```
Uprava podla r. 1996, 1999

- úvod, história,
     literatúra:
     H. Abelson, G.J.Sussman, J.Sussman:
     Structure and Interpretation of Computer Programs,
     MIT Press 1985
- implementácia XSCHEME:
     server barbora: N:\msdos\xscheme\.
- interpret, read-eval-print cyklus
- (exit)
     ukončenie programu, použiť v zátvorkách

- vyjadrovacie prostriedky jazyka pre:
  - elementárne výrazy
  - možnosti kombinácie
  - možnosti abstrakcie
- dva druhy objektov - môžu mať zhodnú syntax
  - procedúry (program)
  - data

- základy jazyka:
- čísla, aj reálne
- zoznamy ako volania funkcií
- +, -, *, /,  aj viacargumentové
pr.: >(* (+ 2 2 2) (+ 3 3 3 ))
     54
- vyhodnocovanie, volná syntax, medzery ako oddelovače
     každý výraz má výslednú hodnotu, ktorá sa vracia

- premenné, mená
- špeciálna forma define pre definíciu hodnôt (globálnych)
     premenných
(define delka 7)
(define obvod (* delka 4))
     ; okamžité vyhodnotenie, vykoná sa: obvod := 28
- okolie (enviroment) ako ,,abstraktná" pamäť
  - obsahuje dvojice meno_premennej - hodnota
  - v okolí sa zhromažďujú väzby z "define"-formy, väzby
formálnych parametrov procedúr na predávané hodnoty a väzby
určené v "let"-forme

- vyhodnocovanie výrazov:
  - najprv sa vyhodnotia podvýrazy (rekurzívne)
```

```
  - aplikuje sa procedúra  (hodnota najľavšieho prvku - operátor)
     na hodnoty ostatných argumentov - operandy
- vyhodnocovanie zloženého výrazu - odspodu, príklad
  - vyhodnocovanie čísel, vstavaných operátorov, ostatných mien
  - špeciálne formy (napr. define, exit) a ich argumenty sa
     vyhodnocujú špeciálne
  - vyhodnocovanie závisí na okolí

- definície (užívateľských) procedúr
  schematicky: (define (<meno> <formálne_parametry>) <telo>)
  pr. (define (kvadrát x) (* x x))
      (define (l1_norma x y) (abs (- x y)))

- príklad zložitejšieho vyhodnocovania:
  - v SCHEME sa používa:
     substitučný model,
     odpovedá volaniu hodnotou, pretože všetky parametry vstupujú
           do fnc. vyhodnotené
pr.: (define (sucet_kvadratov  x y) (+ (kvadrat x) (kvadrat y)))
     (define (f a) (sucet_kvadratov (+ a 1) (* a 2)))
postup vyhodnocovania:
(f 5)                  ; volanie
(sucet_kvadratov (+ 5 1) (* 5 2))  ; rozvinutie funkcie f
     ; a nahrada formálneho param. a aktuálnou hodnotou 5
(sucet_kvadratov 6       10     )  ; vyhodnotenie arg.
(+ (kvadrat 6) (kvadrat 10))       ; rozvinutie fnc. sucet_kvadratov
(+ (* 6 6 )    (* 10 10)   )       ; rozvinutie fnc. kvadrat, 2x
(+ 36          100         )       ; redukcia vstavanej fnc.
136                                ; vysledok

- iný vyhodnocovací model (používaný v iných funcionálnych
           jazykoch):
  expanzia až na elementárne operátory, potom redukcie (tzv.
     volanie menom)
  - nevýhody volania menom: viacnásobné vyhodnocovanie spoločných
     podvýrazov
  - výhoda - lepšia konečnosť
     na tom istom programe sa volanie hodnotou zacyklí, ale volanie
     menom skončí: napr. ak sa zacyklí vyhodnocovanie argumentu,
     ktorý sa potom nepoužíva.

- špeciálne formy:
  - pre podmienený príkaz:
  (cond  (<podm1> <výraz1>)
```

```
        (<podm2> <výraz2>)
        ...
        (<podmN> <výrazN>))
  - podmienky sa postupne zhora vyhodnocujú, prvá, ktorá sa splní
    nech je podm_i, potom sa vyhodnocuje výraz_i a jeho hodnota
    je hodnotou celej podmienky
  - postupné vyhodnocovanie podmienok -
    je dôležité, ale je to odchýlenie od deklarativity
  - príklad (define (abs x)
                (cond ((> x 0) x)
                      ((= x 0) 0)
                      (else    (- x))
           )    )
  - pravdivostné hodnoty - false je (), nil, #f
    - iné hodnoty sa chápu ako true
  - posledná záchytná podmienka - (ELSE ...)
  - ak nie je splnená žiadna podmienka, chovanie závisí na
implementácii:
     1. cond vráti nil
     2. výsledok je nedefinovaný (tj. náhodný)
     3. behová chyba

- vstavané fce <, =, >, <=, >=

- (if <podm> <výraz> <alternatíva>)
  - podmienky ako stráže - dôležitosť postupného vyhodnocovania v
    podmienkach
  pr.: (define (moje_if ...) ...)
    - moje_if vyhodnotí obidve vetvy, aj keď niektorá z nich nemá
          zmysel (tj. spôsobí behovú chybu), prípadne sa zacyklí
    - pr.: faktorial(0) = 0 * faktorial(-1)

- vstavané funkcie and, or - na ľubovoľnom počte argumentov
- (not arg)

Pr. zložitejší:
Newtonova iteračná metóda výpočtu odmocniny x.
x1 (počiatočný odhad) = 1
ďalší odhad: x_(i+1)= priemer(x_i, x/x_i)
požadovaná presnosť: |odhad - x| < 0.001

(define (sqrt x)
  (define (presny? odhad x)                    -- pomocné fnc.
    (< (abs (- (kvadrat odhad) x)) .001 ))
```

```
  (define (priemer x y)
    (/ (+ x y) 2))
  (define (zlepsi odhad x)
    (priemer odhad (/ x odhad)))
  (define (sqrt_iter odhad x)
    (if (presny? odhad x)
        odhad
        (sqrt_iter (zlepsi odhad x) x)))
 (sqrt_iter 1 x))                              -- telo
```

- define má blokovú štruktúru
  - pomocné proc. sú schované
- viazané (~lokálne) x volné (~globálne) premenné
  - formálne parametre
- vyhodnocovanie voľných premenných
  - závisí na "nadradenom" okolí

- programátorské techniky:
- (lineárna) rekurzia:

```
  (define (faktorial n)
    (if (= n 1)
        1
        (* n (faktorial (- n 1)))))
```

  - rekurzia
  - používa zásobník, spotreba pamäti O(n)
  - interný postup výpočtu je tento (vyhodnocovanie if preskakujem):

```
    (f 4)
    (* 4 (f 3))
    (* 4 (* 3 (f 2)))
    (* 4 (* 3 (* 2 (f 1))))
    (* 4 (* 3 (* 2 1)))
    (* 4 (* 3 2))
    (* 4 6)
    24
```

- iterácia:

```
  (define (faktorial n)
    (f_iter 1 1 n))
  (define (f_iter s c n)         súčin čítač n, invariant s=(c-1)!
    (if (> c n)
         s
         (f_iter (* s c) (+ c 1) n)))
```

