Programování pro fyziky

ZS 2021/2022

T. Ledvinka

Ústav teoretické fyziky
MFF UK

(obrázky převážně z wikipedia.org)

Fyzika a počítače

- experimenty (návrh, simulace, řízení, sběr a analýza výsledků)

- simulace a modelování (meteorologie, astrofyzika, kosmologie, … kde nepraktický experiment)

- molekuly, materiály (pochopení a předpovědi vlastností, cílený návrh)

- teorie (výpočty, předpovědi)

- výuka (pochopení, "experimenty")

- AI ?

- Kvantové počítače ?

(Jde to i bez nich, ale je to těžší. Co by za ně dali Newton, Lagrange, Gauss, ...)

Algoritmus

Abu Ja'far Muhammad ibn Musa al-Khwarizmi (780-850)

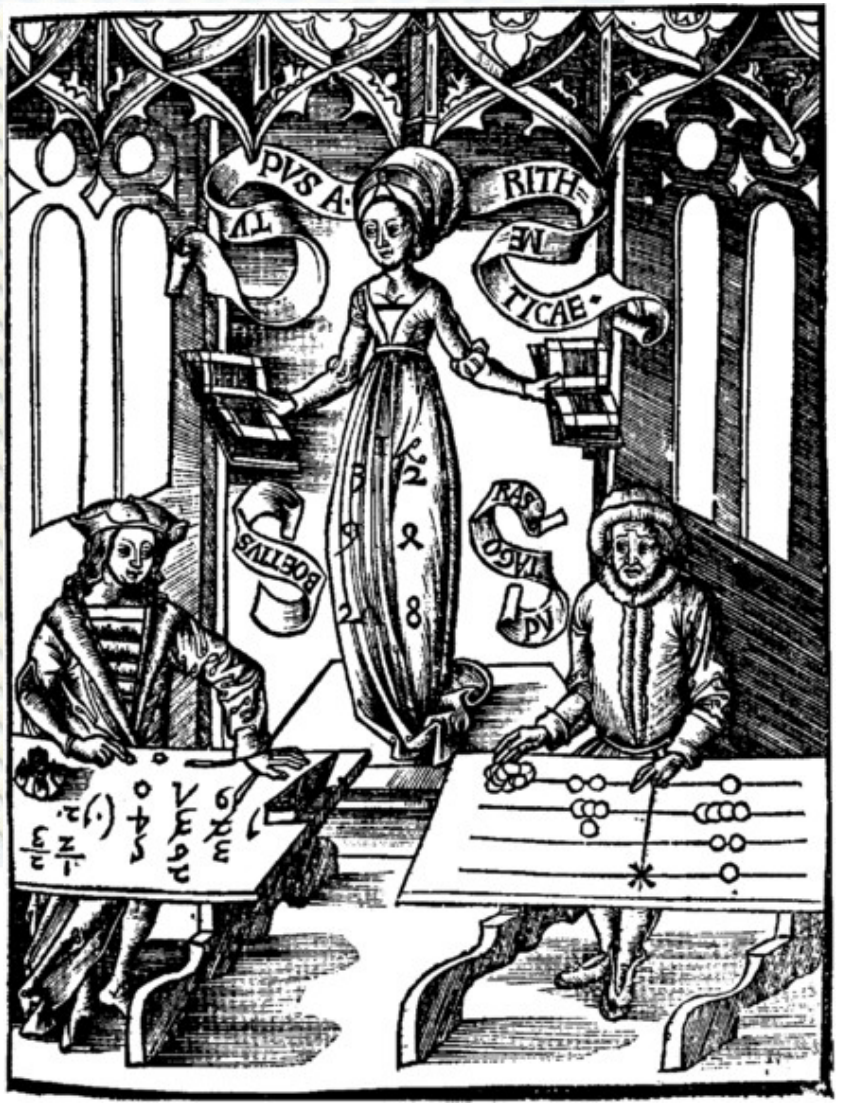
- *Liber Algorismi de numero Indorum*
  - symboly ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’
  - sčítání, odčítání, násobení a dělení
  - „dobrý algoritmus zefektivní výpočet“ (LXXVII * XIII = MI)
- s nástupem abaku pojem algoritmus mizí
- objevuje se znovu až s moderním významem

Zatímco původní význam dnes spadá pod pojem aritmetika,

termín **algoritmus** je dnes pojmem matematickým/informatickým:

Návod jak získat výsledek, kde

- je zadána posloupnost přesně daných kroků
- počet kroků potřebných k nalezení výsledku je konečný
- výsledek je správně
- stejné zadání vede ke stejnému výsledku



Euklidův algoritmus

Návod jak poměřit dvě úsečky nejdelším dílem [Euklides (450-380 př.n.l.) chápe čísla jako úsečky, tj. úsečka je celočíselným násobkem jednotkové úsečky), tedy dnešními slovy jak najít **největší společný dělitel dvou celých čísel**.:

... pokud AB neměří CD, pak, když opakovaně odčítáme úsečku kratší od delší, dostaneme zbytek, který ...

Naše první formulace: **Největší společný dělitel dvou celých kladných čísel nalezneme tak, že dokud jsou obě čísla různá, odečítáme menší od většího.**

• je to algoritmus ?

Velká čísla ?

→ Ať počítají **otroci**.

• ať si nastudují Euklidovy spisy

• vzorový výpočet

• návody (viz dále)

• Proč zmiňovat Euklida? Co bychom vymysleli my? Postupný test 1..min(a,b)? Rozklad na prvočinitele?

• Proč začínat kurs výkladem pojmu algoritmus?

Aniž se o tom bude příliš mluvit, budeme „hledat“ dobré algoritmy na řešení vzorových úloh

Vzor správného „programu“ posvěcený matematiky. Blízký intuitivně pojmu „automatizovat řešení úlohy“.

Návod č.1

NSD dvou čísel reprezentovaných dvěma hromádkami kamenů, zjistím tak že, dokud jsou obě hromádky různě velké odebírám z větší hromádky takový počet kamenů, jaký je na hromádce menší.

Návod č.2

NSD dvou čísel spočtu tak, že si je napíšu každé na samostatnou tabulku, položím je vedle sebe a poté

1. kouknu jestli na tabulkách nejsou stejná čísla, pokud ano jdu na bod 6.
2. pokud je větší číslo napsáno na pravé tabulce, prohodím je
3. vezmu novou tabulku, položím ji úplně napravo
4. napíšu na pravou tabulku rozdíl čísel z levé a prostřední tabulky
5. zahodím levou tabulku a pokračuji na bodě 1.
6. levou tabulku pošlu poštou zadavateli úlohy, pravou zahodím

• jenže otrok neumí číst → přeložím do otrokódu: ⌛🔔☪✍📄✍☝👉✉

• problémy: plýtvání, program a data jsou ukládány odlišně, musí překládat kvalifikovaná síla, ….

Řešení:

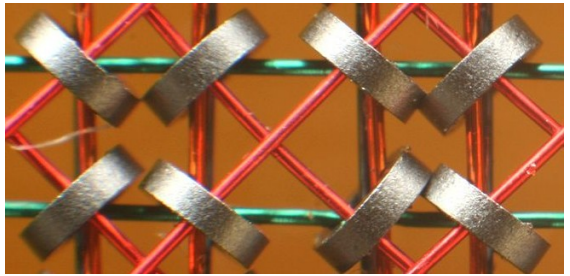
• Omezíme vzletnost jazyka pro zápis algoritmu, aby byl přeložitelný do otrokódu samotným otrokem.

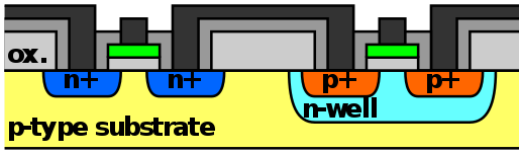
• Místo zahazovacích tabulek použijeme jednu velkou tabuli s přepisovatelnými kolonkami na čísla

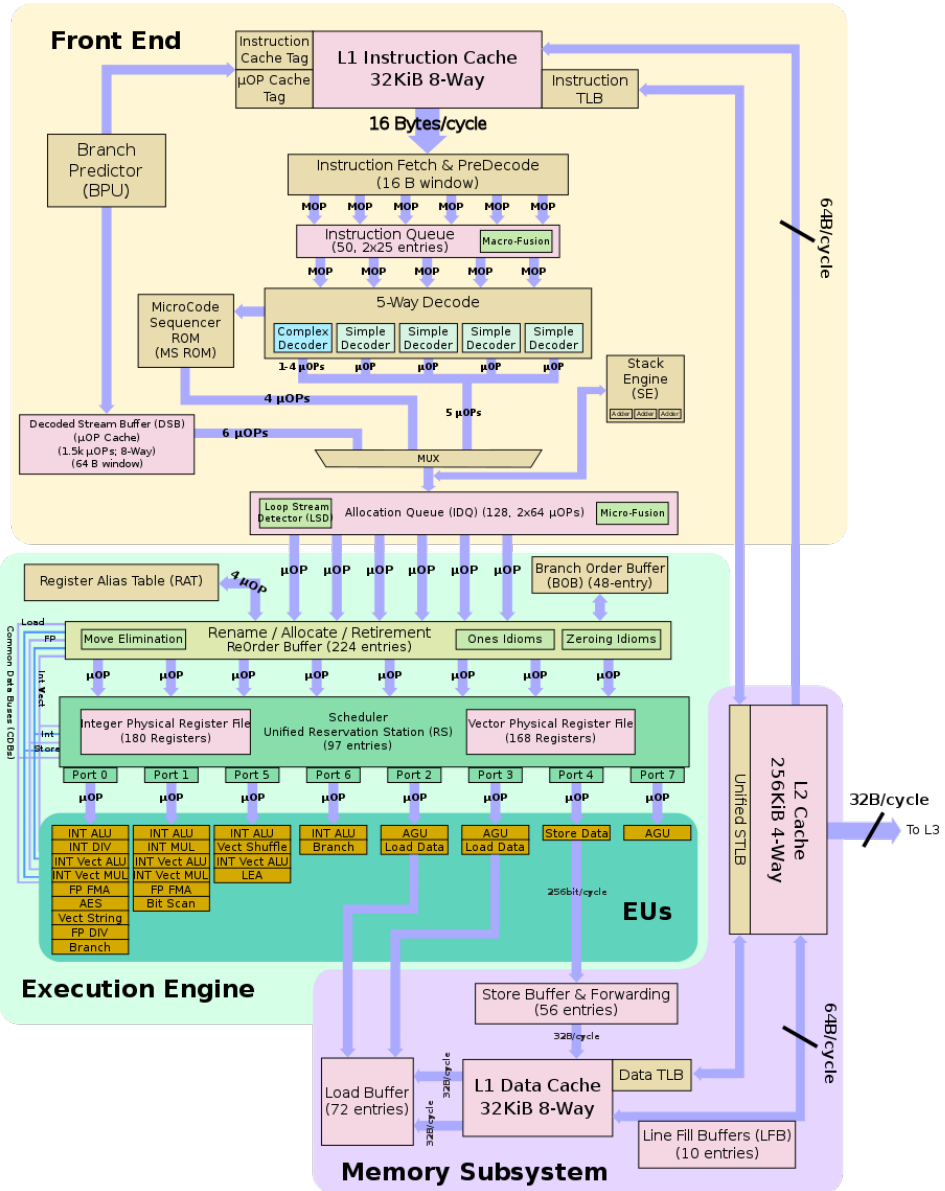
• Instrukce otrokódu budou čísla taktéž zapsaná na této tabuli…

Programovací jazyk

```
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.
```




ox.
n+
n+
p+
p+
n-well
p-type substrate


Front End
Instruction Cache Tag
µOP Cache Tag
L1 Instruction Cache 32KiB 8-Way
Instruction TLB
16 Bytes/cycle
Branch Predictor (BPU)
Instruction Fetch & PreDecode (16 B window)
MOP
Instruction Queue (50, 2x25 entries)
Macro-Fusion
64B/cycle
MicroCode Sequencer ROM (MS ROM)
5-Way Decode
Complex Decoder
Simple Decoder
1-4 µOPs
µOP
4 µOPs
5 µOPs
6 µOPs
Stack Engine (SE)
Decoded Stream Buffer (DSB) (µOP Cache) (1.5k µOPs; 8-Way) (64 B window)
MUX
Loop Stream Detector (LSD)
Allocation Queue (IDQ) (128, 2x64 µOPs)
Micro-Fusion
Register Alias Table (RAT)
Branch Order Buffer (BOB) (48-entry)
Load
FP
Common Data Buses (CDBs)
Int Vect
Int
Store
Move Elimination
Rename / Allocate / Retirement ReOrder Buffer (224 entries)
Ones Idioms
Zeroing Idioms
Integer Physical Register File (180 Registers)
Scheduler Unified Reservation Station (RS) (97 entries)
Vector Physical Register File (168 Registers)
Port 0
Port 1
Port 5
Port 6
Port 2
Port 3
Port 4
Port 7
INT ALU
INT DIV
INT Vect ALU
INT Vect MUL
FP FMA
AES
Vect String
FP DIV
Branch
INT ALU
INT MUL
INT Vect ALU
INT Vect MUL
FP FMA
Bit Scan
INT ALU
Vect Shuffle
INT Vect ALU
LEA
INT ALU
Branch
AGU
Load Data
AGU
Load Data
Store Data
AGU
256bit/cycle
EUs
Execution Engine
Unified STLB
L2 Cache 256KiB 4-Way
32B/cycle
To L3
Store Buffer & Forwarding (56 entries)
32B/cycle
Load Buffer (72 entries)
L1 Data Cache 32KiB 8-Way
Data TLB
Line Fill Buffers (LFB) (10 entries)
64B/cycle
Memory Subsystem

Běžící program

Centrální kouzelník
* umí stěhovat
* umí počítat
* (malá) paměť
* poslouchá na slovo

Paměť

Instrukce

Data

Operační systém

Běžící program
Běžící program
Běžící program
Běžící program

Centrální kouzelník
* umí stěhovat
* umí počítat
* (malá) paměť
* poslouchá na slovo

Paměť Instrukce Data

Paměť Instrukce Data

Paměť Instrukce Data

Paměť Instrukce Data

• problémy: plýtvání, program a data jsou ukládány odlišně, musí překládat kvalifikovaná síla, ….

Řešení:

• Omezíme vzletnost jazyka pro zápis algoritmu, aby byl přeložitelný do otrokódu samotným otrokem.

• Místo zahazovacích tabulek použijeme jednu velkou tabuli s přepisovatelnými kolonkami na čísla

• Instrukce otrokódu budou čísla taktéž zapsaná na této tabuli…

Programovací jazyk

```
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.
```

Pozorování

- Žádné číslování kroků, jednoduchý text
- slova, čísla, symboly
- některá slova mají „shora“ daný význam: **program, var, begin, end, while, do, if, then, else,**

  kde jedině **var** není v anglickém slovníku (je to zkratka z variable)
- shůry je dáno i slovo **integer**. V zápisu `var a,b:integer` říká, že budu pracovat s celými čísly **a** a **b**
- symboly := se vyskytují pohromadě a očividně znamenají **přiřazení** hodnoty.
- `<>` `a` `>` jsou symboly pro porovnání (různé a větší) hodnot
- slovní popis **dokud jsou obě čísla různá, odečítáme menší od většího**, který očividně odpovídá řádkům

```
while a<>b do
  if b>a then b:=b-a else a:=a-b;
```

předchází nějaké protivné deklarace názvu programu a proměnných (proč to ten Wirth nějak nezkrátil?)

a (pochopitelnější) inicializace proměnných (někam ty čísla napsat musím)

a klíčové řádky pak následuje příkaz výstupu.

- proč je text řádku s **if** takový posunutý?
- Výklad: deklarace, příkazy

```
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.
```

Ilustrace (programujeme bez počítačového jazyka):

```
0001: BB CE 07 00 00      mov     r1,7CEh   ; 1998d
0006: B8 D4 09 00 00      mov     r0,9D4h   ; 2516d
000B: 3B C3               cmp     r0,r1
000D: 74 0E               je      001D
000F: 3B D8               cmp     r1,r0
0011: 7D 04               jge     0017
0013: 2B C3               sub     r0,r1
0015: EB 02               jmp     0019
0017: 2B D8               sub     r1,r0
0019: 3B C3               cmp     r0,r1
001B: 75 F2               jne     000F
001D: A1 94 00 00 00      mov     r0,output
0022: 8B D3               mov     r3,r1
0024: E8 57 F3 FF FF      call    WriteInt
```

Poznámky: Ta červená písmenka a číslice to je otrokód, vše ostatní jsou poznámky určené k tomu, aby čtenář viděl víc než jen BBCE070000B8D40900003BC3740E3BD87D042BC3EB022 ... Ta čísla úplně nalevo jsou pořadová čísla bytů kódu (tzv. adresy), prostřední sloupec obsahuje člověkem rozpoznatelné zkratky názvů operací otroka = procesoru, (**move**, **compare**, **j**ump if **e**qual, **sub**tract atp.) a pravý sloupec obsahuje konkrétní informace pro každou operaci, kde např. r0,r1,r3 jsou místa pro rychlé uložení několika málo údajů (registry). Při psaní prvních programů lidé museli vymýšlet přímo ta červená čísílka!!!

Paměť: BB CE 07 00 B8 D4 09 00 00 3B C3 74 0E ... FF FF

Je ten program správně?

• v 60-tých letech se informatici hodně zabývali možností dokázat správnost programu

• Umím dokázat, že můj důkaz, že program je správně, je správně?

• Umím dokázat, že můj důkaz, že můj důkaz, že program je správně, je správně, je správně?

• Nicméně, kromě způsobu jak to dokázat vymysleli také jak psát programy, aby to šlo dokázat

Metoda

• Obložili příkaz dvěma logickými výrazy:

{předběžná podmínka} **příkaz** {výsledná podmínka}

tak aby (co nejslabší) předběžná podmínka vedla po vykonání **příkazu** na výslednou podmínku.

$\{b \neq 0\}$ a:=a mod b; $\{\exists\, n :$ nové-$a + b\, n =$ původní-$a\}$

Výsledky

• velmi pracné i pro jednouché programy

• formulace doktríny strukturovaného programování (likvidace **goto** :-)

Náznak metody důkazu správnosti programu

{ nechť $a>0$ stejně tak jako $b>0$ }

```
while a<>b do begin
```

{ vím, že $a \neq b$ a že příkaz prováděný v cyklu zachová vstupní podmínku $a>0 \wedge b>0$ a nezmění NSD}

```
if b>a then
```

{ jsem-li tady, pak platí, že $b>a$ }

```
b:=b-a
```

{ z předpokladu $b>a$ a provedené operace $b:=b-a$ vyplývá, že změněné $b>0$ má s a tentýž NSD}

```
else
```

{ jsem-li tady, pak platí, že $a>b$ }

```
a:=a-b;
```

{ z předpokladu $a>b$ a provedené operace $a:=a-b$ vyplývá, že změněné $a>0$ má s b tentýž NSD}

{ teď nevím, jestli $a \neq b$ ale stále platí $a>0 \wedge b>0$ a nezměnil se NSD}

```
end;
```

{ jsem-li zde, vím, že $a=b \neq 0$ a je jejich hodnota je rovna NSD původních, protože ani jeden z provedených příkazů nezměnil hodnotu NSD.}

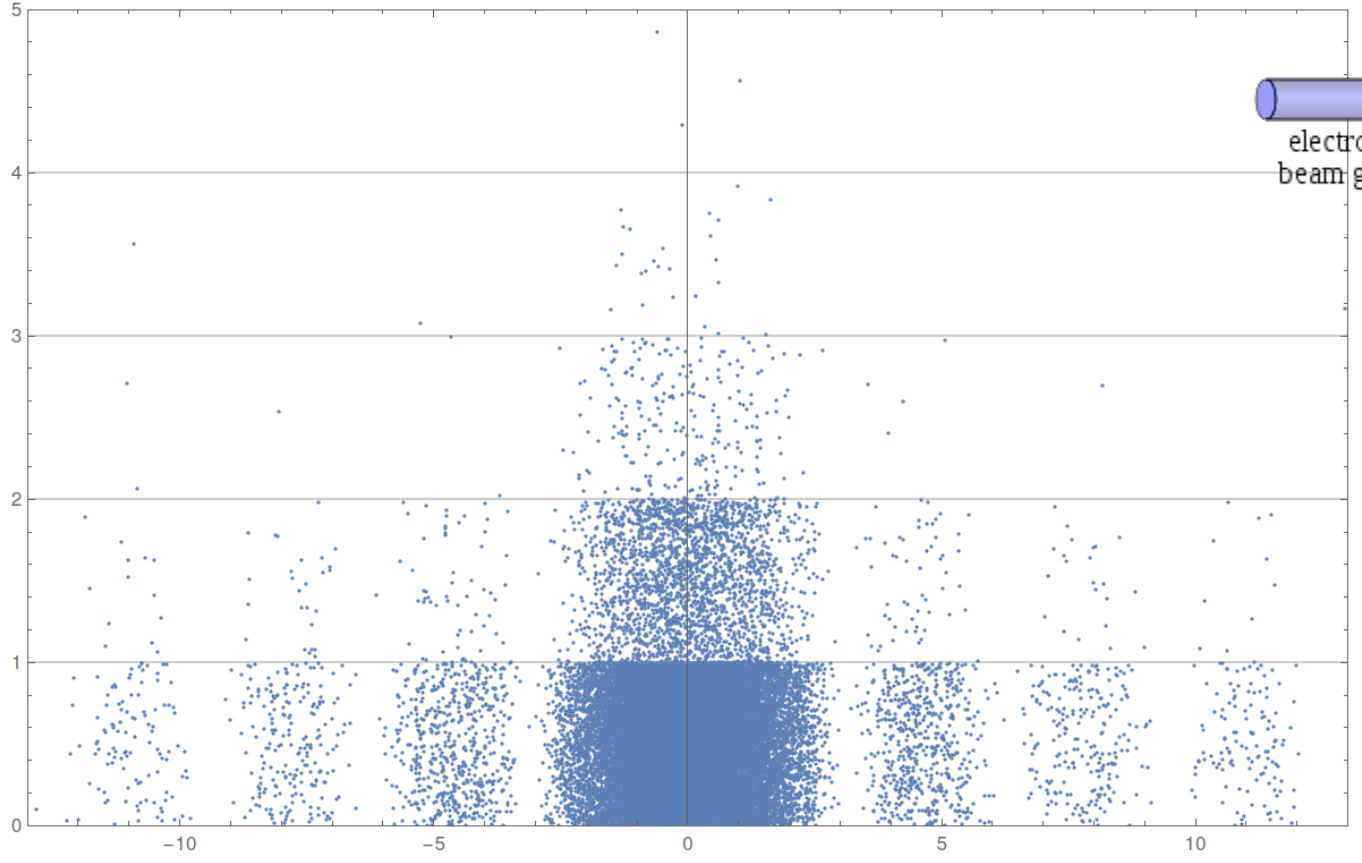
- dokázali jsme, že pokud to skončí, výsledek je NSD
- dokonce lze dokázat, že program skončí

Pascal ? !

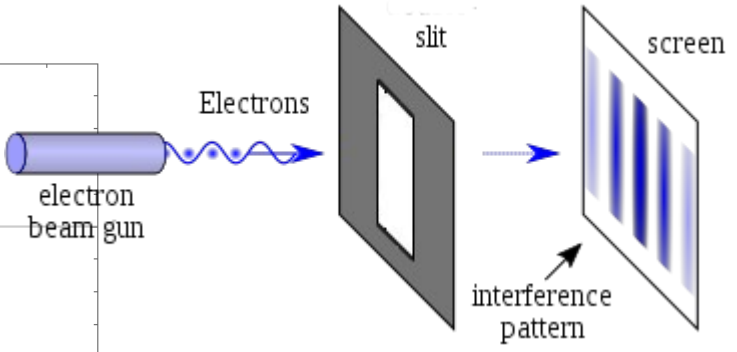
- Ada / ALGOL / Alice / APL / **Assembly Language** / Awk / BASIC / **C** / **C++** / **C#** / COBOL / D / Erlang / F# / FORTH / **FORTRAN** / Go / Haskell / **IDL** / **Java** / **Javascript** / **LabVIEW** / Lisp / Logo / Modula-3 / Objective-C / OCaml / **Pascal** / **Perl** / **PHP** / PL/I / SQL / **PostScript** / PROLOG / **Python** / R / **RegEx** / Ruby / Scala / Simula / Smalltalk / SNOBOL / SQL / Swift / Unix Shell / **Wolfram Mathematica**

```
kandidati = Table[{RandomReal[{-13, 13}], RandomReal[4]}, 500000];
prosli = Select[kandidati, Sinc[#[[1]]]^2 > RandomReal[]*10^Floor[#[[2]]] &];
ListPlot[prosli, PlotRange -> {{-13, 13}, {0, 4}}, Frame -> True, GridLines -> {None, Automatic}]
```

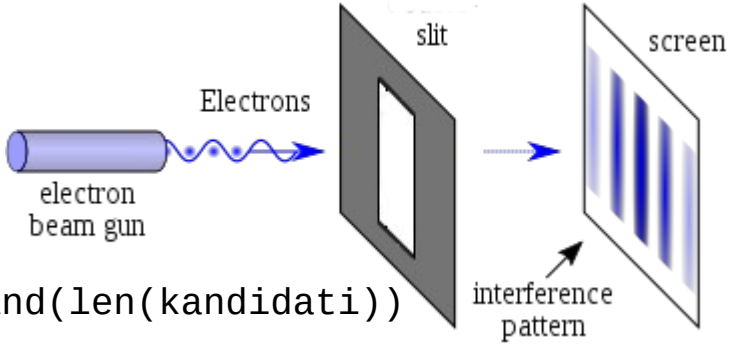
```
kandidati = Table[ {RandomReal[{-13, 13}], RandomReal[5]}, 1000000];
prosli = Select[ kandidati, Sinc[#[[1]]]^2 > 10^Floor[#[[2]]] RandomReal[] &];
ListPlot[ prosli, PlotRange → {{-13, 13}, {0, 5}}, Frame → True, GridLines → {None, Automatic}]
```

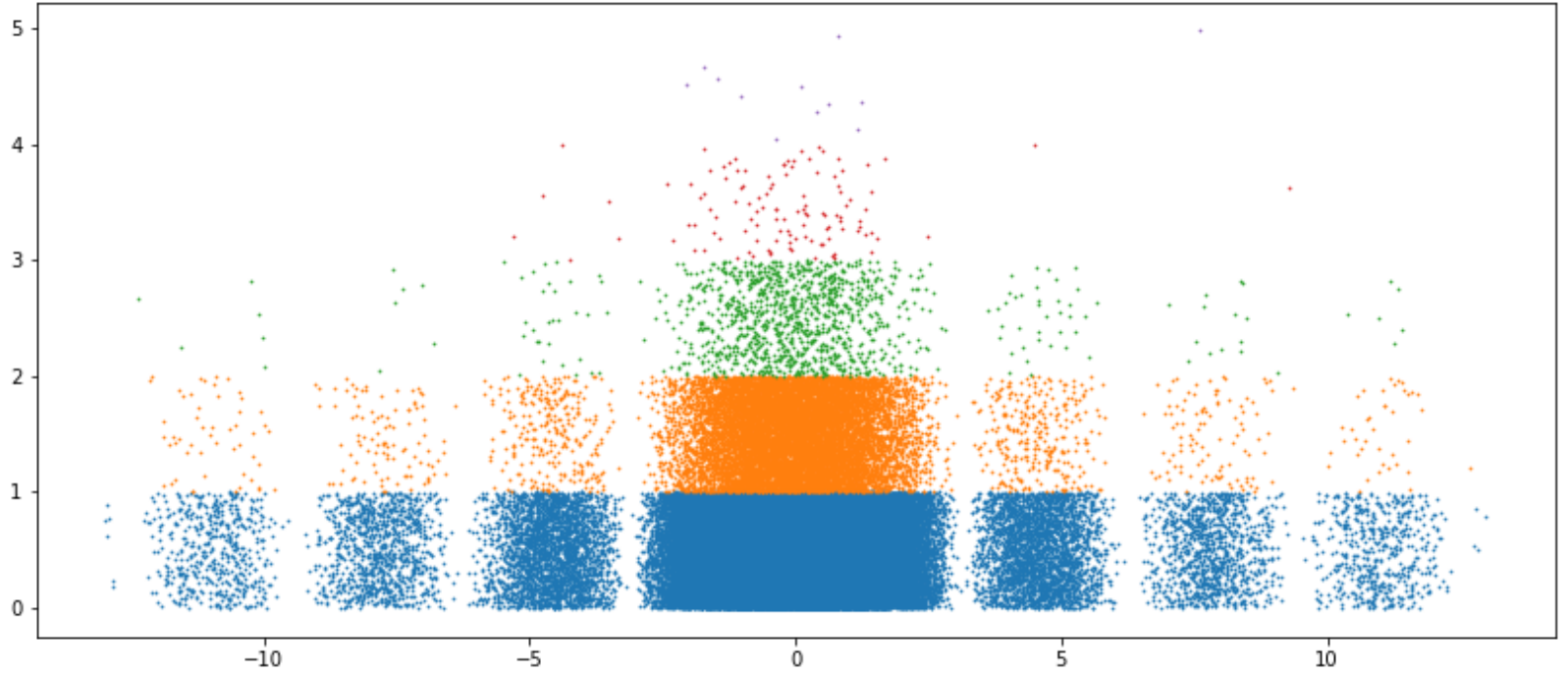


interferenční experiment



```
import numpy as np
import matplotlib.pyplot as plt

for k in range(5):
    kandidati = np.random.uniform(-13,13,10**(6-k))
    vybrat = np.sinc(kandidati/np.pi)**2 > np.random.rand(len(kandidati))
    prosli = kandidati[ vybrat ]
    y = np.random.rand(len(prosli))+k
    plt.plot(prosli,y,'.',ms=1.5)

plt.show()
```

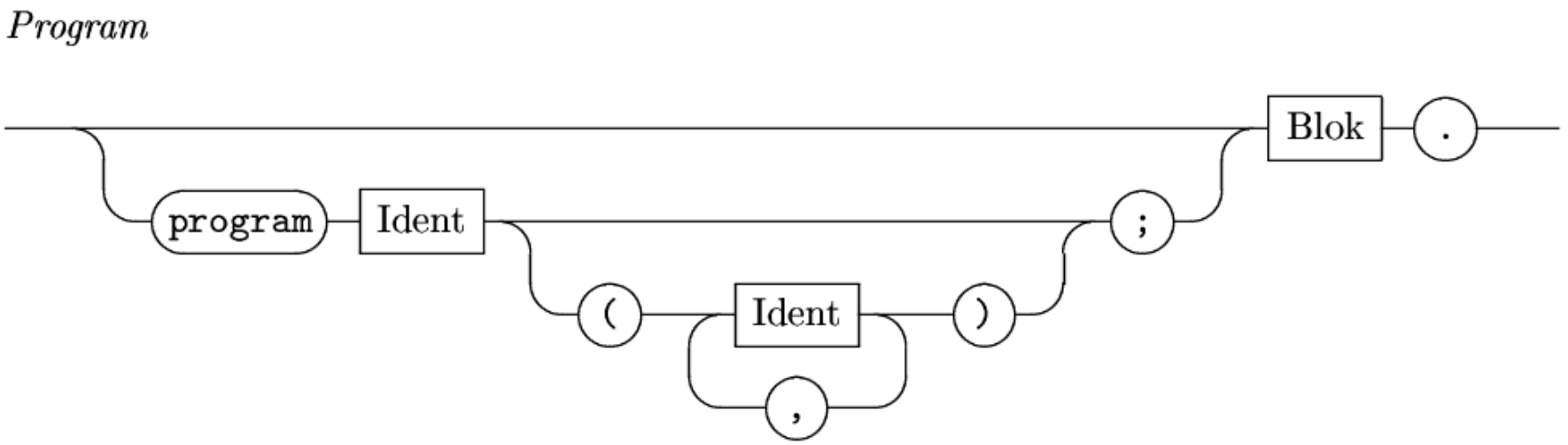




Píšeme program

- Ukázka psaní a spuštění krátkého programu
- Jaké druhy souborů se při tom objeví
- (viz též http://utf.mff.cuni.cz/~ledvinka/?196864)
-

Začneme kolejištěm pro pascalovský program:



Protože je uvozovací část nepovinná, je správně jak následující program

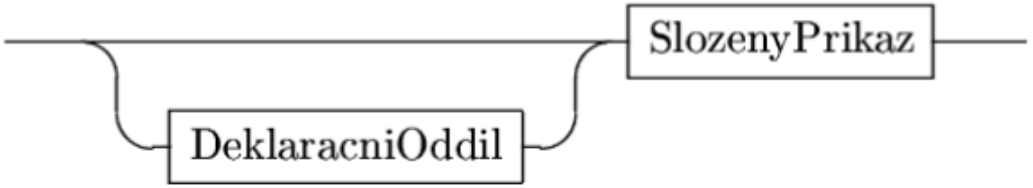
```
program SHlavickou;
begin
  writeln('Jsem spravny program!');
end.
```

tak i tento kratký

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

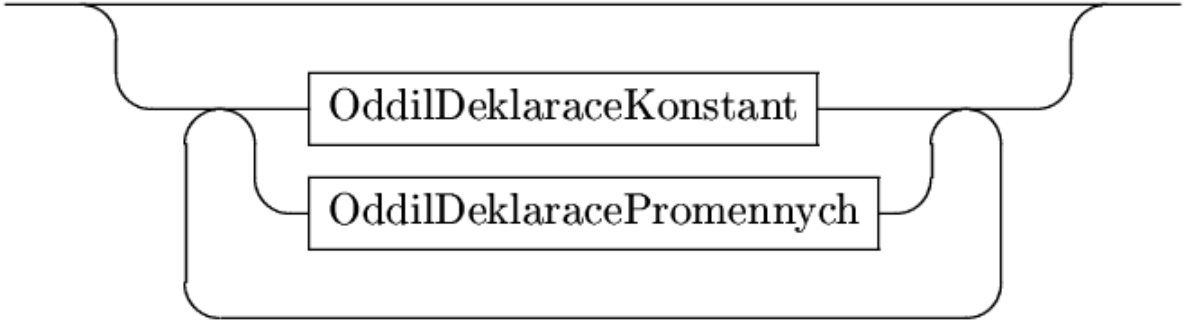
Předchozí programy byly těmi nejjednoduššími, jaké si lze představit. Neobsahují žádnou deklaraci a ten druhý se skládá pouze ze složeného příkazu. Tento složený příkaz ale bývá předcházen deklaračním oddílem, který jak později uvidíme, tvoří těžiště struktrurovaného programu.

