě nesmějí mít stejné typy povinných formálních parametrů (nepovinné jsou ty s deklarací Id : Typ = DefaultHodnota), rozhodně nestačí aby se dvě funkce lišily jen typem vrácené hodnoty. Překladač (i programátor, že) prostě musí mít jasno v tom, která procedura se tím či oním zápisem aktuálních parametrů vlastně zavolá.

```
program  Soucty;

type  tRealVec =  array of  real;

const  Ai :  array  [1..3]  of  integer = (1,2,3);
       Ar :  array  [1..3]  of  real    = (1.0,2.0,3.0);

function  Soucet( const  A:  array of  integer):integer; overload;
var i,s : integer;
begin
  s:=0;
   for  i := Low(A)  to  High(A)  do  s := s+A[i];
  Soucet := s;
end ;

function  Soucet( const  A:  array of  real):real;  overload ;
var  i : integer;
    s : real;
begin
  s:=0;
   for  i := Low(A)  to  High(A)  do  s := s+A[i];
  Soucet := s;
end ;

function  Soucet( const  A,B:  array of  real):tRealVec;  overload ;
var  i : integer;
    s : tRealVec;
begin
    SetLength(s,High(A)+1);
    Assert(High(A)=High(B),'Soucet␣(r+r)␣vektoru:␣Scitana␣pole␣museji␣by␣stejne␣dlouha!
   ');
     for  i:= Low(A)  to  High(A)  do  s[i] := A[i]+B[i];
    Soucet:=s;
end ;

function  Soucet( const  A:  array of  real;  const  B:  array of  integer):tRealVec;
   overload ;
var  i : integer;
    s : tRealVec;
begin
    SetLength(s,High(A)+1);
    Assert(High(A)=High(B),'Soucet␣(r+i)␣vektoru:␣Scitana␣pole␣museji␣by␣stejne␣dlouha!
   ');
     for  i:= Low(A)  to  High(A)  do  s[i] := A[i]+B[i];
    Soucet:=s;
end ;

procedure  WriteVecLn(  const  A: array of real; w: integer = 14; d : integer = 12);
var  i,imax : integer;
begin
  Write('[');
  imax := High(A);
   for  i:= Low(A)  to  imax  do begin
    Write(A[i]:w:d,'␣');
     if  i < imax  then  Write(',');
   end ;
  WriteLn(']');
end ;

begin
  WriteLn(Soucet(Ai));
  WriteLn(Soucet(Ar));
  WriteVecLn(Soucet(Ar,Ai));
  Readln;
end .
```

Nejnovějším hitem v oblasti přetěžování je rozšíření působnosti operací jako je `+, -, *, /` na nové typy. Zejména to má smysl u komplexních čísel, a tak je kdispozici knihovna `ucomplex`, která zavede typ complex a operace nad mím (včetně mnoha funkcí). Stčí napsat `uses ucomplex` a váš program bude umět komplexní čísla. Jak to funguje a jak byste si něco takového mohli napsat sami by mělo být zřejmé z ukázkového programu:

```
program cmplxtest;

type complex = record
                  re,im : real;
               end;

function mkcomplex(const a,b : real) : complex;
begin
  result.re := a;
  result.im := b;
end;

operator * (const p : complex; const a : real) : complex; overload;
begin
    result.re := p.re*a;
    result.im := p.im*a
end;

operator * (const p,q : complex) : complex; overload;
begin
    result.re := p.re*q.re-p.im*q.im;
    result.im := p.re*q.im+p.im*q.re;
end;

var u,v:complex;

begin
    u := mkcomplex(0,1);
    v := u*u;
    writeln( v.re,'␣',v.im);
end.
```

Protože přetěžování funkcí spíše zvyšuje čitelnost programu (s obvyklými důsledky), i když nepochází od Wirtha, nemusíte se za použití této techniky stydět. To naopak neplatí o následujícím tématu.