  - iterácia

```
  - používa stavové premenné
    - úplný ,,stav" výpočtu mám prístupný (tj. pod kontrolou)
    - pamäť : O(1) - pri dobrej implementácii
    - postup výpočtu (vyhodnocovanie if preskakujem):
     (faktorial 4)
     (f-iter  1 1 4)
     (f-iter  1 2 4)
     (f-iter  2 3 4)
     (f-iter  6 4 4)
     (f-iter 24 5 4)
     24

- Fibonacciho čísla, dva programy pre porovnanie
  - rekurzívny, nelineárny
  (define (fib n)
    (cond ((= n 0) 0)
          ((= n 1) 1)
          ( else (+ (fib (- n 1))
                    (fib (- n 2)) ))))

  - iteračný
  (define (fib n)
    (fib_iter 1 0 n))
  (define (fib_iter a b citac)
    (if (= citac 0)
        b
        (fib_iter (+ a b) a (- citac 1))))
  - pri iteračnom spôsobe si v stavových premenných (ktorých je
určitý počet) pamätám stav výpočtu, resp. zaujímavú/potrebnú
časť stavu. Ak budem počítať nejakú inú postupnosť, v ktorej
ďalší člen závisí na konštantnom počte predchádzajúcich členom,
môžem rozšíriť počet stavových premenných - argumentov. Ak budem
počítať postupnosť, kde ďalší člen závisí na všetkých
predchádzajúcich členoch postupnosti, nemôžem predávať jednotlivé
členy samostatne, ale môžem ich predať všetky naraz: budem mať
(jeden) stavový argument, ktorý bude obsahovať dosiaľ spočítanú
časť postupnosti. Samozrejme v tomto prípade bude veľkosť
spotrebovanej pamäti závisieť na reprezentácii stavového
argumentu.

- pr.: Euklidov alg. - najväčší spol. deliteľ
  (define (nsd a b)
    (if (= b 0)
        a                         ; výsledok
```

```
        (nsd b (remainder a b))))  ;
```

-- remainder - vstavaná fnc., zvyšok po delení (modulo)

- (1999) deklarativnost FP
  definicia funkcie je rovnost, tj. zapis
  ```
  (define (f args) telo)
  ```
  sa deklarativne pochopi ako (definicna) rovnost f(args) = telo

  - proceduralne sa rovnosti pouzivaju orientovane, tj. ako prepisovacie pravidla
  - podmienky v tele: funkcia je niekolko podmienenych rovnosti

  - vypocet odpoveda rieseniu sustavy rovnic
    - zadanie (tj. program a volanie) je sustava rovnic v implicitnom tvare - tj. prave strany obsahuju volania inych funkcii
    - riesenie je v explicitnom (vyriesenom) tvare: prava strana neobsahuje volania inych funkcii, len hodnoty, tj. datove struktury (aj zlozene ...)
  - program je teoria, ktoru mozete skumat, upravovat beznymi matematickymi a logickymi metodami
    - skumat: ci odpoveda zamyslanej semantike, ci platia invarianty
    - upravovat: do efektivnejsieho tvaru

- programovanie funkcií vyšších rádov, funkcionálne parametre

pr.: súčet kvadrátov čísel od a do b

```
(define (sum2 a b)
  (if (> a b)
      0
      (+ (kvadrat a)
         (sum2 (+ a 1) b ))))
```

- analogicky by sme napísali funkciu sum1 pre súčet čísel od a do b

- abstrakcia:

Chceme definovať funkciu, ktorá odpovedá "sume", ale nie "sume kvadrátov"

  - abstraktne:

```
  (define ( <meno>  a b)
    (if (> a b)
        0
        (+ ( <funkcia> a)
           ( <meno> ( <next> a) b))))
```

  - <funkcia> a <next> sa stanú parametrami:

- konkrétne v scheme:

```
(define (sum f a next b)         -- f a next sú funkčné param.
  (if (> a b)
      0
      (+ (f a)                         -- volanie f
         (sum f (next a) a next b))))
```

- použitie sum - definícia pôvodných funkcií sum1 a sum2

```
(define (sum2 a b) (sum kvadrat a 1+ b))
(define (sum1 a b) (sum id      a 1+ b))
```

  - ostáva definovať: id, 1+

- použitie sum: v numerickom integrovaní
- iná možná abstrakcia: sum ako iteratívny výpočet
- je možná ešte vyššia abstrakcia:
  - funkcii sum (sčítanie) s operáciami 0 a + je podobná funkcia PI (násobenie) s operáciami 1 a *

- špeciálna forma: lambda definícia anonymnej procedúry
  - má zmysel, len ak sa procedúra použije len jedenkrát píše na mieste jediného použitia
  - syntax:

```
(lambda ( <formálne_parametre> )  <telo> )
```

pr.:

```
(define (sum1 a b )
  (sum (lambda (x) x)                 -- identita
       a
       (lambda (x) (+ x 1))           -- 1+
       b ))
```

  - define pre procedúry je syntaktický cukor:

```
(define (sum1 a b) <telo> ) je interne to isté ako:
(define sum1 (lambda (a b) <telo> ))
```

