REKURZIA
Navrhnúť a implementovať rekurzívne procedúry

j.Scheme nemá zabudovanú silnú podporu cyklov;
rekurzívne volanie procedúry plní úlohu iteratívneho príkazu;
procedúra je rekurzívna, keď vyvoláva vo svojom tele aspoň jednú aktiváciu seba samej;

Výpočet faktoriálu
matematická definícia pre výpočet faktoriálu čísla:

n! = 1 (n=0)
n! = n.(n-1)! (n>0)

```
(define   (fakt  n)
     (if   (= n  0)  1
           (* n  (fakt  (-  n  1) ) ) ) )
```

2 časti rekurzívnej procedúry:
Base case
Obsahuje *limitnú podmienku rekurzie* (0!=1)
Recursive case
Obsahuje *rekurzívny predpis*, ktorý problém redukuje na 1 a viac jednoduchších problémov.
(n!=n.(n-1)!

Výpočet rekurzívnej procedúry je vykonávaný vždy v 2 fázach:

Fáza navíjania

Predpis rekurzie sa postupne aplikuje, problém sa redukuje na jednoduchšie problémy. Fáza navíjania končí dosiahnutím limitnej podmienky rekurzie, kedy možno problém vyriešiť bez použitia rekurzie.

Fáza odvíjania

Riadenie sa postupne vracia až na miesto prvej aktivácie procedúry. V rámci tejto fázy sa častokrát konštruuje výsledok volania procedúry.

Trasovanie výpočtu

Dôležité pre ladenie programu.

Mnohé prekladače jazyka umožňujú zapnúť sledovanie lubovolnej procedúry prostredníctvom *trace* procedúry. Procedúra *untrace* po vyvolaní vypne sledovanie.

Ak zapneme sledovanie našej procedúry *fakt*, po jej vyvolaní máme nasledujúci postup. K programu súčasne musíme priložiť knižnicu, ktorá obsahuje podporu trasovania výpočtu (u nás to realizujeme príkazom *require*):

```
>  (require  (lub “trace.ss“) )
> (trace  fakt)
(fakt)
 >  (fakt  3)
|  (fakt  3)
|    (fakt  2)
|    | ( (fakt  1)
|    |    (fakt  0)
|    |    1
```

| | 1
| 2
| 6
6

Súčet čísiel v intervale

Napíšte procedúru, ktorá spočíta všetky celé čísla na danom intervale.
Nech a, b sú celé čísla, výsledkom procedúry je nasledujúca hodnota:
a + (a+1) + (a+2) +...+ (b-1) + b

Hraničnou podmienkou rekurzie je situácia, kedy a > b. Na takomto intervale neležia žiadne celé čísla, tzn. výsledkom je 0. V prípade, že a ≤ b, problém je možné zjednodušiť – vezmeme číslo a ktoré pričítame k súčtu všetkých čísiel na int. <a+1, b>. takýmto spôsobom dostaneme rekurzívny predpis. V každom ďalšom volaní procedúry sa nám parameter a zvyšuje o jednotku, tzn. v konečnej dobe parameter a dosiahne a aj prevýši parameter b, ktorý sa nemení. Rekurzívnu procedúru implementuje tento kód:

```
(define  (sucet-od-do  1  100)
   (if  (>  a b)  0
        (+  a  (sucet-od-do  (+  a  1)  b) ) ) )
```

Program môžeme spustiť na počítači a overiť si jeho beh trasovaním výpočtu:

```
> (sucet-od-do 1 100)
5050
> (trace sucet-od-do)
(sucet-od-do)
> (sucet-od-do 1 5)
| (sucet-od-do 1 5)
|  (sucet-od-do 2 5)
|  | (sucet-od-do 3 5)
|  |  (sucet-od-do 4 5)
|  |  | (sucet-od-do 5 5)
|  |  |  (sucet-od-do 6 5)
|  |  |  0
|  |  | 5
|  |  9
|  | 12
|  14
| 15
15
```