Přetypování (type cast)

je operace, kdy z výrazu nebo proměnné jednoho typu vyrobíme výraz či proměnnou jiného typu, aniž se s bity a bajty něco děje.

```
program  TypeCasts;

type  tBarva = (Cerna, Modra, Zelena, Cervena, Bila);
      t8byte = array[0..7] of byte;
      pInt64 = $ ^{int64} $;

var  Barva : tBarva;
     i     : Integer;
     i64   : Int64;
     r     : Real;

begin
  {1. Přetypování ordinálního výrazu}
  Writeln( Boolean(0) );
  Writeln( Integer(Modra) ); {tohle by šlo i přes ord}
  Barva := tBarva( 4 );

  {2. Přetypování proměnné}
  byte(Barva) := 4;           {provede totéž, co předchozí příkaz}

  r := Pi;                    {int64(t8byte(Pi)) se samozřejmě nesmí, Pi není proměnná}
  Writeln(int64(t8byte(r))); {Kvůli bezpečnosti je zakázáno
                             přímé přetypování real --> celé číslo}


  {3. Přetypování ukazatele }
  Writeln( ( pInt64( @r ) )^ );
```

```
  Readln;
end .
```

Výše uvedený program ilustruje tři různé druhy přetypování:

- Přetypování výrazu ordinálního typu, kdy z výrazu ord. typu tA učiníme výraz ord. typu tB. Protože ordinální typy jsou vlastně jen intervaly celých čísel, existuje přirozené zobrazení z tA do tB [takové, kdy ord(a) = ord(b)].
- Přetypování proměnné, kdy řekneme kompilátoru, aby na oblast paměti, kde je uložena proměnná typu tA nahlížel jako by tam byla uložena proměnná typu tB. Podmínkou je, aby velikosti proměnných obou typů byly shodné [sizeof(tA) = sizeof(tB)] a aby nešlo o převod mezi celými a reálnými čísly (z důvodu bezpečnosti, aby se někdo nedivil, že int64(PromennaTypuRealSHodnotouRovnouPi) = 4 614 256 656 552 045 848).
- Přetypování ukazatelů, kdy výraz typu ukazatel na tA (tedy $^{tA)}$ předěláme na výraz typu ukazatel na tB, přičemž oba ukazují na oblast začínající na stejné adrese.

V každém případě je přetypování operací velmi post-Wirthovskou a někteří z vás jej nebudou nikdy potřebovat.

Osnova

1. Problémy

Klasické elementární algoritmy (Eukleidův algoritmus, Eratosthenovo síto).

Matematické výrazy, Hornerovo schéma.

Chyby - reprezentace reálných čísel pomocí mantisy a exponentu (a proč je z hlediska chyb +- horší než */).

Příklady výpočtu funkcí daných vzorcem, součtem řady nebo rekurentním vztahem.

Základní numerické algoritmy (hledání kořenů, kvadratura).

Práce s poli, základní operace z lineární algebry, GJ eliminace, jak ji pužít na inversi matice. Interpolace (Lagrange).

Časová náročnost algoritmu.

Vyhledávání prvku v poli, (klíč, k čemu je dobré hledat v setříděném poli)

Vnitřní třídění – O($N^{2)}$ algoritmy jako příklady na práci s poli. Quicksort jako příklad dobrého algoritmu a rozumět principu a proč je obvykle $O(N \log N)$.

Fronta a zásobník, operace vložení a výběru jako další příklad práce s poli. Kruhová fronta.

Vstup a výstup dat (výstup na konsoli, přesměrování, formátování textového výstupu).

Nástin 2D grafiky. Gnuplot, formát vstupních dat, změna rozsahu os, k čemu index a using. Postscript. Náhodná čísla jsme nestihli.

Modularita. Jak rozdělit program na modul a jednodušší program. Proč vůbec moduly.

2. Jazyk Pascal

Proměnné a konstanty.

Celočíselné typy, char, boolean, real.

Výraz, operátory, priorita, standardní funkce.

Přiřazovací příkaz, kompatibilita pro přiřazení, výjimka real:=integer;

Jednoduchý a složený příkaz.

Podmíněný příkaz, cykly.

Program, procedura a funkce.