Blok



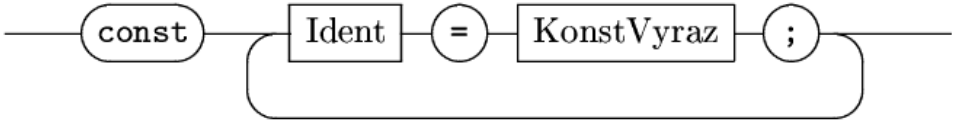
V naší zatím velmi zjednodušené verzi Pascalu budeme uvažovat pouze *proměnnéakonstanty* a tak deklarační oddíl popisuje následující "kolejiště":

DeklaracniOddil



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

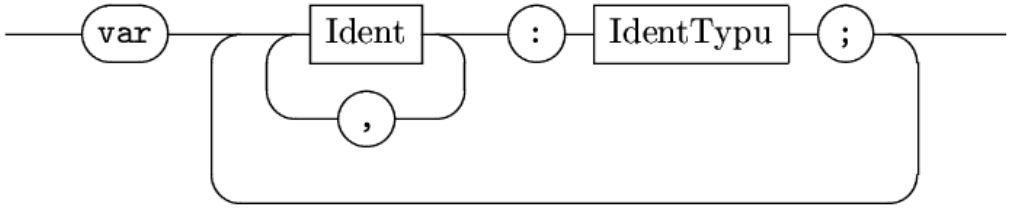
OddilDeklaraceKonstant




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

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

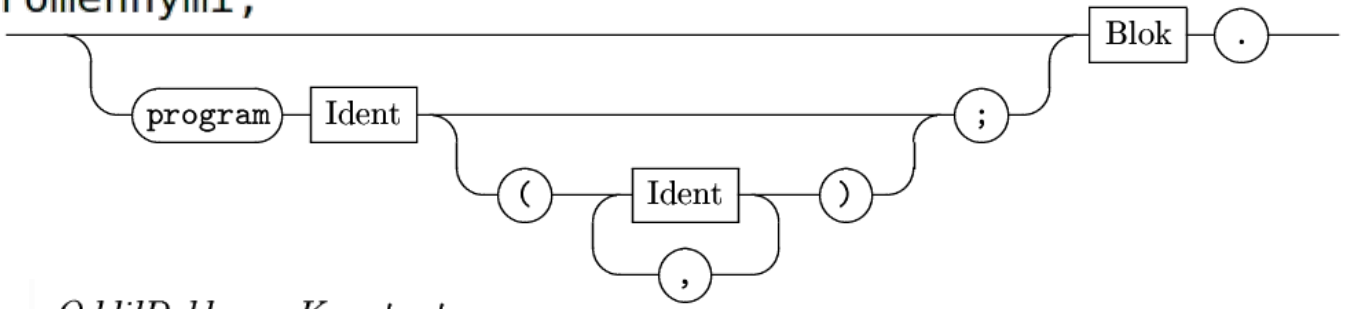
OddilDeklaraceProměnných



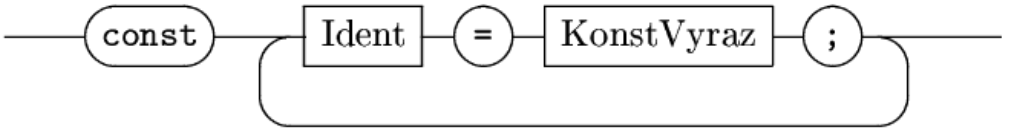
Identifikátory proměnných by měly stručně napovídat, co jsou proměnné zač. Při řešení jednoduchých úloh vystačíme ale s konvencí z hodin matematiky. Indentifikátory typu jsou prozatím shůry dány tyto:

- **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

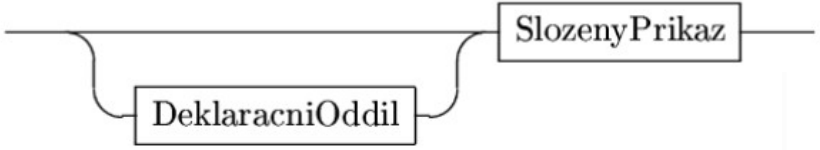
```
program SKonstantouADvemaPromennymi;
const N = 10;
var   i,s : integer;
begin
  i:=1;
  s:=0;
  while i<=N do
  begin
    s:=s+i;
    i:=i+1;
  end;
  writeln('Soucet cisel od 1 do ',N,' je ',s);
end.
```



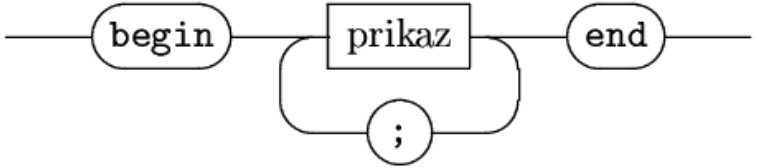
OddilDeklaraceKonstant



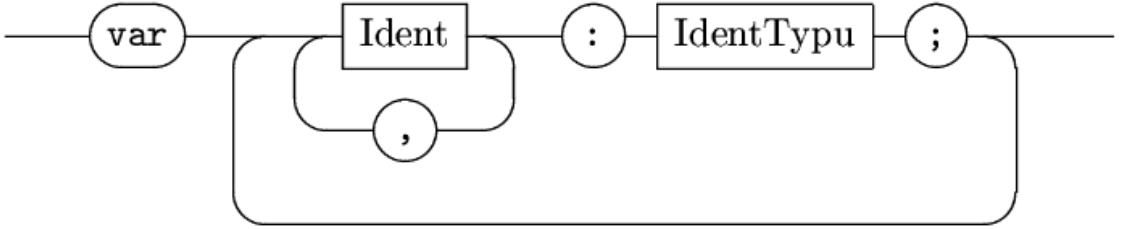
Blok



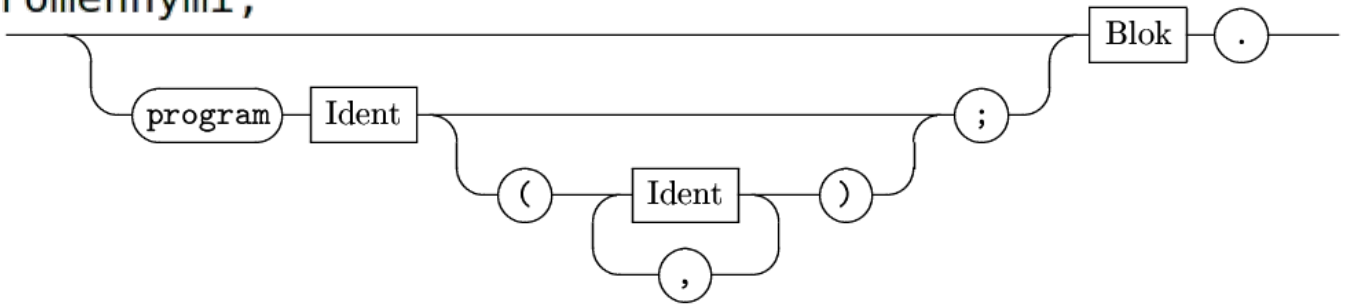
SlozenyPrikaz



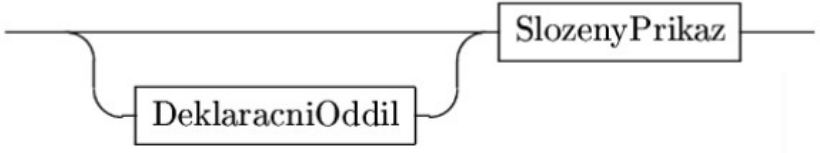
OddilDeklaracePromennych



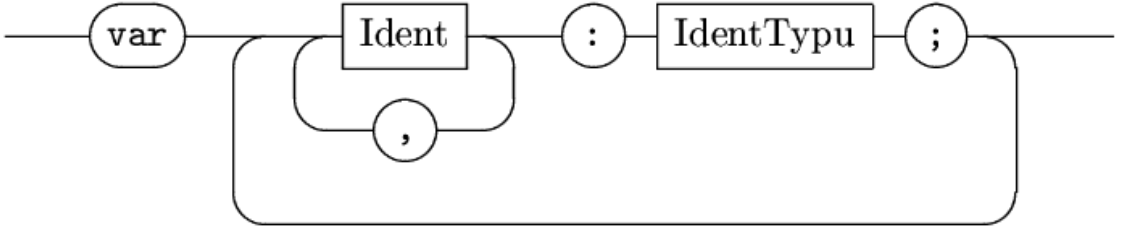
```
program SKonstantouADvemaPromennymi;
const N = 10;
var   i,s : integer;
begin
  i:=1;
  s:=0;
  while i<=N do
  begin
    s:=s+i;
    i:=i+1;
  end;
  writeln('Soucet cisel od 1 do ',N,' je ',s);
end.
```



Blok

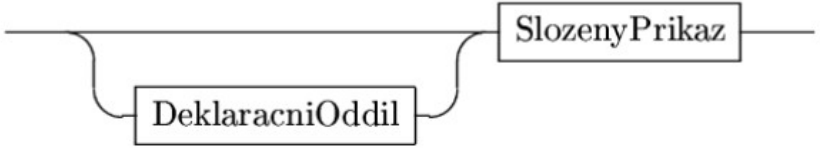


OddilDeklaracePromennych

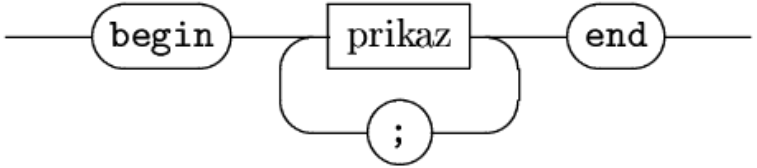


```
program SKonstantouADvemaPromennymi;
const N = 10;
var   i,s : integer;
begin
  i:=1;
  s:=0;
  while i<=N do
  begin
    s:=s+i;
    i:=i+1;
  end;
  writeln('Soucet cisel od 1 do ',N,' je ',s):
end.
```

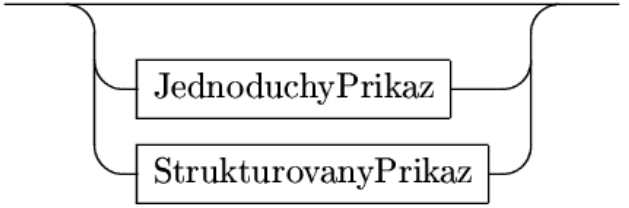
Blok



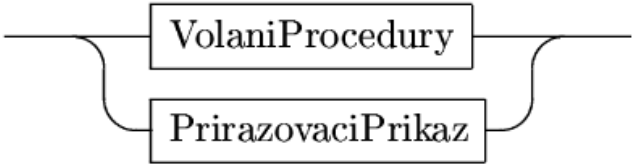
SlozenyPrikaz



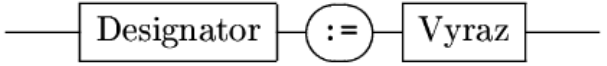
Prikaz



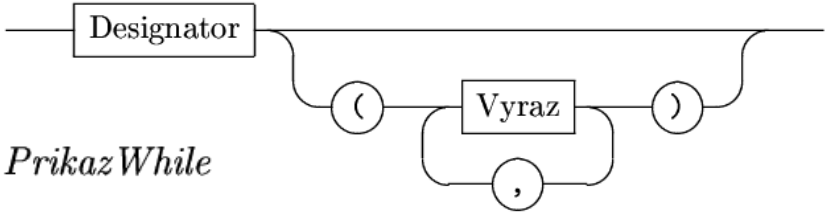
JednoduchyPrikaz



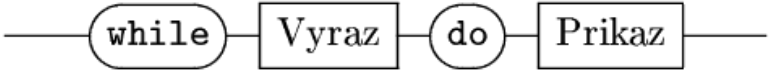
PrirazovaciPrikaz

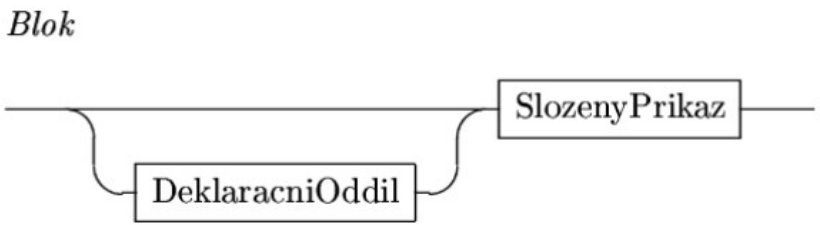


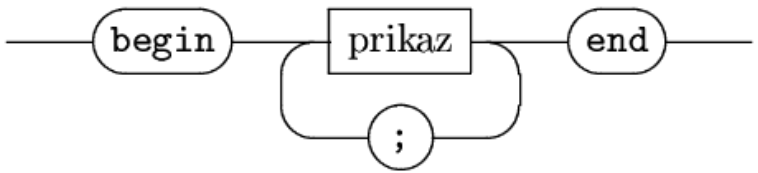
VolaniProcedury

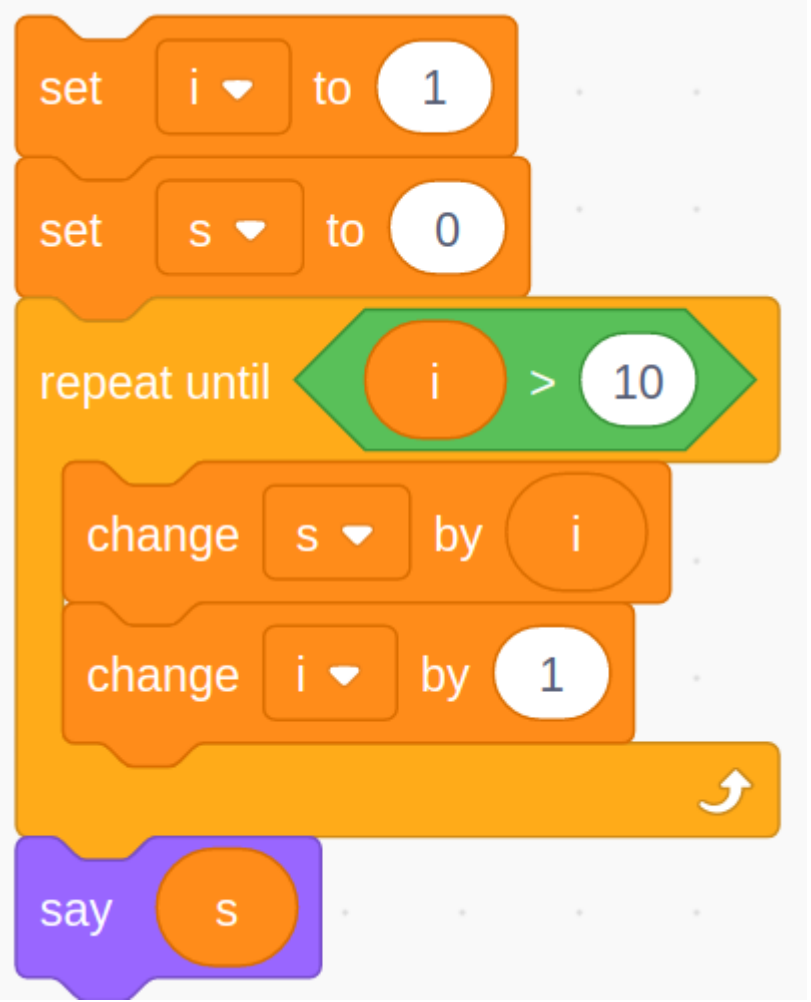


PrikazWhile

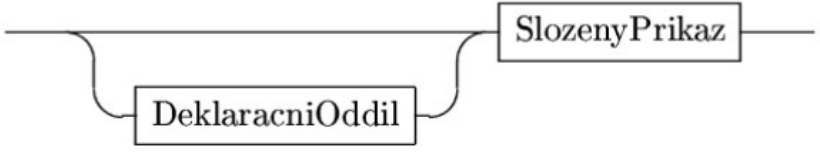



Blok
SlozenyPrikaz
DeklaracniOddil

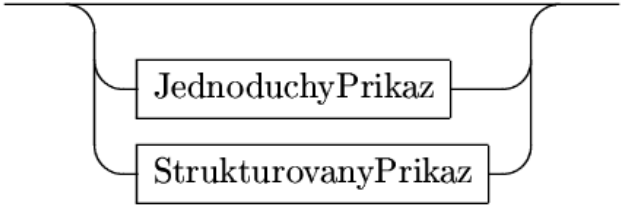

SlozenyPrikaz
begin
prikaz
end
;


set i to 1
set s to 0
repeat until i > 10
change s by i
change i by 1
say s

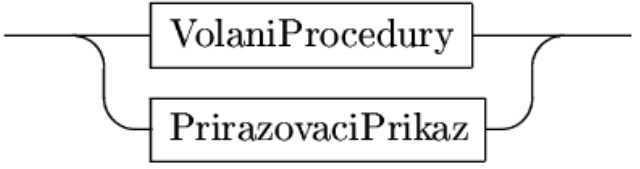
Blok



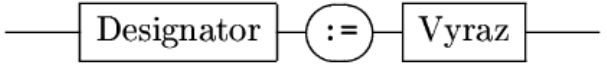
Prikaz



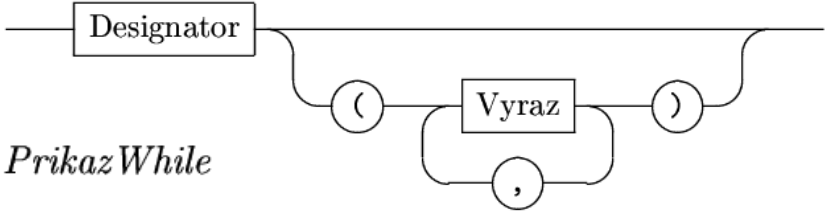
JednoduchyPrikaz



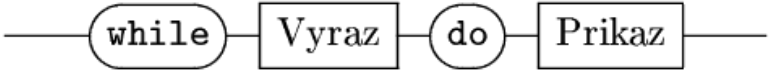
PrirazovaciPrikaz



VolaniProcedury

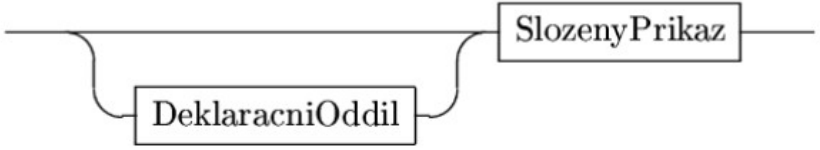


PrikazWhile

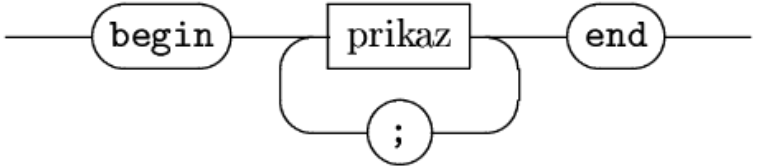


```
program SKonstantouADvemaPromennymi;
const N = 10;
var   i,s : integer;
begin
  i:=1;
  s:=0;
  while i<=N do
  begin
    s:=s+i;
    i:=i+1;
  end;
  writeln('Soucet cisel od 1 do ',N,' je ',s):
end.
```

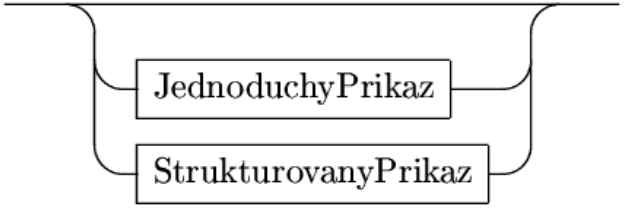
Blok



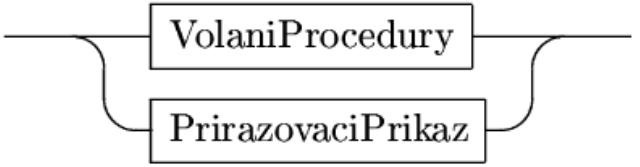
SlozenyPrikaz



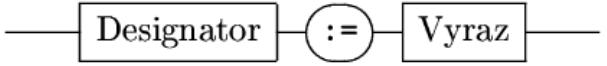
Prikaz



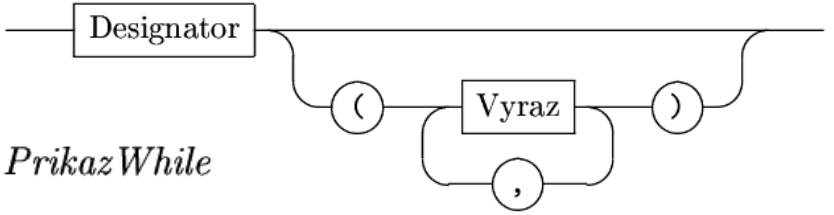
JednoduchyPrikaz



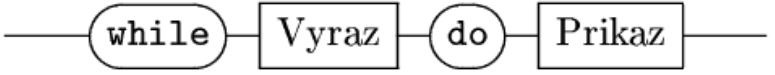
PrirazovaciPrikaz



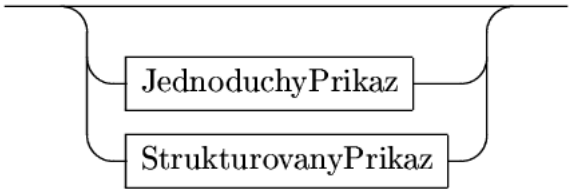
VolaniProcedury



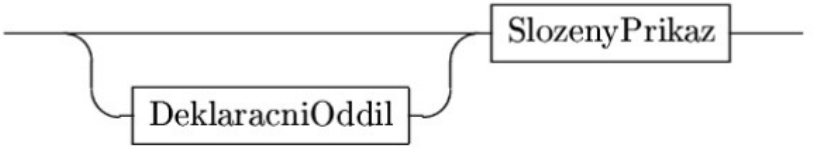
PrikazWhile



Prikaz



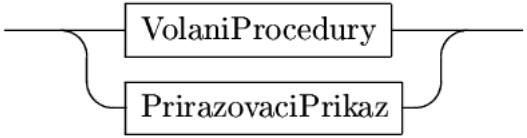
Blok



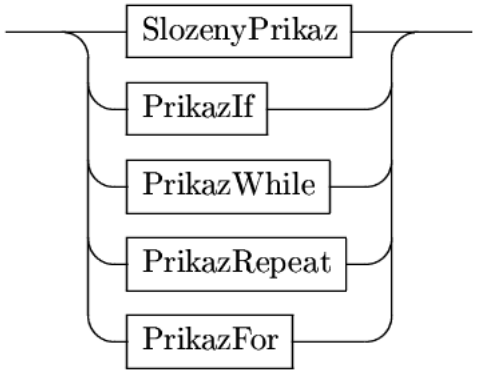
Za prvé, jak vidíme, nic (prázdný příkaz) je také příkaz.

