Funkcionální programování úvod

Probereme základy jazyka LISP. Plný popis jazyka najdete např.na http://www-2.cs.cmu.edu/afs/cs/project/ai-repository/ai/html/cltl/cltl2.html

- **Imperativní jazyky jsou založeny na von Neumann architektuře**
  - **primárním kriteriem je efektivita výpočtu**
  - **Modelem je Turingův stroj**
  - **Základní konstrukcí je příkaz**
  - **Příkazy mění stavový prostor programu**
- **Funkcionální jazyky jsou založeny na matematických funkcích**
  - **Program definuje funkci**
  - **Výsledkem je funkční hodnota**
  - **Modelem je lambda kalkul (Church 30tá léta)**
  - **Základní konstrukcí je výraz (definuje algoritmus i vstup)**
  - **Výrazy jsou:**
    - **Čisté = nemění stavový prostor programu**
    - **S vedlejším efektem = mění stavový prostor**

Funkcionální programování úvod

Vlastnosti čistých výrazů:

- **Hodnota výsledku nezávisí na pořadí vyhodnocování (tzv.Church-Roserova vlastnost)**
- **Výraz lze vyhodnocovat paralelně , např ve výrazu (x*2+3)/(fce(y)*x) lze pak současně vyhodnococat dělence i dělitele. Pokud ale fce(y) bude mít vedlejší efekt a změní hodnotu x, nebude to čistý výraz a závorky paralelně vyhodnocovat nelze.**
- **nahrazení podvýrazu jeho hodnotou je nezávislé na výrazu, ve kterém je uskutečněno (tzv. referenční transparentnost)**

  -vyhodnocení nezpůsobuje vedlejší efekty

  -operandy operace jsou zřejmé ze zápisu výrazu

  -výsledky operace jsou zřejmé ze zápisu výrazu

Funkcionální programování- úvod

Př. nalezení největšího čísla funkcionálně vyjádřeno

a) Ze dvou čísel.

označme symbolem def definici fce

max2(X, Y) def jestliže X > Y pak X jinak Y

b) Ze čtyř

max4(U, V, X, Y) def max2(max2(U, V), max2(X,Y))

c) Z n čísel

max(N) def jestliže délka(N) = 2

pak max2(prvý-z(N), duhý-z(N))

jinak max2(prvý-z(N), max(zbytek(N)))

Prostředky funkcionálního programování jsou:

- **Kompozice složitějších funkcí z jednodušších**
- **rekurze**

Funkcionální programování- úvod

Def.: Matematická fce je zobrazení prvků jedné množiny, nazývané definiční obor fce do druhé množiny, zvané obor hodnot

- **Funkcionální program je tvořen výrazem E.**
- **Výraz E je redukován pomocí přepisovacích pravidel**
- **Proces redukce se opakuje až nelze dále redukovat**
- **Tím získaný výraz se nazývá normální formou výrazu E a je výstupem funkcionálního programu**

Př. Aritmetický výraz E = (4 +7+ 10) * (5 – 2)=(11+10)*(5-2) = 20*(5-2) =20*3=60

s přepis. pravidly určenými tabulkami pro +, *, …

Smysl výrazu je redukcemi zachován = vlastnost referenční transparentnosti je vlastností vyhodnocování. funkcionálního programu

Funkcionální programování- úvod

- **Church-Roserova věta:** Získání normální formy je nezávislé na pořadí vyhodnocování subvýrazů

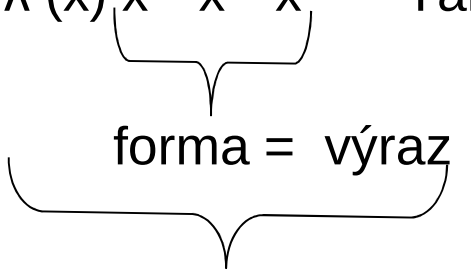
- Funkcionální program sestává z definic funkcí (algoritmu) a aplikace funkcí na argumenty (vstupní data).
- Aparát pro popis funkcí je tzv **lambda kalkul**

  používá operaci aplikace fce F na argumenty A, psáno FA

  „ „ abstrakce ve tvaru λ (x) M [x]

  definuje fci (zobrazení) x → M [x]