Lokalita a zastiňování, rozsah účinnosti deklarace.

Předávání parametrů hodnotou a odkazem, konstantní parametry.

Strukturované typy (pole, záznam, řetězce).

Inicializované proměnné a typované konstanty.

Parametry typu `array of X` (open array) . Dynamická pole. Funkce `Low` a `High`.

Binární operace na celých číslech a zkrácené vyhodnocování logických výrazů.

Textový vstup a výstup. Otypované soubory.

Parametr typu procedura a funkce.

Modularita (unit), práce s dokumentací knihoven.

Dynamické datové struktury.

Ukazatele. Přetypování.

Použití již hotového objektu, základní odlišnosti od modelu záznam+procedury.

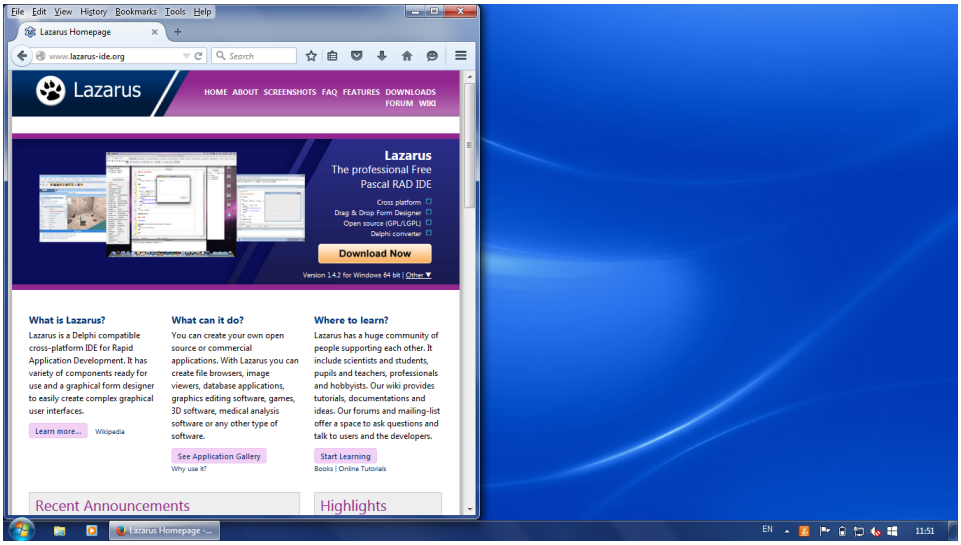
Literatura

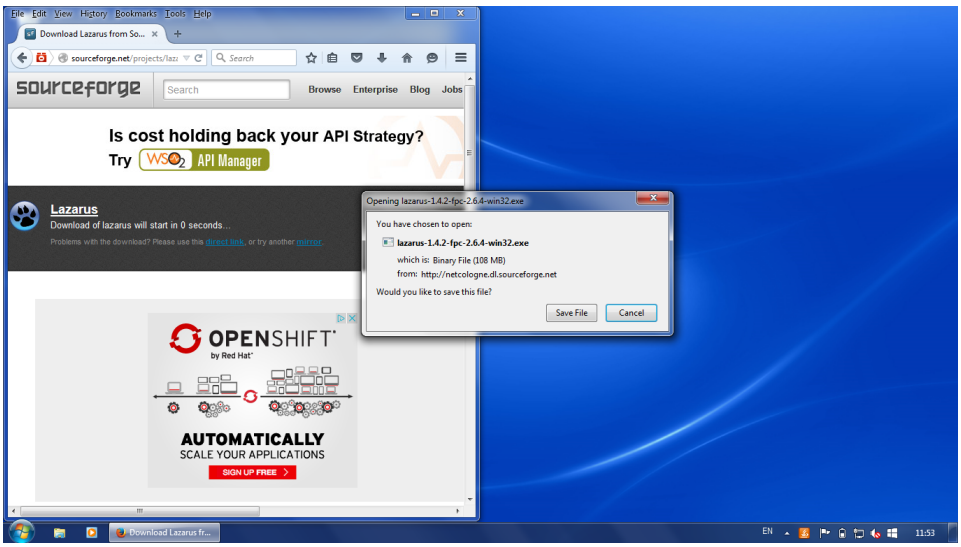
P. Satrapa: Pascal pro zelenáče, Neokortex, Praha 2000
Delphi 6 Help, 1990-2003
Poznámky k přednášce na webu

DODATEK A: Lazarus

Jak nainstalovat prostředí pro psaní programů v Pascalu.