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

JednoduchyPrikaz

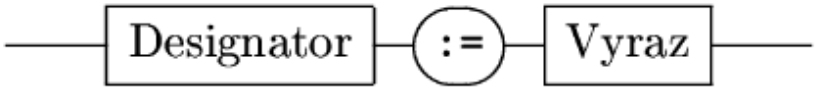


StrukturovanyPrikaz

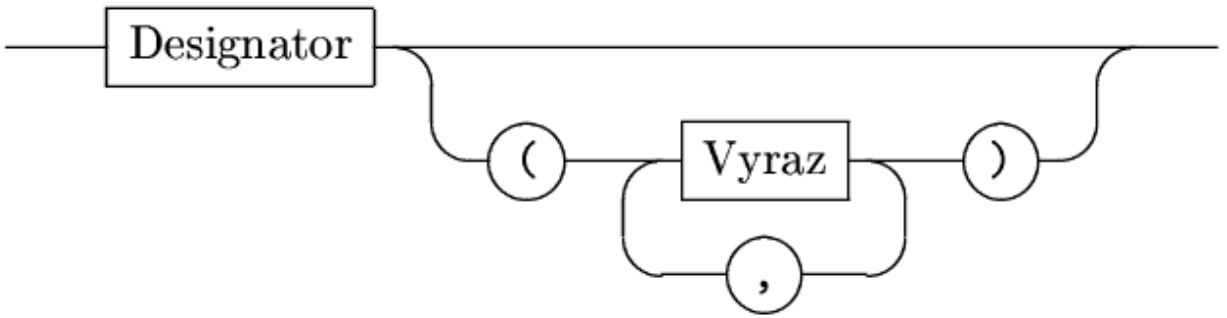


Jednoduché příkazy

PrirazovaciPrikaz



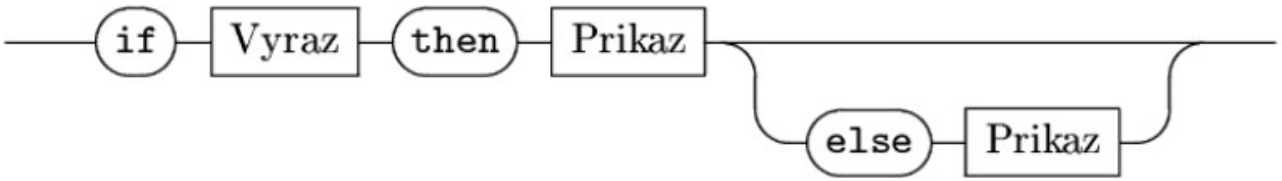
VolaniProcedury



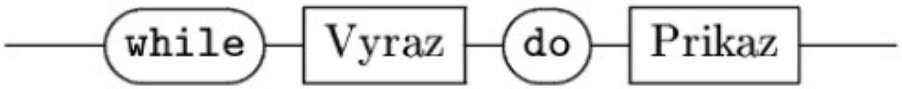
Ještě nejméně týden pro nás bude **designátor** totožný s **identifikátorem**, až se dozvíme, že jsou i další prvky jazyka, hodí se vědět, kde jazyk vyžaduje identifikátor, a kde můžeme použít "něco obecnějšího".
Obě uvedené formy jednoduchého příkazu ilustrují následující dva řádky:

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

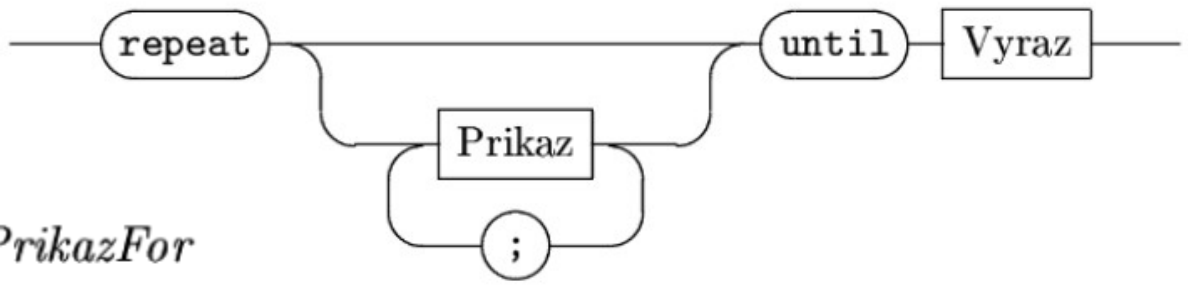
PrikazIf



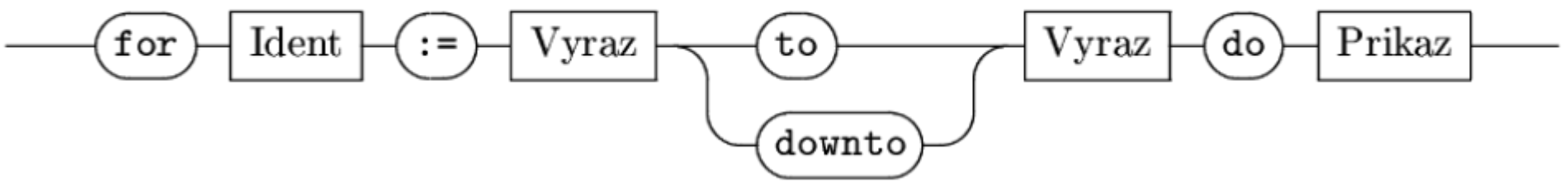
PrikazWhile



PrikazRepeat

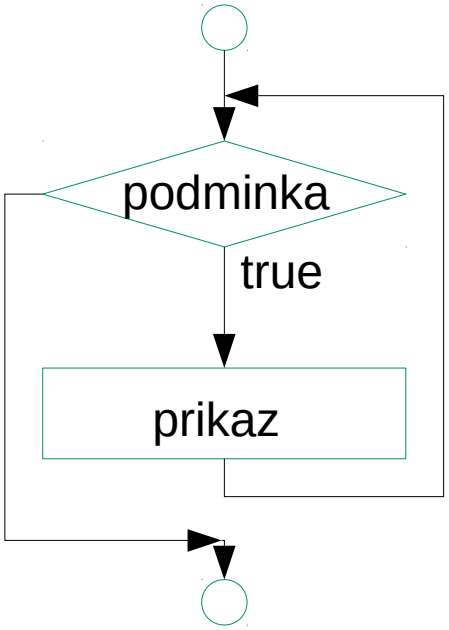


PrikazFor

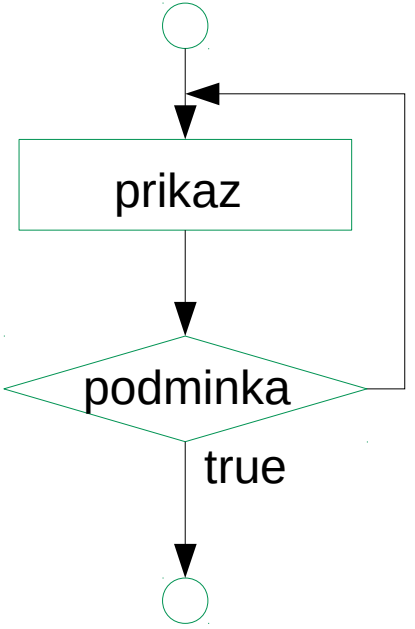


While/Repeat

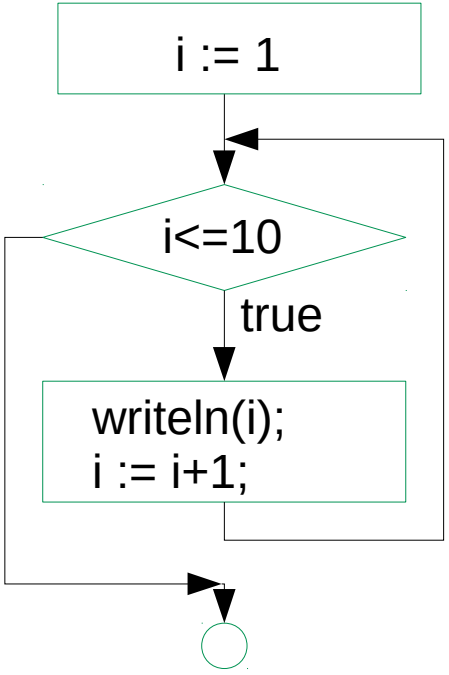
while podminka do prikaz



repeat prikaz until podminka

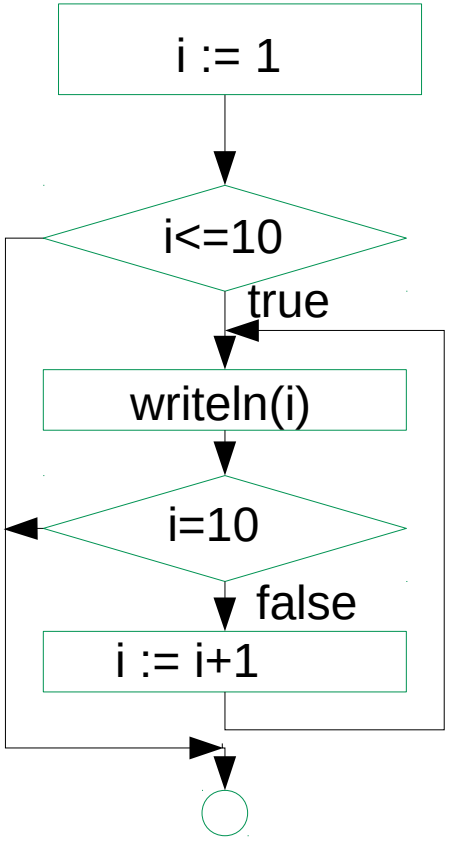


```
i := 1;
while i <= 10 do begin
   writeln(i);
   i := i+1;
end;
```



for

for i := 1 **to** 10 **do** writeln(i)



Vnořené cykly

$$1^3 + 5^3 + 3^3 = 153$$

```
for i := 1 to 9 do
  for j := 0 to 9 do
    for k := 0 to 9 do
      if ... then
```

```
for a := 100 to 999 do begin
    i := ...
    j := ...
    k:= ...
    if ... then
end;
```

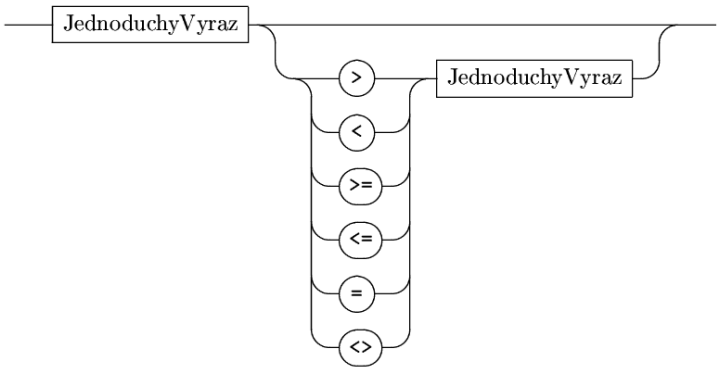
Vnořené cykly

$$1^3 + 5^3 + 3^3 = 153$$

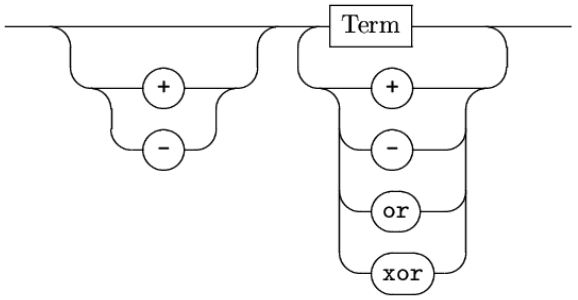
```
for i := 1 to 9 do
  for j := 0 to 9 do
    for k := 0 to 9 do
      if ... then
```

```
for a := 100 to 999 do begin
    i := ...
    j := ...
    k:= ...
    if ... then
end;
```

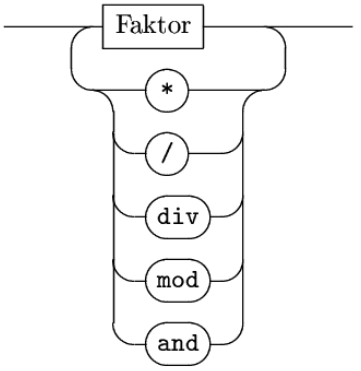
Vyraz



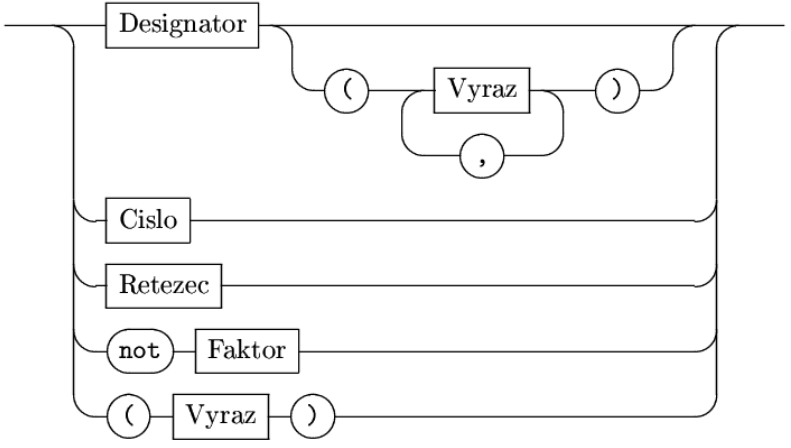
JednoduchyVyraz



Term



Faktor



2*i+1

(x+1)/(x-1)

sin(2*x)

sin(2x)

Typy a aritmetické výrazy

Čím se liší tyto výrazy?

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

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

	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

Typy a aritmetické výrazy

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

	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

Celočíselné výsledky funkcí

`abs(i), sqr(i), trunc(x),round(x), math.`

Reálné výsledky funkcí a operací

`(i/j), abs(x), sqr(x), sqrt(i), sin(j), ...`

Logické výrazy

```
var  i,j : integer;
     x,y : real;
     a,b : boolean;
```

Jednoduché logické výrazy

```
i=j, x>0, i<>0, j<=10, ...
```

Logické funkce

```
odd(i), even(j), math.InRange(x,0,1),
math.SameValue(x,y)
```

Složitější logické výrazy

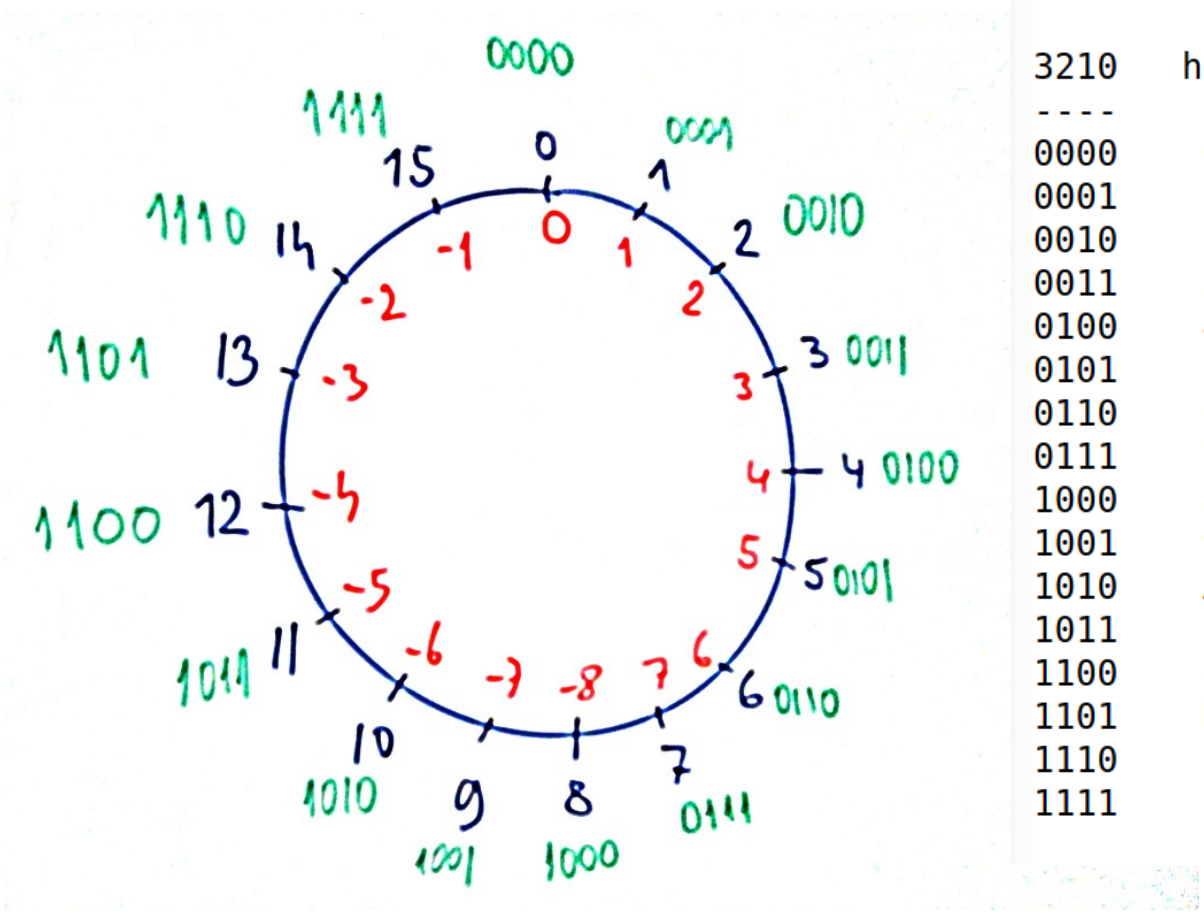
```
(i=j) and (x>0), (j<=10) or (j>=20),
(i=1) or (i=10) or (i=100) or (i=1000)
```

Celá čísla v počítači

```
Z = A*B

        1 0 1 1   (A)
      × 1 0 1 0   (B)
      ---------
        0 0 0 0
 +      1 0 1 1
 +    0 0 0 0
 +  1 0 1 1
 ---------------
 = 1 1 0 1 1 1 0
```

Identif.Typu	Rozsah	Formát ulo▫ení
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

Celá čísla v počítači



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

Logické operace s celými čísly

```
var x,y : Integer; ....     x := 21;
    y := 12;  { nyní 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    }     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 }     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 }     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 );
```

Reálná čísla v počítači

* Do žádného počítače se nevejde ani jedno *opravdové* reálné číslo nezaokrouhlené

* Nezbytně reálná čísla potřebujeme

* Musíme tolerovat, že poslední cifry se *ztrácejí* (typ real má cca 15 cifer)
  1.000 000 000 000 000 + 0.000 000 000 000 000 1 → 1.000 000 000 000 000

* Opakovanými operacemi se chyba může rozšířit (někdy i fatálě)

* Není rozumné testovat reálná čísla na rovnost, cykly v reálné proměnné nahrazovat celými čísly,

* Zapisujeme např. **const** G_SI =6.67408E-11;

* Omezení délky mantisy počtu cifer a rozsahu exponentu

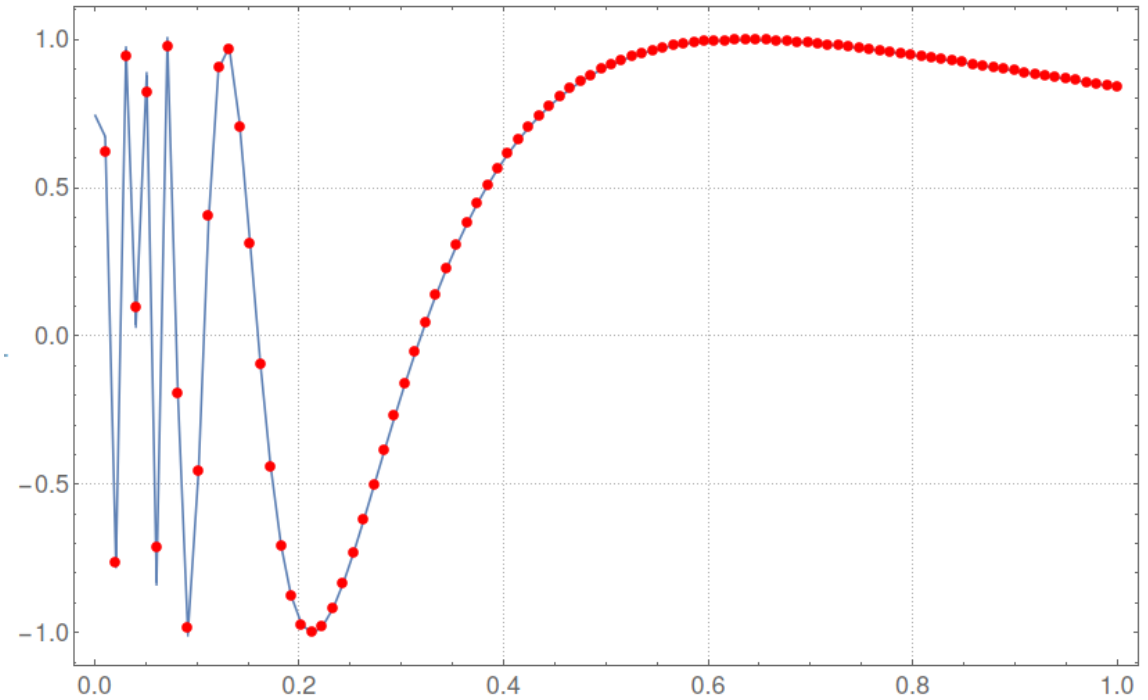
Spočíst a nakreslit

- graf funkce
- graf funkce 2 proměnných
- Ilustrace náhodného procesu
- přesměrování vstupu a výstupu

(nástroj na vytváření obrázků:`http://gnuplot.info`)

Spočíst a nakreslit

- graf funkce sin(1/x)



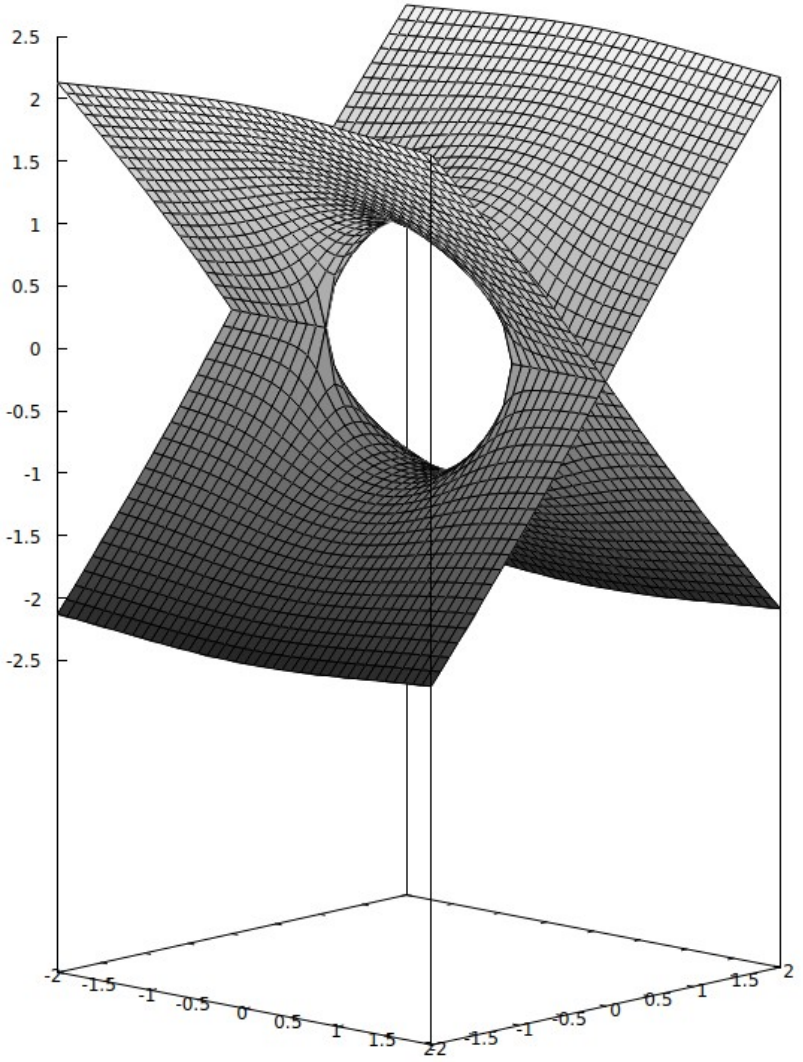
Spočíst a nakreslit

- graf funkce 2 proměnných

$$f(z) = \sqrt{z^2 - 1}$$

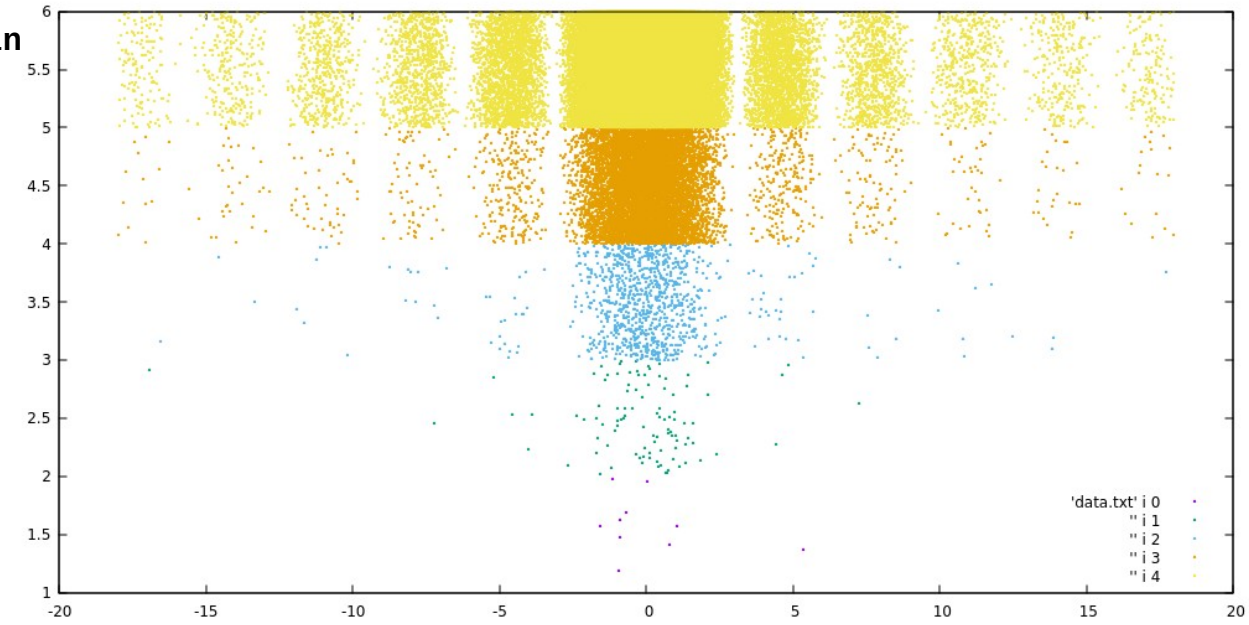
$$\sqrt{x + iy} = \sqrt{\frac{r + x}{2}} \pm i\sqrt{\frac{r - x}{2}}$$

$$x, y \in \mathbb{R},\ r = \sqrt{x^2 + y^2}$$



```
program stinitko;

const a = -18;
      b = +18;

var x,y : real;
    n,p : integer;
    deset_na_p: real;
begin
  p := 1;
  deset_na_p := 10;
  n := 0;

  repeat
     x := a+(b-a)*random;

     if sqr( sin(x) ) > random*x*x then begin
       writeln( x, ' ', random+p);
       n := n+1;
     end;

     if n = deset_na_p then begin;
       writeln;
       writeln;
       p := p+1;
       n := 0;
       deset_na_p:=deset_na_p*10;
     end;
  until p>5;
end.
```

```
C:\user> stinitko > data.txt
C:\user> gnuplot
        G N U P L O T
        Version 5.2
. . . . .
gnuplot> set pointsize 0.3
gnuplot> set key bottom
gnuplot> plot 'data.txt' i 0,'' i 1,'' i 2,'' i 3,'' i 4
gnuplot>
```



Procedury a funkce

- Co když Pascal „neumí“ nějakou funkci?
- Co když délka kódu příliš roste?
- Co když opakovaně potřebuji tentýž vztah?

=> Napišme novou funkci

[Satrapa] 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.

Euklidův algoritmus jako funkce

Jako příklad budiž zmíněn opět Euklidův algoritmus. Co kdybychom chtěli spočíst nějvětší 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.

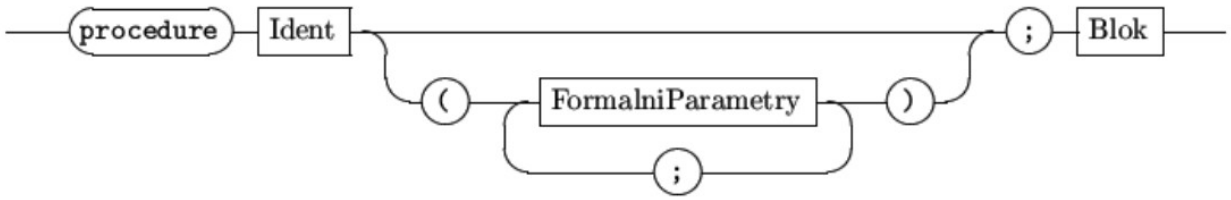
```
function  GCD(a,b : integer) : integer;
var  c:integer;
begin
   repeat
      c := amod b;
      a := b;
      b := c; { (a,b) := (b,a mod b); }
  until b=0;
   GCD := a;
end ;  {function GCD}
```

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:

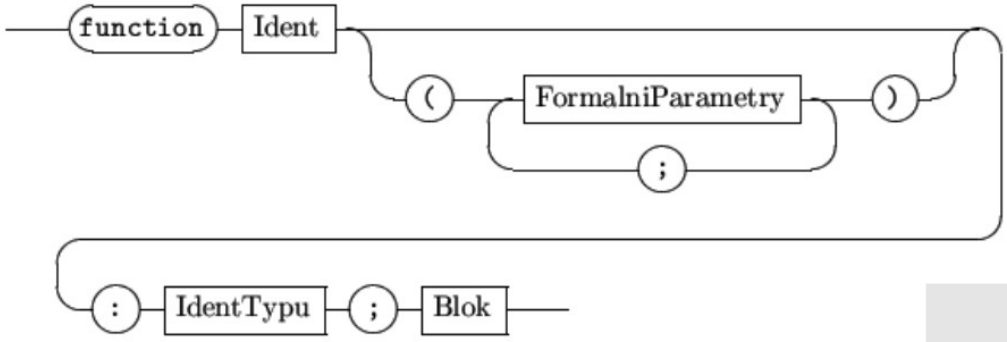
```
n := GCD(44,55) ;
```

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.

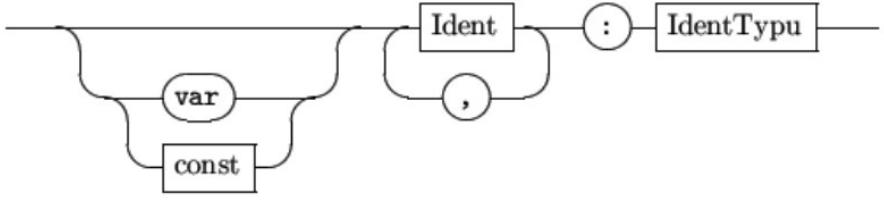
DeklaraceProcedury



DeklaraceFunkce



FormalniParametry




```
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}
```

```
unit output;

interface


implementation

// Type your code here, or load an example.

function myF(const num: real): real;
begin
    myF := sin(num);
end;

function soucet(const num: real): real;
begin
    soucet := myF(num)+myF(num+1)+myF(num+2)+myF(num+3);
end;


end.

```

A Output... Filter... Libraries Add new... Add tool...

```
DEBUGSTART_$OUTPUT:

myf(real):
        pushq   %rbp
        movq    %rsp,%rbp
        leaq    -32(%rsp),%rsp
        movsd   %xmm0,-8(%rbp)
        fldl    -8(%rbp)
        fsin
        fstpl   -16(%rbp)
        movsd   -16(%rbp),%xmm0
        movq    %rbp,%rsp
        popq    %rbp
        ret

soucet(real):
        pushq   %rbp
        movq    %rsp,%rbp
        leaq    -64(%rsp),%rsp
        movsd   %xmm0,-8(%rbp)
        movsd   -8(%rbp),%xmm0
        call    myf(real)
        movsd   %xmm0,-40(%rbp)
        movsd   -8(%rbp),%xmm0
        addsd   _$OUTPUT$_Ld1,%xmm0
        call    myf(real)
        movsd   %xmm0,-24(%rbp)
        movsd   -24(%rbp),%xmm0
        addsd   -40(%rbp),%xmm0
        movsd   %xmm0,-24(%rbp)
        movsd   -8(%rbp),%xmm0
        addsd   _$OUTPUT$_Ld2,%xmm0
        call    myf(real)
        movsd   %xmm0,-32(%rbp)
        movsd   -32(%rbp),%xmm0
        addsd   -24(%rbp),%xmm0
        movsd   %xmm0,-32(%rbp)
        movsd   -8(%rbp),%xmm0
        addsd   _$OUTPUT$_Ld3,%xmm0
        call    myf(real)
        addsd   -32(%rbp),%xmm0
```

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 .
```

Inicializace lok. proměnných

```
program lokalni;

function f(i : integer):integer;
var j:integer;
begin
  f := j;
end;

begin
  Writeln(f(1));
  Writeln(f(1));
end.
```

Předávání parametrů: Globální promměné

```
program  ProcSGlobProm;

var  i;

procedure  MojeProc;
begin
    i:=0;
end ;

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

Předávání parametrů: Hodnotou

```
program  ProcSParHodnotou;

var  i;

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

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

Předávání parametrů: Odkazem

```
program  ProcSParOdkazem;

var  i;

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

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

Předávání parametrů odkazem II.

