Lisp 8-1

Parametry funkcí

Volitelné parametry (&optional):
- volitelné parametry jsou nastaveny na nil, pokud nejsou při volání funkce zadány

```
(defun alfa (a b c &optional d e f)
  (list a b c d e f))
(alfa 1 2 3 4) -> (1 2 3 4 nil nil)
```

- volitelné parametry mohou mít přednastavené hodnoty

```
(defun beta (a b c &optional (d 100) (e 200))
  (list a b c d e))
(beta 1 2 3) -> (1 2 3 100 200)
(beta 1 2 3 4) -> (1 2 3 4 200)
```

- přednastavená hodnota může vycházet z povinných argumentů

```
(defun zeta (a b c &optional (d a) (e c))
  (list a b c d e))
(zeta 1 2 3) -> (1 2 3 1 3)
(zeta 1 2 3 4) -> (1 2 3 4 3)
```

Klíčové parametry (&key):
- vhodné pro funkce, kde raději argumenty pojmenujeme a pak nezáleží na jejich pořadí
- pojmenování argumentu uvozuje prefix ":"

```
(defun alfa (a b c &key d e)
  (list a b c d e))
(alfa 1 2 3 :e 100 :d 200) -> (1 2 3 200 100)
```

- nenastavené parametry obsahují nil

```
(defun beta (a &key b c d e)
  (list a b c d e))
(beta 1 :d 100 :c 200) -> (1 nil 200 100 nil)
```

- hodnota může být přednastavena

```
(defun zeta (a &key b (c pepa) d (e 300))
  (list a b c d e))
(zeta 1 :d 800 :e 400) -> (1 nil pepa 800 400)
```

Volitelný počet parametrů (&rest):
- nutné pro funkce s volitelným počtem parametrů, jako např. AND
- volitelné parametry předány jako seznam

```
(defun alfa (a b c &rest s)
  (list a b c s))
(alfa 1 2 3 u v w x y z) -> (1 2 3 (u v w x y z))
```

**AND jako funkce s proměnným počtem parametrů:**

```
(defun my_and (&rest s)
   (cond ((null s) t)
         ((eval (car s)) (apply #'my_and (cdr s)))
         (t nil)
)  )
```

příklad:

```
 (my_and t t t nil) -> nil
 (my_and t t t) -> t
 (my_and '(eq 1 1) '(eql 100 100) '(equal '(1 2) '(1 2))) -> t
```

**OR jako funkce s proměnným počtem parametrů:**

```
(defun my_or (&rest s)
   (cond ((null s) nil)
         ((eval (car s)) t)
         (t (apply #'my_or (cdr s)))
)   )
```

příklad: (my_or nil nil t nil) -> t

```
  (my_or2 nil nil) -> nil
  (my_or2 '(eq 1 2) '(eql 100 200) '(equal '(1 2) '(1 2))) -> t
```

Řídící strategie v Lispu:

**rekurze** x **mapování** x **klasická iterace**

- všeobecně píšeme především čitelný kód
- při programování matematických funkcí se držíme strategie použité při matematické definici
- pokud řešení problému vyžaduje vnořené seznamy, pak se hodí rekurze
- pokud problém spočívá v transformaci jednoho seznamu do jiného, pak se hodí mapovací funkcionály
- pokud se problém skládá z opakovaného provádění daných operací nad danou množinou, pak se hodí klasická iterace

Iterativní prostředky:

(**dotimes** (proměnná počet_opakování výsledek) výraz1 výraz2 ...)

- proměnná postupně proběhne rozsah hodnot (0,počet_opakování>
- výsledek tvoří návratovou hodnotu kostrukce dotimes, pokud není uveden, vrací se nil

**Výpočet $m^n$ pomocí iterace:**

```
(defun m_na_n (m n)
  (let ((vysledek 1))
     (dotimes (citac n vysledek)
           (setf vysledek (* m vysledek))
) ) )
```

příklad:

```
(m_na_n 2 4) -> 16
```

Lisp 8-3

(dolist (proměnná seznam výsledek) výraz1 výraz2 ...)
- opakování se děje přes počet prvků seznamu
- stejné vlastnosti jako dotimes

**Délka seznamu pomocí iterace:**

```
(defun delka(seznam)
  (let ((vysledek 0))
    (dolist (citac seznam vysledek)
      (setf vysledek (+ 1 vysledek))
) ) )
```

příklad:
(delka '(a b c d)) -> 4

Obecná konstrukce - do
(do ((proměnná počáteční_hodnota další_hodnota)) (podmínky_ukončení) výraz1 výraz2 ...)