Př. λ (x) x * x * x Tak to píší matematici, v programu je to uzávorkováno



definuje bezejmennou fci x * x * x lambda výrazem

Funkcionální programování- úvod

- Lambda výrazy popisují bezejmenné fce
- „ „ „ jsou aplikovány na parametry např.

  (λ (x) x * x * x) 5 = 5 * 5 * 5 = 125

  popis funkce argumenty

  aplikace (vyvolání) funkce
- Ve funkcionálním zápisu je zvykem používat prefixovou notaci ~ funkční notaci ve tvaru funktor(argumenty)

  př. ((lambda (x) (* x x)) 5)

  ((lambda (y) ((lambda (x) (+ (* x x) y)) 2)) 3)

  → 7 přesvědčíme se v Lispu?

Funkcionální programování- úvod

V předchozím příkladu výraz ((lambda (x) (+ (* x x) y)) 2)) obsahuje vázanou proměnnou x na 2 a volnou proměnnou y. Ta je pak ve výrazu ((lambda (y) ((lambda (x) (+ (* x x) y)) 2)) 3) **vázána na 3**

V LISPu lze zapisovat také

((lambda (y x) (+ (* x x) y)) 2 3) dá to také 7 ?

((lambda (y x) (+ (* x x) y)) 3 2)

Ta upovidanost s lambda má důvod v přesném určení pořadí jaký skutečný argument odpovídá jakému formálnímu.

Pořadí vyhodnocování argumentů lze provádět:

- **Všechny se vyhodnotí před aplikací fce = eager (dychtivé) evaluation**
- **Argument se vyhodnotí těsně před jeho použitím v aplikaci fce = lazy (líné) evaluation**

Pochopit význam pojmu

- **Volná proměnná**
- **Vázaná proměnná**

Funkcionální programování- LISP

- Vývoj, verze: Maclisp, Franclisp, Scheme, Commonlisp, Autolisp
- Použití:
  - UI (exp.sys., symb.manipulace, robotika, stroj.vidění,přiroz.jazyk)
  - Návrh VLSI
  - CAD
- Základní vlastnost: **Vše je seznamem** (program i data)

Ovládání CLISPu

Po spuštění se objeví prompt [cislo radku]>

Interaktivně lze zadávat příkazy např (+ 2 3). LISP ukončíte zápisem (exit).

Uděláte-li chybu, přejdete do debuggeru, v něm lze zkoušet vyhodnocení, navíc zápisem Help se vypíší další možnosti, např Abort vás vrátí o 1 úroveň z debuggerů zpět (při chybování v debug.se dostanete do dalšího debug. vyšší úrovně). Program se ale většinou tahá ze souboru

K zatažení souboru s funkcemi použijte např.

(LOAD “d:\\moje\\pgs\\neco.lsp”) když se dobře zavede, odpovi T

Funkcionální programování- LISP

Datové typy:

-atomy:
- -čísla (celá i reálná)
- -znaky
- -řetězce "to je řetězec"

S - výrazy
- -symboly (T, NIL, ostatní)
  k označování proměnných
  a funkcí. T je pravda, NIL nepravda
- -seznamy (el1 el2 ...elN), NIL je prázdný seznam jako ()

- Seznamy mohou být jednoduché a vnořované
  např (sqrt (+ 3 8.1)) je seznam použitelný k vyvolání fce
- Velká a malá písmena se nerozlišují v CommonLispu

Funkcionální programování- LISP

Postup vyhodnocování (často interpretační):

Cyklus prováděný funkcí eval:

1. **Výpis promptu**
2. **Uživatel zadá lispovský výraz (zápis fce)**
3. **Provede se vyhodnocení argumentů**
4. **Aplikuje funkci na vyhodnocené argumenty**
5. **Vypíše se výsledek (fční hodnota)**

- **Pár příkladů s aritmet. funkcemi spustíme v Lispu**

(+ 1111111111111111 222222222222222)

má nekonečnou aritmetiku

(sqrt (* 4 pi) zná matem fce a pi

2*2 způsobí chybu

- **Při chybě se přejde do nové úrovně interpretu**
- **V chybovém stavu lze napsat help ¬**
- **Abort = návrat o úroveň výše**

Funkcionální programování- LISP

- Funkce pracují s určitými typy hodnot
- Typování je prostředkem zvyšujícím spolehlivost programů
- Typová kontrola:

  -statická znáte z Javy

  -dynamická (Lisp, Prolog, Python) – nevyžadují deklaraci typů argumentů a funkčních hodnot

Zajímavosti -komplexní čísla (* #c(2 2) #c(1.1 2.1)) dá výsledek #c(-1.9999998 6.3999996)

-operátory jsou n-ární (+ 1 2 3 4 5) dá výsledek 15

-nekonečná aritmetika pro integer

Funkcionální programování- LISP

Elementární funkce (teoreticky lze s nimi vystačit pro zápis jakéhokoliv algoritmu, je to ale stejně neohrabané jako Turingův stroj

- **CAR alias FIRST** **selektor-vyběr prvého prvku**
- **CDR alias REST** **selektor-výběr zbytku seznamu (čti kúdr)**
- **CONS** **konstruktor- vytvoří dvojici z agumentů**
- **ATOM** **test zda argument je atomický**
- **EQUAL** **test rovnosti argumentů**

Ostatní fce lze odvodit z elementárních

např. test prázdného seznamu funkcí NULL

(NULL X) je stejné jako (EQUAL X NIL)

Lisp mu předloženou formuli vyhodnocuje

(prvouČástPovažujeZaFunkci dáleNásledujíJejíArgumenty)

Argumenty se LISP snaží vyhodnotit, což mnohdy nechceme

Jak zabránit vyhodnocování – je na tgo fce QUOTE zkracovaná apostrofem

Př. (FIRST ‘(REST (1 2 3))) (FIRST (REST ‘(1 2 3)))

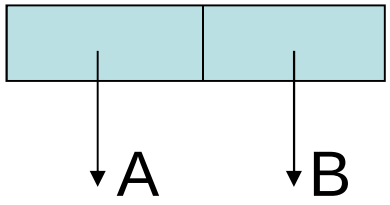
dá hodnotu REST dá hodnotu 2

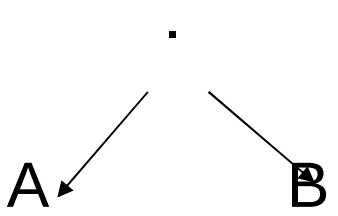
Funkcionální programování- LISP

Zobrazení, jak CONS vytváří lispovské buňky – tzv tečka dvojice

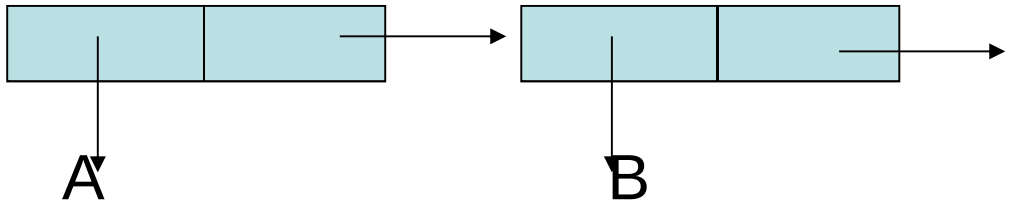
(CONS ‘a ‘b) $\rightarrow$ (A . B) cons dvou atomů je tečka dvojice