Modifikácie procedúry

1. Spočítavať budeme iba každé druhé číslo v danom intervale - vzor typickej rekurzívnej procedúry.

```
(define (sucet2-od-do a b)
  (if (> a b) 0
      (+ a (sucet2-od-do (+ a 2) b) ) ) )

> (sucet2-od-do 1 100)
2500
> (trace sucet2-od-do)
```

```
(sucet2-od-do)
>  (sucet2-od-do  1  5)
| (sucet2-od-do  1  5)
| (sucet2-od-do  3  5)
|   |  (sucet-od-do  5  5)
|   |    (sucet-od-do  7  5)
|   |   | (sucet-od-do  5  5)
|   |  0
|   | 5
|  8
| 9
9
```

2. modifikujeme sucet-od-do: procedúra ktorá spočítava všetky párne čísla v intervale <a, b>. Rekurzívna procedúra môže obsahovať viac rekurzívnych predpisov.

```
(define  (sucet-parne  a  b)
  (cond  ( (>  a  b)  0)
        (else  (+ a  (sucet-parne  (+  a  1)  b) ) ) ) )
```

Pri sčítaní iba párnych čísiel – potrebujeme rozlíšiť 2 prípady:

a je párne (even zabudovaný predikát)
číslo pričítame k výsledku (ako predtým vo vetve else)

a nie je párne (nepričítame číslo), pokračujeme v pričítavaní čísiel od a+1 do b.

```
(define   (sucet-parne   a  b)
  (cond  ( (>  a  b)   0)
         ( (even? a)  (+  a (sucet-parne (+ a  1)  b) ) )
         (else  (sucet-parne   (+  a  1)   b) ) ) )
```

Voľba jedného z dvoch rekurzívnych predpisov závisí od hodnoty čísla a.

Lissajousove krivky

vykresľuje ich bod ktorý sa pohybuje harmonickým pohybom súčasne v 2och na seba kolmých smeroch
Tvar krivky je daný pomerom frekvencie a fázového posunu a závisí na 3 faktoroch:
f_x - frekvencia, s akou sa bod pohybuje v jednom smere;
f_y – frekvencia, s akou sa pohybuje bod v druhom smere
ϕ – fázový posun, ktorý vyjadruje, ako sa pohyb v jednom smere „predbieha“ v porovnaní s pohybom v druhom smere;

matematické vyjadrenie súradníc bodu:
$x(t) = \sin(f_x t)$
$y(t) = \sin(f_y t + \phi)$
parameter t reprezentuje čas, ak $t=2\pi$, krivka opísaná bodom bude úplne vykreslená.

Vykresľovanie kriviek na obrazovke

Využijeme grafickú knižnicu. Nasledujúci kód reprezentuje pohyb bodu na obrazovke. Pred spustením kódu je potrebné pridať knižnicu *canvas* ako teachpack do jazyka Scheme. Kód napíšeme do okna definícií. Tieto procedúry pracujú pomocou tzv. vedľajšieho efektu. Ďalej sa využíva fakt, že tieto procedúry môžu obsahovať viac ako jeden výraz.

```
(define (lissajous-loop t fx fy phi)
  (graphics-line-to (sin (* fx t) )
                    (sin (+ (* fy t) phi) ) )
```

```
  (lissajous-loop   (+ t 0.0001)   fx   fy   phi) )

(define  (lissajous  fx  fy  phi)
  (graphics-init   500  500  -1  -1  1  1)
  (graphics-move-to   0  (sin  phi) )
  (lissajous-loop  0  fx  fy  phi) )
```

Procedúra *lissajous* akceptuje 3 parametre – hodnoty frekvencií f_x, f_y a fázový posun ϕ. Procedúra beží v nekonečnom v cykle, pre zastavenie musíme použiť „Break“ tlačidlo. Spustiť procedúru môžeme s týmito parametrami:

```
> (lissajous  2  3  1)
```

Na obrazovke sa vykreslí obrazec.