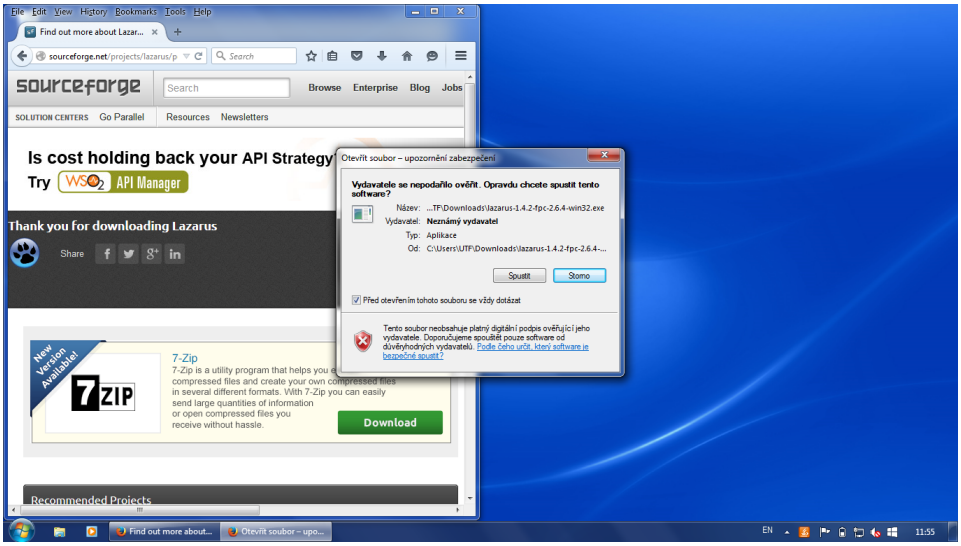
Instalační program lze nalézt zde: `http://www.lazarus-ide.org/` . (Obrázky na této stránce zvětšíte/zmenšíte kiknutím.)