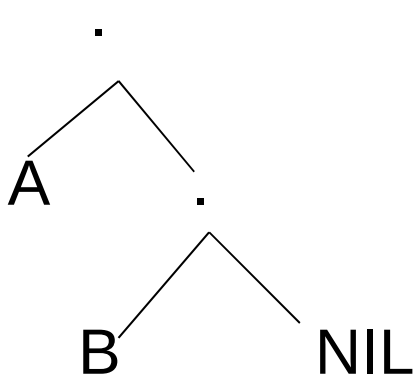
Lispovská buňka





(CONS ‘a ‘(b)) cons s druhým argumentem seznam je seznam





Funkcionální programování- LISP

Převod tečka notace do seznamové notace:

- **Vyskytuje-li se „. „ před „(„ lze vynechat „. „ i „(„ i odpovídající „)„**
- **Při výskytu „. NIL„ lze „. NIL„ vynechat**

Př. ‘((A . Nil) . Nil) je seznam ((A)) !!je třeba psát mezera tečka mezera

‘(a . Nil) je seznam (A)

‘((a . Nil) . ((b . (c . Nil)) . Nil)) je seznam ((A) (B C))

Seznam je takový S-výraz, který má na konci NIL

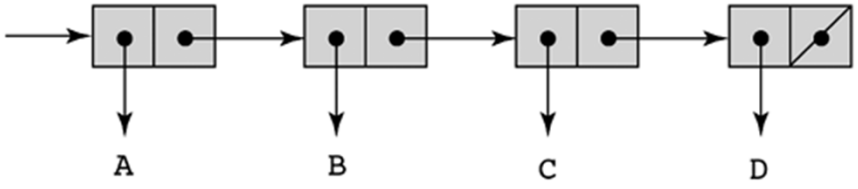
Forma (také formule) je vyhodnotitelný výraz

K řádnému programování v Lispu a jeho derivátech je potřebný editor hlídající párování závorek, který k freewarové verzi nemám