Experiment: Máme kombináciu parametrov:

f_x	f_y	phi
1	1	0
1	1	0.1
1	2	0
1	2	0.2
2	3	0
2	3	0.3
3	4	0
3	4	0.4
4	5	0
4	5	0.5
5	6	0
7	8	0
3	5	0

Zastavenie nekonečného cyklu

Pokiaľ sú frekvencie f_y, f_y celé kladné čísla, celá Lissajousova krivka sa vykreslí v čase $t=2\pi$. Budeme chcieť overiť toto tvrdenie a upraviť program takým spôsobom, aby sa nekonečný cyklus zastavil, keď platí $t\geq 2\pi$. V našom kóde chýba základný prípad. Kedykoľvek zavoláme procedúru *lissajous-loop* procedúra vyvolá sama seba. Žiadny mechanizmus tu neexistuje ktorý by zastavil ukončil.

Pri každom novom vyvolaní procedúry *lissajous-loop* sa zvýši hodnota parametra t o malý prírastok. Pokiaľ zadanie hovorí o zastavení v čase $t\geq 2\pi$, pravdepodobne sa bude jednať o tento parameter. Do programu je preto potrebné pridať podmienku, ktorá ohraničí rekurzívne volanie iba na situácie, kedy $t\leq 2\pi$, pričom tým efektívne zastaví nekonečný cyklus.

Príklad A, B, C

Zastavenie prekresľovania kriviek

Pri používaní modifikovanej verzie procedúry z predchádzajúcej kapitoly zistíme, že v niektorých prípadoch sa program zastaví ihneď po vykreslení krivky, zatiaľ čo inokedy program prekreslí krivku viac než jedenkrát cez seba. Napr.pri zadaní f_x=5, f_y=7, *phi*=1 je krivka vykreslená iba raz. Ak f_x=6, f_y=8,*phi*=1 je krivka vykreslená dvakrát. Predpokladáme, že frekvenciu tvoria celé kladné čísla.

Vyplňte nasledujúcu tabuľku na základe experimentoania so systémom.

fx	fy	phi	Počet prekreslení
3	5	1	
2	7	0	
5	7	1	
2	4	0	
6	8	0	
6	8	2	
6	9	0	
5	15	3	

Príklad D, E, F, G, H

Vedľajší efekt procedúr, korytnačia grafika

Hlavný a vedľajší efekt procedúry

Procedúry pracovali prostredníctvom svojho hlavného efektu (main effect). Hlavný efekt

procedúry predstavuje jej výsledok, jej návratovú hodnotu, ktorú je možné ďalej vaužiť vo výpočtoch. Návratovú hodnotu procedúry je napr.možné uložiť do premennej alebo ju použiť ako vstup do inej procedúry.
Procedúra *sucet-od-do* poskytuje ľahké zrátanie čísiel od 1 do 5, alebo súčet čísiel od 10 do 15. Takto získané hodnoty môžeme sčítať:

```
> (sucet-od-do  1  5)
15

> (sucet-od-do  11  15)
65

> (+ (sucet-od-do  1  5)  (sucet-od-do  10  15) )
80
```

Táto procedúra pracovala prostredníctvom svojho hlavného efektu. *Hlavný efekt* v jazyku Scheme je jednoducho daný výsledkom vyhodnotenia (posledného výrazu) tela procedúry. V iných programovacích jazykoch je obvykle zvýraznený kľúčovým slovom *return.* Okrem svojho vlastného efektu môže mať procedúra aj *vedľajší efekt* (side effect). Ako svoj vedľajší efekt môže npr.procedúra vypísať správu na obrazovku, vykresliť obrázok, zapísať súbor na disk atď. Programátor by mal rozlišovať medzi hlavným a vedľajším účinkom procedúr a snažiť sa obmedziť použitie vedľajšieho efektu na najnutnejšie situácie. Najobvyklejší prípad použitia vedľajšieho efektu je potom vo všeobecnosti realizácia vstupov a výstupov.