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

```
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;
  ...
```

Jak napsat funkci

- Aby dělala co má
- Vhodný název
- Jasně určené vstupy
  ideálně: vhodně pojmenované parametry,
  občas nezbytné: globální proměnné jako argumenty
- Jasně daný způsob vracení výsledku
- Co ještě funkce dělá (side-effects)

```
// 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;
```

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 .
```

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:

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

```
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;
```

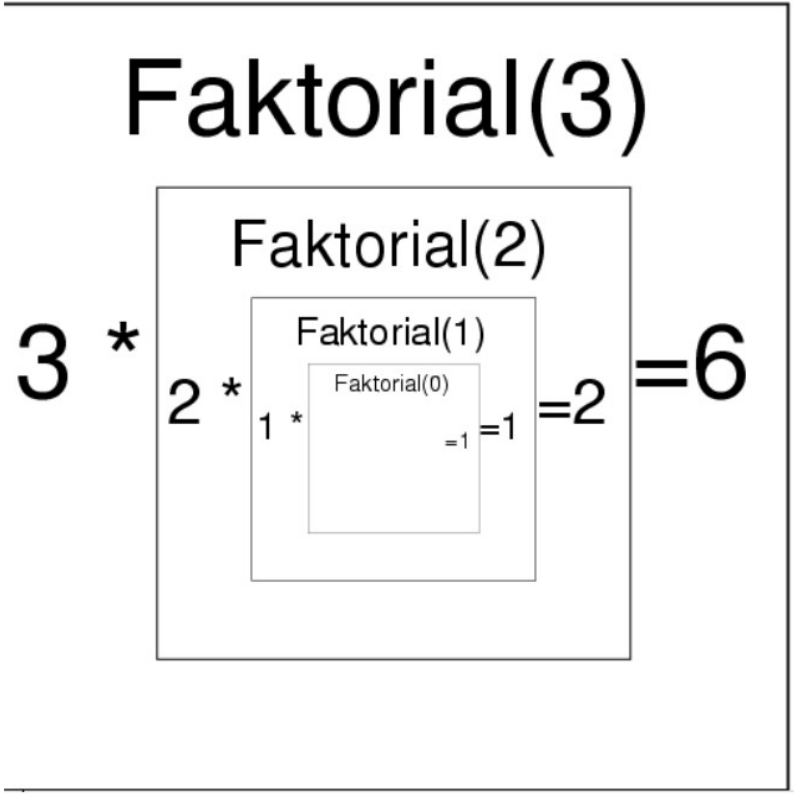
Procedury a funkce (pokrač.)

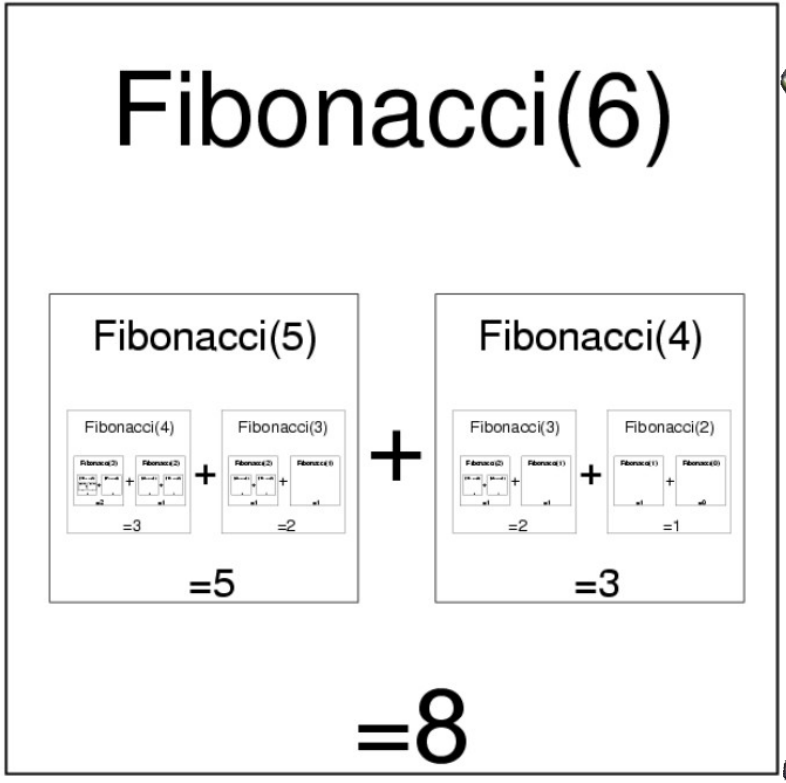
- Program lze dělit na podprogramy
  * vede to ke zjednodušení
  * jednodušší kód má větší šanci být správně
- Komunikace
  * prostřednictvím parametrů
    - předávání hodnotou → vstupní parametr
    - předávání odkazem → vstupní i výstupní parametr
  * návratové hodnoty (u funkcí)
  * globálních proměnných
- Procedury představují nové příkazy
- Funkce vystupují jako části výrazů

Procedury a funkce (pokrač.)

```
v := rychlost(q);

if v > maximalni_rychlost(teplota) then
                                    brzdi(max_dovolne_zrychleni);

while pocet_prvku(seznam) < kapacita do
        pridej(seznam, dalsi_nactena_hodnota(cidlo) );
```

Funkce a rekurze





Funkce a rekurze

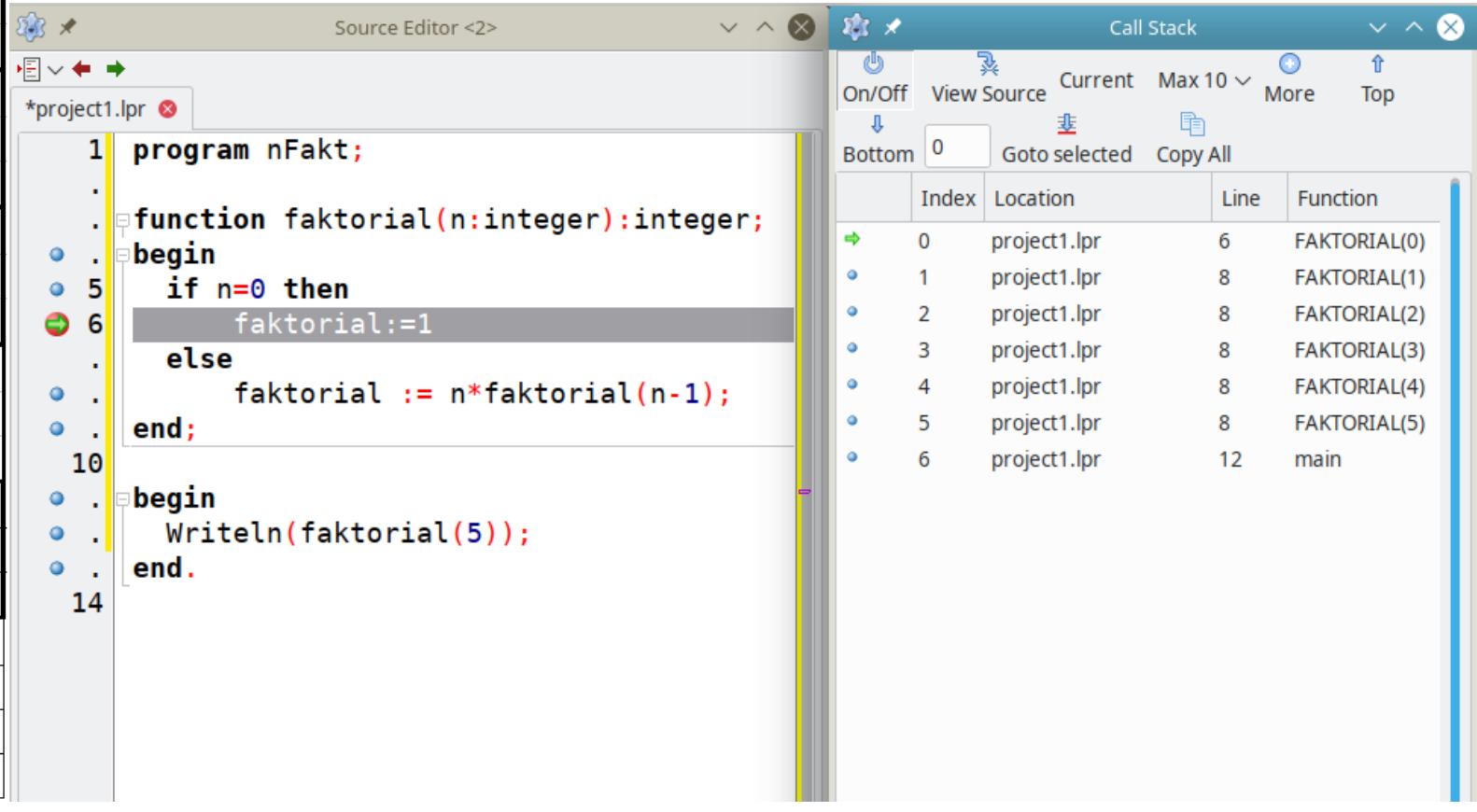
```
1  {$mode Delphi}
2  unit output;
3
4  interface
5
6  function faktorial(const num: Integer): Integer;
7
8  implementation
9
10 // Type your code here, or load an example.
11
12 function faktorial(const num: Integer): Integer;
13 begin
14     if num=0 then faktorial := 1
15     else faktorial := num * faktorial(num-1);
16 end;
17
18 end.
19
```

```
1  DEBUGSTART_$OUTPUT:
2
3  faktorial(longint):
4          pushq   %rbp
5          movq    %rsp,%rbp
6          leaq    -16(%rsp),%rsp
7          movl    %edi,-8(%rbp)
8          testl   %edi,%edi
9          jne     .Lj6
10         movl    $1,-12(%rbp)
11         jmp     .Lj7
12 .Lj6:
13         movl    -8(%rbp),%eax
14         leal    -1(%eax),%edi
15         call    faktorial(longint)
16         movl    -8(%rbp),%edx
17         imull   %edx,%eax
18         movl    %eax,-12(%rbp)
19 .Lj7:
20         movl    -12(%rbp),%eax
21         movq    %rbp,%rsp
22         popq    %rbp
23         ret
24
```

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

Zásobník volání

Význam	Hodnota
...	...
argument	5
návr. hodnota funkce	?
návr. adresa	Hlavní program.12
argument	4
návr. hodnota funkce	?
návr. adresa	Faktorial.8
argument	3
návr. hodnota funkce	?
návr. adresa	Faktorial.8
argument	2
návr. hodnota funkce	?
návr. adresa	Faktorial.8
argument	1
návr. hodnota funkce	?
návr. adresa	Faktorial.8
argument	0
návr. hodnota funkce	1
návr. adresa	Faktorial.8
(nepoužívá se)	
(nepoužívá se)	
(nepoužívá se)	
(nepoužívá se)	

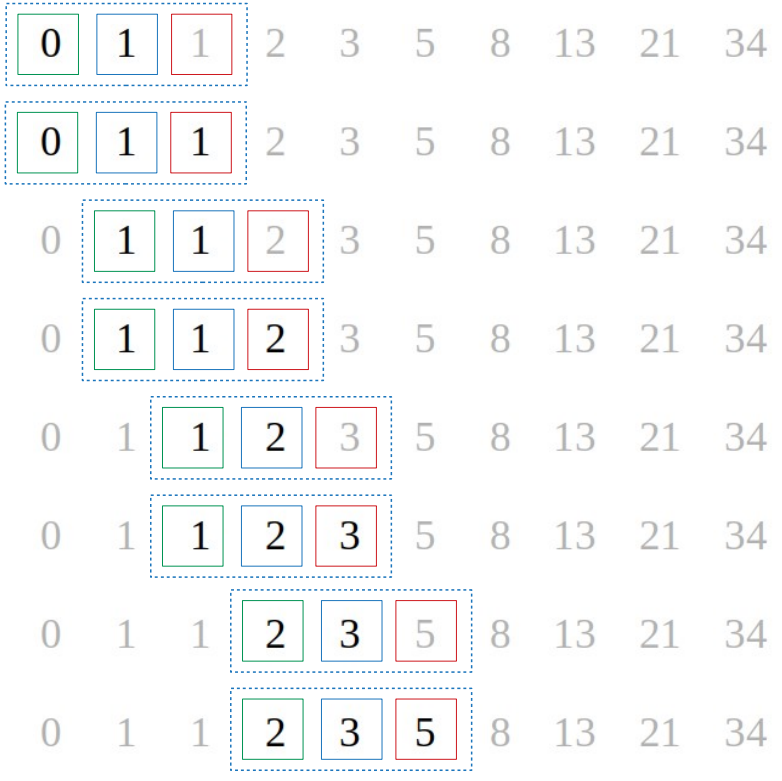


Obejdeme se bez rekurze?

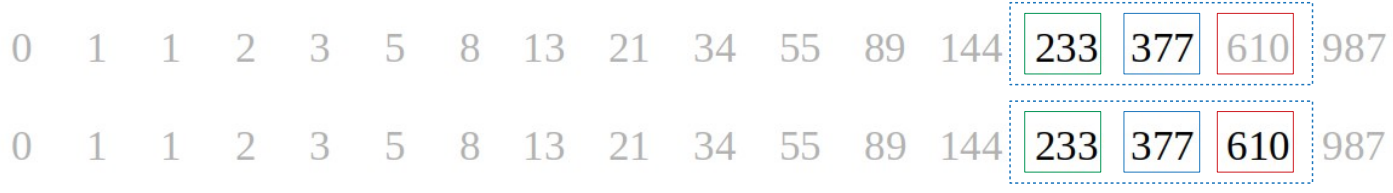
- Vystačíme s cyklem:
  faktorial,
  Fibonacci,
  Legendrovy (např.) polynomy
- Když ne:
  Použijeme rekurzivní volání (není to zas tak drahé)
  Simulujeme zásobník jinak (výjimečně)

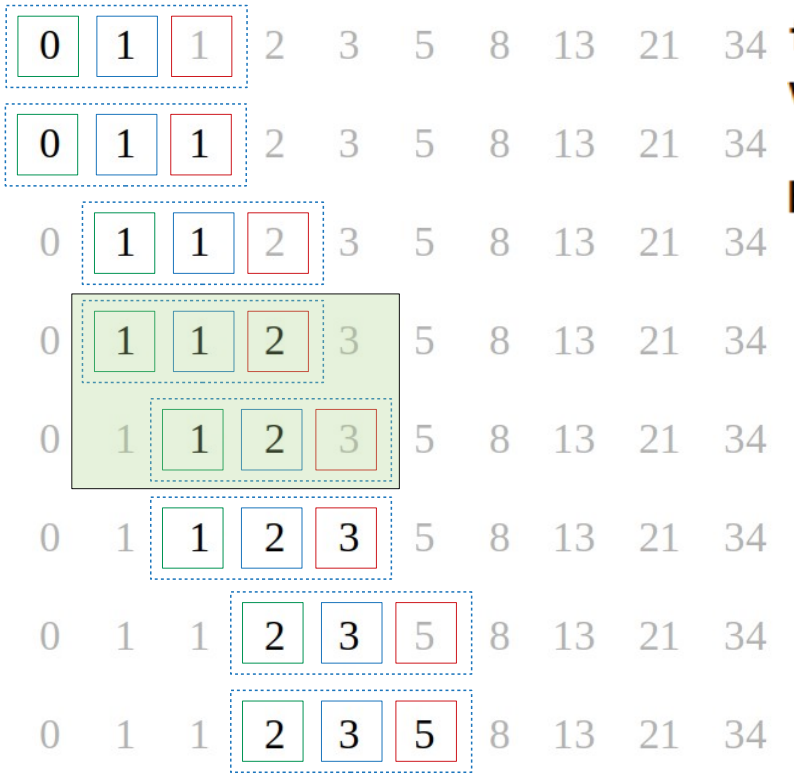
```
function fib(n:integer):integer;
begin
  if n=0 then fib := 0
  else if n=1 then fib := 1
  else fib := fib(n-1)+fib(n-2);
end;
```

```
function fib2(n:integer):real;
var a,b,c: real;
    i : integer;
begin
  a := 0;      // inicializace
  b := 1;
  for i:=2 to n do begin
    c := a+b;   // scitani sousednich clenu posloupnosti
    a := b;
    b := c;
  end;
  // vratit vysledek
  if n>0 then fib2 := b
  else fib2 := 0;
end;
```

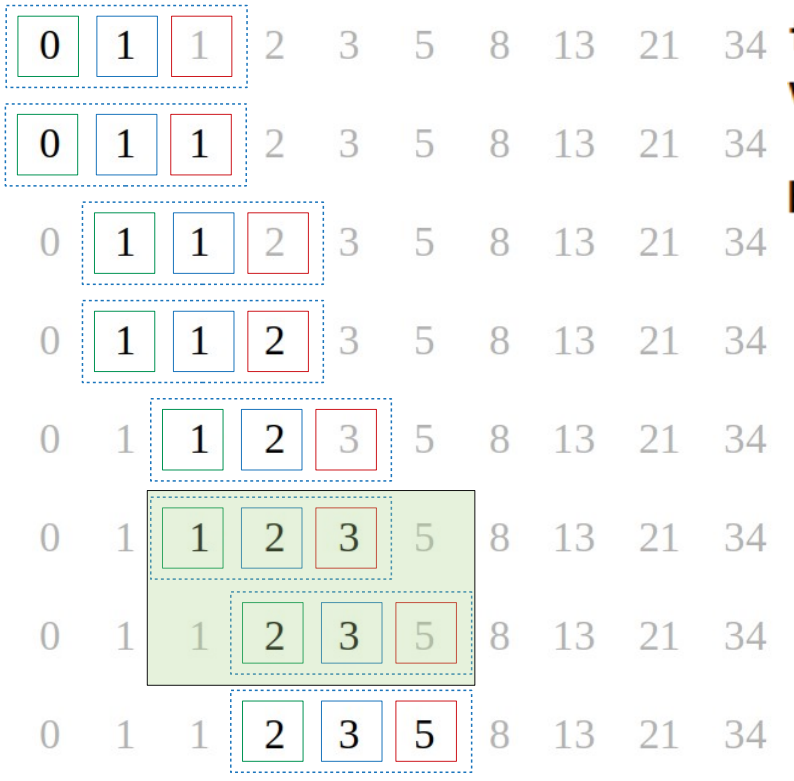



```
function fib2(n:integer):real;
var a,b,c: real;
    i : integer;
begin
  a := 0;      // inicializace
  b := 1;
  for i:=2 to n do begin
    c := a+b;   // scitani sousednich clenu p
    a := b;
    b := c;
  end;
  // vratit vysledek
  if n>0 then fib2 := b
  else fib2 := 0;
end;
```






```
function fib2(n:integer):real;
var a,b,c: real;
    i : integer;
begin
  a := 0;      // inicializace
  b := 1;
  for i:=2 to n do begin
    c := a+b;    // scitani sousednich clenu p
    a := b;
    b := c;
  end;
  // vratit vysledek
  if n>0 then fib2 := b
  else fib2 := 0;
end;
```




```
function fib2(n:integer):real;
var a,b,c: real;
    i : integer;
begin
  a := 0;      // inicializace
  b := 1;
  for i:=2 to n do begin
    c := a+b;    // scitani sousednich clenu p
    a := b;
    b := c;
  end;
  // vratit vysledek
  if n>0 then fib2 := b
  else fib2 := 0;
end;
```

Polynomy určené koeficienty (Hornerovo schéma)

$$1 - 162\,x + 4320\,x^2 - 44\,352\,x^3 + 228\,096\,x^4 - 658\,944\,x^5 + 1\,118\,208\,x^6 - 1\,105\,920\,x^7 + 589\,824\,x^8 - 131\,072\,x^9$$

$$1 + x\,(-162 + x\,(4320 + x\,(-44\,352 + x\,(228\,096 + x\,(-658\,944 + x\,(1\,118\,208 + x\,(-1\,105\,920 + (589\,824 - 131\,072\,x)\,x)))))))$$

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;
```

Polynomy určené koeficienty (Hornerovo schéma)

Poděkování: https://godbolt.org/

Výraz: `1 - 162*x + 4320*x*x - 44352*x*x*x + 228096*x*x*x*x - 658944*x*x*x*x*x;`

Pascal , fpc:	15 *	5 +
C, gcc:	15 *	5 +
C, icc:	5 *	5 +
Fortran, gcc	9 *	5 +
Fortran, ifort	5 *	5 +

Pozn. Některé překladače je třeba k optimalizaci 5* 5+ donutit

Výraz: `1 + x *(-162 + x* (4320 + x* (-44352 + x* (228096 - 658944* x) )));`

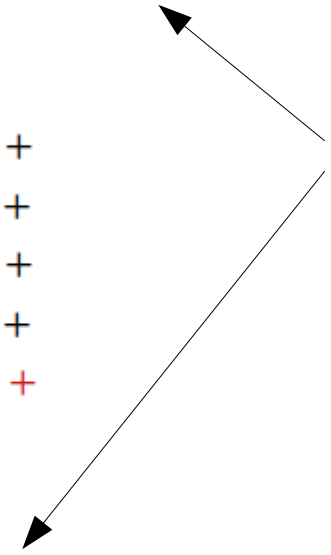
Všichni: 5 * 5+

Polynomy určené koeficienty (Hornerovo schéma)

Poděkování: https://godbolt.org/

Výraz: `1 - 162*x + 4320*x*x - 44352*x*x*x + 228096*x*x*x*x - 658944*x*x*x*x*x;`

Pascal , fpc:	15 *	5 +
C, gcc:	15 *	5 +
C, icc:	5 *	5 +
Fortran, gcc	9 *	5 +
Fortran, ifort	5 *	5 +



Pozor, pro x=Pi se hodnoty výrazů (~2E8) liší (o 3E-8)

Výraz: `1 + x *(-162 + x* (4320 + x* (-44352 + x* (228096 - 658944* x) )));`

Všichni: 5 * 5+

Reálná čísla v počítači

* Do žádného počítače se nevejde ani jedno *opravdové* reálné … lené

* Nezbytně reálná čísla potřebujeme

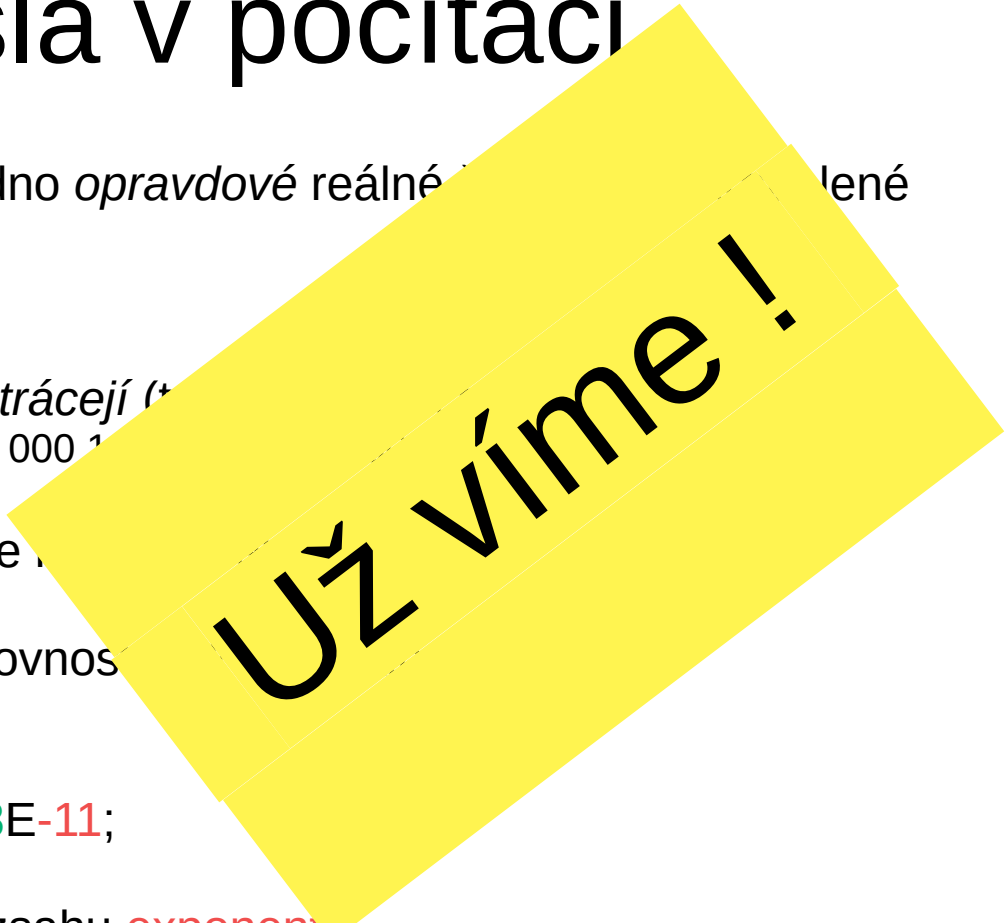
* Musíme tolerovat, že poslední cifry se *ztrácejí* (t…
  1.000 000 000 000 000 + 0.000 000 000 000 000 …

* Opakovanými operacemi se chyba může …

* Není rozumné testovat reálná čísla na rovnos…
  nahrazovat celými čísly,

* Zapisujeme např. **const** G_SI =6.67408E-11;

* Omezení délky mantisy počtu cifer a rozsahu exponentu



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)

         0100000000001001001000011111101101010001000100001011010001100 0
 3.141592653590 = +1.1001001000011111101101010001000100001011010001100 0 x 2^(1)

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

Například π je tedy nahrazeno aproximací

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

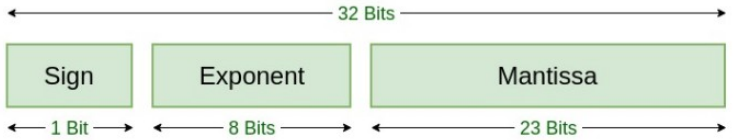
Proč zrovna 64 bitová reálná čísla ?

na začátku bylo ~~slovo~~ písmeno

písmen (in English) je 2x26, číslic 10, +další znaky => potřebujeme aspoň 7 **bit**

zaokrouhlení (8 je lepší něž 7) => **byte**

jedno písmeno na uložení reálného čísla nestačí => 4 bytes == **single precision**



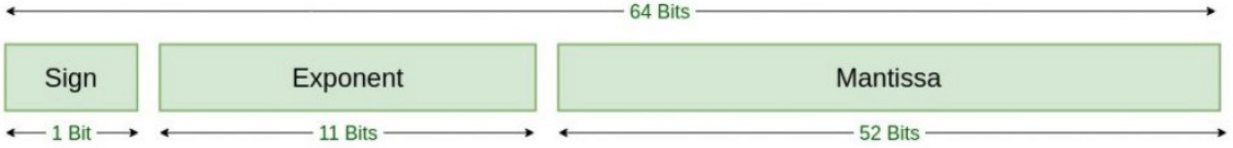
Single Precision
IEEE 754 Floating-Point Standard

23 bitová mantisa dovoluje přesnost $2^{-23} \sim 10^{-7}$ a to je poměrně málo

6 bytes == Turbo Pascal real (1984)

Byte	0	1	2	3	4	5
Bit	01234567	01234567	01234567	01234567	01234567	01234567
Value	EEEEEEEE	MMMMMMMM	MMMMMMMM	MMMMMMMM	MMMMMMMM	SMMMMMMM