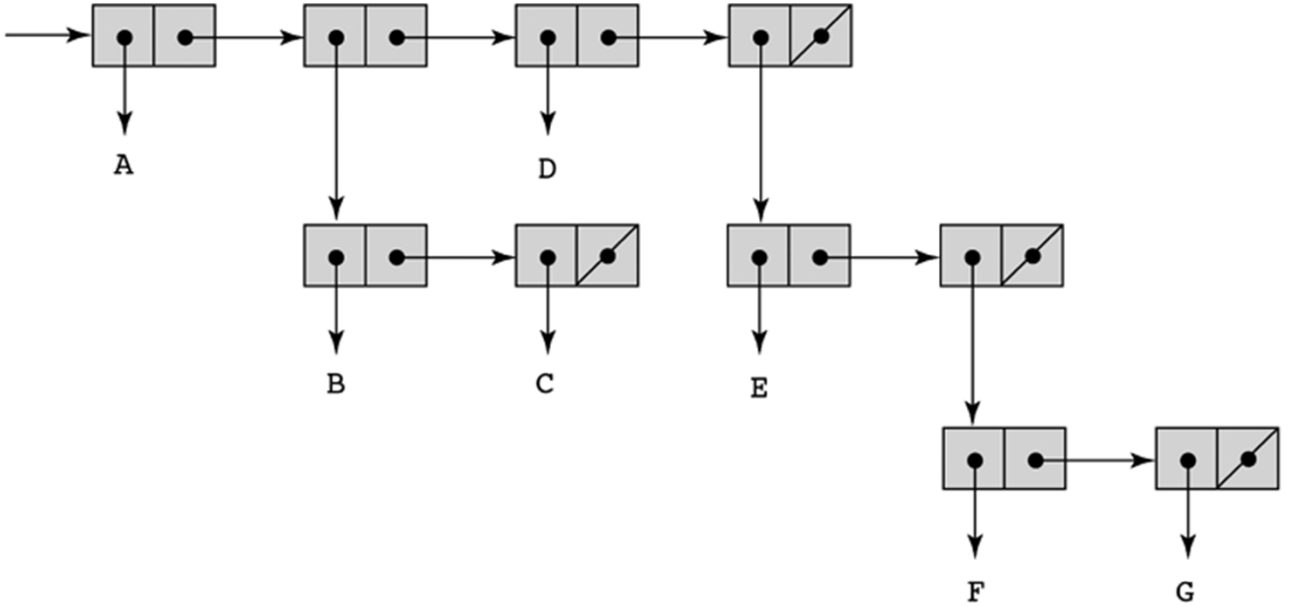
Funkcionální programování- LISP

Interní reprezentace

seznamů



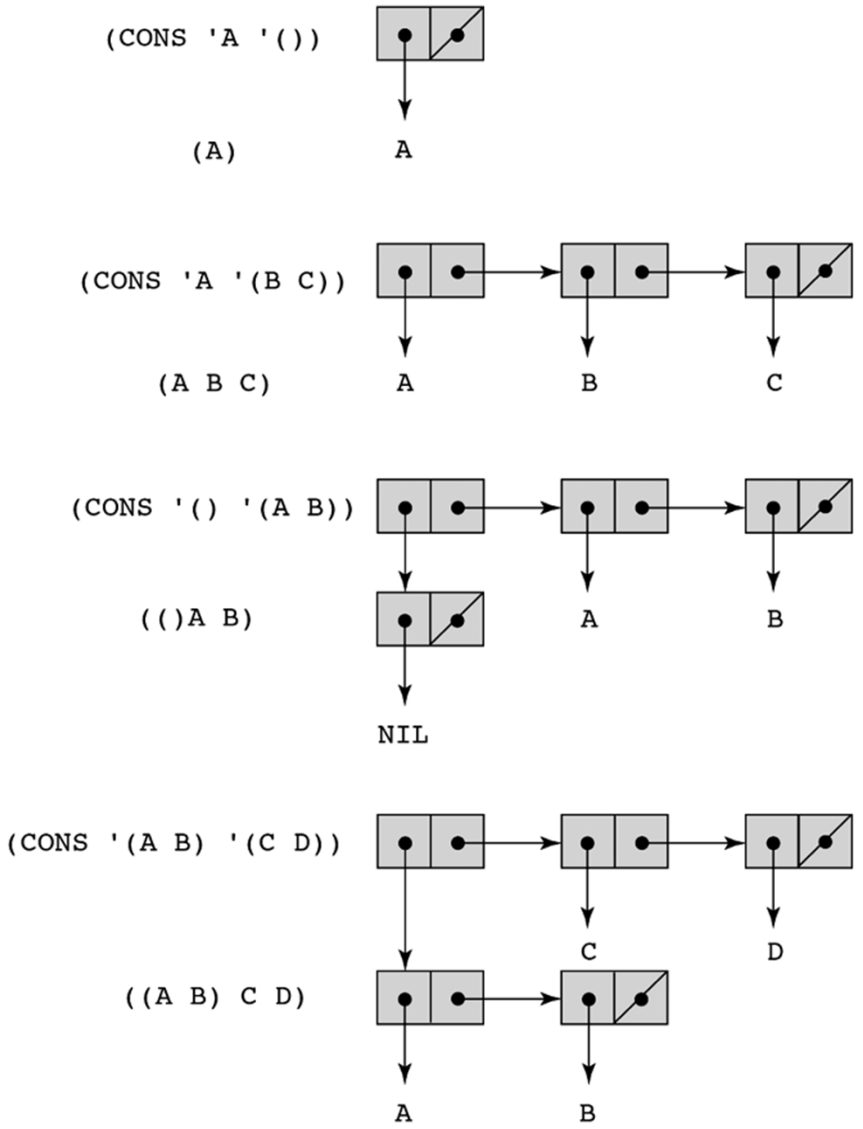
(A B C D)