Upravme kód procedúry *sucet-od-do* tak, že na jej začiatok pridáme príkazy *display, newline.* Procedúra *display* akceptuje jeden parameter a vypíše ho na obrazovku. Táto procedúra pracuje svojim vedľajším efektom, preto nás nezaujíma, čo vracia táto procedúra, a výsledok tejto procedúry nevyužívame v ďalších výpočtoch.

Procedúra *newline* vypíše na obrazovku znak nového riadku, takže ďalšie výpisy začínajú opäť od ľavého okraja okna.

```
(define  (sucet-od-do  a  b)
  (display  "Scitam cislo")
  (display  a)
  (newline)
  (if  (>  a b)  0
       (+  a  (sucet-od-do  (+  a 1)  b) ) ) )
```

Pokiaľ znova vyvoláme takto upravenú procedúru, uvidíme jej hlavný aj vedľajší efekt.

```
>  (sucet-od-do  1  5)
Scitam  cislo  1
Scitam  cislo  2
Scitam  cislo  3
Scitam  cislo  4
Scitam  cislo  5
Scitam  cislo  6
15
```

Hlavný efekt je návratová hodnota 15, zatiaľ čo vedľajším efektom sú výpisy v okne prostredia interakcií jazyka Scheme. Efekt vyhodnotenia výrazu je:

```
 > (+ (sucet-od-do 1 5) (sucet-od-do 11 15)
Scitam cislo 1
Scitam cislo 2
Scitam cislo 3
Scitam cislo 4
Scitam cislo 5
Scitam cislo 6
Scitam cislo 11
Scitam cislo 12
Scitam cislo 13
Scitam cislo 14
Scitam cislo 15
Scitam cislo 16
80
```

V tomto príklade sme tiež zistili, že telo procedúry môže obsahovať viac než jeden výraz jazyka Scheme. Pokiaľ nastane takáto situácia, sú jednotlivé výrazy vyhodnocované postupne. Hlavný efekt procedúry je daný ako výsledok vyhodnotenia posledného výrazu. Všetky ostatné výrazy majú význam len pre svoj vedľajší efekt. To býva zdrojom častých chýb, keď sa uvádzajú do tela procedúry výrazy bez vedľajšieho efektu. Tento nefunkčný kód pre výpočet súčtu druhých mocnín je typický príklad chybného použitia:

```
(define (sucet-stvorca x y)
   (sqr x)
   (sqr y)
   (+ x y) )
```

Prvé dva výrazy v tele procedúry síce počítajú hodnotu druhej mocniny x a y, ale tieto výsledné hodnoty nie sú nikde uchovávané a nie sú použité pre další výpočet. Správna verzia je nasledovná:

```
(define (sucet-stvorca x y)
   (+ (sqr x) (sqr y) ) )
```

begin špeciálna forma
umožňuje zoskupenie niekolkých príkazov do jedného bloku.

```
> (sucet-od-do 1 5)
Scitam cislo 1
Scitam cislo 2
Scitam cislo 3
Scitam cislo 4
Scitam cislo 5
Scitam cislo 6
15
```

Forma *begin* akceptuje ľubovolné množstvo výrazov, ktoré postupne vyhodnocuje. Hlavným

efektom, tzn. návratovou hodnotou je výsledok vyhodnotenia posledného výrazu, napr. môžeme kód upraviť nasledujúcim spôsobom:

```
(define  (sucet-od-do  a  b)
  (if  (>  a  b)
       (begin
          (display  “Hotovo“)
          0)
       (begin
          (display  “Scitam cislo“)
          (display  a)
          (newline)
          (+  a  (sucet-od-do  (+  a 1)  b) ) ) ) )
```