8 bytes == **double precision**



Double Precision
IEEE 754 Floating-Point Standard

52 bitová mantisa dovoluje přesnost $2^{-52} \sim 10^{-16}$ a to je *většinou* dost

```
   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:

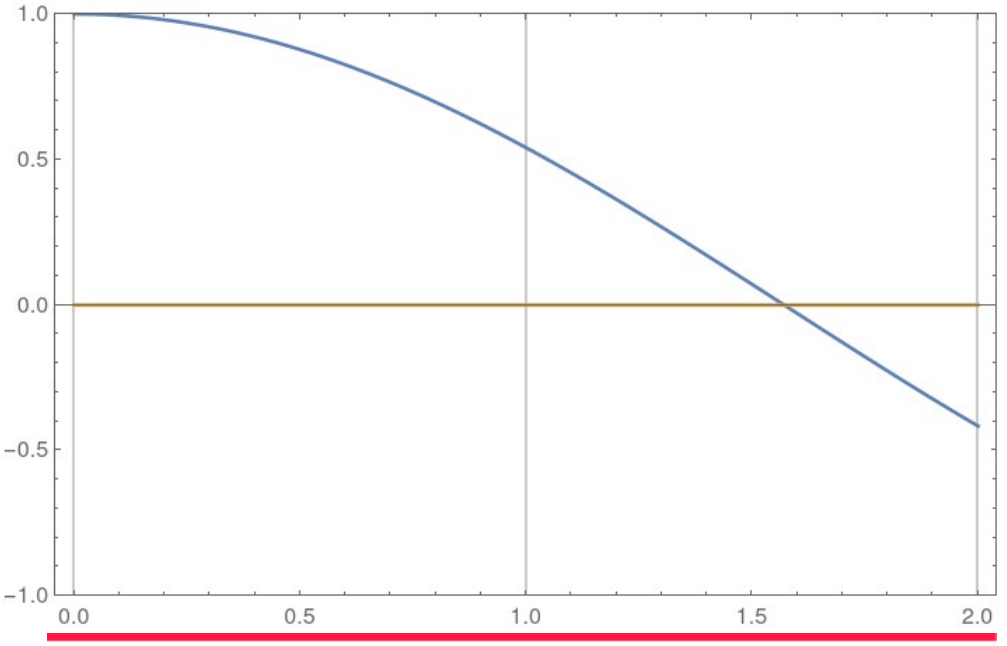
```
x  := 0.005;
x3 := x*x*x;
y  := 105*((x3-15*x)*Cos(x) + (15-6*x*x)*Sin(x))/(x3*x3*x);
%
```

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

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

Hledání kořene -- Metoda půlení intervalu



$2^3 \sim 10$

$2^{10} \sim 1000$

$2^{50} \sim 10^{15}$

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:

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

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

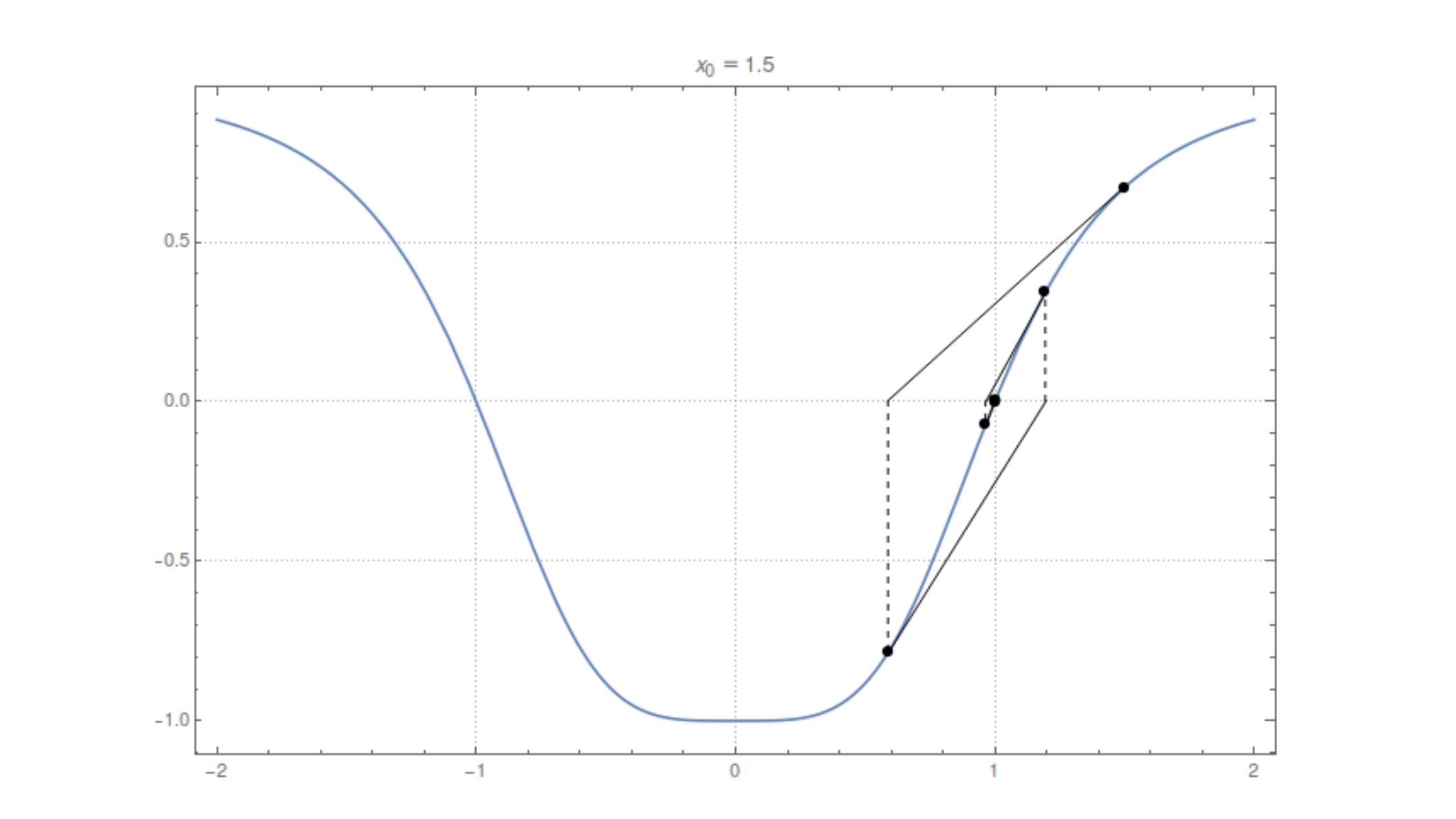
$$\boxed{x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}},$$

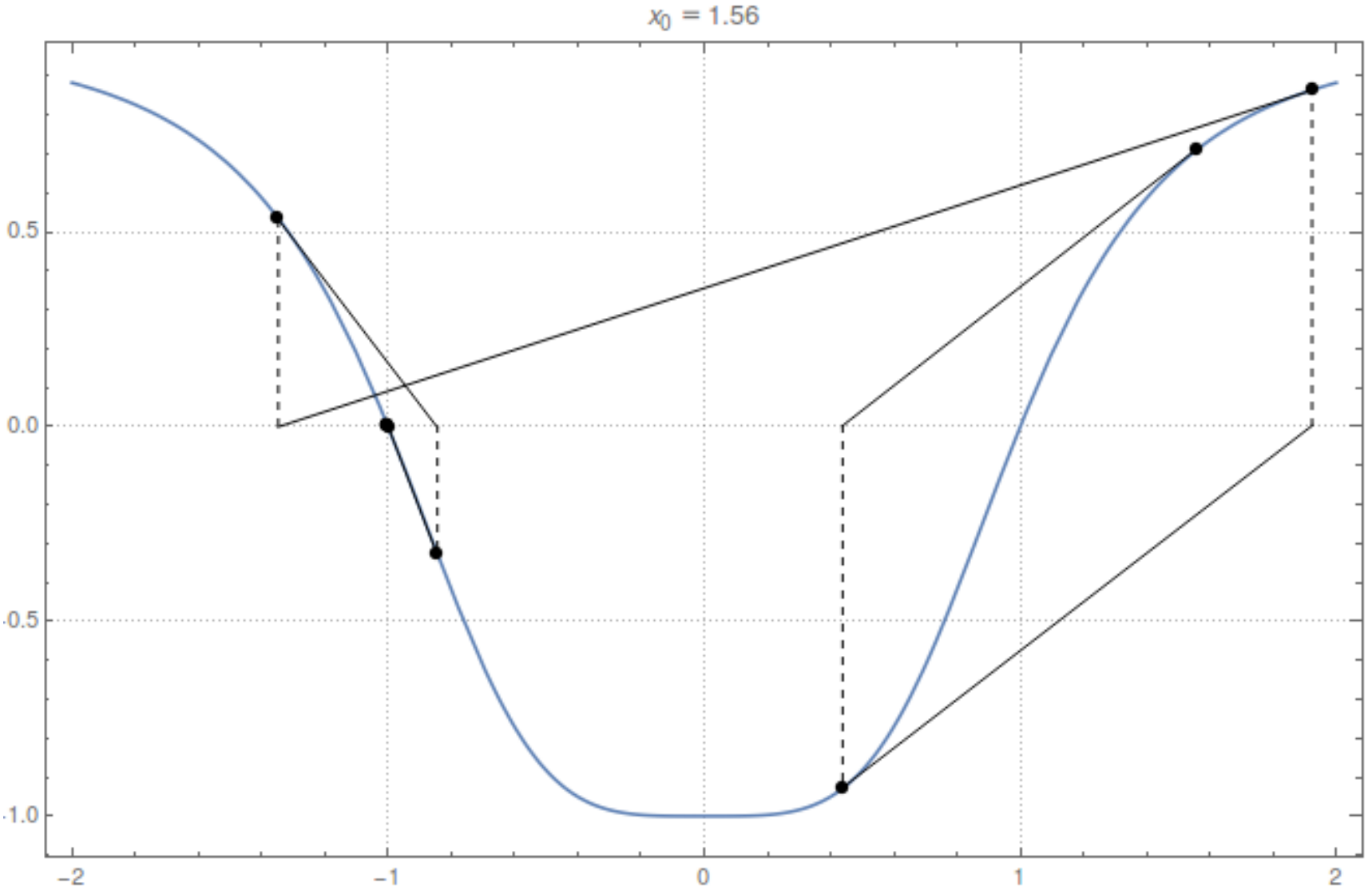
Příklad:

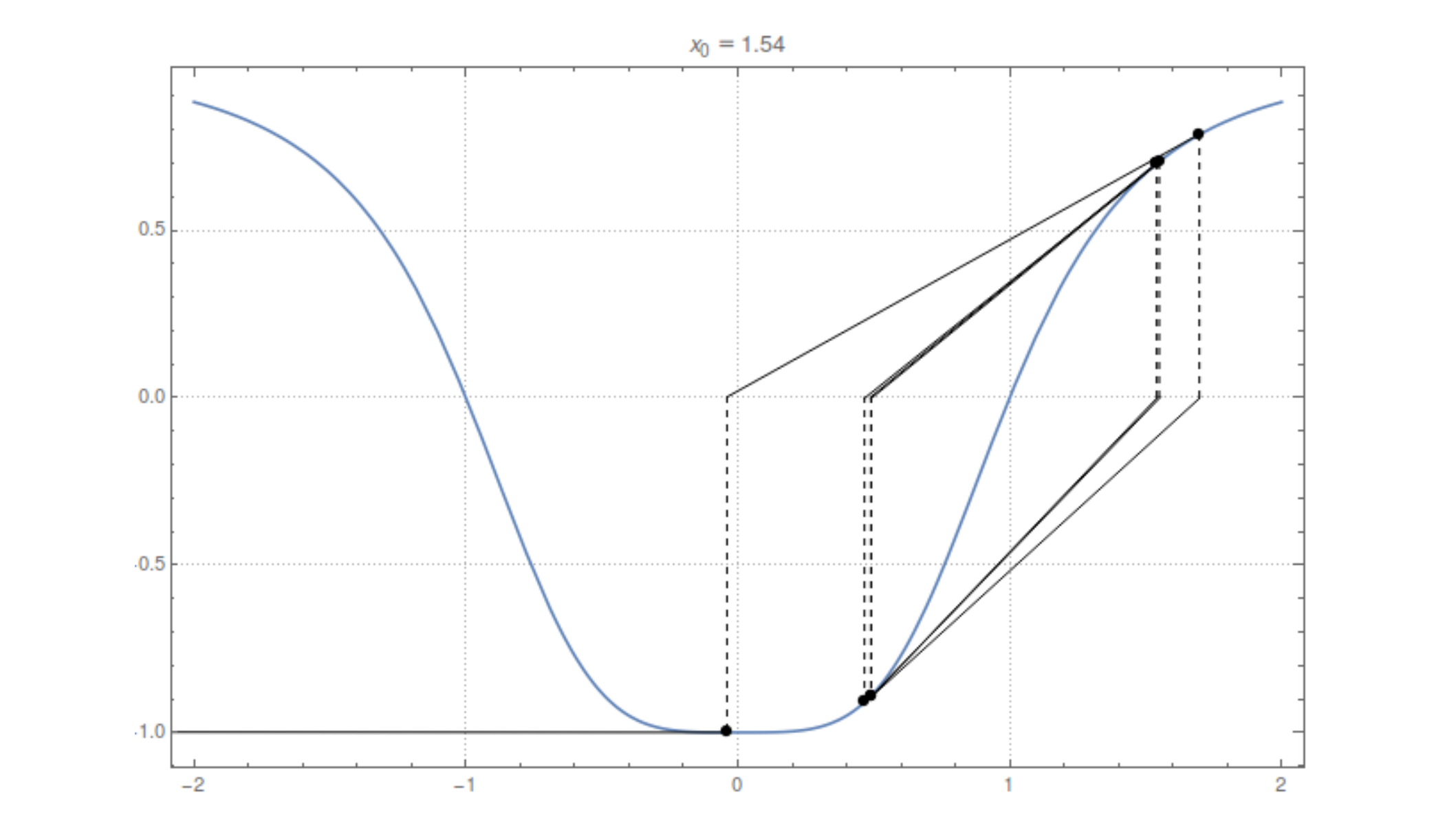
$$f(x) = b - \frac{1}{x}$$

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

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


$x_0 = 1.5$
0.5
0.0
-0.5
-1.0
−2
−1
0
1
2


$x_0 = 1.56$
0.5
0.0
-0.5
-1.0
−2
−1
0
1
2


$x_0 = 1.54$
0.5
0.0
0.5
1.0
−2
−1
0
1
2

Metoda

Tečen (Newtonova)

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

Regula falsi

$[a_0, b_0]$ = [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]

Sečen

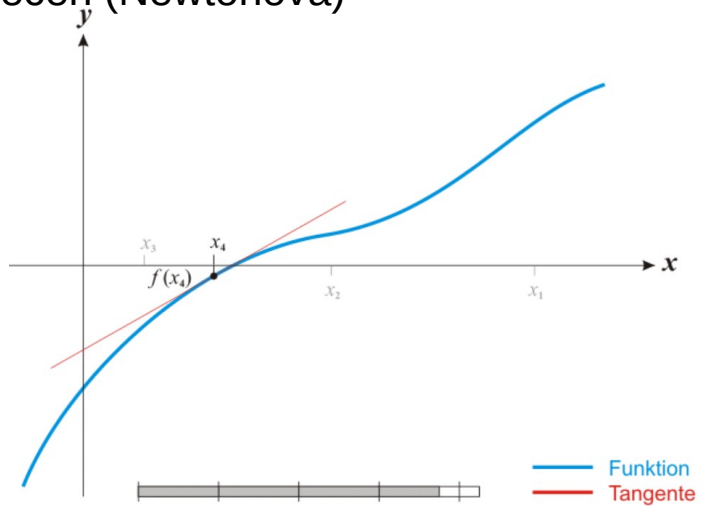
$[a_0, b_0]$ = [1.000000000000000, 1.200000000000000]
$[a_1, b_1]$ = [1.200000000000000, 1.120000000000000]
$[a_2, b_2]$ = [1.120000000000000, 1.110400000000000]
$[a_3, b_3]$ = [1.110400000000000, 1.111116800000000]
$[a_4, b_4]$ = [1.111116800000000, 1.111111114751999]
$[a_5, b_5]$ = [1.111111114751999, 1.111111111111092]
$[a_6, b_6]$ = [1.111111111111092, 1.111111111111111]

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

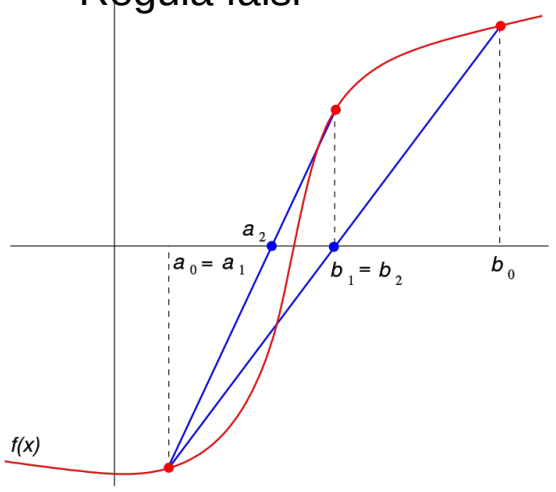
$$c = a\frac{f(b)}{f(b)-f(a)} - b\frac{f(a)}{f(b)-f(a)} = b - \frac{f(b)}{\frac{f(b)-f(a)}{b-a}}$$

Metoda

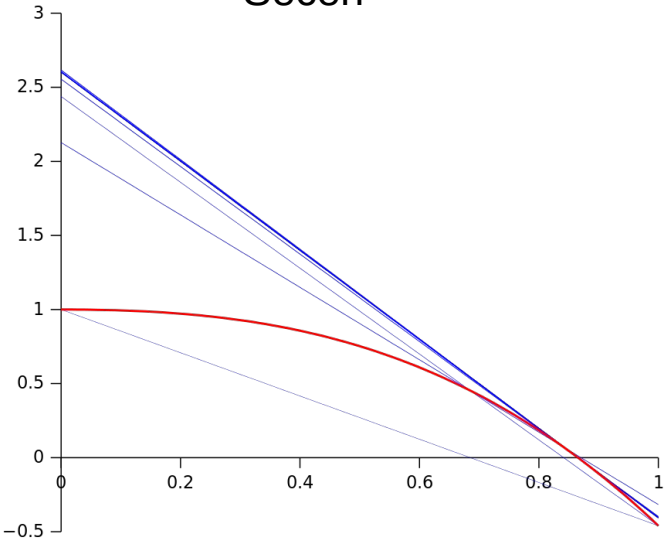
Tečen (Newtonova)



Regula falsi



Sečen



Dotaz ?

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).

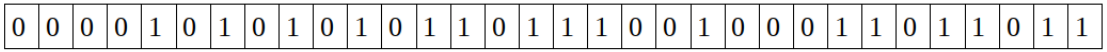
```
#$20=#32: '␣' '!' '"' '#' '$' '%' '&' ''' '(' ')' '*' '+' ',' '-' '.' '/'
#$30=#48: '0' '1' '2' '3' '4' '5' '6' '7'  '8' '9' ':' ';' '<' '=' '>' '?'
#$40=#64: '@' 'A' 'B' 'C' 'D' 'E' 'F' 'G'  'H' 'I' 'J' 'K' 'L' 'M' 'N' 'O'
#$50=#80: 'P' 'Q' 'R' 'S' 'T' 'U' 'V' 'W'  'X' 'Y' 'Z' '[' '\' ']' '^' '_'
#$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.

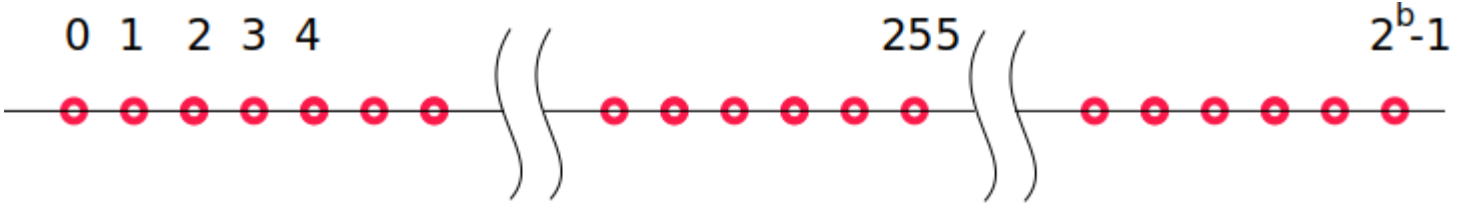
Ordinální typy

Hardware:

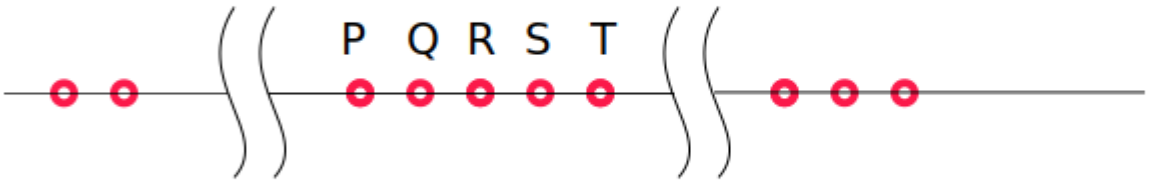


Abstrakce:

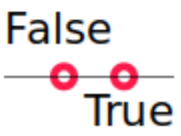
LongWord

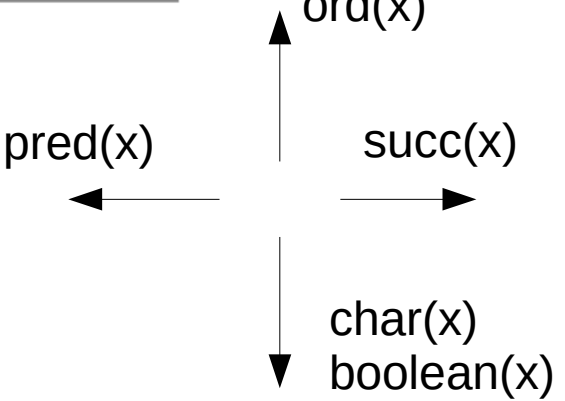


Char



Boolean





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;
```

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);
```

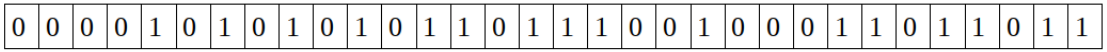
Vyplata := Plat[1, 3];

	1	2	3
1	50000	60000	90000
2	52000	63000	96000
3	54000	66000	102000
4	56000	69000	108000

	eRadovy	eClenVyboru	ePredsedaKlubu
1	50000	60000	90000
2	52000	63000	96000
3	54000	66000	102000
4	56000	69000	108000

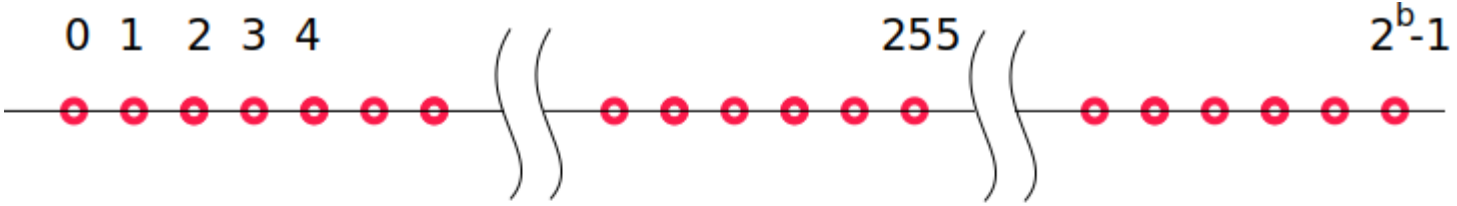
Ordinální typy

Hardware:

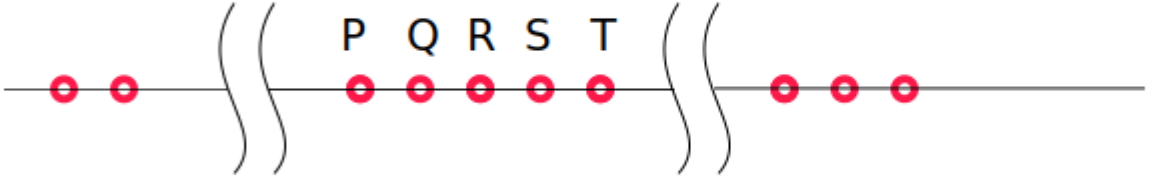


Abstrakce:

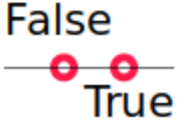
LongWord



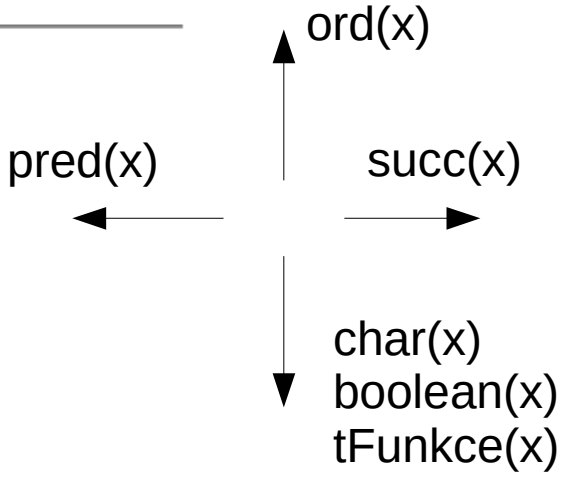
Char



Boolean



tFunkce



Pole (array)

1	2	5	1	8	12	4	5	1	7	8	9

K čemu:

Přístup k prvku pole

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

Přiřazení

```
b := a;
```

Argument funkcí a procedur

```
x := nejvetsi(a);
```

Jak deklarovat?

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

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 !!!
```

Přiřazení

```
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!
```

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);
```

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);
```

Kdy předávat pole odkazem ?

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

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

s vektorovou algebrou byl *špatně* hned dvakrát. jednou

```
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 ;
```

```
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 ;
```

```
const Dim = 30;
type arr = array[1..Dim] of real;

function secti(const a,b : arr):arr;
  var i : integer;
begin
  for i := 1 to Dim do
    secti[i] := a[i]+b[i];
end;

procedure secti2(const a,b : arr; var s:arr);
  var i : integer;
begin
  for i := 1 to Dim do
    s[i] := a[i]+b[i];
end;
```

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;
```

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

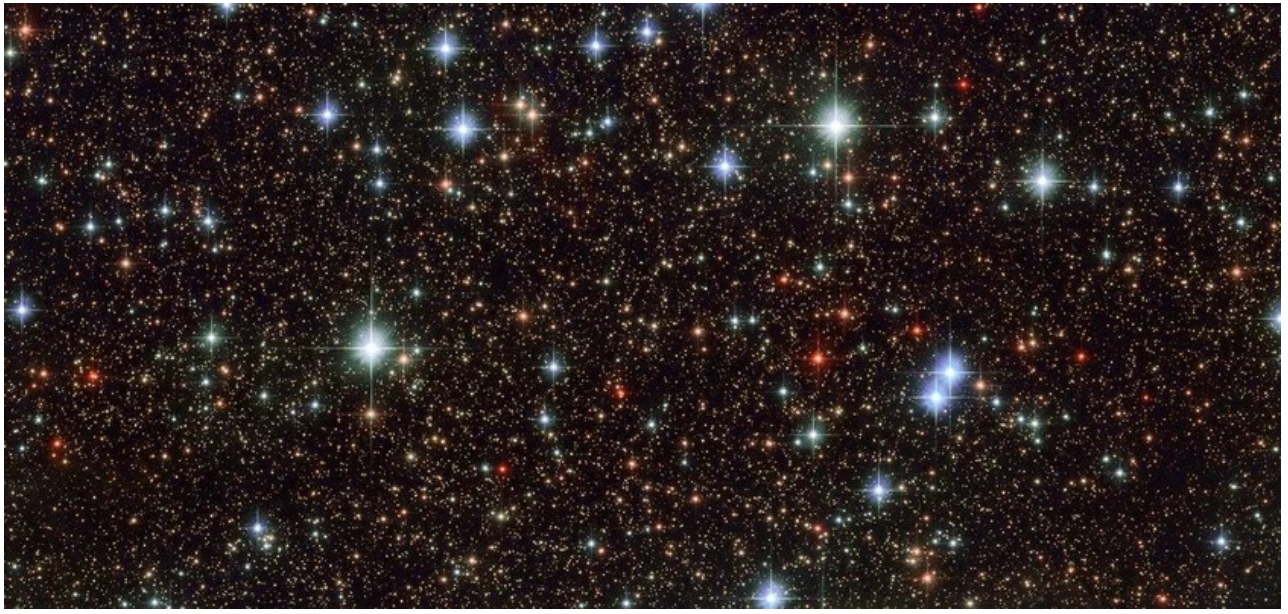
```
var Faktorial : array [0..170] of real; // 171! se do proměnné typu real nevejde
    Binomial  : array [0..1000,0..1000] of real; //zkuste urcit presnejsi meze !!!
```

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 prvočísel 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.
```



```
type tHvezda = record
                 Jmeno: tJmenoHvezdy;
                 Souradnice: array[1..3] of real;
                 Svitivost: real;
                 ....
                 PocetPlanet: integer;
               end;
```

Co hvězda to:

Jméno

Souřadice

Svítivost

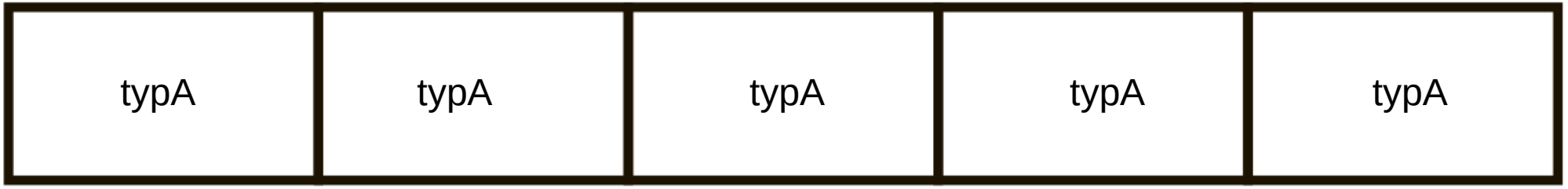
Barva (Spektrální třída)

Hmotnost

Počet planet

ESA/Hubble & NASA

Pole (array)



Záznam (record)