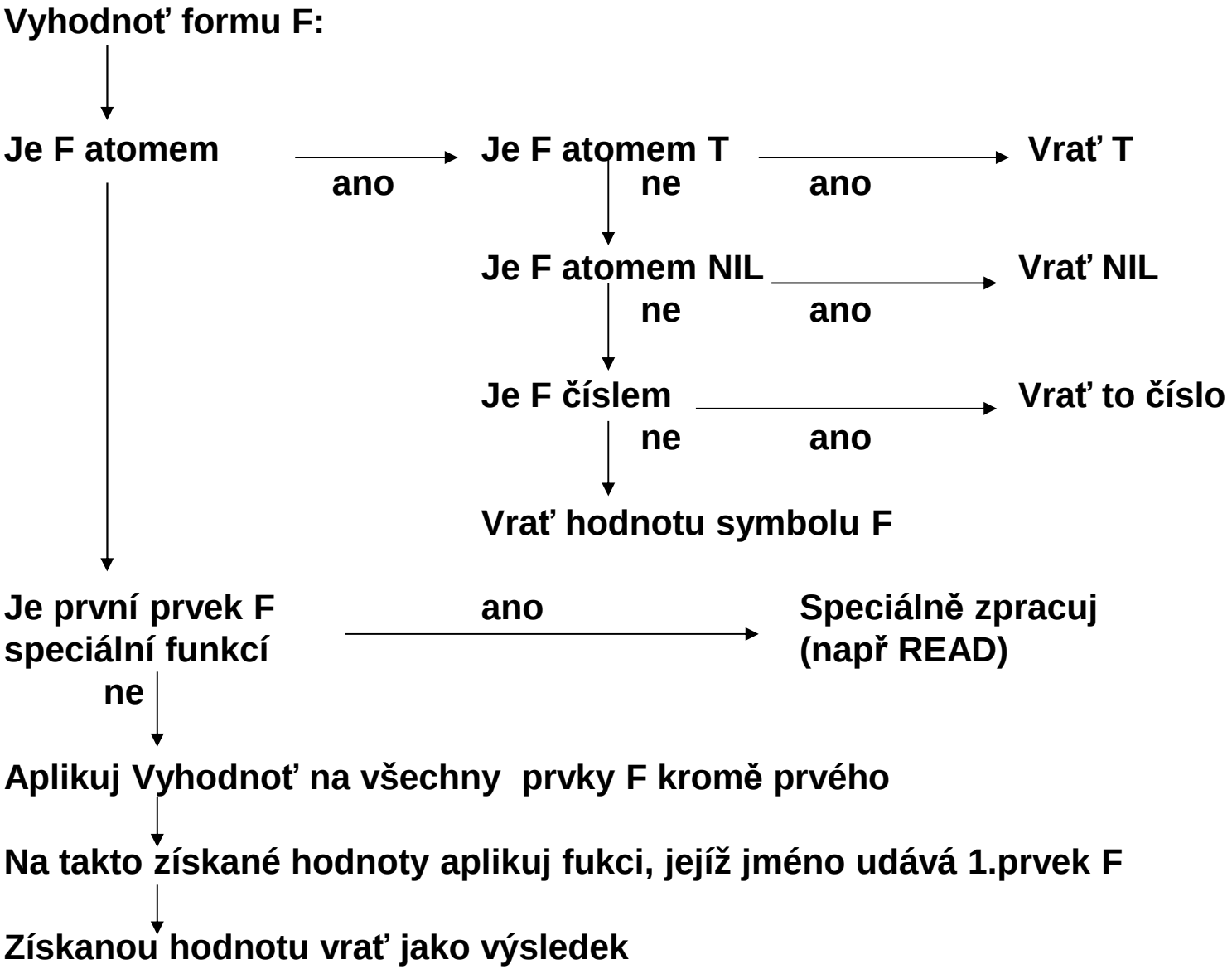


(A (B C) D (E (F G)))