  - spec. forma lambda je konstruktor funkcii, vyhodnotenie (s danymi argumentami) je selektor
  - funkcie su objekty "prvej kategorie": je ich mozne predavat ako parametre, vracat ako hodnoty, integrovat do datovych struktur ...
  - anomymne funkcie su casto jednoucelove parametre funkcii vyssich radov
  - ! potreba lambda funkcii svedci o inom sposobe programovania:

```
    1) hodnota vyrazu (tj volania) funkcie sa pouziva jedenkrat,
v kontexte podla miesta vyskytu
    2) programy je mozne parametrizovat kodom, tj. lambda funkciou
       - aj anonymnym, jednoucelovym
       - aj dynamicky vytvorenym

- zlozene d.s.
  - na implementačnej úrovni je definovany konstruktor a selektory:

  jediná zložená datová štruktúra - dvojica, pár (angl. pair)
  konštruktor: (cons x y) - vráti dvojicu
  selektory: car, cdr - prvá a druhá zložka
  platí:
  >(define a (cons 1 2))
  a
  >(car a)
  1
  >(cdr a)
  2
  >a
  (1 . 2) ; vypis parov

  - dvojica v pamäti - cons vyhradí pamäť a zapíše adresy svojich
     argumentov:
                       ŪÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ
    implementácia       pointer na x  pointer na y
                       R̄ÄÄÄÄÄÄÄÄÄÄÄÄÄÄĀÄÄÄÄÄÄÄÄÄÄÄÄÄÄŮ

- pracovať len s dvojicami je nepríjemné:
  konštrukcia zoznamov, ako d.š.:
  >(list 1 2 3)
  (1 2 3)
  - zoznam je syntaktický cukor pre postupnosť cons
(asociovanu doprava)
    ekv.: (cons 1 (cons 2 (cons 3 ())))

- reprezentacia termov, s obecnym poctom argumentov:
  hlava je funkcny symbol, telo je zoznam argumentov
  matematicky :  1 + 2 * (3 + 4)
  reprezentacia:        (+ 1 (* 2 (+ 3 4)))

- pr. n-tý prvok zoznamu
  (define (nth n x)
     (if (= n 0)       -- zacneme od 0 (na rozdiel od slajdov)
```

```
         (car x)
         (nth (- n 1) (cdr x)) ))
          ^spracovanie ^tela zoznamu

- pr. dĺžka zoznamu
  (define (length x)
    (if (null? x)      ; test na prázdnosť zoznamu x
        0
        (+ 1 (length (cdr x))) ))
                     \-----/
typicky: hodnota sa pouzije na mieste
     vyskytu ako parameter dalsej funkcie
=> nepotrebujeme priradenie
  - ale: mozne nevyhody:
    - viacnasobne pocitanie hodnoty (neefektivita)
    - definovanie lokalnych pomocnych podvyrazov (prehladnost)
    => specialna forma let

- špeciálna forma: let
  definícia lokálnych väzieb
  - syntax:
    (let (( <prem1> <výraz1)
          ( <prem2> <výraz2)
           ...
          ( <premN> <výrazN))
      <telo> )
  - vyhodnotia sa všetky <výraz>y, potom sa <prem>enné asociujú
so spočítanými hodnotami a potom sa <telo> vyhodnotí v takto
vzniknutom okolí.
  - let je ,,blok", rozsah platnosti premenných je telo, po jeho
ukončení sa ,,oddefinujú"
  - výrazy sa výhodnocujú súčasne, s použitím externého okolia
     (na rozdiel od použitia niekoľkych define za sebou)
    => vyhodnocovanie a priradenie je paralelné, tj. výmenu hodnôt
     dvoch premenných môžeme napísať
     (let ((a b) (b a)) <telo>)
  - let je syntaktický cukor pre:
  ((lambda ( prem1  prem2  ... premN ) telo )  -- aplikuj fnc
            vyraz1 výraz2 ... výrazN )         -- na argumenty
     - pri vyhodnocovaní funkcie sa najprv vyhodnotia aktuálne
           parametre, potom sa naviažu na formálne parametre
           a potom sa vyhodnocuje telo
- použitie: pr.:
  - definícia pomocných výpočtov, napr. viacnásobne použitých
```

```
     volaní; ak by sa nepoužilo let, každé volanie fce kvadrát sa
     vyhodnocuje zvlášť
  (define (g x)
    (let ((y (kvadrat x)))
      (/ y (+ y 1))))

  - forma let neumožňuje definíciu vzájomne rekurzívnych
(pomocných) funkcií použitých v tele hlavnej funkcie
  => špec. forma LETREC so zhodnou syntaxou, ale použitie
     <prem>enných vo <výraz>och sa odkazuje na práve definované
     <prem>enné, nie na ich hodnoty z globálneho okolia
    - implementacne zavisle

  - implementacia vazieb: okolie je zoznam dvojic meno-hodnota.
    Pri vyhodnocovani sa hlada prvy vyskyt premennej v okoli.
    Let prida dvojice na zaciatok (zoznamu) okolia.
    Vyhodnocovaci stroj si pamata zaciatok okolia.
    - technicke detaily: dva sposoby vazby premennych:
        staticky a dynamicky.
      - staticky (v Scheme): premenna ma hodnotu z okolia v ktorom
bola definovana
      - dynamicky (Lisp): premenna ma hodnotu z okolia, v ktorom
bola volana

- define, lambda a let umožňujú myslieť a programovať
funkcionálne.

--- (zložené) datové štruktúry:
- datové abstrakcie
  - chceme nejakú reprezentáciu pre racionálne čísla, komplexné
     čísla ...
- popis lineárnej kombinácie: ax+by
  (define (lin_komb a b x y)
     (add (mul a x) (mul b y)))
  táto definícia/program by mal slúžiť, ak x a y sú čísla,
     vektory alebo matice
  - lin_komb nemôže poznať vnútornú štruktúru (typ) x, y
    => abstraktné data
  - lin_komb nemôže poznať konkrétne funkcie add, mul
    => generické operátory
- problém viacerých reprezentácií:
    komplexné čísla môžu byť zadané/reprezentované v súradnicovom
alebo polárnom tvare. Chceme, aby operátory prijímali obidva
tvary.
```

```
  - možné riešenie:
    data si sebou nesú (aj) informáciu o type:
    { ,,suradnicový_tvar", x, y }
    { ,,polárny_tvar",     uhol, velkost }
    - (rozne) konstruktory pre komplexne cisla
    (define (mk_polar velkost uhol)
(cons 'polar (cons velkost uhol)) )
    (define (mk_rect real imag)
(cons 'rect (cons real imag)) )
  - operátory si majú možnosť zistiť, aké data dostali.

- vytváranie datových štruktúr
  - nemáme správu pamäti, pointry (pod svojou kontrolou)
  - konštruktor : vznik d.š. zo zložiek, (podobné ,,new"
     v Pascale), vracia ,,objekt"
  - selektor(y) : prístup na zložky v ,,objekte"
  - konštruktor a selektory umožňujú výrobu d.š. a prácu s d.š.
na nízkej úrovni abstrakcie, majú odtieniť konkrétnu
reprezentáciu d.š.

- pr. racionálne čísla
  konštruktor, vracia ,,racionálne_číslo" :
     (make_rac <citatel> <menovatel>)
  selektory - vracajú zložky:
     (citatel <rac>)
     (menovatel <rac>)
  - pozn.: konštruktor a selektory musia byť kompatibilné, ale
selektory nemusia vraciať presne to, čo dostal konštruktor. Napr.
pri (každej) konštrukcii rac. čísla môže prebehnúť vydelenie citateľa
a menovateľa ich najväčším spoločným deliteľom.

- operácie s rac. číslami:
  - sčítanie rac. čísel
    vytvorím nové rac. číslo pre súčet - pomocou konštruktoru
    na argumenty pristupujem len pomocou selektorou
    (define (+rac x y)
       (make_rac (+ (* (citatel x)   (menovatel y))
                    (* (menovatel x) (citatel   y)))
                 (* (menovatel x) (menovatel y)) ))

- (jedna možná) reprezentácia rac. čísel:
  (define (make_rac x y) (cons x y))
  (define (citatel x)   (car x))
  (define (menovatel x) (cdr x))
```

```
- abstrakčné bariéry (v prípade rac. čísel):
    užívatelský program ...
      (program) používa rac. čísla
        - navrhuje užívateľ
    (interface) +rac -rac *rac /rac =rac ...
      (program) pravidlá pre počítanie s rac. číslami
        - navrhuje tvorca operácií s  rac. číslami
    (interface) make_rac citatel menovatel
      (program) implementácia r.č. ako dvojíc
        - navrhuje tvorca d.š. pre rac. čísla
    (interface) cons car cdr
      (program) implementácia dvojíc
        - navrhuje implementátor scheme

- priklady na rozne spracovanie hornerovej schemy
  A) vyhodnotenie Hornerovej schemy
  - vstup je zoznam koeficientov a hodnota
  - vysledok je cislo
; koeficienty su vo vsetkych prikladoch od najnizsieho
(define (eval_hs koefs x)
(if (NULL? Koefs)
    ()
    (+ (car koefs)
       (* x (eval_h (cdr koefs) x)) )))

  B) vytvorenie funkcie, ktora vyhodnocuje H.s.
  - vstup su len koeficienty, vystup je funkcia, az ta dostane
hodnotu
  - vracaju sa (anonymne) funkcie
  - pri pouziti v tele je vytvorenu funkciu nutne
aplikovat na hodnotu x
(define (mk_hsf koefs) ; "chyba" parameter pre x
    (if (null? koefs)
       (lambda (x) 0)
       (lambda (x) (+ (car koefs)
                      (* x
                         ((mk_hsf (cdr koefs)) x) ))) ))
  - pouzitie
>((mk_hsf '(3 2 1)) 2)
17
; testy
(define (t2 x) ((mk_hsf '(3 2 1)) x))
(write (map (mk_hsf '(3 2 1)) '(0 1 2 3)))  (newline)
```

```
  ;>>(3 6 11 18)

 C) Vytvorenie symbolickeho vyrazu
  - vstup su koefs a MENO premennej
  - vystup je (vyhodnotitelny) vyraz (s neefektivitou na konci)
(define (mk_hsexp koefs prem) ; parameter prem je MENO premennej
    (if (null? koefs)
       0
       (list '+ (car koefs)
             (list '* prem (mk_hsexp (cdr koefs) prem)) )))

  - pouzitie:
    - symbolicka hodnota y sa vyskytuje vo vyraze
>(mk_hsexp '(3 2 1) 'y)
(+ 3 (* y (+ 2 (* y (+ 1 (* y 0))))))

  - varianta z prednasky:
(define (mk_hsexp1 koefs prem) ; parameter prem je MENO premennej
    (cond ((null? koefs) 0) ; prazdny zoznam
  ((null? (cdr koefs)) (car koefs)) ; jednoprvkovy,
; vracia sa cislo z hlavy
          (else
            (list '+ (car koefs)
              (list '* prem (mk_hsexp (cdr koefs) prem))) ))))
>(mk_hsexp '(3 2 1) 'y)
(+ 3 (* y (+ 2 (* y 1))))

  - varianta s backquote:
    ...
    (else `(+ ,(car koefs)
              (* ,prem ,(mk_hsexp (cdr koefs) prem))) )) ))

- pouzitie pre vyhodnotenie
  - pre funkcie eval a apply neexistuje typ v typovanom jazyku
    => nejdu priamociaro previest do typovaneho jazyka (Haskell)
ad B)
  - vytvorena funkcia je v internom tvare:
>(mk_hsf '(3 2 1))
#<Procedure #ee2>

  - aplikacia funkcie na zoznam argumentov
>(apply (mk_hsf '(3 2 1)) '(2) ))     ; ocakava 1 argument
11 ; vysledok
    - apply sa v pripade, ze je (v case "kompilacie") znamy pocet
```

```
argumentov, moze nahradit klasickym volanim (f a1 ... an). Ak nie je
znamy pocet argumentov, musime pouzit apply

ad C)
  - vstavana funkcia eval vyhodnocuje vyrazy, tj. je to interpret
zdrojoveho kodu
>(define y 2) ; s let to nefunguje
>(eval (mk_hsexp '(3 2 1) 'y))
11
  - u eval su technicke problemy s vazbou premennych:
    - hodnota y musi byt definovana

- symbolické data - dosiaľ sme pracovali len s číslami
  ako odlíšime symbolické data, ktoré sa nemajú vyhodnocovať od
     premenných, ktoré sa majú vyhodnocovať?
  idea: špeciálna forma quote

  (quote data) - vráti data bez vyhodnotenia
  'data        - to isté, ale skrátene; syntaktický cukor

  - pr.:
  >(list 'a 'b)        -- nevyhodnocuje a ani b
  (a b)
  >(car '(a b c))      -- (1) , argumentom car je zoznam (a b c)
  a                    -- jeho prvý prvok je a.
  >(list 'car (list 'quote '(a b c)))
  (car (quote (a b c))) -- ,,syntakticky" zhodné s (1), ale toto
                            je výsledok (tj. data)

  - všetko sa vyhodnocuje, ale konštanty sa vyhodnotia na seba
    => nemusia byť uvedené apostrofom
    čísla       3     3.14
    reťazce     "pgm/meno.txt"
    znaky       #\a
    logické     #t    #f

- iné datové štruktúry: binárne vyhľadávacie stromy
     vrchol má tri zložky: L, R    ľavý a pravý podstrom
                           H       hodnotu
     dve možnosti reprezentácie vrcholu:
     a)    .                     b)    .
          / \                         / \
         H   .                       L   .
            / \                         / \
```

```
          L   R                        H   .
                                          / \
                                         R   ()
```

```
  - realizácia:
  a) (define (make_t L H R)        ; konštruktor
           (cons H (cons L R)) )
     (define (getl T) (cadr T))    ; selektory
     (define (getr T) (cddr T))
     (define (geth T) (car T))
  b) (define (make_t L H R)        ; konštruktor
           (list L H R) )
     (define (getl T) (car T))     ; selektory
     (define (getr T) (caddr T))
     (define (geth T) (cadr T))
  - nad takto definovaným interface môžem budovať ďalšie
     procedúry
  - syntaktický cukor: (cadr x) je skrátena verzia (car (cdr x)),
analogicky pre ostatne cxxr a cxxxr

- príklady
; append - spájanie zoznamov
(define (app2 a b)
  (if (null? a)
       b
       (cons (car a) (app2 (cdr a) b))))

; reverse - naivné obrátenie zoznamu
(define (rev a)
  (if (null? a)
      ()
      (app2 (rev (cdr a)) (list (car a)))))
- DC: obrátenie v lineárnom čase s použitím akumulátora

- funkcie vyšších rádov: obsahujú funkcionálne parametre
MAP ... funkcia, ktorá aplikuje daný funkcionálny parameter
(jedného argumentu) na každý prvok zoznamu a z výsledkov utvorí
zoznam
(zjednodušená verzia, vstavaná funkcia MAP je obecnejšia)
  (define (MAP f l)
    (if (null? l)                 ; test prázdneho zoznamu
        '()                       ; výsl. je prázdny zoznam
        (cons (f (car l))         ; výsl. je zoznam, hlava vznikne
                                  ; aplikáciou f na hlavu l
              (MAP f (cdr l)) ))) ; telo získame z rekurzie
```