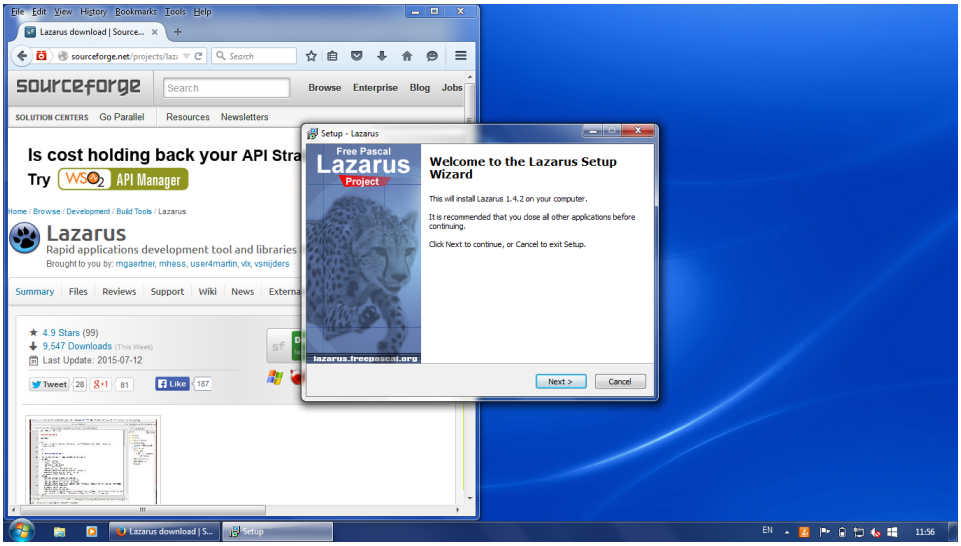




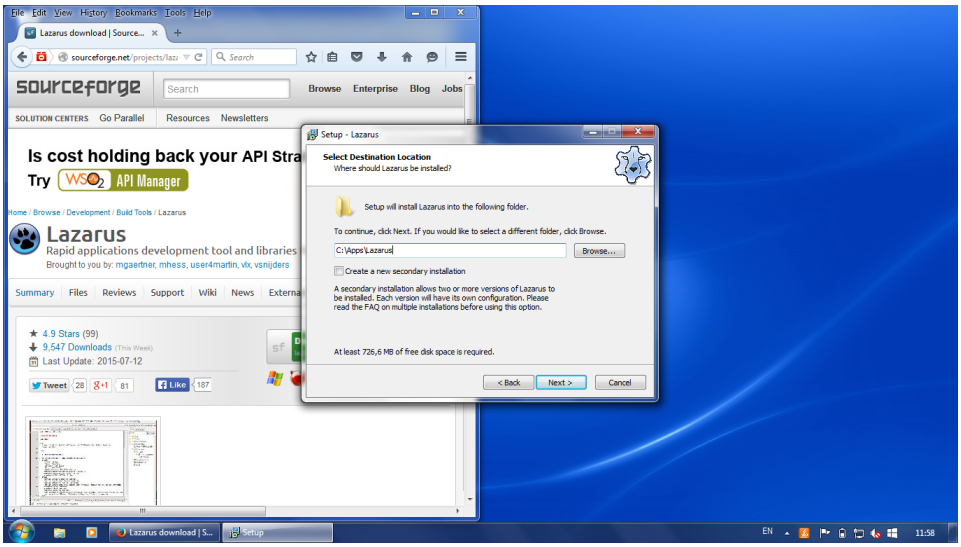
Instalace

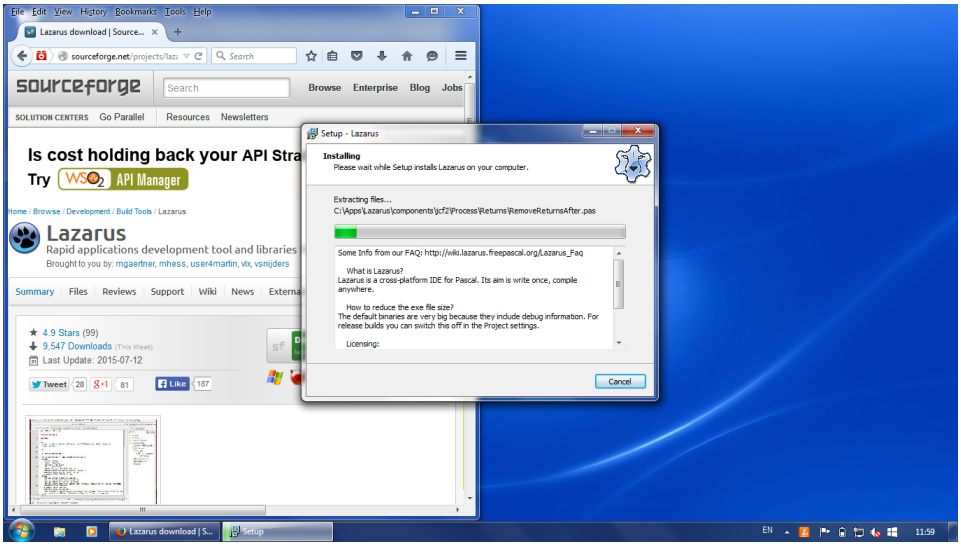
Při instalaci je potřeba vhodně odklikávat "Spustit", "OK", "Next"a podobná tlačítka (vše se bude lišit počítač od počítače).





Pozor, je potřeba zvolit vhodný adresář, kam se překladač bude instalovat. Velmi se **nedoporučuje** použít adresář s mezerou nebo diakritikou v názvu (tedy např. *Program Files*).