```
type tHvezda = record
                 Jmeno: tJmenoHvezdy;
                 Souradnice: array[1..3] of real;
                 Svitivost: real;
                 ....
                 PocetPlanet: integer;
               end;
```

Pole:

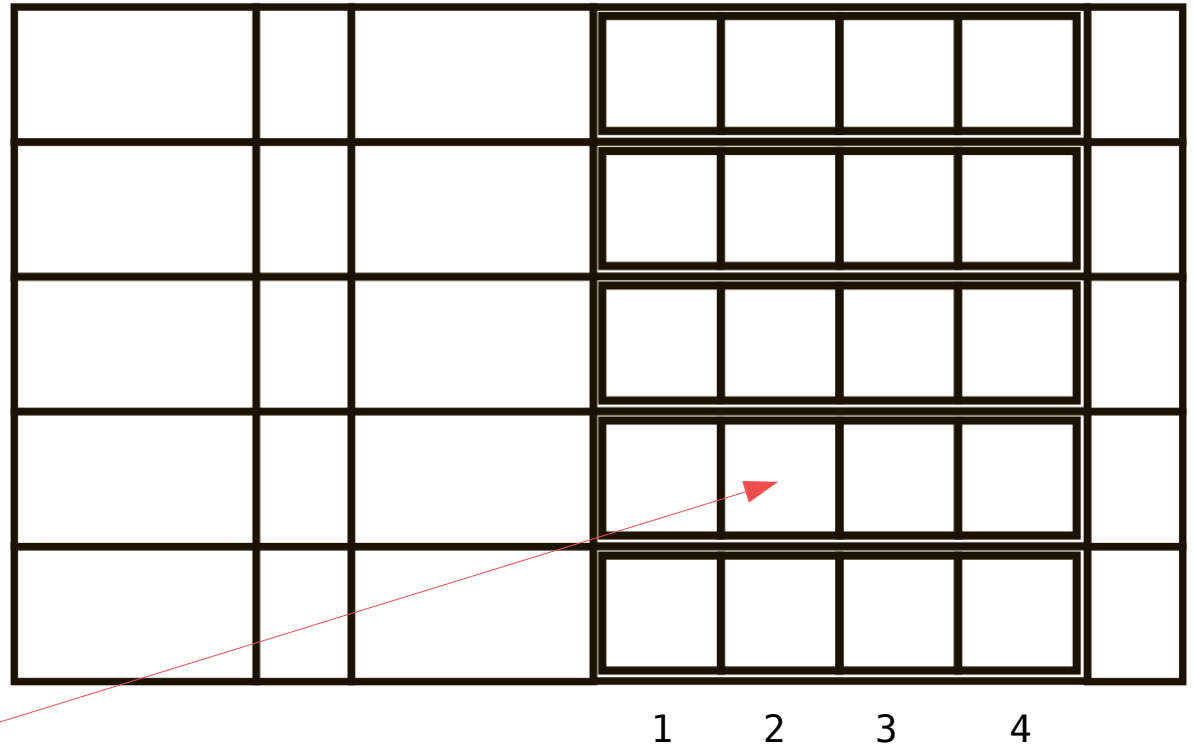
p[1]+p[2]*p[3]

p[iPoloha]+p[iRychlost]*p[iCas]

Záznam:

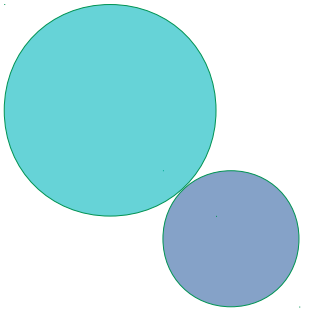
z.poloha+z.rychlost*z.cas

```
type typX =

     array[1..5] of
       record
         A: typA;
         B: typB;
         C: typC;
         D: array[1..4] of char;
         E: typD;
       end;


var X : typX;
```




```
...
    X[4].D[2]
...
```

```
type tHmotnyDisk2D = record
      x,y   : real; // 2D poloha
      vx,vy : real; // 2D rychlost
      r     : real; // polomer kruznice
      m     : real; // hmotnost (dulezita pri srazce)
end;

procedure testSrazeni;
const a : tHmotnyDisk2D =   ( x:10.0; y: 20.0; vx: 7 ; vy: 1  ; r: 4; m: 1);
      b : tHmotnyDisk2D =   ( x:20.0; y: 20.0; vx: 1 ; vy: 9  ; r: 6; m: 2);
      tpq = 5.2;
      eps = 1E-11;
var   Ek,px,py,Fk,qx,qy,cas,d2:real;

begin
   // kotrola, ze se disky dotykaji
   d2 := sqr(a.x-b.x) + sqr(a.y-b.y);
   if abs( d2 - sqr(a.r+b.r) )>eps then begin
       writeln('Test dotyku disku nedopadnul dobre.');
       writeln('Misto ', a.r+b.r , ' jsou disky vzdaleny ', sqrt(d2) );
       halt;
   end;
   // kontrola, ze se pri srazce zachovava hybnost a energie
   px := a.m*a.vx+b.m*b.vx; py := a.m*a.vy+b.m*b.vy;
   Ek := 0.5*a.m*(sqr(a.vx)+sqr(a.vy))+0.5*b.m*(sqr(b.vx)+sqr(b.vy));
   odraz(a,b);
   qx := a.m*a.vx+b.m*b.vx; qy := a.m*a.vy+b.m*b.vy;
   Fk := 0.5*a.m*(sqr(a.vx)+sqr(a.vy))+0.5*b.m*(sqr(b.vx)+sqr(b.vy));
   if abs( px-qx ) + abs( py-qy ) > eps then begin
       writeln('Test funkce odraz nedopadnul dobre.');
       writeln('Misto [',px:8:5,',',py:8:5,'] vratila po srazce hybnost [',qx:8:5,',',qy:8:5,']' );
       halt;
   end;
   if abs( Ek-Fk ) > eps then begin
       writeln('Test funkce odraz nedopadnul dobre.');
       writeln('Misto ',Ek:8:5,' vratila po srazce energii ',Fk:8:5 );
       halt;
   end;
end;
```



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, obzvášt̆e 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

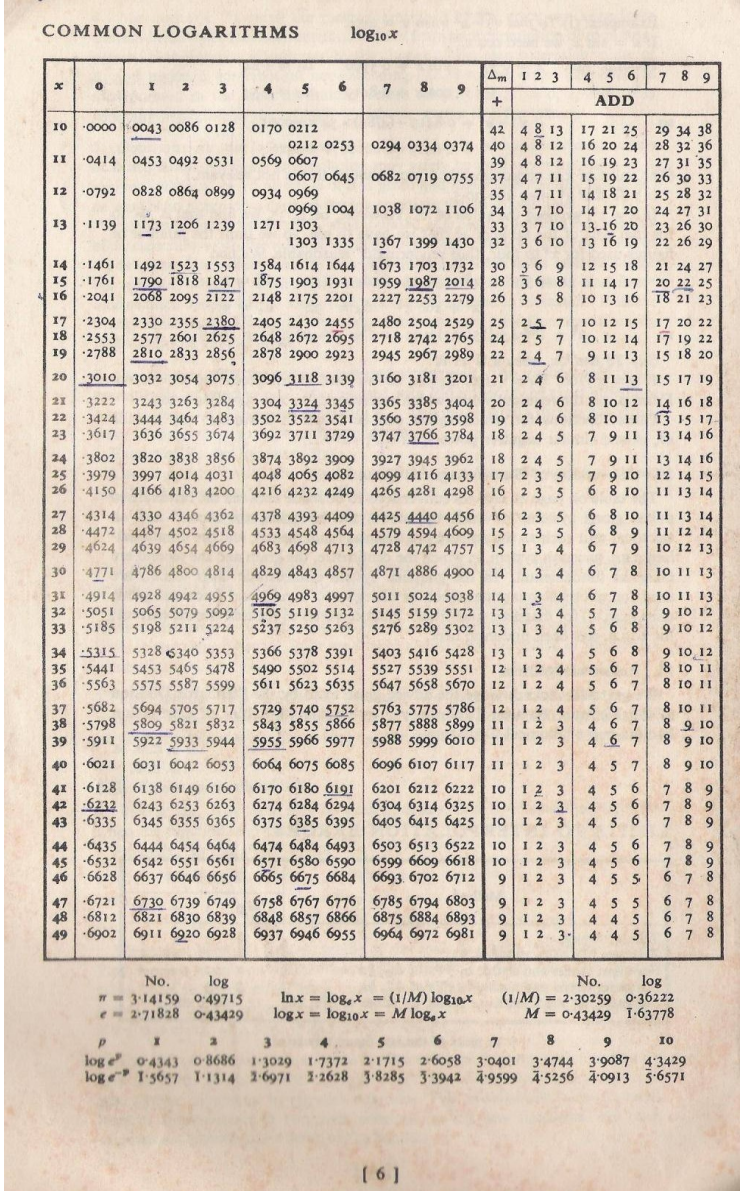
Strukturované programování mělo za cíl učinit příkaz skoku až na výjimky zbytečným. Tento svůj cíl splnilo, protože jsme jej doposud na přednášce nepotřebovali. Přesto existují situace, kdy je jejich použití na místě. K dipozici máme následující příkazy skoku:

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)

1	2	9	4	2	5	8	7	1	3	2	9

```
function jeVSeznamu(const seznam:tSeznam; x:tPrvekSeznamu):boolean;
var i:integer;
begin
  jeVSeznamu := true;
  for i := 1 to delkaSzenamu do
    if seznam[i] = x then exit;
  jeVSeznamu := false;
end;
```

ESA/Hubble & NASA



COMMON LOGARITHMS $\log_{10} x$

x	0	1	2	3	4	5	6	7	8	9	Δ_m	1 2 3	4 5 6	7 8 9
											+	ADD		
10	·0000	0043	0086	0128	0170	0212					42	4 8 13	17 21 25	29 34 38
						0212	0253	0294	0334	0374	40	4 8 12	16 20 24	28 32 36
11	·0414	0453	0492	0531	0569	0607					39	4 8 12	16 19 23	27 31 35
						0607	0645	0682	0719	0755	37	4 7 11	15 19 22	26 30 33
12	·0792	0828	0864	0899	0934	0969					35	4 7 11	14 18 21	25 28 32
						0969	1004	1038	1072	1106	34	3 7 10	14 17 20	24 27 31
13	·1139	1173	1206	1239	1271	1303					33	3 7 10	13 16 20	23 26 30
						1303	1335	1367	1399	1430	32	3 6 10	13 16 19	22 26 29
14	·1461	1492	1523	1553	1584	1614	1644	1673	1703	1732	30	3 6 9	12 15 18	21 24 27
15	·1761	1790	1818	1847	1875	1903	1931	1959	1987	2014	28	3 6 8	11 14 17	20 22 25
16	·2041	2068	2095	2122	2148	2175	2201	2227	2253	2279	26	3 5 8	10 13 16	18 21 23
17	·2304	2330	2355	2380	2405	2430	2455	2480	2504	2529	25	2 5 7	10 12 15	17 20 22
18	·2553	2577	2601	2625	2648	2672	2695	2718	2742	2765	24	2 5 7	10 12 14	17 19 22
19	·2788	2810	2833	2856	2878	2900	2923	2945	2967	2989	22	2 4 7	9 11 13	15 18 20
20	·3010	3032	3054	3075	3096	3118	3139	3160	3181	3201	21	2 4 6	8 11 13	15 17 19
21	·3222	3243	3263	3284	3304	3324	3345	3365	3385	3404	20	2 4 6	8 10 12	14 16 18
22	·3424	3444	3464	3483	3502	3522	3541	3560	3579	3598	19	2 4 6	8 10 11	13 15 17
23	·3617	3636	3655	3674	3692	3711	3729	3747	3766	3784	18	2 4 5	7 9 11	13 14 16
24	·3802	3820	3838	3856	3874	3892	3909	3927	3945	3962	18	2 4 5	7 9 11	13 14 16
25	·3979	3997	4014	4031	4048	4065	4082	4099	4116	4133	17	2 3 5	7 9 10	12 14 15
26	·4150	4166	4183	4200	4216	4232	4249	4265	4281	4298	16	2 3 5	6 8 10	11 13 14
27	·4314	4330	4346	4362	4378	4393	4409	4425	4440	4456	16	2 3 5	6 8 10	11 13 14
28	·4472	4487	4502	4518	4533	4548	4564	4579	4594	4609	15	2 3 5	6 8 9	11 12 14
29	·4624	4639	4654	4669	4683	4698	4713	4728	4742	4757	15	1 3 4	6 7 9	10 12 13
30	·4771	4786	4800	4814	4829	4843	4857	4871	4886	4900	14	1 3 4	6 7 8	10 11 13
31	·4914	4928	4942	4955	4969	4983	4997	5011	5024	5038	14	1 3 4	6 7 8	10 11 13
32	·5051	5065	5079	5092	5105	5119	5132	5145	5159	5172	13	1 3 4	5 7 8	9 10 12
33	·5185	5198	5211	5224	5237	5250	5263	5276	5289	5302	13	1 3 4	5 6 8	9 10 12
34	·5315	5328	5340	5353	5366	5378	5391	5403	5416	5428	13	1 3 4	5 6 8	9 10 12
35	·5441	5453	5465	5478	5490	5502	5514	5527	5539	5551	12	1 2 4	5 6 7	8 10 11
36	·5563	5575	5587	5599	5611	5623	5635	5647	5658	5670	12	1 2 4	5 6 7	8 10 11
37	·5682	5694	5705	5717	5729	5740	5752	5763	5775	5786	12	1 2 4	5 6 7	8 10 11
38	·5798	5809	5821	5832	5843	5855	5866	5877	5888	5899	11	1 2 3	4 6 7	8 9 10
39	·5911	5922	5933	5944	5955	5966	5977	5988	5999	6010	11	1 2 3	4 6 7	8 9 10
40	·6021	6031	6042	6053	6064	6075	6085	6096	6107	6117	11	1 2 3	4 5 7	8 9 10
41	·6128	6138	6149	6160	6170	6180	6191	6201	6212	6222	10	1 2 3	4 5 6	7 8 9
42	·6232	6243	6253	6263	6274	6284	6294	6304	6314	6325	10	1 2 3	4 5 6	7 8 9
43	·6335	6345	6355	6365	6375	6385	6395	6405	6415	6425	10	1 2 3	4 5 6	7 8 9
44	·6435	6444	6454	6464	6474	6484	6493	6503	6513	6522	10	1 2 3	4 5 6	7 8 9
45	·6532	6542	6551	6561	6571	6580	6590	6599	6609	6618	10	1 2 3	4 5 6	7 8 9
46	·6628	6637	6646	6656	6665	6675	6684	6693	6702	6712	9	1 2 3	4 5 5	6 7 8
47	·6721	6730	6739	6749	6758	6767	6776	6785	6794	6803	9	1 2 3	4 5 5	6 7 8
48	·6812	6821	6830	6839	6848	6857	6866	6875	6884	6893	9	1 2 3	4 4 5	6 7 8
49	·6902	6911	6920	6928	6937	6946	6955	6964	6972	6981	9	1 2 3	4 4 5	6 7 8

	No.	log
π =	3·14159	0·49715
e =	2·71828	0·43429

$\ln x = \log_e x = (1/M) \log_{10} x$

$\log x = \log_{10} x = M \log_e x$

	No.	log
$(1/M)$ =	2·30259	0·36222
M =	0·43429	$\bar{1}$·63778

p	1	2	3	4	5	6	7	8	9	10
$\log e^p$	0·4343	0·8686	1·3029	1·7372	2·1715	2·6058	3·0401	3·4744	3·9087	4·3429
$\log e^{-p}$	$\bar{1}$·5657	$\bar{1}$·1314	$\bar{2}$·6971	$\bar{2}$·2628	$\bar{3}$·8285	$\bar{3}$·3942	$\bar{4}$·9599	$\bar{4}$·5256	$\bar{4}$·0913	$\bar{5}$·6571

[6]

Interpolace (jen polynomiální)

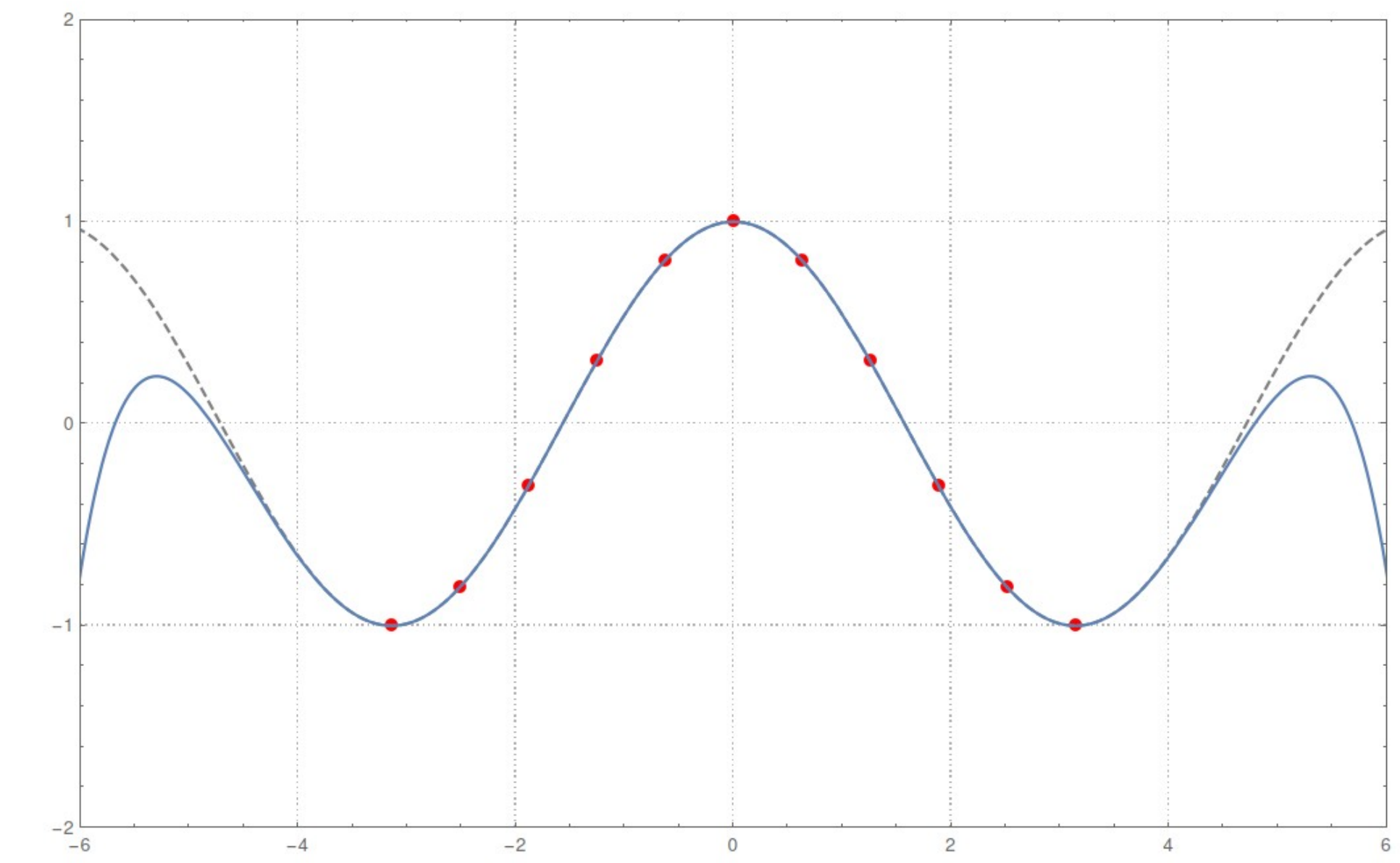
log10(Pi) = 0.49714987269413385435126828829089887365 1

Interpolace vs extrapolace

Intrepolace vs “splajny“ (spline)

Lagrangeův interpolační vzorec

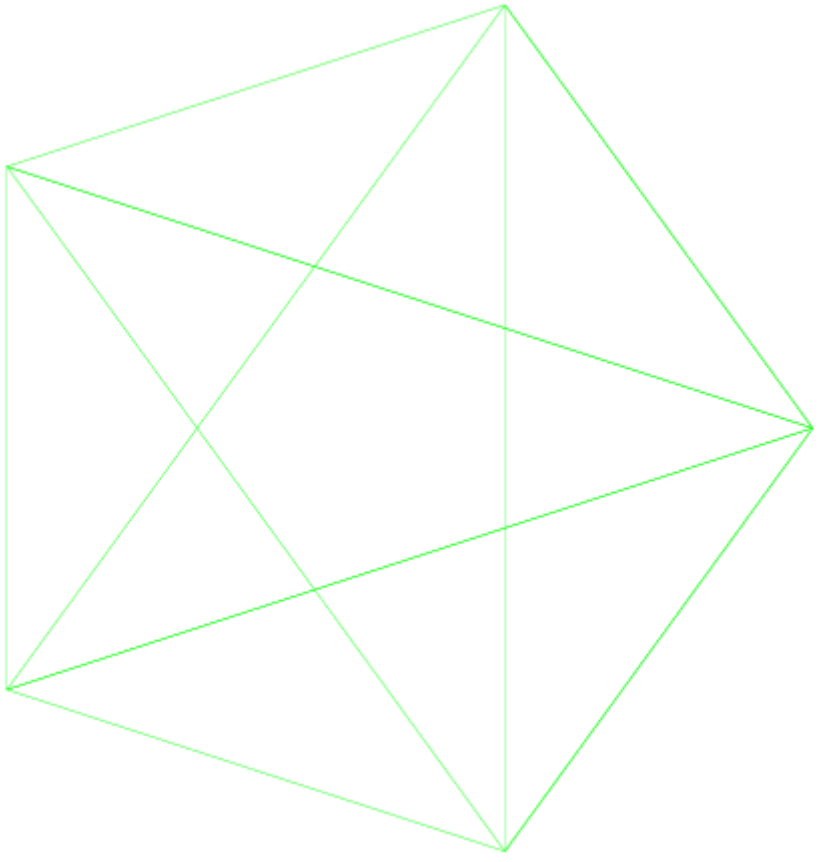
Varování před vysokými řády

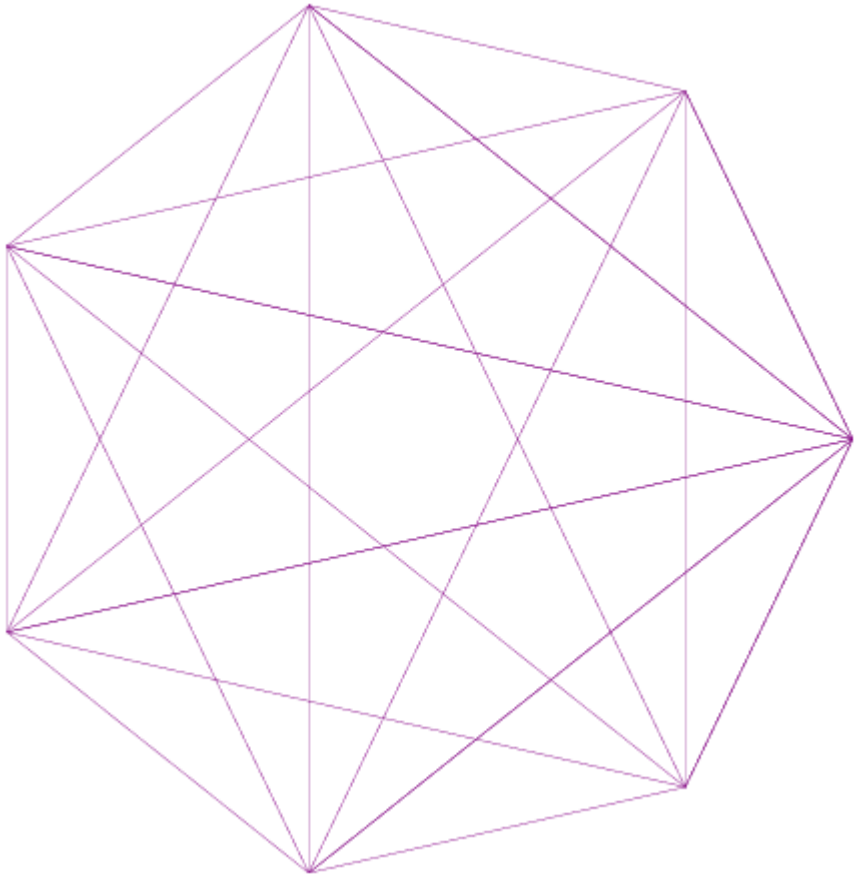

2
1
0
−1
−2
−6
−4
−2
0
2
4
6

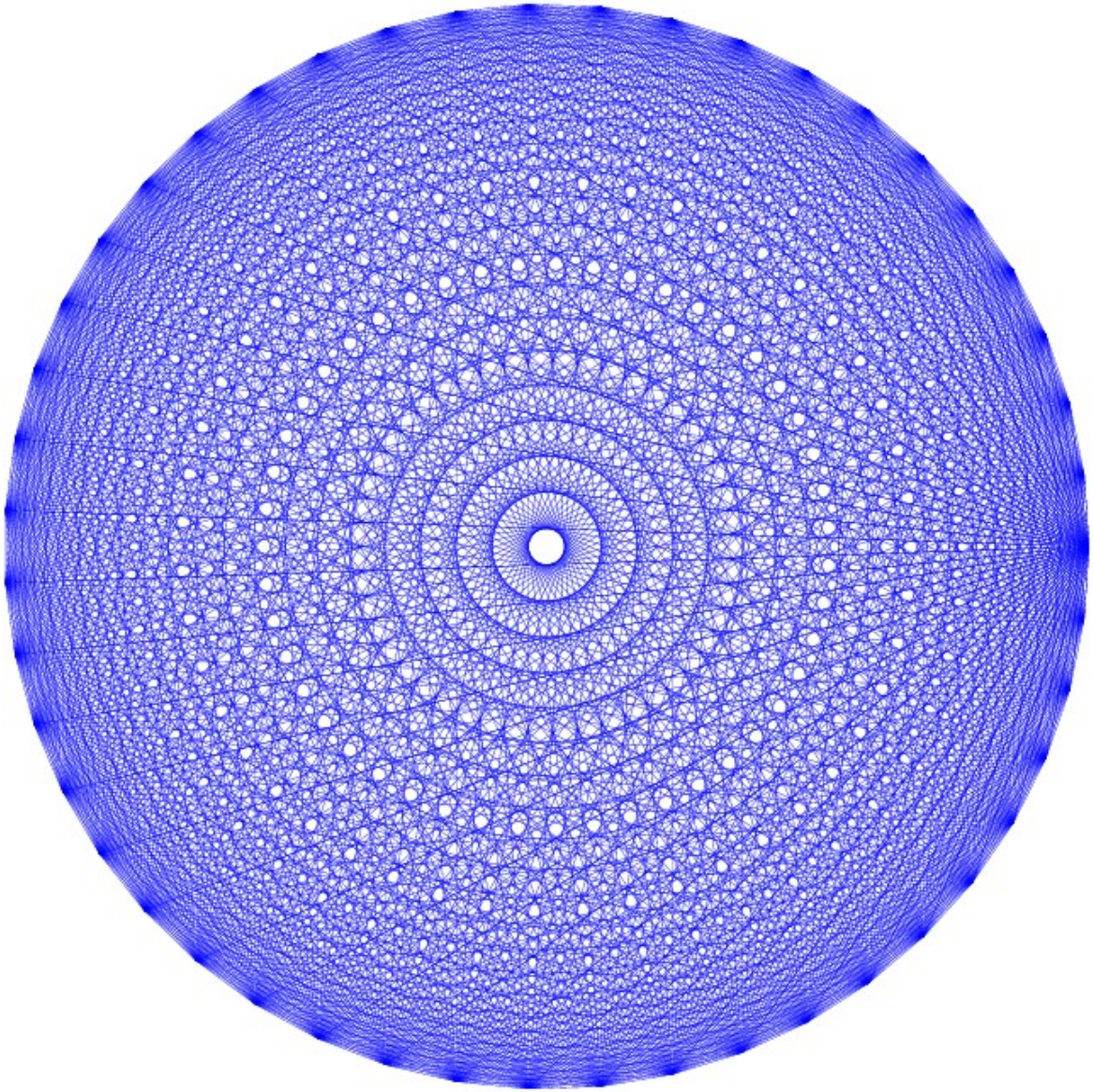
Lagrangeův interpolační vzorec

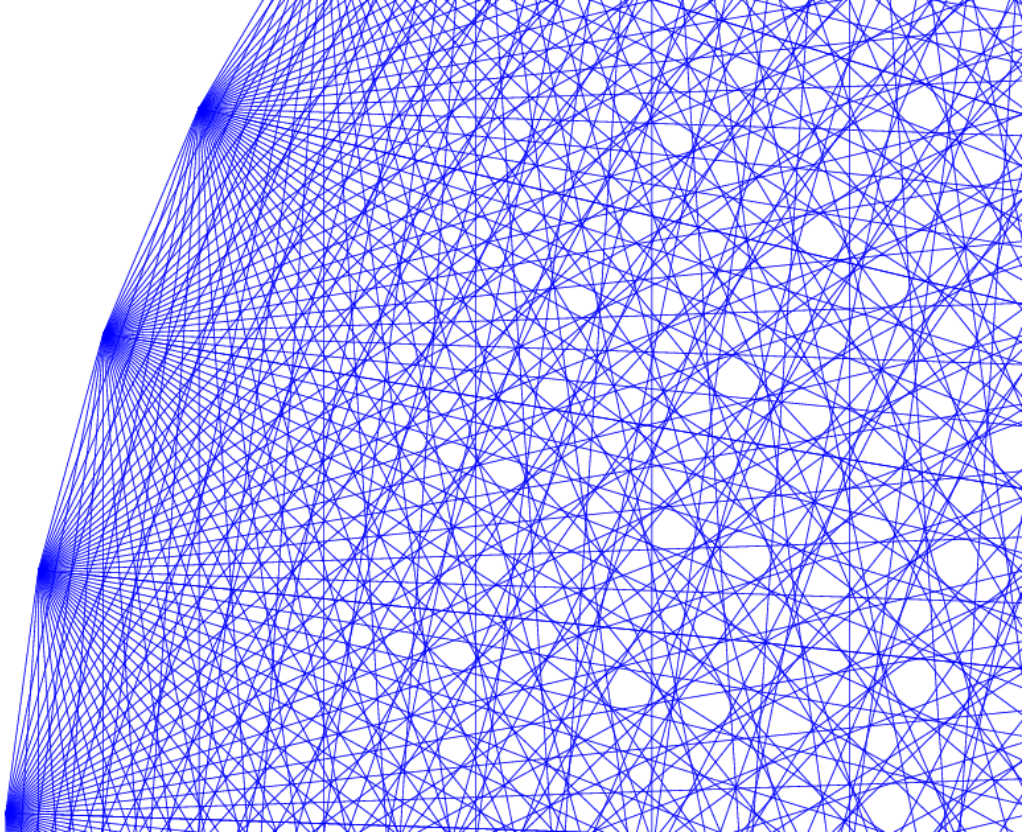
$$f(x) = \frac{(x-x_2)(x-x_3)\ldots(x-x_n)}{(x_1-x_2)(x_1-x_3)\ldots(x_1-x_n)}y_1 + \frac{(x-x_1)(x-x_3)\ldots(x-x_n)}{(x_2-x_1)(x_2-x_3)\ldots(x_2-x_n)}y_2 + \ldots + \frac{(x-x_1)(x-x_2)\ldots(x-x_{n-1})}{(x_n-x_1)(x_n-x_2)\ldots(x_n-x_{n-1})}y_n$$