pr. volania:

```
> (map (lambda (x) (* x x)) '(2 3 4))
(4 9 16)
> (map kvadrat '(2 3 4))
(4 9 16)
```

- vstavané MAP má premenný počet parametrov - zoznamov, ktoré je schopné spracovať. Funkciu danú ako parameter volá postupne na prvé prvky zoznamov, druhé prvky zoznamov ... a z výsledkov zostaví výsledný zoznam. Z popisu plynie, že všetky zoznamy musia mať zhodnú dĺžku.

pr. volaní

```
>(map < '(1 4) '(2 3))
(#t ())
  ; vracia sa dvojprvkový zoznam, jeho hodnoty sú true #t
     a false () a vzniknú z volaní (< 1 2) a (< 4 3)
>(map + '(1 2 3 4) '(5 6 7 8))
(6 8 10 12)
```

FILTER ... funkcia, ktorá dostane funkcionálny parameter (jednoho argumentu) a zoznam. Zo zoznamu vypusti všetky prvku, pre ktoré nie je splnený daný fnc. parameter

```
(define (filter f l)
  (if (null? l)
      '()                                    ; sprac. l=()
      (if (f (car l))                        ; test hlavy
          (cons (car l) (filter f (cdr l)))  ; pridanie hlavy
          (filter f (cdr l)) )))             ; vynechanie hlavy
```

- všimnite si, že hoci sú v tele použité dve rekurzívne volania, každé je v inej vetve if-príkazu. Preto sa z nich vykoná najviac jedno a výpočet zostáva lineárny.

- použitie:

```
(define (between a z x) (and (< a x) (< x z)))
>(filter  (lambda (x) (between 3 5 x)) '(1 2 3 4 5 6))
(3 4 5)
```

MAP a FILTER vyjadrujú parametrický spôsoby spracovania zoznamu, pričom operácie, ktoré sa majú uskutočniť na jednotlivých prvkoch, sú dané funkcionálnym parametrom.
Ak výsledok výpočtu nemá byť zoznam, ale ľubovoľná datová štruktúra (jednoduchá alebo rekurzívna), musíme mať obecnejšie parametrické spracovanie zoznamu.

