LISP – Definice funkcí

(DEFUN jméno-fce (argumenty) tělo-fce)

•Přiřadí jménu-fce lambda výraz definovaný tělem-fce, tj. (LAMBDA (argumenty) tělo-fce). Vytvoří funkční vazbu symbolu jméno-fce

Struktura symbolu:

Jméno	vazba na hodnotu	seznam vlastností	funkční vazba

•Argumenty jsou ve fci lokální

•DEFUN nevyhodnocuje své argumenty

•Hodnotou formy DEFUN je nevyhodnocené jméno-fce

•Tělo-fce je posloupností forem. Při vyvolání fce se všechny vyhodnotí. Funkční hodnotou je hodnota poslední z forem

LISP – Definice funkcí

```
>(DEFUN max2 (x  y)  (IF  (>  x  y)  x  y ) )
max2
(max2  10  20 )
20
>DEFUN  max4  (x  y  u  v)  (max2  (max2  x  y)  (max2  u  v) ) )
max4
>(max4  5  9  12  1)
```

Interaktivní psaní lispovských programů způsobuje postupnou demenci $\Rightarrow$ možnost load ze souboru

(LOAD “jmeno-souboru”)

(LOAD “D:\\moje\\LISP\\soubor.lsp”)

Př. 1max.lsp

LISP – Definice funkcí

```
(defun max2(x y) (if (> x y) x y))
(defun ma(n) (if
;;;        (equal (list-length n) 2)
                (equal (delka n) 2)
                  (max2 (car n) (car (cdr n)))
                  (max2 (car n) (ma (cdr n)))
              )
)
(defun delka(n)
    (if (equal n nil)
              0
              (+ 1 (delka (cdr n))) ))
```

LISP – Definice funkcí

Př. 2sude-poradi.lsp

```
 ;;;vybira ze seznamu prvky sude v poradi
(defun sude (x)
   (cond
   ((not (null (cdr x)))
         (cons(car (cdr x))(sude (cdr (cdr x)))))
   (t nil)
))
```

LISP – Definice funkcí

Př.3NSD-Fakt.lsp

```
(defun nsd (x y)
   (cond ((zerop (- x y)) y)
                 ((> y x) (nsd x (- y x)))
                 (t (nsd y (- x y)))
))

(defun fakt (x)
   (cond        ((= x  0) 1)
                (T (* x  (fakt (- x 1))))
))
```

LISP – Definice funkcí

Redefinice append (:c ignorujeme zamknutí makra)

```
(DEFUN APPEND (X  Y)   ;;;pro dva argumenty
   (IF (NULL  X)
               Y
               (CONS  (CAR  X) (APPEND (CDR  X)  Y))
) )

(DEFUN MEMBER  (X  S)  ;;; je jiz take mezi standardnimi
   (COND     ((NULL  S)  NIL)
        ((EQUAL  X  (CAR  S))  T)          ;;; S
        (T  (MEMBER  X  (CDR  S)))
) )
```

(MEMBER ‘X ‘(A B X Z)) → T versus (X Z)

LISP – Další standardní funkce

Ad aritmetické

(- 10 1 2 3 4) → 0

(/ 100 5 4 3) → 5/3

Ad operace na seznamech

(LIST-LENGTH ‘(1 2 3 4 5)) → 5

(LAST ‘(1 3 2 8 (4 5))) → ((4 5))

```
(DEFUN LAST (S)
   (COND ((NULL (CDR S))
                        S
                        (LAST (CDR S))
))
```

LISP – Další standardní funkce

Ad vstupy a výstupy

(OPEN soubor :DIRECTION směr) otevře soubor a spojí ho s novým proudem, který vrátí jako hodnotu.

Hodnotou je jmeno proudu (= souboru)

Např.