**Obrácení seznamu pomocí iterace:**

```
(defun obrat (s)
  (let ((obraceny nil))
    (do ((  prom          ;proměnná cyklu
            s             ;počáteční hodnota
            (cdr prom)    ;další hodnota
         ))
        ((null prom))     ;podmínka ukončení

       (setq obraceny (cons (car prom) obraceny))       ;tělo cyklu
    )
    obraceny
) )
```

příklad:
(obrat '(a b c d)) -> (d c b a)

Formy prog1, prog2, implicitní progn:

(prog1 výraz1 výraz2 ... ) -> návratová hodnota je hodnota prvního výrazu
(prog2 výraz1 výraz2 ... ) -> návratová hodnota je hodnota druhého výrazu
(progn výraz1 výraz2 ... ) -> návratová hodnota je hodnota posledního výrazu

- progn je implicitní pro většinu konstrukcí

```
(defun alfa (s)
  (prog1 (first s)
         (second s)
         (third s)
) )
(alfa '(a b c d)) -> a
```

```
(defun alfa2 (s)
  (prog2 (first s)
         (second s)
         (third s)
) )
(alfa2 '(a b c d)) -> b
```

```
(defun alfa3 (s)
  (progn (first s)
         (second s)
         (third s)
) ) ;progn by zde nebylo potřeba
(alfa3 '(a b c d)) -> c
```

Lisp 8-4

Makra

```
(defmacro název (argumenty)
        výraz1
        výraz2
        ....
)
```

- při vyhodnocování makra se do těla předávají nevyhodnocené argumenty
- provede se vyhodnocení výrazů v rámci prostředí určeného definicí makra
  => mluvíme o **expanzi makra**, výsledkem expanze je hodnota posledního výrazu
- výsledek expanze považujeme ze E-výraz (vyhodnotitelný S-výraz), který se znovu vyhodnotí, tentokrát v rámci místa aplikace funkce

```
(defmacro demo1 (arg)
  (list  'list arg "& arg))
(demo1 'Elf) -> (Elf & Elf)

(defmacro demo2 (promena arg)
  (set promena (list 'list arg "& arg)))
(setq x nil)
(setq z 'elf)
(demo2 x z) -> (elf & elf)      ;výsledek  druhého vyhodnocení v rámci prostředí aplikace fce
x -> (list z '& z)              ;výsledek expanze

(defmacro moje_IF (podminka THEN tval ELSE fval)
  (list 'cond (list podminka tval) (list fval)))
```

- po expanzi:
```
(cond ((podminka tval)
       ((fval))
)
```
- slova THEN, ELSE pouze syntaktická vata
- používání lisp je celkem nepřehledné, bohužel nejde použít apostrof (prostředí quote)
  '(cond (podminka tval) (fval)) protože by se dovnitř nedostaly skutečné argumenty
- syntaktické vylepšení => obrácený apostrof "`"
  - chová se stejně jako prostředí quote, ale čárka "," uvedená před S-výraz ruší prostředí quote pro tento S-výraz

```
        (setq fotbal 'hokej)
        (nesnasim fotbal) -> chyba
        '(nesnasim fotbal) -> (nesnasim fotbal)
        `(nesnasim ,fotbal) -> (nesnasim hokej)

(defmacro moje_IF (podminka THEN tval ELSE fval)
  `(cond (,podminka ,tval) (,fval)))

(setq a 5)
(setq b 3)
(moje_IF (> a b) THEN 'a_je_vetsi ELSE 'a_je_mensi) -> a_je_vetsi
```

Lisp 8-5

**Fronta pomocí maker:**

```
(defmacro init (fronta)
  (list 'setq fronta nil))

(defmacro pridat (prvek fronta)
   (list 'setq fronta (list 'append fronta (list 'list prvek))))

(defmacro odebrat (fronta)
   (list 'prog1
        (list 'car fronta)
        (list 'setq fronta (list 'cdr fronta))))
```

příklad:
(init fronta1)
(pridat 'a fronta1)
(pridat 'b fronta1)
(pridat 'c fronta1)
fronta1 -> (a b c)
(odebrat fronta1) -> a
fronta1 -> (b c)

**Zásobník pomocí maker:**

```
(defmacro init (zasobnik)
  (list 'setq zasobnik nil))

(defmacro pridat (prvek zasobnik)
   (list 'setq zasobnik (list 'cons prvek zasobnik))))

(defmacro odebrat (zasobnik)
   (list 'prog1
        (list 'car zasobnik)
        (list 'setq zasobnik (list 'cdr zasobnik))))
```

příklad:
(init zasobnik1)
(pridat 'a zasobnik1)
(pridat 'b zasobnik1)
(pridat 'c zasobnik1)
zasobnik1 -> (c b a)
(odebrat zasobnik1) -> c
zasobnik1 -> (b a)

Lisp 8-6

Základní seznámení s objekty v Lispu:

```
(defclass circle ()
   (radius center))
```

(make-instance 'circle)
- vytvořili jsme instanci třídy circle
- bohužel se nijak nedostaneme k jejím datovým položkám

```
(defclass circle ()
  ((radius :accessor circle-radius)
    (center :accessor circle-center)))
```

(setf (circle-radius (make-instance 'circle)) 2)
- vytvořili jsme instanci
- pomocí přístupové funkce circle-radius jsme zpřístupnili položku radius
- radius jsme nastavili na hodnotu 2

```
(defclass circle ()
  ((radius :accessor circle-radius :initarg :r :initform :1)
    (center :accessor circle-center :initarg :c :initform '(0 . 0))))
```
- máme vytvořeny přístupové funkce
- konstruktor má volitelné parametry "c" a "r" pro nastavení center a radius
- pokud volitelné parametry nebudou použity máme nastaveny inicializační hodnoty

```
(setq kruh1 (make-instance 'circle :r 20))
(circle-radius kruh1) -> 20
(circle-center kruh1) -> (0 . 0)
```

Lisp umožňuje vícenásobnou dědičnost:

```
(defclass super-circle (circle shapes screen)
  .... )
```

defprop, defmethod, defstruct ... více v literatuře, doporučuji on-line dostupnou knihu OnLisp.