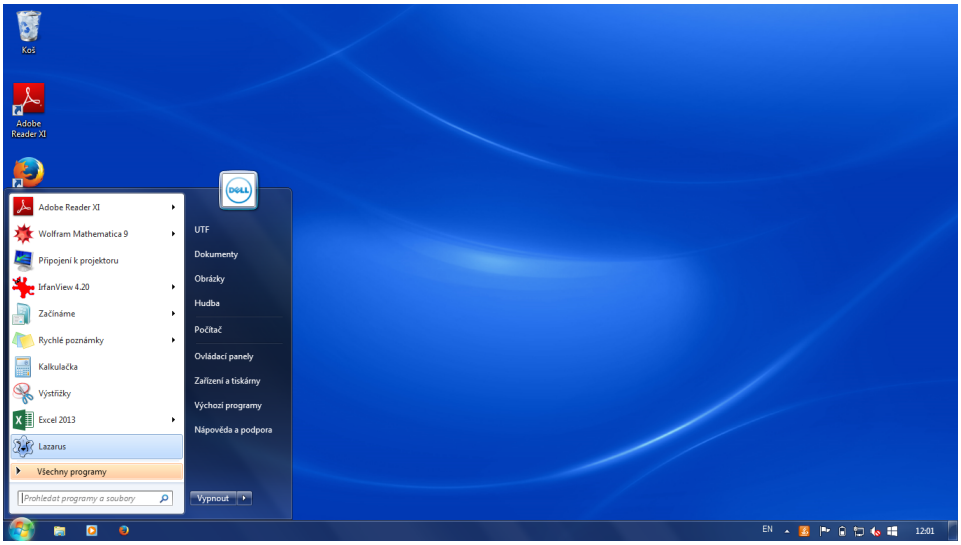




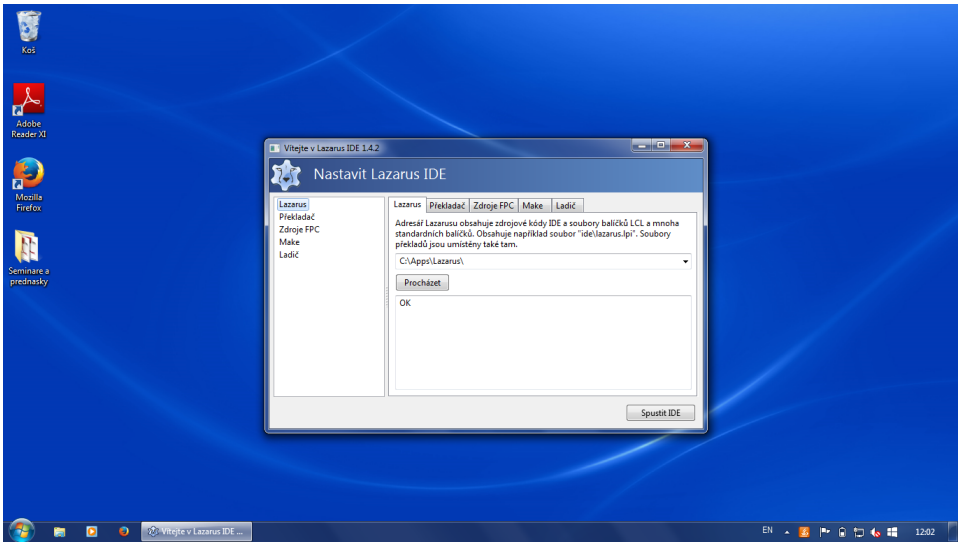
Začínáme!

(Pár poznámek pro ty, co nikdy nic nekompilovali. Uživatelé Linuxu to jistě zvládnou bez nápovědy. Macintosh nemám v dosahu, ale Lazarus by tam měl chodit také.)