Funkcionální programování- LISP

Výsledky fce CONS





Obr. Schéma lispovského vyhodnocování

Kromě CONS jsou další konstruktory APPEND a LIST

APPEND:: (seznam x seznam … x seznam) → seznam

Vytvoří seznam z argumentů – vynechá jejich vnější závorky

Př. (APPEND ‘(A B) ‘(B A)) dá (A B B A)

(APPEND ‘(A) ‘(B A)) dá (A B A)

Common Lisp připouští libovolně argumentů Append

(APPEND NIL ‘(A) NIL) dá (A)

(APPEND () ‘(A) ()) dá také (A)

(APPEND) dá NIL

(APPEND ‘(A)) dá (A)

(APPEND ‘((B A)) ‘(A) ‘(B A)) dá ((B A) A B A)

výsledky (A B B A) (A B A)

výsledky (A) (A) nil (A) ((B A) A B A)

Konstruktory APPEND a LIST

LIST:: (seznam x seznam x … x seznam) → seznam

Vytvoří seznam ze zadaných argumentů

Př.(LIST ‘A ‘(A B) ‘C) →

(A (A B) C)

(LIST ‘A) →

(A)

(LIST) →

nil

(LIST ‘(X (Y Z)) ‘(X Y)) →

((X (Y Z)) (X Y))

(APPEND ‘(X (Y Z)) ‘(X Y)) →

(X (Y Z) X Y)

(CONS ‘(X (Y Z)) ‘(X Y)) →

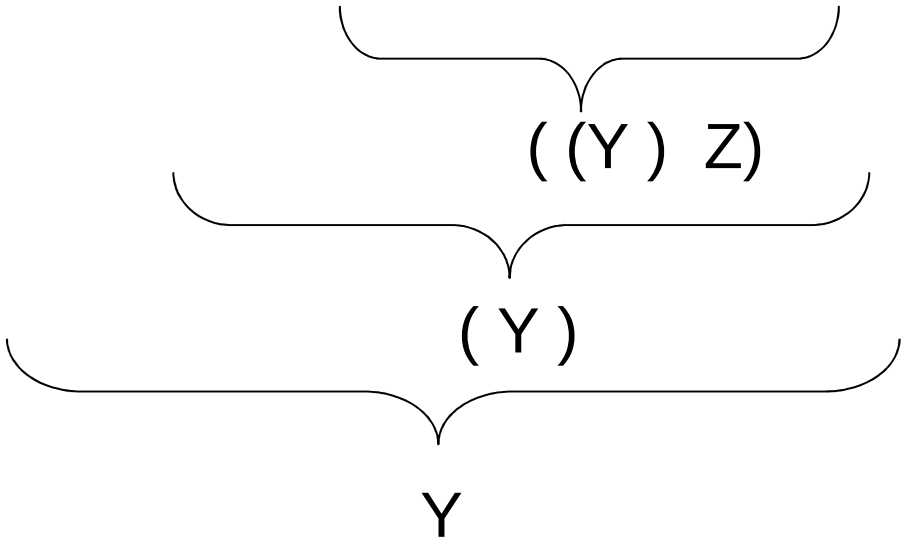
((X (Y Z)) X Y)

Selektory CAR a CDR lze vnořovat zkráceně

CxxR **CAAR, CADR, CDAR, CDDR**

CxxxR **CAAAR, …**

Př. (CAADR ‘(X (Y Z)) = (CAR (CAR (CDR ‘(X ((Y) Z)))))



(CADAR ‘ ((1 2 3) (4 5)))

2

LISP - Testy a větvení

ATOM je argument atomický? (atom 3) dá T

NULL „ „ prázdný seznam?

NUMBERP „ „ číslem?

SYMBOLP „ „ symbolem?