FOLDR ... zvinutie zoznamu sprava

```
- volanie (FOLDR op e '(x1 x2 ... xn)) vykoná výpočet
  x1 op (x2 op (... xn-1 op (xn op e)... ))
(define (foldr f e l)
  (if (null? l)
      e
      (f (car l) (foldr f e (cdr l))) ))
  - e je často neutrálny/jednotkový prvok k operácii f
príklady:
- (define (suma l) (foldr + 0 l))  ; scita vsetky čísla
- (define (PI   l) (foldr * 1 l))  ; vynásobí všetky čísla
- (define (length l)   ; počítanie dĺžky zoznamu
     (foldr (lambda (x y) (+ y 1)) 0 l))
- (define (append x y) (foldr y cons x)) ; spajanie dvoch zoznamov
- (define (mappend l) (foldr append () l)) ; spajanie zoznamov
- (define (insert x l)
     (cond ((null? l) (list x))
           ((< (car l) x) (cons (car l) (insert x (cdr l)) ))
           (else (cons x l)) ))
  (define (insertsort l)     ; znamy insertsort
     (foldr insert () l))
  - map pomocou fold:
  (define (map f l)
    (foldr (lambda (x y) (cons (f x) y)) () l) )
    -(hodili by sa nejake operacie na funkciach: skladanie ...)

- máme binárnu funkciu max počítajúcu maximum, chceme maximum
     zoznamu
  (define (maxl l) (foldr max (car l) (cdr l)))
     ; maximum nemá zmysel z prázdneho zoznamu
  - podobne priemer nemá zmysel počítať z prázdneho zoznamu
    => vymyslieť (a naprogramovať) inú operáciu, ktorá
       spracováva neprázdne zoznamy
    => foldr1, ktorá spracuje jeden (posledný) prvok zoznamu
       - pre počítanie priemeru sa musí výsledok (dvojica čísel pre
súčet a počet) ešte po skončení rekurzie vydeliť - tj. potrebujeme
postspracovanie mimo rekurziu

FOLDL ... zvinutie zoznamu zlava
- volanie (FOLDL op e '(x1 x2 ... xn)) vykoná výpočet
  ((...((e op x1) op x2) ... op xn-1) op xn
(define (foldl f e l)
  (if (null? l)
      e
      (foldl f (f e (car l)) (cdr l)) ))
```

```
  - e sa používa ako akumulátor medzivýsledkov
príklady:
- mnohé asociatívne z už uvedených vyššie
- (define (reverse l)        ; obrátenie zoznamu
     (foldl (lambda (x y) (cons y x)) () l))   ;
  - odpoveda pouzitiu akumulatoru

- niektore operacie nemaju definovanu hodnotu na prazdnom zozname,
napr min a max. Pre tieto operacie je vhodny mierne upraveny
prechod strukturou, foldr1, resp. foldl1
  (define (foldr1 op l)
    (cond ((null? l) (write 'error_in_foldr1)) ; prazdne l
          ((null? (cdr l)) (car x)) ; jednoprvkovy l
          (else (op (car l) (foldr1 op (cdr l)))) ))
    - foldr1 moze potrebovat predspracovanie prvku pomocou map.
Ina moznost by bola zadat specialnym funkcionalnym parametrom,
 ako sa spracuje posledny prvok

  - foldl1 sa od foldl odlisuje na zaciatku, preto rekurziu nemusime
explicitne, ale s pomocou fold
  (define (foldl1 e op l)
    (if (Null? l)
        (write '(error in_foldl1 empty_list))
        (foldl (car l) op (cdr l)) ))

- nie vzdy su priebezne vysledky vo forme pozadovanej pre vystup.
Napriklad aritmeticky priemer je mozne napisat pomocou fold, ale
musim si pamatat dve hodnoty pre pocet a sucet prvkov.
Az po skonceni rekurzie sa spocita pozadovany vysledok
  (define (priemer l)
    (/ (suma l) (length l)) ; 2x fold
  - takto napisany priemer ma dve nevyhody:
    1) priemer nie je definovany pre prazdny zoznam
       => lepsie je pouzit foldr1
    2) struktura sa prechadza dvakrat
       => vylepsit:
Operacia bude pracovat s dvojicami (pocet,sucet),
map prevedie zoznam cisel na zoznam dvojic.
Let potrebujeme na zapamatanie vysledku, z ktoreho potom
vytahujeme jednotlive zlozky.
  (define (priemer1 l)
    (let ((a (foldr1 (lambda (x y)
                       (cons (+ (car x) (car y))
                             (+ (cdr x) (cdr y)) ))
```

```
                (map (lambda (x) (cons 1 x)) l) )))
        (/ (cdr a) (car a)) ))
```

- parametrické operácie je možné definovať aj na iných d.š., napr.
na maticiach:
MAPMAT ako parametrická unárna operácia s maticou po prvkoch:
funkcia MAPMAT, ktorá dostane funkcionálny parameter f
a aplikuje ho na všetky prvky matice. MAPMAT musi poznať
a využívať vnútornú štruktúru matice.
- matica je reprezentovaná ako zoznam zoznamov: môžeme napísať
explicitnú dvojitú rekurziu, ale tiež môžeme použiť map na
spracovanie každej úrovne zoznamov => dostaneme dve vnorené
volania map:

```
(define (mapmat f m)
     (map                          ; aplikácia na riadky
           (lambda (v) (map f v))  ; sprac. prvkov v riadku
              m ))                 ;
```

použitie:

```
>(mapmat (lambda (x) (* 10 x)) '((1 2) (3 4)))
((10 20) (30 40))
```

- DC: definovať MAPMAT pre riedku reprezentáciu matíc, tj. matica
je daná ako zoznam trojíc pre všetky nenulové koeficienty:
     (index_i, index_j, hodnota_ij)

MAP2MAT ako paramerická binárna operácia s maticami po prvkoch

```
(define (map2mat f m1 m2)
     (map               ; aplikácia na dvojice zhodných riadkov
        (lambda (v1 v2) (map f v1 v2)) ; spracovanie riadkov
           m1 m2))
```

definícia sčítania (analogicky odčítania) matíc:

```
(define (+mat m1 m2) (map2mat + m1 m2))
```

DC: zobecnenie násobenie matíc, kde operácia skalárneho súčinu
dvoch vektorov sa abstrahuje na formálny parameter.

--- funkcie ako výstupné hodnoty:
- definícia zloženia dvoch funkcií jednej premennej:

```
(define (slozeni f g)
     (lambda (x) (f (g x))) )      ; vracia sa funkcia
```

- použitie:

```
>((slozeni kvadrat 1+) 3)
16
```

  - vyhodnotí sa volanie funkcie "slozeni", ktoré vytvorí a vráti

```
nejakú funkciu (resp. reprezentáciu funkcie), tá sa potom
aplikuje na číslo 3 a vyhodnotí sa.

- príklad: definícia funkcionálu "+n", ktorý k číslu n vráti
funkciu jednoho argumentu, ktorá pričítava n (vracia sa inkrement
o n)
(define (+n n) (lambda (x) (+ x n))
  použitie:
     >((+n 3) 5)
     8
     >(map +n '(1 2 3))
     -- vráti trojprvkový zoznam funkcií, kde prvá funkcia
           inkrementuje o 1, druhá o 2 a tretia o 3


------------------
- vstavané procedúry: podrobnejšie v manuále
     (LOAD subor)
           načíta súbor
           pr.: (load "pgm/moje.s")
             ! aj v ceste su lomitka
     (EXIT) ukončenie práce

- testy ...
(null? x)       vracia #t, ak x je ()
- aritmetika ...
- deštruktívne priradenia
(set! premenná výraz)
(set-car! dvojica výraz)
(set-cdr! dvojica výraz)

- sekvenčné výpočty
(BEGIN <výraz> ...)
(SEQUENCE <výraz> ...)

- definovanie a získavanie vlastností symbolov
- vstup/vystup

- vyhodnocovanie
MAP - už bolo
(APPLY fnc zoznam_argumentov)
     aplikuje funkciu na zoznam argumentov
     pr.: >(apply * '(2 3 4))
           24
```

```
     - argumenty sú data, ktoré môžem ich vytvoriť (načítať)
     počas behu programu a následne použiť pri vyhodnocovaní
(EVAL výraz)
     vyhodnotí výraz
     pr.: >(EVAL '(* 2 3 4))
          >(EVAL (LIST * 2 3 4))
          24
     - výraz, ktorý sa doteraz spracovával a tvoril ako data,
     môžem prehlásiť sa zdrojový kód a nechať ho vyhodnotiť
     => stráca sa rozdiel medzi datami a programom

; Erathostenovo sito
; najprv pomocné predikáty:
; výroba intervalu ako d.š.
(define (int od do)
  (if (> od do)
      ()
      (cons od (int (1+ od) do)) ))

; filter zoznamu l podľa predikátu pred
(define (fl pred l)
  (cond ((null? l) '())
        ((pred (car l)) (cons (car l) (fl pred (cdr l))) )
        ( else (fl pred (cdr l))) ))

; deliteľnosť
(define (delit x y)
  (= (remainder x y) 0))
- použitie:
(define (delit2 x) (delit x 2))

; ladiaci test
(write (fl delit2 (int 1 10)))

; vlastné E. sito
(define (sito l)
  (if (null? l)
    ()
    (let ((prvoc (car l))) ; "význačná lok. hodnota" -prvočíslo
      (write (list prvoc)) ; ladiaca tlač
      (cons prvoc          ; nájdené ďalšie prvočíslo
           (sito (fl (lambda (x) (not (delit x prvoc))) (cdr l))) ))))
                 ; zoznam prefiltrujeme podľa nedeliteľnosti
                 ; aktuálnym prvočíslom
```

```
; ladenie
(write (sito (int 2 20)))
(define (test c)
  (sito (int 2 c)) )
(write (test 30))
```