(SETQ S (OPEN “d:\\moje\\data.txt” :direction :output)) ;;; :input

Standardně je vstup z klávesnice, výstup obrazovka

(CLOSE proud) zapíše hodnoty na disk a uzavře daný proud (přepne na standardní)

Např. (CLOSE S)

(READ) (READ proud) načte lispovský objekt

(PRINT a)	(PRIN1 a)	(PRINC a)
řádka, a, mezera	jen a	řetězec bez uvozovek
(PRINT a proud)	"	"

(TERPRI) nový řádek

Př.5Average.lsp Výpočet průměrné hodnoty

```
(defun sum(x)
    (cond ((null x) 0)
          ((atom x) x)
          (t (+ (car x) (sum (cdr x))))))
(defun count (x) ;;; je take mezi standardnimi
    (cond ((null x) 0)
          ((atom x) 1)
          (t (+ 1 (count (cdr x))))))

(defun avrg ()          ;;;main program je posloupnost forem
    (print "napis seznam cisel")
    (setq x (read))
    (setq avg (/ (sum x) (count x)))
    (princ "prumer je ")
    (print avg))
```

Př. 6hanoi.lsp výstup příkazem format

(FORMAT	**cíl**	**řídící řetězec**	**argumenty)**
na obrazovku	t	~%	odřádkování
netiskne ale vrátí	nil	~a	řetězcový argument
Na soubor	proud	~s	symbolický výraz
		~d	desítkové číslo

```
(DEFUN hanoi (n from to aux)
   (COND   ((= n 1) (move from to))
         (T       (hanoi (- n 1) from aux to)
                  (move from to)
                  (hanoi (- n 1) aux to from)
)))
(DEFUN move (from to)
   (format T  "~%move the disc from ~a to ~a." from to)
)
```

LISP – Další standardní funkce

Ad funkce pro řízení výpočtu

(WHEN test formy) ; je-li test T je hodnotou hodnota poslední formy

(DOLIST (prom seznam) forma) ; váže prom na prvky až do vyčerpání seznamu a vyhodnocuje formu

Př. (DOLIST (x ‘(a b c)) (print x)) → a b c

(LOOP formy) ;opakovaně vyhodnocuje formy až se provede forma return

Př. (SETQ a 4) → 4

(LOOP (SETQ a (+ a 1)) (WHEN (> a 7) (return a))) → 8

(DO ((var1 init1 step1) … (varn initn stepn)) ;;inicializace

(testkonce forma1 forma2 ...formam) ;;na konci je provede

formy-prováděné-při-každé-iteraci)

Př. 7fibo.lsp (N-tý člen = člen N-1 + člen N-2)

```
(defun fibon(N)
    (cond
        ((equal N 0) 0)   ;;;trivialni pripad
        ((equal N 1) 1)   ;;;  “      “
        ((equal N 2) 1)   ;;;  “      “
        (T (foo (- N 2)))
))

(defun  foo(N)
    (setq F1 1)    ;;; clen n-1
    (setq F2 0)    ;;; clen n-2
    (loop
        (setq F (+ F1 F2))          ;;; clen n
        (setq F2 F1)                ;;;  novy clen n-2
        (setq F1 F)                 ;;; clen n-1
        (setq N (- N 1))
        (when (equal N 0) (return F))
))
```

Co se tiskne? (8DOpriklad.lsp)

```
(DO (( x  1 (+  x  1))  (y  10  (*  y  0.5))) ;soucasna inicializace
   ((>  x  4)  y)
   (print  y)
   (print ‘pocitam)
)
```

10

POCITAM

5.0

POCITAM

2.5

POCITAM

1.25

POCITAM

0.625

Př.8NTA.lsp (nalezení pořadím zadaného členu seznamu)

```
(setq v "vysledek je ")
(defun nta (S x)
   (do ((i 1 (+ i 1)))
      ((= i x) (princ v) (car S)) ;test konce a vysledna forma
      (setq S (cdr S))
   ))
(defun delej () (nta (read) (read)))
```

Dá se také zapsat elegantně neefektivně = rekurzivě

```
(defun nty (S x)
   (cond ((= x 0) (car S))
            (T (nty (cdr S) (- x 1)))
   ))
```

LISP – Další standardní funkce

(EVAL a) vyhodnotí výsledek vyhodnocení argumentu

Př.

```
>(EVAL (LIST ‘REST (LIST ‘QUOTE ‘(2  3  4))))
(3  4)
>(EVAL (READ))
(+  3  2)
5
>(EVAL (CONS ‘+  ‘(2  3  5)))
10
(SET ‘A  2)
(EVAL  (FIRST  ‘(A  B)))
2
```

LISP – Další standardní funkce

(LET ((prom1 vyraz1) (prom2 vyraz2) … (prom vyrazm))

forma forma … forma)

Dovoluje zavest lokalni proměnné promk s počátečními hodnotami vyrazk. Vyrazy se nejprve všechny vyhodnotí a pak přiřadí.

