Lisp 5-1

Opakování & rozšíření

zastavení vyhodnocování

(list 3 4 5) -> (3 4 5)
(car (list 3 4 5)) -> 3
(car (3 4 5)) -> 3 not a function
(car '(3 4 5)) -> 3
(list t nil "1 retezec" '(+ 2 3) (+ 2 3) 'neco) -> (t nil "1 retezec" (+ 2 3) 5 neco)

práce se seznamy

(car '(1 2 3)) -> 1
(cdr '(1 2 3)) -> (2 3)
(caddr '(1 2 3)) -> 3
(nth 2 '(5 6 7)) -> 7 *;opakovaný car, indexujeme od nuly*
(nth 3 '(5 6 7)) -> nil
(nthcdr 0 '(5 6 7)) -> (5 6 7) *;opakovaný cdr*
(nthcdr 1 '(5 6 7)) -> (6 7)
(reverse '(a b c)) -> (c b a)
(length '(a (b c) d)) -> 3

matematika

(float (/ 3 5)) -> 0.6
(max 1 2 3 4 5) -> 5
min, round, floor, mod, rem ...

funkce

```
(defun moje (x y z)
        (fce1 x y)
        (fce2 y z)
        (+ x y z))
```

- je to makro, nevyhodnocuje svoje parametry
- návratová hodnota je jméno funkce
- jako vedlejší efekt se definice funkce sváže s jejím jménem

(moje arg1 arg2 arg3)
*- zkontroluje se zda symbol **moje** ma asociovanou definici funkce*
- argumenty se vyhodnotí
- hodnoty se naváží na parametry
- vyhodnotí se výrazy v těle funkce .. lexikální vazba proměnných
- výsledkem funkce je hodnota posledního výrazu

komentáře

;;; mimo funkci
;; ve funkci
; za výrazem

dokumentace

(defun neco (x) "dokumentace ....." (výpočet))
(documentation 'neco 'function) -> "dokumentace ....."

navázání hodnot

- hodnoty parametrů funkcí jsou navázány na proměnné při každém volání funkce

- navázání pomocí set a setq
  - umožňují změnu hodnoty proměnné
  - nepatří do čistého Lispu
  - používat pouze v top-level, ne definici funkce

- navázání také můžeme udělat pomocí ***let***

```
(let ( (param1 init_vyraz1)
       (param2 init_vyraz2)
       (              ..... ))
     (vyraz_tela1)
     (vyraz_tela2)
     (           .....  ))
```

- init výrazy se vyhodnotí **paralelně** (tj. žadné předpoklady o pořadí !!!)
- hodnoty se naváží
- vyhodnotí se výrazy těla ... lexikální vazba parametrů
- poslední výraz je výsledkem bloku let

```
(defun name_spaces(s)
  (let (  (first (first s))
          (last (first (reverse s))))
       (list first last)))
```

- místo funkcí car jme použili ekvivalentní first
- namespace proměnných a namespace funkcí je očividně oddělený

```
(defun zastineni (x)
  (let ((x 7))
       (let ((x 5))
            (list x))))
```

(zastineni 3) -> 5

- důsledek lexikální vazby proměnných
- na proměnnou je navázána ta hodnota, která je v kontextu nejblíže

Lisp 5-3

Lexikální a dynamická vazba

- proměnné v Lispu **automaticky** používají lexikální vazbu
- proměnná v lexikální vazbě získává svoji hodnotu v nejbližším kontextu **napsaného** kódu

- existuje i možnost dynamické vazby
- proměnná v dynamické vazbě získává svoji hodnotu v nejbližším kontextu **spuštěného** kódu
- dynamickou proměnnou získáme pomocí operátorů defvar, defparameter, defconstant nebo declare special
- dle konvence se dynamické proměnné označují na začátku i na konci hvězdičkou
- používat opatrně, špatně se hledá chyba

```
(defun scope () x)
(defun scope2 (x) (scope))
(defun scope3 (y) (scope))

(scope2 10) -> unboud variable 'x'
(scope3 10) -> unbound variable 'x'
(defvar x 300)
(scope2 10) -> 10    ;x je dynamická a nejbližší runtime vazba proběhla ve scope2
(scope3 10) -> 300   ;x je dynamická a nejbližší runtime vazba proběhla v top-level
```

```
(defun scope () x)
(defun scope2 (x) (declare (special x)) (scope))
(defun scope3 (y) (scope))

(scope2 20) -> 20
(scope3 20) -> unbound variable 'x'
```

```
(defun scope () x)
(defun scope2 (x) (declare (special x)) (scope))
(defun scope3 (x) (declare (special x)) (scope))

(scope2 20) -> 20
(scope3 30) -> 30
```