- iná realizácia car, cdr a cons
car, cdr a cons musia spĺnať podmienky:
(car (cons x y)) = x
(cdr (cons x y)) = y
tieto pravidlá popisujú koretnosť interface

cons realizované pomocou dispatch procedúry

```
(define (cons x y)
     (define (dispatch m)
           (cond ((= m 0) x)
                 ((= m 1) y)
                 (else (error "CONS - unexpected arg." m)) ))
     dispatch)   ; cons vracia funkciu
```

realizácia selektorov - volanie vrátenej funkcie a vhodnými parametrami

```
(define (car z) (z 0))
(define (cdr z) (z 1))
```

ešte iná možnosť realizácie:

```
(define (cons x y) (lambda (m) (m x y))
(define (car z) (z (lambda (p q) p)))
(define (cdr z) (z (lambda (p q) q)))
```

- stráca sa rozdiel medzi programom a datami, pretože data sú realizované pomocou väzieb zachytených vo funkciách

- lokálne premenné pre stavy a podobnosť s OOP
Pretože môžem vracať funkcie ako výsledky volaní, nič nebráni tomu, aby sa pri vytváraní nejakej d.š. nevytvorili súčasne funkcie, ktoré s ňou budú pracovať
- banková aplikácia:
vytvorím účet, ktorý si pamätá zostatok, sú definované funkcie na vyberanie a vkladanie na účet

```
(define (make_ucet zostatok)       ; konštruktor účtu
     (define (vyber suma)          ; def. lokálnej fnc.
       (if (>= zostatok suma)
           (sequence (set! zostatok (- zostatok suma))
```

```
                    zostatok)
              "Nedostatok hotovosti"))
      (define (vloz suma)           ; dalšia lok. fnc.
        (set! zostatok (+ zostatok suma))
        zostatok)
      (define (spracuj m)     ; dispatch fnc.
        (cond ((eq? m 'vyber) vyber) ; fnc. vyber volám id. vyber
              ((eq? m 'vloz)  vloz)
              (else (error "MAKE_UCET: Neznáma správa" m))
      ) )
  spracuj) ; vracia sa dispatch funkcia
použitie:
>(define acc (make_ucet 1000))
acc
>((acc 'vyber) 600)
400
>((acc 'vyber) 500)
"nedostatok hotovosti"
>((acc 'vloz) 900)
1300
  -- (acc 'vyber) sa vyhodnotí na funkciu lokálne známu pod
menom vyber, ktorá sa aplikuje na hodnotu 600, resp. 500 a vracia
výsledok, ktorý sa potom vypíše
  - dispatch funkcia, v tomto prípade spracuj, umožnuje k symbolu
(vyber, vloz) vrátiť "metodu", ktorá sa môže použiť pre zmenu
stavu "objektu".
 - Na rozdiel od OOP:
     - dispatch funkciu si musíme naprogramovať sami
     - nemáme tu žiadnu "typovú" kontrolu, ktorá by počas
       prekladu (načítavania zdrojáku) poznala, že prípustné
       argumenty pre acc sú len 'vyber a 'vloz
     - není tu hierarchia a dedičnosť

----
CVICENIA. 2002, 25.11.
(obľúbené) Hornerovo schéma: (define (hs koefs x) ...)
- viz inde
- otázka vhodného poradia koeficientov: dve možnosti:
 - pre rekurzívne spracovanie koef. od abs. členu a_0
(používa "hs (cdr koefs)" )
 - pre vyhodnocovanie pomocou akumulátora: naopak od najvyššieho členu
a_n
- vracanie funkcií:
namiesto funkcie dvoch argumentov vyrobim funkciu, ktorá pre
```

dané koeficienty vyrobí funkciu ("skompiluje"), ktorá počíta hodnotu polynomu pre dané x. Tj. navratová hodnota(!) funkcie gen_hs je funkcia (v nejakej internej reprezentácii, ktorú je možné vyhodnotiť, ale nie vypísať)

```
(define (gen_hs koefs)
        (lambda (x) (hs koefs x))
)
```

- lambda je konštruktor funkcie (...), vzniká anonymná funkcia, ktorú použijem na mieste výskytu (alebo si ju pomenujem)
- otázka: ako sú zviazané koefs.

```
; volanie
> ((gen_hs '(2 1 5)) 3)
; ab       cd        e
```

vysvetlenie:
podvýraz <d> je zoznam koeficientov, vďaka quote <c> sa nevyhodnocuje. Výraz <b> sa vyhodnotí na anonymnú funkciu, ktorá sa vyhodnotí v rámci výrazu <a> a parametrom <e>

```
; povodné volanie: pre porovnanie:
> (hs '(2 1 5) 3)
```

- quote, aj v (písanom) jazyku:
Řekněte odpověď vs. řekněte "odpověď".

- generovanie vyváženého stromu
  vstup: usporiadaný zoznam, pocet pouzitelnych prvkov
  vystup: dvojica vyvazeny vyhl̄. strom, nepoužite prvky zo zoznamu.
 -

```
(define (vyvazeny s n)
  (cond ((= n 0) (cons nil s))  ; vraciam prazdny strom a pov. zoznam
        ((= n 1) (cons (mk_tree nil (car s) nil))       ; jednoprv. str.
                       (cds s) ))       ; a zvyšok zoznamu
        (else
         (letrec                ; rekurzivne let
           ((nl (div (- n 1) 2))        ; velkost laveho
            (nr (div n       2))        ; velkost praveho
            (outl (vyvazeny s nl))      ; vyroba laveho podstromu
            (outr (vyvazeny (cdr  ; uberame 1 prvok zo zvysku pre akt.v.
                             (cdr outl)) ; zvysok po vyrobe laveho
                            nr))        ; velkost praveho, 2. arg.
           )   ; koniec "rekurzívnych" def. (sú pouzitelné vnorené let)
```

```
          (cons        ; výroba výsledku: dvojice
            (mk_tree (car outl)  ; lavy podstrom ako car z outl
                     (car        ; vybranie vnutorneho vrcholu
                       (cdr outl)) ; zo zvyšku po lavom
                     (car outr)) ; vybratie praveho podstromu
            (cdr outl)         ; vybratie zvyšku
) )     )) )              ; ...
```

- ak používam len! konštruktory a selektory, potom možem zmeniť reprezentáciu stromov bez zmeny programov (ale konštr. prazdneho stromu je v programe natvrdo).

```
(define (mk_tree l x r) ; konštruktor neprázdneho stromu
        (cons l (cons x r))) ; trojica, ako dve dvojice
(define (lavy s)        ; def. selektorov
        (car s))        ; vybratie
(define (pravy s)
        (cddr s))       ; vybratie
... ; pre vrchol : DC
```

- iná repr.: ako trojprvkový zoznam. Menej efektívna (pamäťovo a následne aj časovo), podrobnost viz inde (tuším). DC: koštruktory a selektory

DC:
zipWith f as bs: volá funkciu f na odpovedajúce si páry prvkov. Zoznamy sú obecne rozne dlhé.