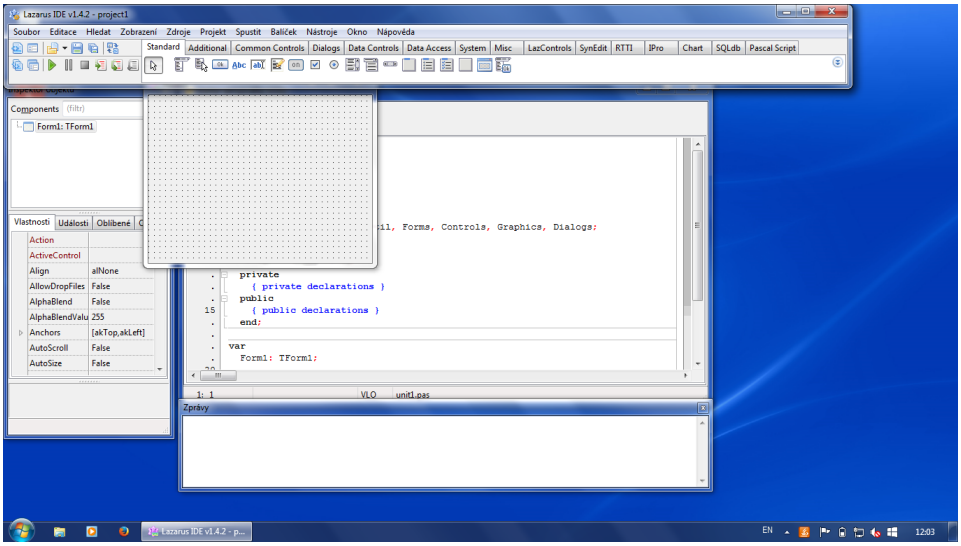
Nejprve nainstalovaný program spustíme.



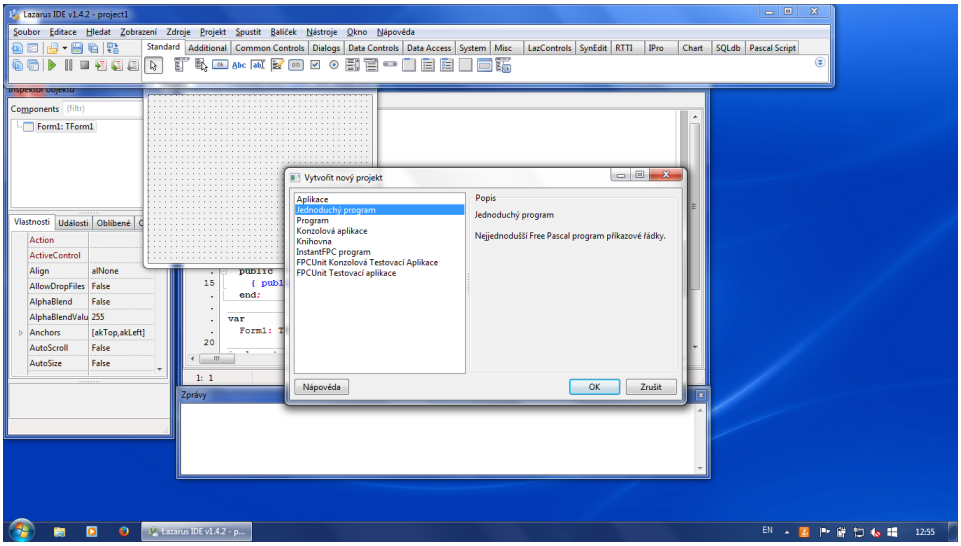
Při prvním spuštění se nás ještě zeptá, zda nechceme změnit konfiguraci. To ale nemáme zapotřebí.



Poté se již program "spustí". Jde o druh programu zvaný *Integrované vývojové prostředí* (zkratka IDE). Na začátku předpokládá, že máme zájem o psaní okénkových programů. V našem předmětu ale vystačíme s programy pro příkazovou řádku (viz níže).



Programům se zde říká vznešeně "Projekt". Proto, když chceme začít psát nový program vybereme v menu "Projekt/Nový projekt ...". Zde máme na výběr několik možností, nám stačí volba "Jednoduchý program".



Tak se otevře okno s jednoduchým textovým editorem a v něm je minimální platný program v Pascalu.