Poté se vyhodnotí formy, poslední udává hodnotu LET formy

Př.

```
(LET ((pv1 (+ x  y)) (pv2 (- x y)))
        (SQRT (+ (* pv1 pv2) (* pv1 pv1) (* pv2 pv2))) )
```

Nepočítá opakovaně podvýrazy

Pozn. LET* je obdobná, ale vyhodnocuje prom postupně

LISP – Další standardní funkce

Ad sekvenční vyhodnocování forem

(PROG (lok-proměnné) forma-1 forma-2 … forma-n)

(RETURN forma) **(GO návěští)**

Př.
```
(DEFUN Iteracne-MEMBER (atm S)
      (PROG ( )
        opakuj
          (COND ((NULL S) (RETURN NIL))
                ((EQUAL atm (CAR S)) (RETURN T))
          )
          (SETQ S (CDR S))
          (GO opakuj)
  ))
```

LISP – Další standardní funkce

```
(DEFUN iteracne-LENGHT (S)
   (PROG (sum)
        (SETQ  sum  0)
      opakuj
        (COND  ((ATOM  S)  (RETURN  sum))
        )
        (SETQ  sum  (+  1  sum))
        (SETQ  S  (CDR  S))
        (GO  opakuj)
))
```

LISP – rozsah platnosti proměnných

```
(DEFUN co-vraci (Z)
   (LIST (FIRST Z) (posledni-prvek))
)
(DEFUN  poslední-prvek ( )
   (FIRST (LAST  Z))
)
>(SETQ  Z  ‘(1  2  3  4))
(1  2  3  4)
>(co-vraci ‘(A  B  C  D))
```

(A 4) u statickeho rozsahu platnosti -CLISP

(A D) u dynamickeho rozsahu platnosti -GCLISP

- **Lisp pracuje se symbolickými daty.**
- **Dovoluje funkcionální i procedurální programování.**
- **Funkce a data Lispu jsou symbolickými výrazy.**
- **CONS a NIL jsou konstruktory, FIRST a REST jsou selektory, NULL testuje prázdný seznam, ATOM, NUMBERP, SYMBOLP, LISTP testují typ dat, =, EQ, EQUAL, testují rovnost, <, >, … testují pořadí**
- **SETQ, SET přiřazují symbolům globální hodnoty**
- **DEFUN definuje funkci, parametry jsou v ní lokální.**
- **COND umožňuje výběr alternativy.**
- **AND, OR, NOT jsou logické funkce.**
- **Proud je zdrojem nebo konzumentem dat. OPEN jej otevře, CLOSE jej zruší.**
- **PRINT, PRIN1, PRINC TERPRI zajišťují výstup.**
- **READ zabezpečuje vstup.**
- **LET dovoluje definovat lokální proměnné.**
- **EVAL způsobí explicitní vyhodnocení.**
- **Zápisem funkcí a jejich kombinací vytváříme formy (vyhodnotitelné výrazy).**
- **Lambda výraz je nepojmenovanou funkcí**
- **V Lispu má program stejný syntaktický tvar jako data.**

Co to pocita? – 91Co.lsp

```
(DEFUN  co1 (list)
   (IF (NULL list)  ( )
                    (CONS (CAR list) (co2 (CDR list)))
))

(DEFUN  co2 (list)
   (IF (NULL list)  ( )
                     (co1  (CDR list ))
))
```