Lisp 5-4

Closures - uzávěry

- closure vzniká kombinací funkce a sady vazeb volných proměnných ve funkci
- konkrétně pomocí **defun** nebo **#'** před lambda výrazem

```
(defun elf (x) (list x y))
```
- *elf je closure s dynamicky vázaným y*

```
(elf 10) -> unbound variable 'y'
(let ((y 30)) (elf 10)) -> unbound variable 'y'
(let ((y 30)) (declare (special y)) (elf 10)) -> (10 30)


(let ((y 30)) (defun elf (x) (list x y))
```
- *elf je closure s lexikálně vázaným y (označujeme též jako lexikální closure)*

```
(elf 10) -> (10 30)
(let ((y 300)) (elf 10)) -> (10 30) ; máme přeci lexiální vazbu


(setq alfa '#(lambda (x) (list x y)))
```
- *alfa má navázaný closure s dynamicky vázaným y*

```
(funcall alfa 10) -> unbound variable 'y'
(let ((y 100)) (funcall alfa 10)) -> unbound variable 'y'
(let ((y 100)) (declare (special y)) (funcall alfa 10)) -> (10 100)


(let ((y 100)) (setq alfa #'(lambda (x) (list x y)))
```
- *alfa má navázaný closure s lexikálně vázaným y*

```
(funcall alfa 10) -> (10 100)
(let ((y 200)) (funcall alfa 10)) -> (10 100) ;máme přeci lexikální vazbu
```

Ukázka použití closure ke studiu

```
(defun make-counter ()
  (let ((count 0))
    (list
        #'(lambda () (setq count 0)) ;setq umožňuje modifikovat proměnnou
        #'(lambda () (setq count (1+ count)) ;predikat 1+ je snad jasný (+ 1 count)
        #'(lambda ()  count))))
```

- **make-counter** vrací list třech lexikálních closure
- první prvek listu je funkce, která resetuje čítač
- druhý prvek listu je funkce, která zvyšuje čítač
- třetí element listu je funkce, která vrací hodnotu čítače
- všechny tři funkce žijí s tou samou lexikální proměnnou **count**
- při každém volání **make-counter** vzniká jiná proměnná **count**

```
(setq citac1 (make-counter))
(setq citac2 (make-counter))

(funcall (second citac1)) -> 1 ;zvýšili jsme čítač, význam second odpovídá (nth 1 ...)
(funcall (second citac1)) -> 2
(funcall (third citac1)) -> 2
(funcall (third citac2)) -> 0 ;opravdu je citac2 nezávislý
```

func pro vstup a výstup

(princ <object>) ... popis objektu

(terpri) ... nový řádek

(format <stream> <formatstring> <obj>*) ... formátovaný výstup
- <stream> t - tiskne na obrazovku, vrací nil
  nil - netiskne, ale vrací výstup jako návratovou hodnotu
- <formatstring> ~% - nový řádek
  ~A - řetězec bez ""
  ~S - řetězec s ""
  ~C - znak

(read) ... čte další objekt (atom, číslo, seznam), nic nevyhodnocuje

Dvojnásobek zadané hodnoty:

```
(defun dvojnasobek nil
  (let (x)
    (princ "zadej číslo: ") ;reprezentací stringu je zase string
    (setq x (read)) ;co přečtem z klávesnice uložíme do proměnné
    (terpri)          ;odřádkujeme
    (princ "dvojnasobek je: ")
    (princ (* 2 x)) ;vypíšeme výsledek
    (terpri)))
```

příklad: (dvojnasobek)

Formátovaný výstup - příklady

```
(format t "text") -> text     ;tisk na obrazovku, nil návratová hodnota
(format nil "text") -> "text"   ;jen návratová hodnota

(setq x 123.45)
(format t "x=~S" x) -> x=123.45
(setq y (format nil "x=~S" x))
(princ y)
 -> x=123.45     ;tisk na obrazovku
    "x=123.45"   ;návratová hodnota
```

Lisp 5-6

Průnik dvou seznamů:

```
(defun prunik(a b)
  (declare (special b))  ;dynamická vazba z parametru funkce aby šel použít v lambda výrazu
  (mapcan #'(lambda (x) (my_member x b)) a))
```

příklad: (prunik '(a a x b) '(v b f a))

```
(defun my_member(z s)
  (declare (special z))  ; dynamická vazba z parametru funkce aby sel použít v labda výrazu
  (mapcan #'(lambda (x)
                (cond ((equal z x) (list z))
                      (t nil))
            ) s)
)
```

příklad: (my_member 'a '(b d a)) -> (a)

Výběr prvků splňující zadanou funkci:

```
(defun vybrat_pokud (s fce)
  (cond ((null s) nil)
      ((funcall fce (car s)) (cons (car s) (vybrat_pokud (cdr s) fce)))
      (t (vybrat_pokud (cdr s) fce))
)  )
```

příklad: (vybrat_pokud '(1 -3 2 -4) '(lambda (n) (and (evenp n) (minusp n)))) -> (-4)

Rotace seznamu:

```
(defun rotace_zpet(s)
  (cons (car (reverse s)) (reverse (cdr (reverse s)))))
```

;tohle už známe a provádět 2x reverse není moc efektivní

```
(defun rotace_zpet2(s)
  (let ((x (reverse s)))
    (cons (car x) (reverse (cdr x)))))
```

;efektivnější verze s použitím lokální proměnné

příklad: (rotace_zpet '(a b c d)) -> (d a b c)
příklad: (rotace_zpet2 '(a b c d)) -> (d a b c)