```
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;
```









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?

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 krabic, které chceme efektivně naskládat do kontejnerů.

Nyní se můžeme ptát

* Jak rychle odhadnout, zda mi stačí výpočetní prostředky, co mám

* Jak se bude program chovat při růstu onoho charakteristického čísla N.
  (I ve fyzice se ale projevuje tendence počítat složitější a složitější problémy - třeba ty jednodušší už někdo vyřešil)

Řešení

* Lze zkoumat experimentálně a provést extrapolaci

* Existuje možnost to „odhadnout“ analýzou (nahlédnutím do) kódu

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 krabic, které chceme efektivně naskládat do kontejnerů.

Nyní se můžeme ptát

* Jak rychle odhadnout, zda mi stačí výpočetní prostředky, co mám

* Jak se bude program chovat při růstu onoho charakteristického čísla N.
  (I ve fyzice se ale projevuje tendence počítat složitější a složitější problémy - třeba ty jednodušší už někdo vyřešil)

Řešení

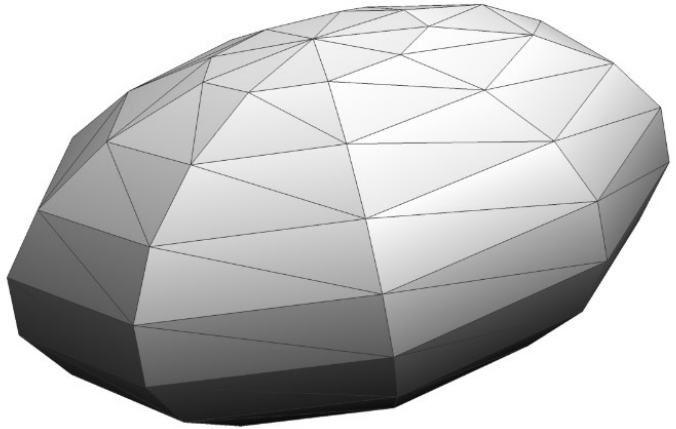
*** Lze zkoumat experimentálně a provést extrapolaci**

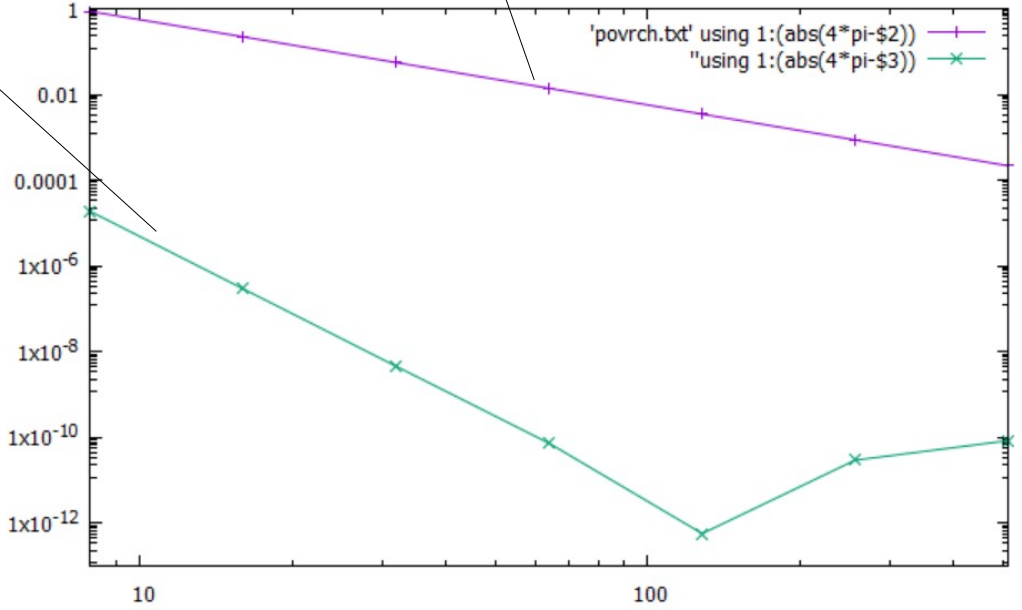
* Existuje možnost to „odhadnout“ analýzou (nahlédnutím do) kódu

$$A_{i_1,j_1,\ i_2,j_2,\ i_3,j_3} = \frac{1}{2}\left|\left(\vec{X}_{i_2,j_2} - \vec{X}_{i_1,j_1}\right) \times \left(\vec{X}_{i_3,j_3} - \vec{X}_{i_1,j_1}\right)\right|$$

$$A_{\mathcal{E}} \approx A_{m,n}(a,b,c) = \sum_{i=0}^{n-1}\sum_{j=0}^{m-1}\left(A_{i,j,\ i+1,j,\ i+1,j+1} + A_{i,j,\ i,j+1,\ i+1,j+1}\right)$$

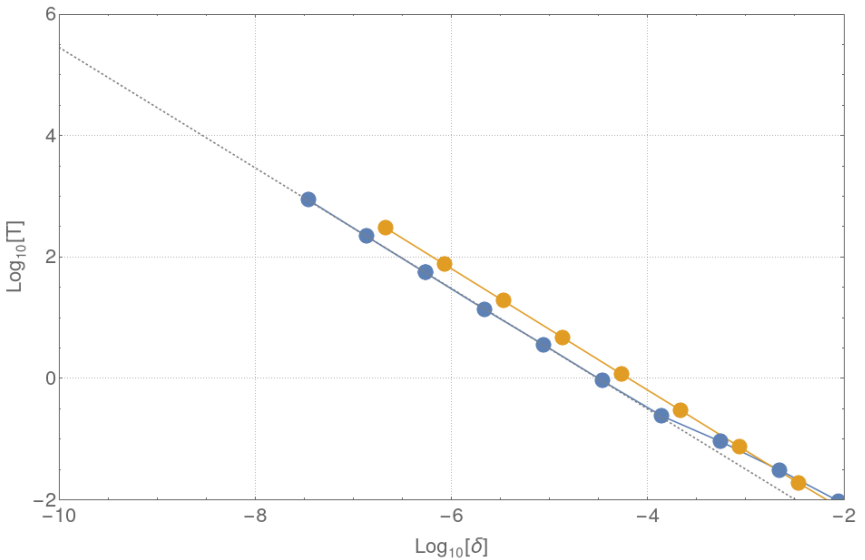
$$A_{\mathcal{E}} \approx \tilde{A}_{m,n}(a,b,c) = \frac{1}{120}\left(5A_{m,n} - 128A_{2m,2n} + 243A_{3m,3n}\right)$$





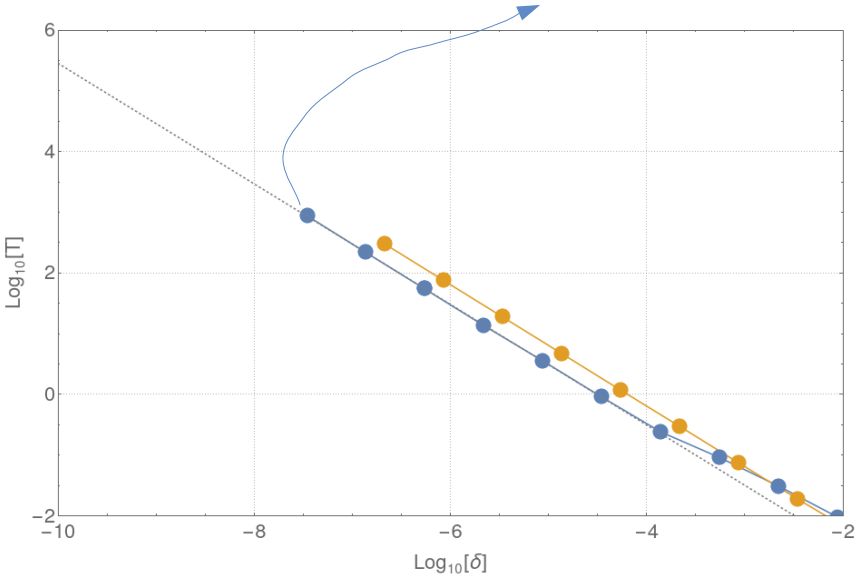
abyste s chybou $< 10^{-10}$ nalezli výsledek

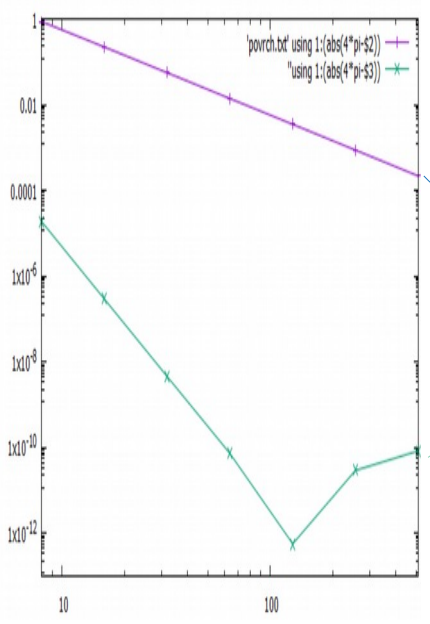
čas



chyba

čas


6
4
2
0
−2
−10
−8
−6
−4
−2
Log10[T]
Log10[δ]


'powrch.txt' using 1:(abs(4*pi-$2))
"using 1:(abs(4*pi-$3))
1
0.01
0.0001
1x10-6
1x10-8
1x10-10
1x10-12
10
100

chyba

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 krabic, které chceme efektivně naskládat do kontejnerů.

Nyní se můžeme ptát

* Jak rychle odhadnout, zda mi stačí výpočetní prostředky, co mám

* Jak se bude program chovat při růstu onoho charakteristického čísla N.
  (I ve fyzice se ale projevuje tendence počítat složitější a složitější problémy - třeba ty jednodušší už někdo vyřešil)

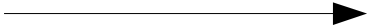
Řešení

* Lze zkoumat experimentálně a provést extrapolaci

*** Existuje možnost to „odhadnout“ analýzou (nahlédnutím do) kódu**

```
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;
```

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



T = 27 N + 50

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}

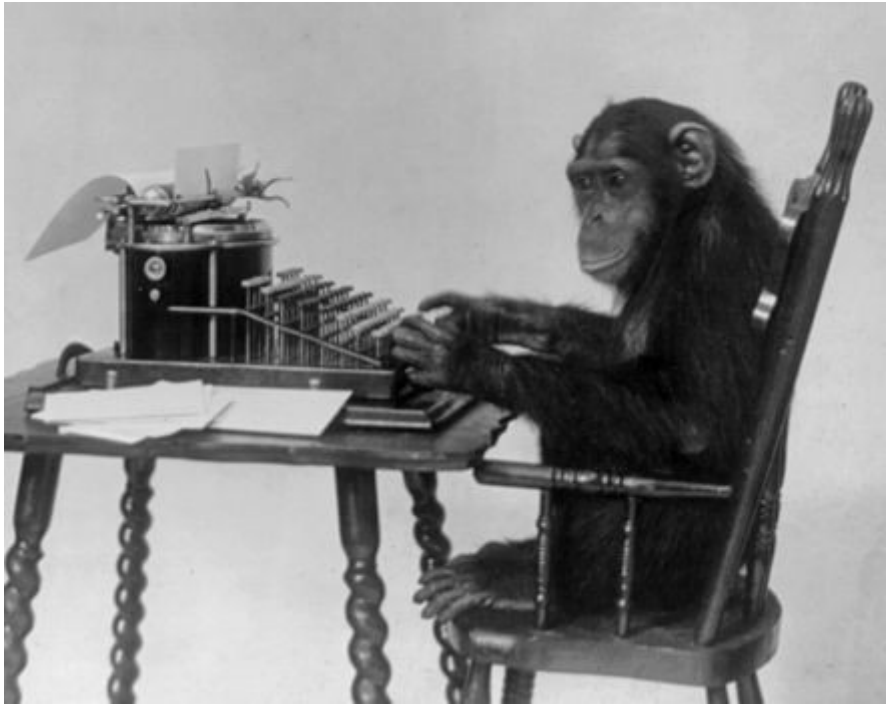
Mimochodem, od velkého třesku uplynulo asi $4x10^{26}$ nanosekund.

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

O(n!)

O(2^n)



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.

Obrázky: Wikipedia

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

$$O(N^{2)} + O(N) * (O(N) + O(N) + O(N^2)) = O(N^3)$$

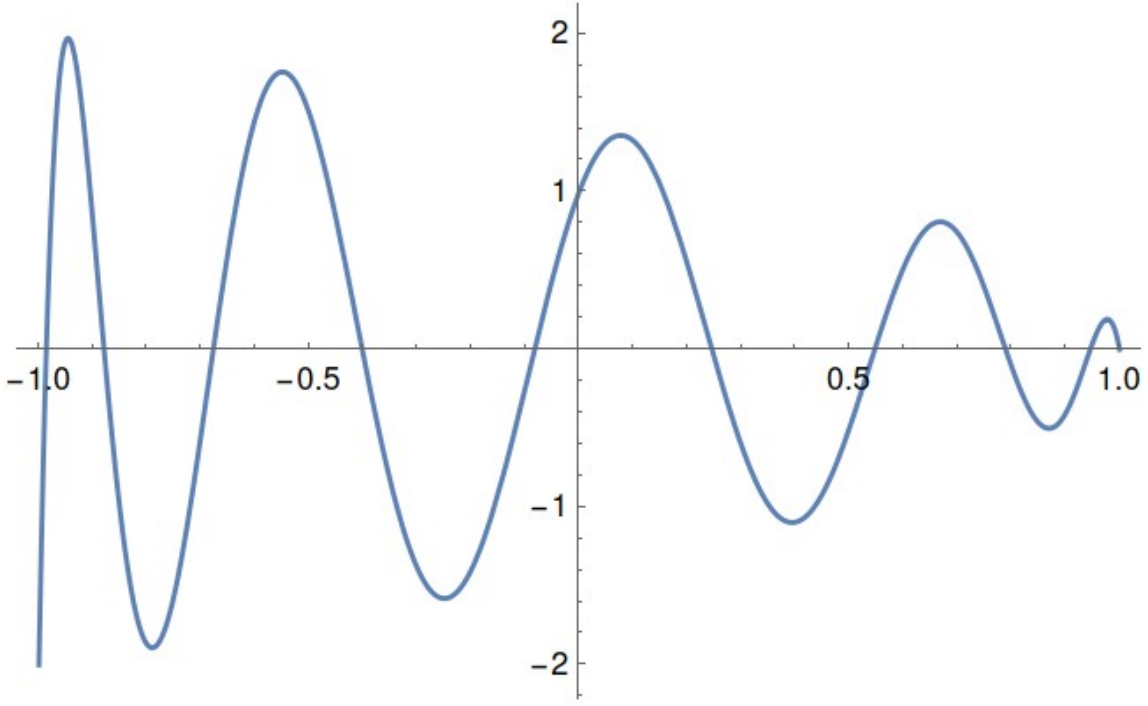
V tabulce si pak snadno najdeme, pro jaká N jde o zelený, oranžový či hnědý problém.

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.

Opět polynomy

$$1 + 9x - 50x^2 - 120x^3 + 400x^4 + 432x^5 - 1120x^6 - 576x^7 + 1280x^8 + 256x^9 - 512x^{10}$$

$$1 + x\,(9 + x\,(-50 + x\,(-120 + x\,(400 + x\,(432 + x\,(-1120 + x\,(-576 + x\,(1280 + (256 - 512x)\,x)))))))))$$



Out[48]//MatrixForm=

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

In[49]:= `prohod[1, 2]`

Out[49]//MatrixForm=

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

In[50]:= `vynasobRadek[1, 1 / 2]`

Out[50]//MatrixForm=

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

`prictikRadku[3 , - radek[1] ]`

Out[ ]//MatrixForm=

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

Soustava rovnic

- GJ eliminace soustavy

  $A\,x = b$

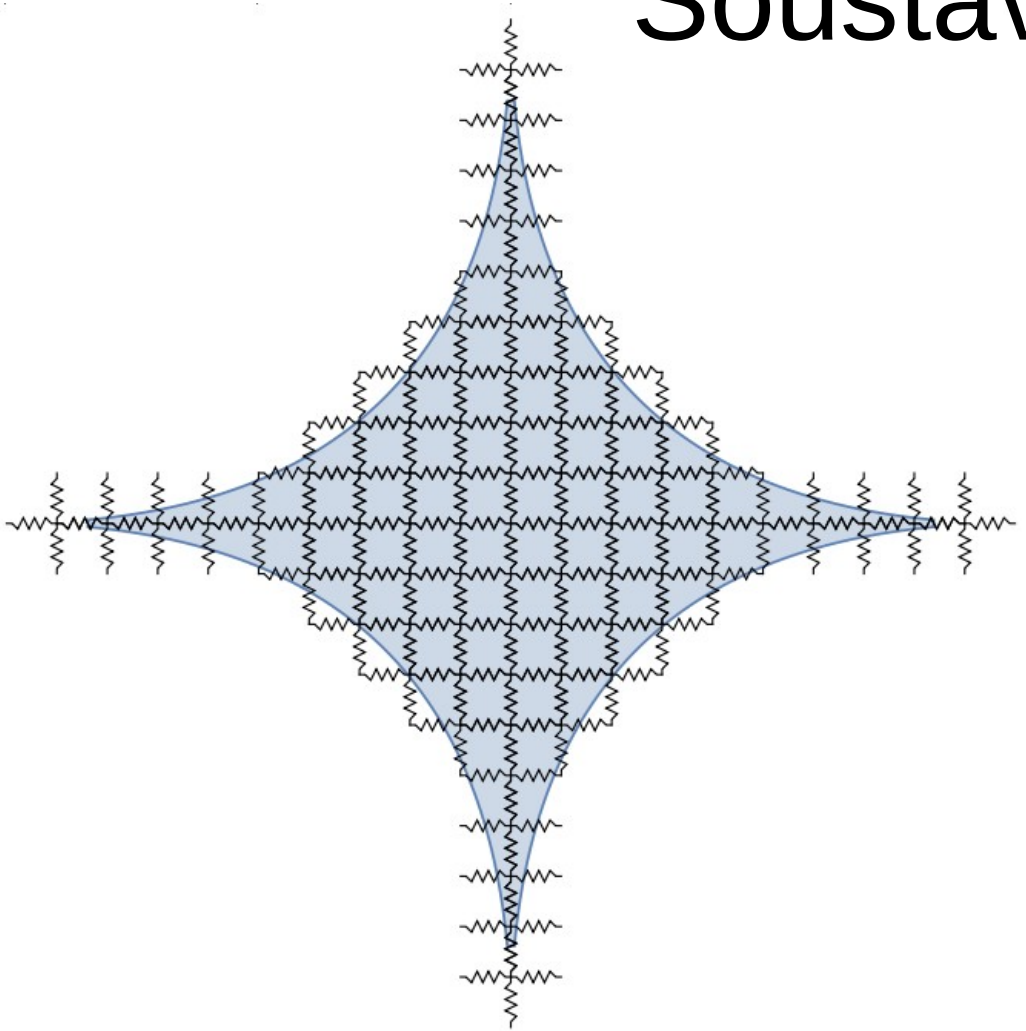
  - prohazovaní řádek
  - násobení
  - přičítání

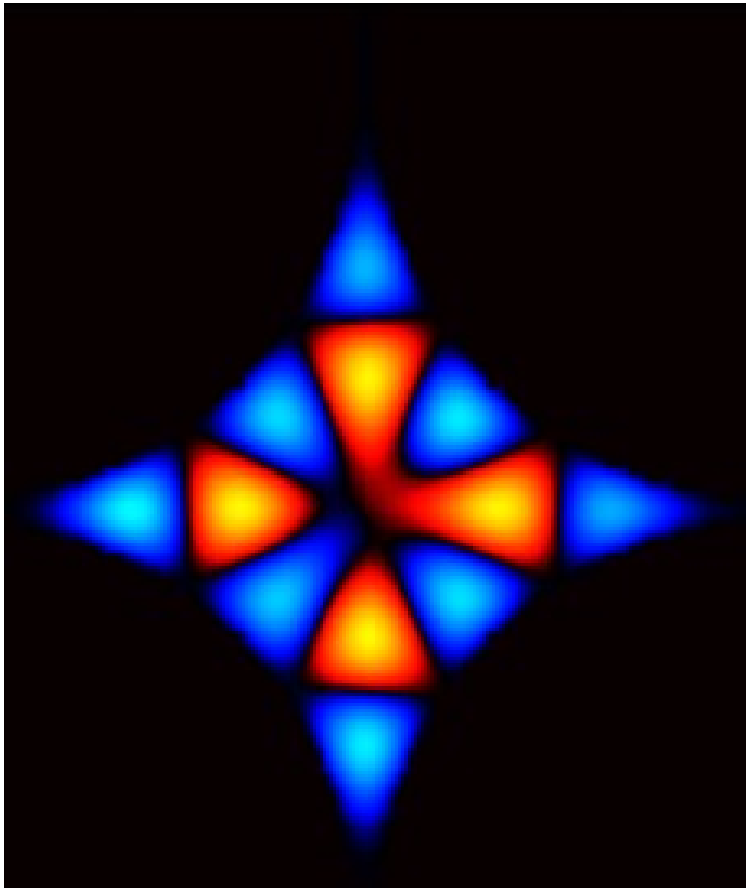
  → výsledek

  $1\,x = b'$

- Gaussova eliminace je rychlejší
- Efektivní řešení: LU rozklad

Soustava rovnic





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)

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
```

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).

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
```

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

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:

https://en.wikipedia.org/wiki/Quicksort

7	9	1	6	5	4	2	3	8

- 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.