Jak to napsat jinak

```
(DEFUN ODDS (list)
   (IF (OR (NULL list) (NULL (CDR list)))  list ;cast then
         (CONS (CAR list)  (ODDS (CDDR list)))  ;    else
))

(DEFUN EVENS (list)
   (IF (NULL list) ()
                  (ODDS (CDR list))
))
```

LISP – schéma rekurzivního výpočtu

S jednoduchým testem

```
(DEFUN fce (parametry)
   (COND     (test-konce    koncová-hodnota);;primit.příp.
             (test    rekurzivní-volání) ;;redukce úlohy
))
```

S násobným testem

```
(DEFUN fce (parametry)
   (COND     (test-konce1          koncová-hodnota1)
             (test-konce2          koncová-hodnota2)
                    . . .
              (test-rekurze rekurzivní-volání)
                    . . .
))
```

Př.92rekurze.lsp

```
;;odstrani vyskyty prvku e v nejvyssi urovni seznamu S
(DEFUN delete (e  S)
    (COND        ((NULL S) NIL)
            ((EQUAL e (CAR S)) (delete e (CDR S)))
            (T (CONS (CAR S) (delete e (CDR S)))) ))

;;zjisti maximalni hloubku vnoreni seznamu;; MAX je stand. fce
(DEFUN max_hloubka (S)
    (COND        ((NULL S) 0)
            ((ATOM (CAR S)) (MAX 1 (max_hloubka (CDR S))))
            (T (MAX (+ 1 (max_hloubka (CAR S)))
                        (max_hloubka (CDR S)) )) ));;nasobna redukce

;;najde prvek s nejvetsi hodnotou ve vnorovanem seznamu
(DEFUN max-prvek (S)
    (COND        ((ATOM S) S)
            ((NULL (CDR S)) (max-prvek (CAR S)))
            (T (MAX (max-prvek (CAR S)) ;;nasobna redukce
                        (max-prvek (CDR S)) ))  ))
```

LISP - Funkcionály

Funkce, jejichž argumentem je funkce nebo vrací funkci jako svoji hodnotu. Vytváří programová schémata, použitelná pro různé aplikace. (Higher order functions)

Př. *pro každý prvek s seznamu S proveď f(s)*

to je schéma

Programové schéma pro zobrazení

f : (s1, s2, . . ., sn) → (f (s1), f (s2), . . ., f (sn))

```
(DEFUN zobrazeni (S)
    (COND        ((NULL S) NIL)
                 (T (CONS        (transformuj (FIRST S))
                                 (zobrazeni (REST S)) ))
))
```

LISP - Funkcionály

Programové schéma filtru

```
(DEFUN filtruj (S)
   (COND        ((NULL S) NIL)
                ((test-prvku (FIRST S))
                        (CONS (FIRST S) (filtruj (REST S))) )
                (T  (filtruj (REST S)))  ))
```

Programové schéma nalezení prvého prvku splňujícího predikát

```
(DEFUN najdi-prvek (S)
   (COND        ((NULL S) NIL)
                ((test-prvku (FIRST S))  (FIRST S))
                (T (najdi-prvek (REST S))) ))
```

Programové schéma pro zjištění zda všechny prvky splňují predikát

```
(DEFUN zjisti-všechny (S)
   (COND        ((NULL S) T)
                ((test-prvku (FIRST S) (zjisti-všechny (REST S)))
                (T NIL) ))
```

LISP - Funkcionály

- Při použití schéma nahradíme název funkce i jméno uvnitř použité funkce skutečnými jmény.
- Abychom mohli v těle definice funkce použít argument v roli funkce, je třeba informovat LISP, že takový parametr musí vyhodnotit pro získání popisu funkce.

?Př.?