---------------
cvičenia 1996
mergesort

rok 1995
na svojom prvom cvičení som robil (ale až po tretej prednáške):
návrh d.š. pre datumy, konštruktor, selektory
  rozdiel medzi (cons rok (cons mesiac den)) a
                (list rok mesiac den)
  príznak ,,typu"
append zoznamov
reverse zoznamov, rekurzívne aj iteratívne
multiappend - dostáva zoznam zoznamov, všetky spája do jednoho
  abstrakcia - spájaciu operáciu a neutrálny prvok na spájenie ()
     nahradiť parametrami
  definícia sumy (a ústne súčinu) zoznamu
ďalšia abstrakcia - namiesto sumy chcem priemer
  - potrebujem ,,pracovné hodnoty" - dvojice počet_prvkov,suma

```
    -''- transformáciu z vstupných hodnôt do pracovných
                       z pracovných na výsledné - v interface
  - práca s dvojicami - pomocou lambda procedúr

- ďalšie nápady
  - filter zoznamu - napr. len párne čísla,
                             (ne)deliteľné daným číslom
    - abstrakcia filtra - s parametricky daným testom
  - Erathostenovo sito
    - potrebuje vygenerovanie intervalu
    - používa (abstraktný) filter
  - násobenie riedkych matíc, ? sčítanie
  - nejaké symbolické výpočty
    - derivovanie, ? zjednodušovanie
    - grafové úlohy

---- 1999
- symbolické výpočty a generovanie kódu:

- tvorba symbolického výrazu k Hornerovej schéme
  vstup je zoznam argumentov, výstup je výraz obsahujúci čísla a
konštantu x
(define (h2v koefs)
  (cond ((null? koefs) (write "error in h2v -- prázdny zoznam koef."))
        ((null? (cdr koefs)) (car koefs)) ; posledny koef.
        (else (list '+                  ; výstup je súčet
                    (list '*            ; súčinu
                          (h2v (cdr koefs))     ; polynomu
                          'x )                  ; a x
                    (car koefs) )) ))   ; s prvým koeficientom
- volanie:
>(h2v '(3 2 1))
(+ (* (+ (* 1 x) 2) x) 3)
  - poradie argumentov sme si zvolili vo funkcii h2v

- kompilácia výrazu:
  funkcionálne programovanie môže vracať funkcie, tj. ak zadám výraz /pre
jednoduchosť jednej premennej/, vráti sa funkcia jednej premennej, ktorá
počíta tento výraz. Ešte jednoduchšie: výraz je polynom počítaný
hornerovou schémou, výsledok je funkcia jednej remennej, ktorá počíta
hodnotu polynomu.
  - vytvorenie funkcie pre hornerovu schému:
    vstup je zoznam koeficientov vo vhodnom poradí, tj. od najnižšieho,
výstup je funkcia. Z technických dôvodov prázdny zoznam považujeme za
```

polynom vracajúci vždy 0. /DC: upraviť program, aby končil u posledného koeficientu./

```
  (define (mk_horner koefs)
    (if (null? koefs)
       (lambda (x) 0)
       (lambda (x) (+ (car koefs)
                      (* x
                         ((mk_horner (cdr koefs)) x) ))) ))
  - volanie:
  >((mk_horner '(3 2 1)) 2)
  11
  >(map (mk_horner '(3 2 1)) '(0 1 2 3)) ; vyhodnocuje viackrát
  (3 6 11 18)
  >(let ((h (mk_horner '(3 2 1)) )) (cons (h 1) (h 2)) )
  (6 . 11)
```

- DC varianta: tvorba funkcie v akumulátore

- kompilácia výrazu:
  vstupom je výraz s /binárnymi/ funkčnými symbolmi +, - a *, číslami a "premennou" x. Chceme vytvoriť odpovedajúcu funkciu v jednej premennej.

```
(define (v2f v)
    (cond ((eq? (car v) '+) (lambda (x) (+ ((v2f (cadr v)) x)
                                           ((v2f (caddr v)) x) )) )
          ((eq? (car v) '-) (lambda (x) (- ((v2f (cadr v)) x)
                                           ((v2f (caddr v)) x) )) )
          ((eq? (car v) '*) (lambda (x) (* ((v2f (cadr v)) x)
                                           ((v2f (caddr v)) x) )) )
          ((eq? v 'x) (lambda (x) x))
          ((number? v) (lambda (x) v))
          (otherwise (write "error")) ))
```

- map pre binárne stromy:
  konštruktor (make_t l x r) pre neprázdne stromy, hodnota () pre prázdne stromy
  deštruktory pre neprázdny strom:
  (getl s), (getr s), (getx s)
  - vlastné map

```
  (define (mapt f s)
     (if (null? s)
        '()
        (make_t (mapt (getl s))
                (f s)
```

```
              (mapt (getr s)) ) ))
  - map nezavisi na konkretnej d.s., len na konstruktoroch

  pouzitie:
  vytvorenie páru, pričítanie 1, funkcia pre pridanie konštanty
  >(mapt (lambda (x) (cons x x)) STROM)
  >(mapt (lambda (x) (+ x 1)) STROM)
  (define (pridej1k_t konst)
      (mapt (lambda (x) (cons konst x)) STROM) )

- zip pre b.s.
  (define (zipWith f s1 s2) ; v pripade nezhody vracia prazdny strom
    (cond ((null? s1) '() )
          ((null? s2) '() )
          (#t (make_t (zipWith f (getl s1) (getl s2))
                (f (getx s1) (getx s2))
                (zipWith f (getr s1) (getr s2)) ) )))
  - aplikácia: zip, ktoré vytvorí dvojice
  (define (zip s1 s2)
    (zipWith cons s1 s2)) ; hm

- fold pre binárne stromy:
  (define (foldbt e f s)
    (if (null? s)
        e
        (f (foldbt (getl s))
           (getx s)
           (foldbt (getr s)) ) ))

  - aplikácie foldbt:
    - výška stromu:
(define (vyska s) (foldbt 0 (lambda (l x r) (+ 1 (max l r))) s))

    - velkost stromu:
(define (size x) (foldbt 0 (lambda (l x r) (+ l 1 r)) s))
    - mapt pomocou foldbt
(define (mapt f s)
  (foldbr () (lambda (fl x fr) (make_t fl (f x) fr)) s) )

    - konštrukcia funkcie "member" pre daný konkrétny strom, ktorá
nevyužíva usporiadanie, tj. prechádza celý strom. (Pretože v SCHEME je
vyhodnocovací model call-by-value, foldbt musí prejsť celý strom)
(define (memberfbt s)
  (foldbt (lambda (k) #f)
```

```
        (lambda (fl x fr)
           (lambda (k) (or (fl k) (eq? k x) (fr k))) ) ))
```

- iná je konštrukcia funkcie, ak využijeme usporiadanie stromu. Uvedomte si, že pri konštrukcii prechádzame celý strom, pri použití skonštruovanej funkcie už len príspušnú cestu.

```
(define (findfbt s)
  (foldbt (lambda (k) #f)
        (lambda (fl x fr)
          (lambda (k) (cond ((eq? k x) #t)
                            (( < k x) (fl k))
                            (otherwise (fr k)) ))) ))
```

6,10.12.'99

- na tychto prikladoch by ste si mali uvedomit, ze vytvorenie pomocnych/ knizničných funkcii umozni spracovavat cele datove struktury naraz. Naviac dovoľuje na tejto vyššej urovni o datovych strukturach rozmýšľať, tj. neobsluhovat jednotlive pointry /spojových štruktúr/, položky /záznamov/, prípady /tagu/ ...