```
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.
```

```
procedure Quicksort(var A: array of integer;
                    l, r: Integer);
var  i, j : Integer;
     swp, rozhod: integer;
begin
  rozhod := A[(l + r) div 2];
  htmlLog(l,r,-1,-1);
  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
      htmlLog(l,r,i,j);
      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;
```

54	59	71	84	60	85	54	84	42	62	64	38	43
54	59	71	84	60	85	54	84	42	62	64	38	43
43	59	71	84	60	85	54	84	42	62	64	38	54
43	38	71	84	60	85	54	84	42	62	64	59	54
43	38	42	84	60	85	54	84	71	62	64	59	54
43	38	42	54	60	85	84	84	71	62	64	59	54
43	38	42	54	60	85	84	84	71	62	64	59	54
38	43	42	54	60	85	84	84	71	62	64	59	54
38	43	42	54	60	85	84	84	71	62	64	59	54
38	42	43	54	60	85	84	84	71	62	64	59	54
38	42	43	54	60	85	84	84	71	62	64	59	54
38	42	43	54	60	85	84	84	71	62	64	59	54
38	42	43	54	60	85	84	84	71	62	64	59	54
38	42	43	54	60	54	84	84	71	62	64	59	85
38	42	43	54	60	54	59	84	71	62	64	84	85
38	42	43	54	60	54	59	64	71	62	84	84	85
38	42	43	54	60	54	59	64	62	71	84	84	85
38	42	43	54	60	54	59	64	62	71	84	84	85
38	42	43	54	59	54	60	64	62	71	84	84	85
38	42	43	54	59	54	60	64	62	71	84	84	85
38	42	43	54	54	59	60	64	62	71	84	84	85
38	42	43	54	54	59	60	64	62	71	84	84	85
38	42	43	54	54	59	60	64	62	71	84	84	85
38	42	43	54	54	59	60	64	62	71	84	84	85
38	42	43	54	54	59	60	62	64	71	84	84	85
38	42	43	54	54	59	60	62	64	71	84	84	85
38	42	43	54	54	59	60	62	64	71	84	84	85
38	42	43	54	54	59	60	62	64	71	84	84	85
38	42	43	54	54	59	60	62	64	71	84	84	85
38	42	43	54	54	59	60	62	64	71	84	84	85

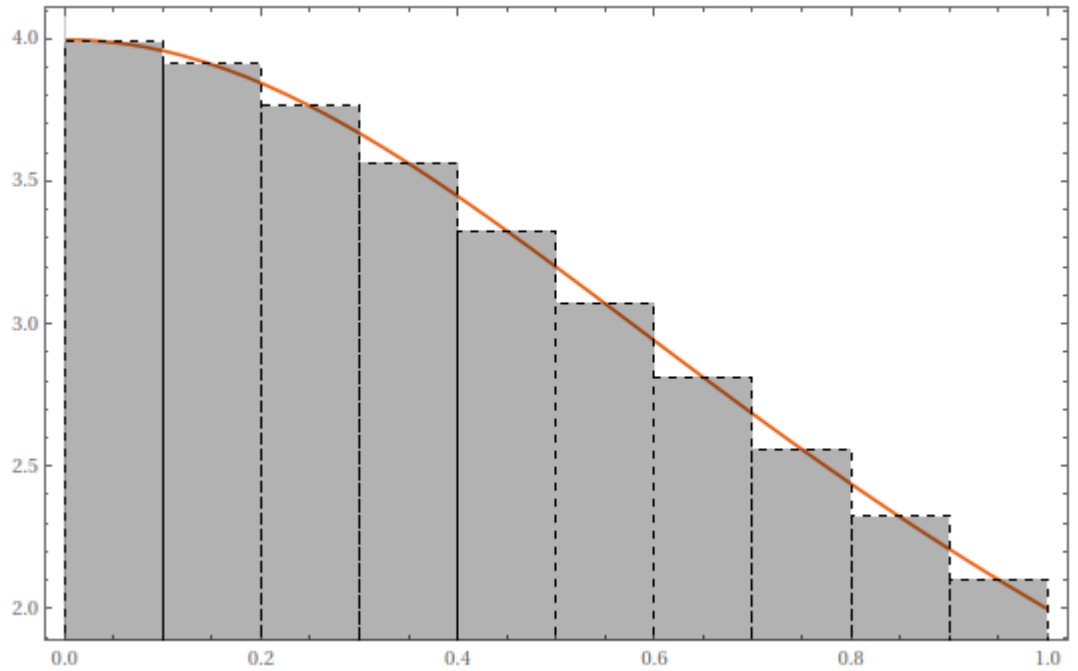
Co kdybychom chtěli třídit/řadit něco jiného ?

- co vše budeme muset psát znovu?
- neuděláme při tom chybu?

Odbočka:

Numerická kvadratura

= výpočet určitých integrálů

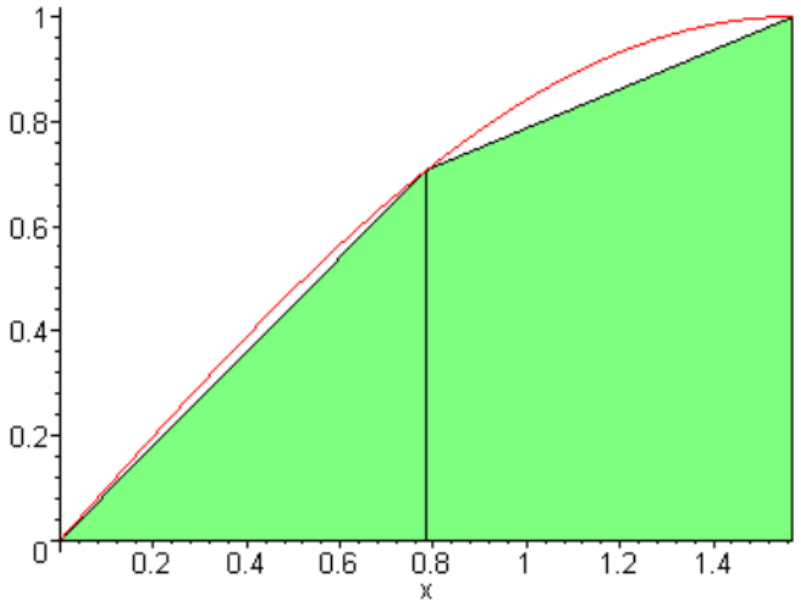


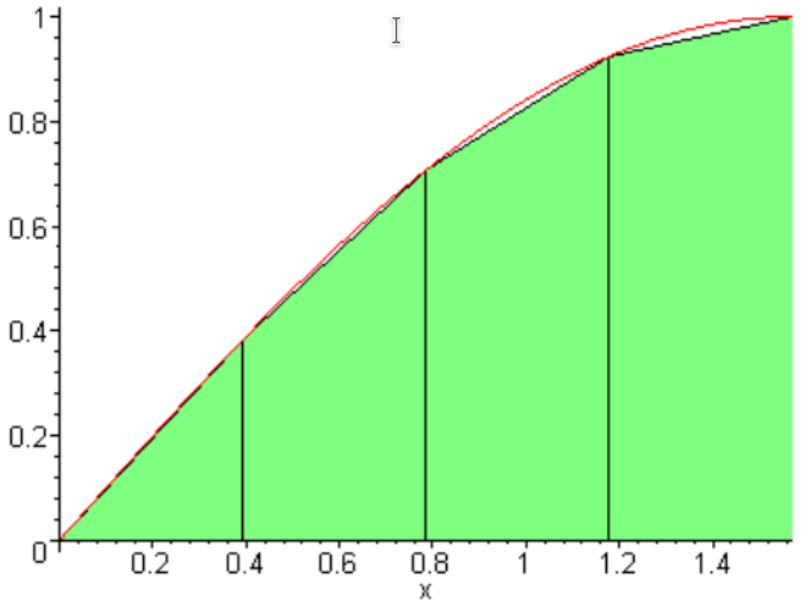
Numerická kvadratura (výpočet určitého integrálu funkce jedné proměnné)

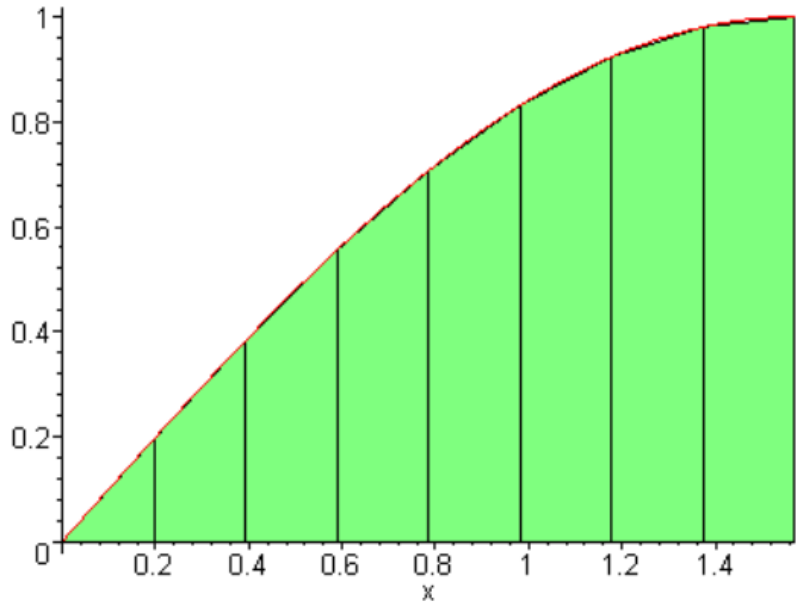
Nejdříve si ukážeme jak spočítat tzv. lichoběžníkovým pravidlem přibližnou hodnotu určitého integrálu. Řekněme, že počítáme

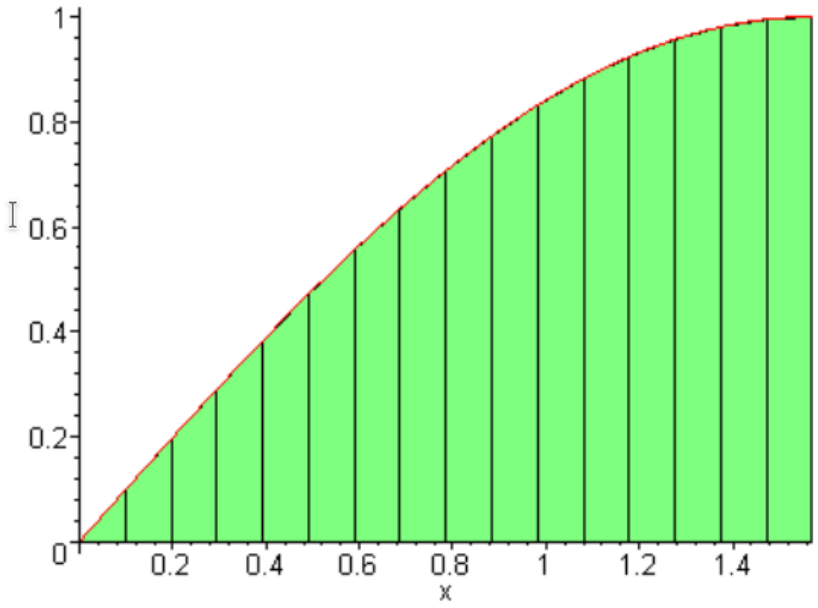
$$\int_0^{\pi/2} \sin(x)\, dx$$

Pro rozdělení integračního intervalu na 2, 4, 8 a 16 podintervalů (kde funkci nahradíme lineární interpolací) vypadá lichoběžníkové pravidlo takto:









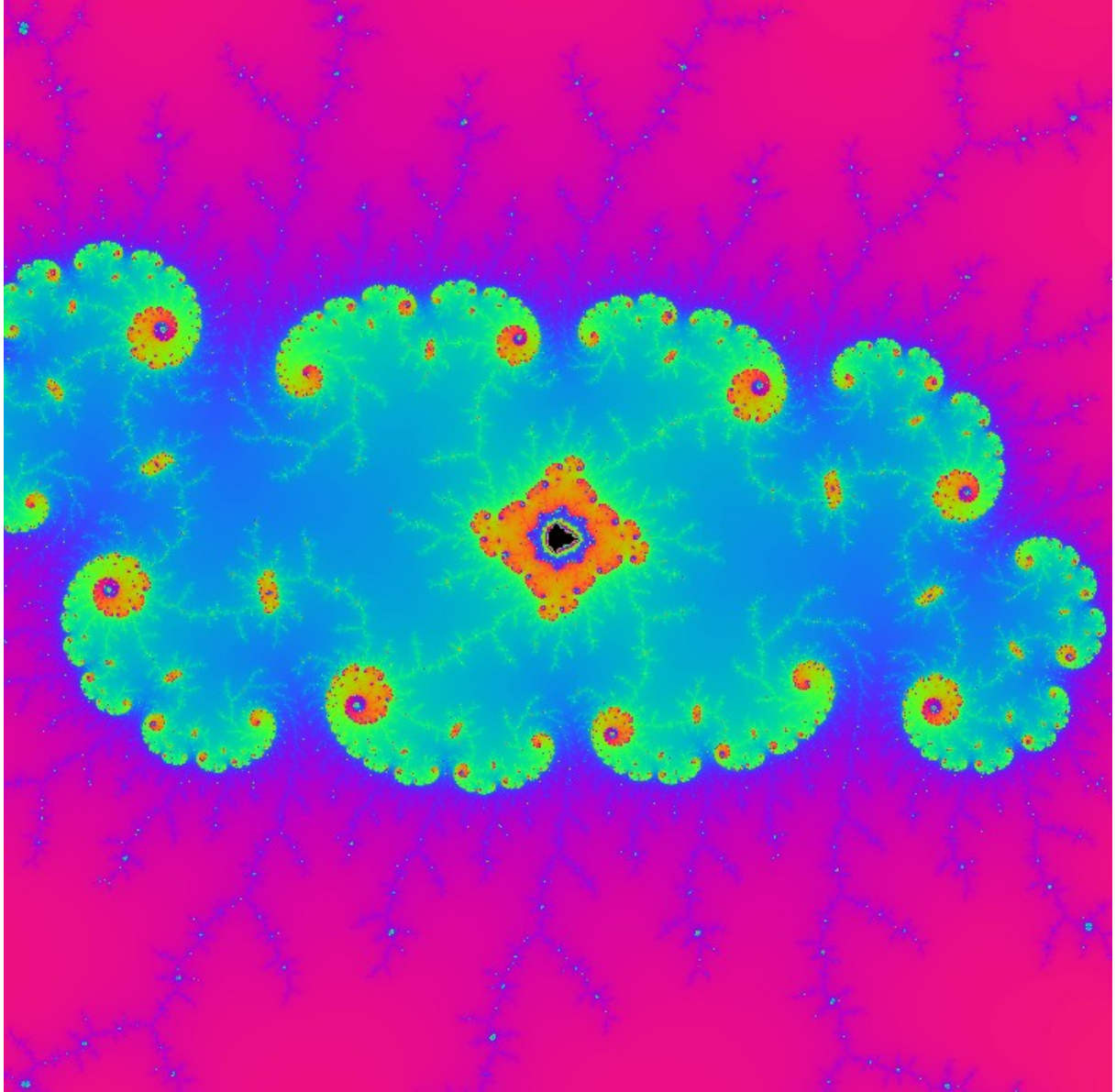
Protože určitý integrál je lineární zobrazení z prostoru funkcí na intervalu <a,b>do R, nepřekvapí že lichoběžníkové pravidlo se redukuje na lineární kombinaci funkčních hodnot, konkrétně

$$\int_a^b f(x)\,dx \doteq dx\left(\frac{1}{2}f(x_0)+f(x_1)+f(x_2)+f(x_3)+\ldots+f(x_{n-1})+\frac{1}{2}f(x_n)\right)$$

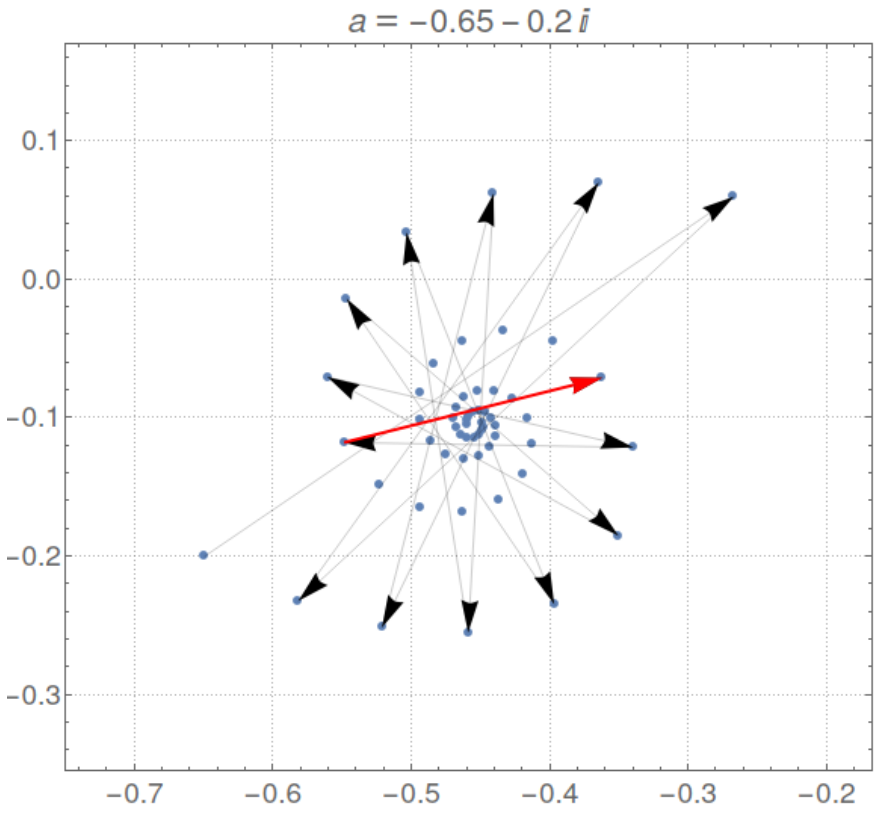
$$dx=\frac{b-a}{n} \qquad x_k=a+k\,dx \qquad x_0=a \qquad x_n=b$$

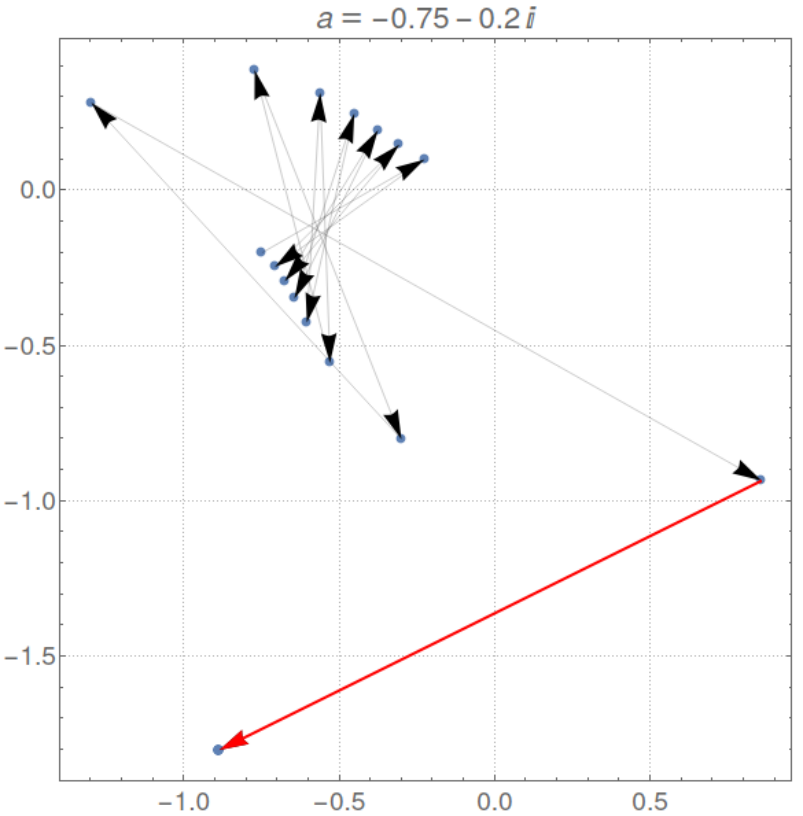
```pascal
type tRFunkce = function(x: real): real;

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;
```

```
unit  Tridicka;

interface

type  tPorovnaciFunkce = function( j,k : integer ) : integer
     tPrehazovaciProcedura = procedure( j,k : integer );

procedure  Setrid(Porovnej:tPorovnaciFunkce; Prehod:tPrehazo

implementation

procedure  Setrid(Porovnej:tPorovnaciFunkce; Prehod:tPrehazo
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;
```

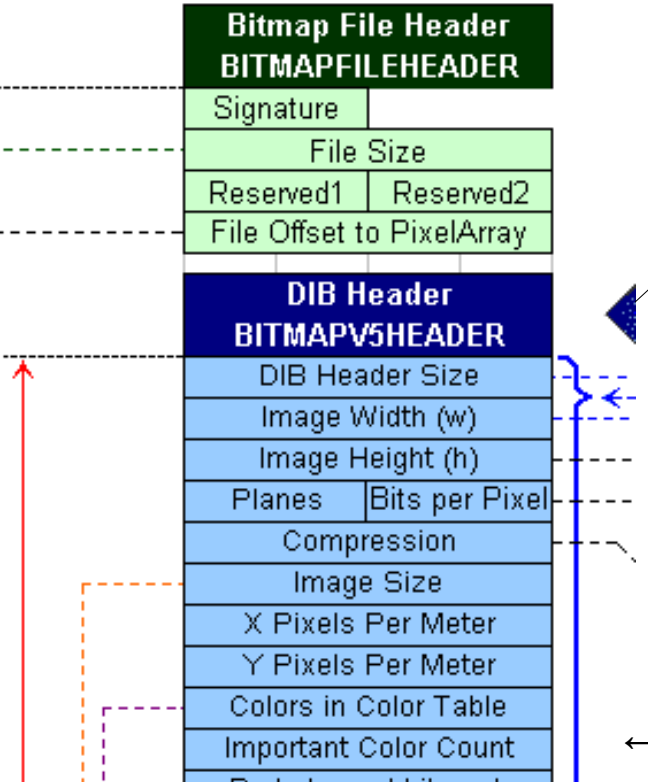


$$z_{n+1} = z_n^2 + a$$


a = −0.65 − 0.2 i
0.1
0.0
−0.1
−0.2
−0.3
−0.7
−0.6
−0.5
−0.4
−0.3
−0.2


a = −0.75 − 0.2 i
0.0
−0.5
−1.0
−1.5
−1.0
−0.5
0.0
0.5

```
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:0.12e-08),
( X:-1.25586912146251;    Y:-0.382814298251953;   w:5e-07),
( X:-1.74652309431234;    Y:1.06246014559333e-06; w:2e-08));

const KteryVyrez = 5;

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;

var   img : tRGBmatrix;
       i,j: integer;

const hImg= 1024;
      wImg= 1024;

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.
```



← wikipedia

```
//  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 ;
```

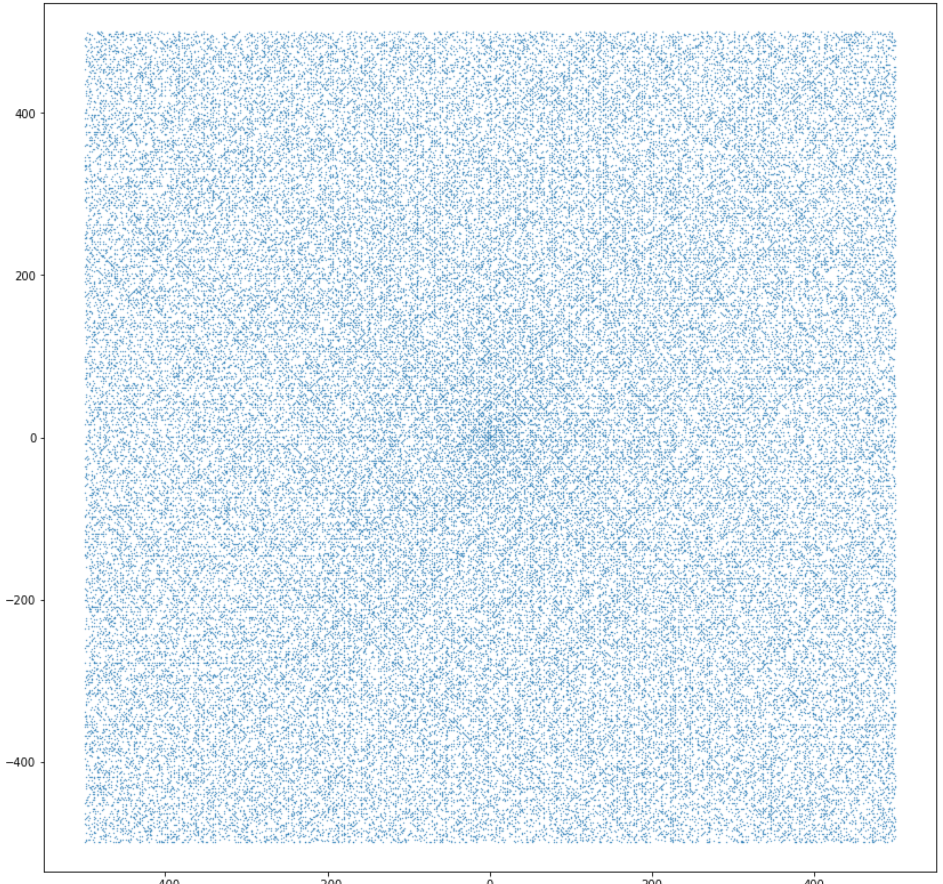
```
// https://en.wikipedia.org/wiki/BMP_file_format
type tBMP_Header = packed record
    AcsiiB        : Char    ; // B
    AcsiiM        : Char    ; // M
    Size          : LongInt ; // 54 + DataSize
    Unused1       : Word    ;
    Unused2       : Word    ;
    DataOfs       : LongInt ; // 54
    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
    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;
```

```
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;
```

Když umíte Pascal ...

... umíte programovat



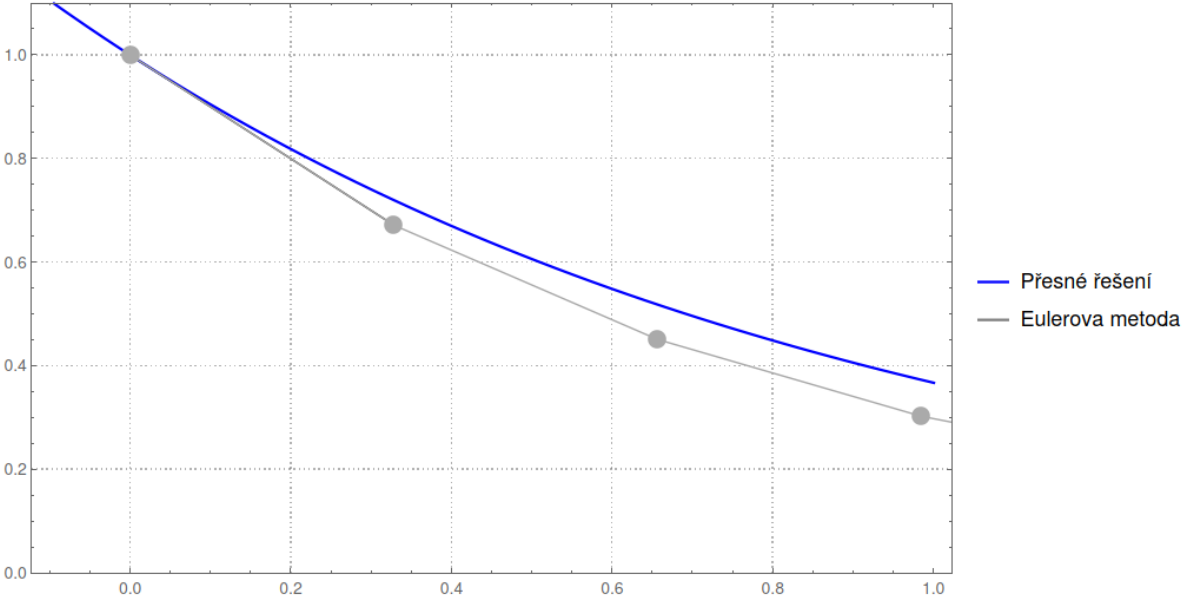
```
# program pro Vypocet NSD



a = 4477
b = 2072

print( 'NSD(',a,',',b,')=', sep='', end='' )

while not b==0:
    a,b = b, a % b



print( a )

# konec programu
```

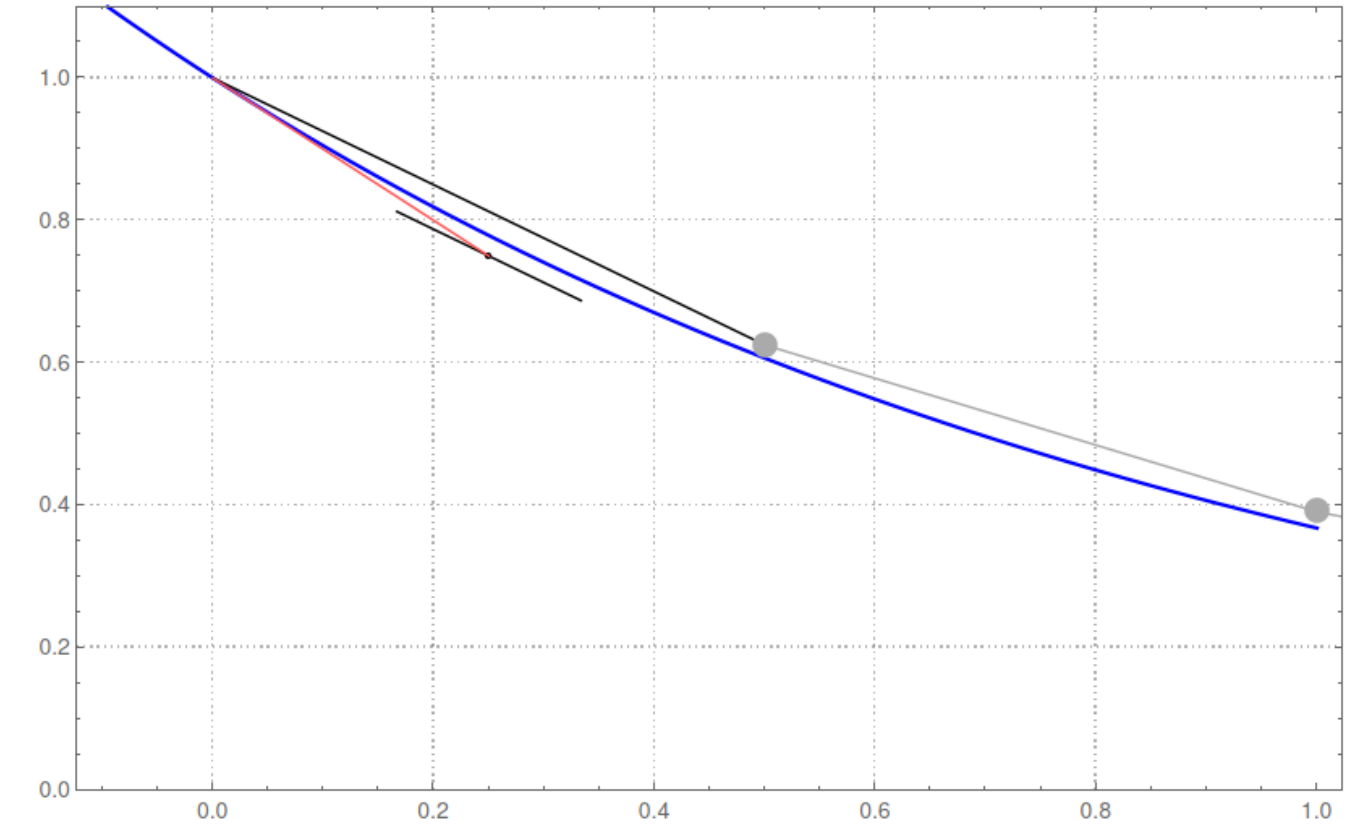
```
program VypocetNSD;

var a,b,c : integer;

begin
  a := 4477;
  b := 2072;

  write( 'NSD(',a,',',b,')=');

  while not (b=0) do begin
     c := a mod b;
     a := b;
     b := c;
  end;

  writeln( a ) ;
end.
```

```
program cv09_sito;
{$R+}
{$mode Delphi}
const N = 100000;
var maDelitele: array[2..N] of boolean;
// vyuzivame inicializace na {false, false, ....}

var p,np, x,y,j: integer;

begin
  // 1. SITO  pro vsechna p takova, ze p^2 <= N
  for p:=2 to trunc(sqrt(N)) do
    // kdyz to je prvocislo, tj
    if not maDelitele[p] then begin
      // nasobky p pocinaje np=2*p oznacim
      np:=2*p;
      while np<=N do begin
        maDelitele[np] := true;
        np := np+p
      end;
    end;

  // 2. SPIRALA
  x:=1;
  y:=1;  //souřadnice čísla 2 na spirále (např.)

  for j:=2 to N do begin
      if not MaDelitele[j] then writeln( x,' ',y);

      if x+y<=0 then
          if x>=y then x:=x+1 else y:=y-1
      else
          if x> y then y:=y+1 else x:=x-1
  end;

end.
// GNUPLOT:
//    set size ratio -1
//    plot '1.txt' pt 15 ps 0.2
```

```
N = 1000000                       # Posledni testovane cislo
primes = [False,False]+(N-1)*[True]

i = 2
while i*i<=N :                    # prochazim vsechna cisla tak, ze zacnu u dvojky
    if primes[i] :                # a kdyz je to prvocislo
        for j in range(2*i,N+1,i):# jeho nasobky
            primes[j]=False       # uz nejsou prvocisla, ze
    i += 1

import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (14,14)
datax = []
datay = []

(x,y) = (1,1)                     # souradnice cisla 2 na spirale (napr.)

for i in range(2,N+1):
  if primes[i]:
    datax.append(x)
    datay.append(y)

  # jdi na dalsi bod spiraly
  if x+y<=0:
    if x>=y:
      x+=1
    else:
      y-=1
  else:
    if x>y:
      y+=1
    else:
      x-=1

plt.axis('equal')
plt.scatter(datax,datay,s=0.1)
```

$$\frac{dU(t)}{dt} = F(t, U(t))$$



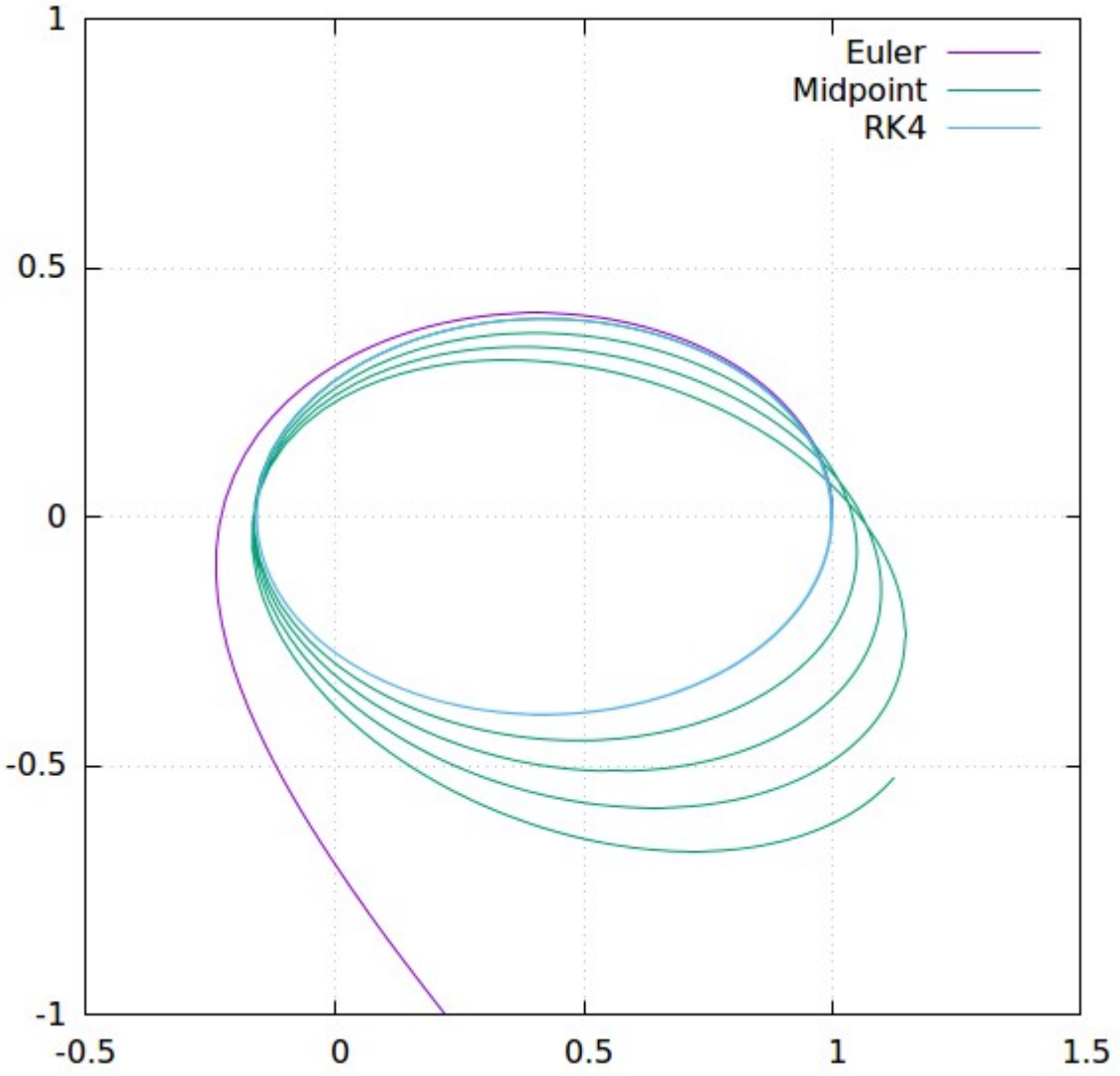


$n = 10 \quad \Delta_{mid} = -7.10^{-4} \quad \Delta_{Euler} = 0.02$

$n = 10^3 \quad \Delta_{mid} = -6.10^{-8} \quad \Delta_{Euler} = 2.10^{-4}$

$n = 10^5 \quad \Delta_{mid} = -6.10^{-12} \quad \Delta_{Euler} = 7.10^{-6}$

$n = 10^7 \quad \Delta_{mid} = -6.10^{-16} \quad \Delta_{Euler} = 7.10^{-8}$


Euler
Midpoint
RK4
1
0.5
0
-0.5
-1
-0.5
0
0.5
1
1.5