Schéma aplikace funkce na každý prvek seznamu

```
(DEFUN aplikuj-funkci-na-S (funkce  S)
    (COND        ((NULL  S) NIL)                    ?
                 (T (CONS        (funkce (FIRST S))
                                 (aplikuj-funkci-na-S  funkce (REST
    S)) ) )
```

-FUNCALL je funkcionál, aplikuje funkci na argumenty

```
(DEFUN aplikuj-funkci-na-S (funkce  S)
    (COND        ((NULL  S) NIL)
                 (T (CONS        (funcall funkce (FIRST S))
                                 (aplikuj-funkci-na-S  funkce (REST
    S)) ) )
```

```
                                        ?
(aplikuj-funkci-na-S    car    '((a b) (c d)) )
(aplikuj-funkci-na-S    'car    '((a b) (c d)) )
(a  c)
```

LISP - Funkcionály

Tvar:

(<jméno funkcionálu > <získání popisu fce>
<argumenty fce>)

(FUNCTION jméno fce) dtto #'jméno fce

(FUNCTION lambda výraz)

Vyhodnocení formy FUNCTION vrací hodnotu funkční vazby symbolu „jméno fce“ (u vestavěných kompilovaných funkcí adresu kódu funce, u ostatních lambda výraz z definice funkce).

(FUNCTION je komplementem DEFUN)

Odlišnost:

FUNCTION	QUOTE
Získání popisu funkce	zabráněnívyhodnocení

LISP - Funkcionály

(FUNCALL #'fce argumenty) aplikuje fci na argumenty

(FUNCALL #'CONS '(a b) '(1 2))

((a b) 1 2)

(SETQ PRVNI #'CAR)

(FUNCALL PRVNI '(a b c))

a

(APPLY #'fce seznam) aplikuje fci na prvky seznamu

(APPLY #'CAR '((1 2 3)))

1

(SETQ f #'<)

(APPLY f '(1 2 3 4))

(FUNCALL f 1 2 3 4)

(**APPLY #'f '(argumenty) ≡ (f 'argument … 'argument)**

LISP - Funkcionály

MAPCAR aplikuje fci na prvky seznamů, které jsou dalšími argumenty, z výsledků vytvoří seznam

(MAPCAR (FUNCTION +) ‘(1 2 3 4) ‘(1 2 3))

(2 4 6)

(MAPCAR #’- ‘(1 2 3) ‘(1 2 3 4) ‘(2 2 2 2))

(-2 -2 -2)

(MAPCAR #’APPEND ‘((a b) (c)) ‘((x) (y z)))

((a b x) (c y z)

(SETQ f #’<) (MAPCAR f ‘(1 2 3)) ‘(12 3 2))

(T T NIL)

MAPLIST aplikuje fci na seznamy, pak na CDR každého ze seznamů pak na CDDR ..., až jeden ze seznamů bude NIL

(MAPLIST #’APPEND ‘((a b) (c)) ‘((x) (y z)))

(((a b) (c) (x) (y z)) (c) (y z)))

LISP - Funkcionály

FIND-IF nalezne prvý prvek seznamu, vyhovující predikátu

FIND-IF-NOT „ ----------„ ne „----------„

(FIND-IF #’SYMBOLP ‘(3 (a) b 1 c))

B

(FIND-IF #’(LAMBDA (N) (> N 5)) ‘(2 3 1 8 9))

8

COUNT-IF a (**COUNT-IF_NOT**) zjistí počet prvků seznamu, které (ne)splňují predikát

(COUNT-IF #‘SYMBOLP ‘(3 (a) b 1 c))

2

REMOVE-IF a **REMOVE-IF-NOT**

(REMOVE_IF #’SYMBOLP ‘(1 A (1) (2 3 4) B))

(1 (1) (2 3 4))

(MAPCAR #'(LAMBDA (N) (* N N)) '(1 2 3 4))

(1 4 9 16)

```
(defun deriv (e x);; 95DERIV.LSP
  (cond ((equal e x) 1)
    ((atom e )   0)
    ((equal (car e) '+)
       (cons '+ (maplist (function (lambda(j) (deriv (car j) x)))
                                 (cdr e))))
    ((equal (car e) '*)
          (cons '+ (maplist (function (lambda(j) (cons '*
                                 (maplist (function (lambda(k) (cond
                                                 ((equal k j)(deriv (car k) x))
                                                 (t (car k))
                                              ))) (cdr e))
                                             ))) (cdr e))))
    ((equal (car e) 'sin)
          (list '* (list 'cos (cadr e)) (deriv (cadr e) x)))
    (t '(neznamy operator)) ))
```