- pr. antiunifikácia:
Sú dané dva termy. Chceme vyrobiť term /v Prologovksom zmysle/, z ktorého by sme mohli obiva termy dostat substitúciou. Ako reprezentáciu premennej budeme pužívať výraz 'var .
  Postup: ak sú obidva termy jednoduché konštanty, potom ak sú zhodné, výsledok je táto konštanta, ináč 'var. Ak sú obidva termy zoznamy zhodnej dĺžky, potom výsledok je zoznam, ktorého prvky získame rekurzívne. Ináč, včetne zoznamov rôznej dĺžky, výsledok je 'var.

```
(define (au x y)
  (cond ((atom? x) (eq? x y))  ; ak sa zhoduju, y je tiez atom
        ((and (list? x) (list? y)
              (= (length x) (length y)) )
          (aul x y) )          ; antiunif. pre zoznamy
        (#t 'var)              ; inak nezhoda -> premenna
))
(define (aul x y )
  (zipwith au x y))
(define (zipwith f xs ys)      ; obecny sposob rekurzie
  (if (or (null? xs) (null? ys))
      '()
      (cons (f (car x) (car y))
            (zipwith (cdr xs) (cdr ys)) )))
```

- iny sposob: zip do dvojic a potom map, ale s upravenou funkciou

```
(define (zipwith f xs ys))
  (map
     (uncurry2 au)   ; (lambda (x) (au (car x) (cdr x))) ;
     (zip xs ys) ))
```

- z funkcie dvoch premennych pre zipwith musime vyrobit funkciu jednej premennej pre map. Ak uvazujeme obvyklu dvojicu vyrobenu pomocou cons, je prevod (uncurry2 :: (a->b->c) -> a*b->c , pre dvojice)

```
(define (uncurry2 f)
  (lambda (x) (f (car x) (cdr x))) )
```

- pr. výstavba stromu AC, dopredná fcia.
Daný je zoznam slov, slová sú /kvôli jednoduchosti/ po písmenách. Chceme vybudovať vyhľadávací strom. Každý uzol obsajhuje tri informácie: 1/ písmeno, ktoré vedie do tohoto vrcholu, 2/ či tu slovo končí /'ano alebo 'ne/ 3/ zoznam podstromov.

Po navrhnutí tohoto programu mozeme rozšíriť storm a spracovanie, aby namiesto informácie, či tam slovo končí, bolo uvedené konkrétne slovo. Ďalej možeme zaviesť spätnú funkciu

Idea: Tvoríme funkciu ac1. Máme zoznam slov /pre jednoduchosť po písmenách/, vyberieme prvé písmeno z každého slova a utvoríme dvojicu písmeno, slovo. Usporiadame /není to nutné/ a rozdelíme na triedy "ekvivalencie" podľa prvého písmena. Dostaneme zoznam dvojíc, |písmeno.slova|. Na každý prvok tohoto zoznamu zavoláme ďalšiu funkciu ac2. Tá vyrobí požadovanú trojicu |písnemo, koncovú_info, zoznam podstromov| takto. Písnemo je dostupné, koncová info sa získa prechodom zoznamu a otestovaním , či taj je nejaké /jediné/ jednopísmenné slovo. potom sa utrhne prvé písmeno, prípadný prázdny zbytok odfiltruje a na zostávajúce časti slov sa pustí rekurzívne fcia ac1.

/Celkový návrh nie je ideálny, ale názorný. Boli by možné optimalizácie/.

- predpoklad o vstupných slovách: sú neprázdne

```
(define (ac1 slova)
  (map ac2           ; spracovanie tried rozdelenych podla 1. pism.
      (partition              ; rozdeleni podla 1. zlozky
       (sort                  ; usp. podla 1 zlozky
        (map (lambda (x) (cons (car x) x)))   ; tvorenie dvojic
             slova)) ))       ; slova
```

- fcia, ktorá vyrobí požadované trojice:

```
(define (ac2 x)
```

```
   (let ((z (cdr x)))
     (cons (car x)
      (cons (konec z) (ac1 (neukoncene z)) )) ))
```

- celkove volanie: ak chceme vratit vysledok v podobe stromu, trojice:
  (ac2 (cons 'fiktivne pismeno slova))

- konec je pomocna funkcia, ktora vracia symbol 'ano alebo 'ne podla toho, ci sa v slovach vyskytuje jednopismenne slovo (tj. take, ktore sa nebude spracovavat rekurzivne).

```
 (define (konec x)
  (if           ; podla toho, ci najdeme jednopismenne slovo
   ( NULL? (filter      ; filtrujeme, viz niekde inde
     (lambda (y) (NULL? (cdr y))) ; test na jednopism slova
     (cdr x)            ; na druhej zlozke vstupu: slovach
   'ne                  ; nenaslo sa
   'ano )))             ; naslo sa
```

- fcia neukoncene filtruje prve pismeno a potom prazdne slovo, ak existuje
 (define (neukoncene ...))

- zmena d.š. : informáciu o tom, či nejaké slovo končí , sme zjednodusili na 'ano/'ne. Pre dalšiu časť algoritmu by bolo vhodné anio nahradit konkrétnym slovom. pri súčasnom návrhu d.š. ale zo slova postupne uberáme písmená a na konci už nemáme dosť informácií, tj. celé slovvo. Jedna z možností je ku každémuu slovu pridať to slovo ešte raz, najlep3ie v podobe vhodnej pre dal3ie spracovanie alebo pre v7stup. (pridat tuto informaciu dokazeme jednoducho pomocou map)
Môžeme ho pridať do dvojice ako prvu zložku, ale to bude zmanenať zásah do kódu na mnohých miestach. Pridáme ho na koniec zoznamu písmen, za posledné. Rekurzívne spracovanie slov sa nezmení, zmení sa ošetrenie koncových podmienkok, tj. fcí. konec a neukončené.
 Reprezentacia slova bude vyzerat takto:
 '(h e r her) alebo '(h e r (h e r))

- v súvislosti s týmito zmenami o výhode typov: zmena reprezentácie spôsobí zmenu typu /alebo zlé otypovanie/. Zmena typu /vstupu/ u volajucej funkcie sposobi nutnost zmeny typu u volanej funkcie. Takto sa zmena reprezentacie vdaka typovej kontrole odpropaguje celým programom, bez nutnosti ladenia.

- /continuation/

```
- /komb. pevného bodu:
   F(YF) = YF, Y= \f.(\x.fxx)(\x.fxx)
   Theta TF->F(TF) , T= (\xy.y(xxy))(\xy.y(xxy))
   Tarau to učí !?/
- /haskell: length x = fold 0 (const (1 +)) x /

------
pár odladených príkladov: rok 1995

; obecny akumulator - s jednotkovým/koncovým prvkom e
;   a s kombinačnou funkciou c, doprava asociatívny
(define (acu e c l)
  (if (null? l)
      e
      (c (car l) (acu e c (cdr l))) ))

; použitia akumulátora - suma a suma druhých mocnín
(define (sum  l) (acu 0 + l))
(define (sum2 l) (acu 0 (lambda (x y) (+ (* x x) y)) l))

; ladiace testy
(write (sum (int 1 10)))
(write (sum2 (int 3 5)))

; použitie acc - minimum
(define (min1 l) (acu (car l) min (cdr l) ))  ;; len neprazdne
; ladiace testy
(write (min1 '(2 3 4)))
(write (min1 '(5 4 3)))

; použitie acc - append
(define (app1 l) (acu () (lambda (x y) (cons y x)) l))

; howgh
```