LISTP „ „ seznamem?

Př. (NULL NIL) → (NULL ( )) →

(LISTP ( )) → (LISTP NIL) →

(SYMBOLP NIL) → (SYMBOLP ( )) →

všechny mají hodnotu T, protože () i nil jsou prázdný seznam

Př.

(NUMBERP ‘(22)) (NUMBERP ’22) (NUMBERP 22)

NIL T T

Př.

(SYMBOLP ‘A) je T

Když ale do A přiřadíme (SETQ a 2) bude (SYMBOLP ‘A) dávat NIL

LISP - Testy a větvení

Je tam několik testů rovnosti, nemusíte si je pamatovat, berte to jako poznámku

= jsou hodnoty argumentů (může jich být více) stejná čísla?

EQ „ „ „ stejné atomy?

EQUAL „ „ „ stejné s-výrazy? Se stejnou hodnotou

> „ „ „ v sestupném pořadí?

>= „ „ „ v nestoupajícím pořadí?

< **<=** je obdobné

Př. (EQ 3.0 3) dá NIL **(= 3.0 3)** dá T

(EQ ‘(A) ‘(A)) dá NIL **(EQ ‘A ‘A)** dá T

(EQUAL ‘(A) ‘(A)) i **(EQUAL (+ 2 3 3) (CAR ‘(8))**

dají T protože jejich argumenty vypisují stejnou hodnotu A či 8

(< 2 3 5 (* 4 4) 20) → T neboť posloupnost je vzestupná

(= 1.0 1 1.00 (- 3 1 1.00)) → T je to divné, ale bere je jako stejná

LISP - Testy a větvení

AND, OR mají libovolně argumentů, NOT má jen jeden

Všechny hodnoty různé od NIL považují za pravdivé

Argumenty vyhodnocují zleva doprava. Hodnotou fce je hodnota naposledy vyhodnoceného argumentu.

Používá zkrácený výpočet argumentů. Tzn pokud při AND vyhodnotí argument jako NIL, další již nevyhodnocuje. Obdobně pro OR vyhodnotí-li argument jako T, další nevyhodnocuje

Př. (AND (NULL NIL) (ATOM 5) (+ 4 3)) dá 7

(OR NIL (= 2 (CAR '(2)) (+ 1 1) (- 3.0 1.0) 0) 9) dá 9

IF forma

(IF podm then-část else-část)

Větvení se potřebuje k vytváření uživatelových fcí

Př. ((lambda (x) (if (< x 0) (* x x (- x)) (* x x x))) -5) dá 125

LISP - Testy a větvení

COND je spec. fcí s proměnným počtem argumentů

```
(COND (podm1  forma11  forma12  …  forma1n)
          (podm2  forma21  forma22  …  forma2m)
                  . . .
          (podmk  formak1  formak2  …  formako) )
```

Postupně vyhodnocuje podmínky, dokud nenarazí na prvou, která je pravdivá. Pak vyhodnotí formy patřící k pravdivé podmínce. Hodnotou COND je hodnota poslední z vyhodnocených forem. Při nesplnění žádné z podm, není hodnota COND definována (u Common Lispu).

Pseudozápis pomocí IF:

```
COND  if podm1 then
                  { forma11  forma12  …  forma1n}
        else
        if podm2 then
                  { forma21  forma22  …  forma2m}
        else
                  . . .
        if podmk then
                  { formak1  formak2  …  formako}
        else NIL
```

LISP - Přiřazování

Přiřazení je operace, která pojmenuje hodnotu a uloží ji do paměti

- **Je ústupkem od čistě funkcionálního stylu**
- **Může zefektivnit a zpřehlednit i funkcionální program**
- **Mění vnitřní stav výpočtu (vedlejší efekt přiřazení)**

Zahrnuje funkce pro: -I/O,
-pojmenování uživ. fcí
-pojm. hodnot symbolů (SET, SETQ)

(SETQ jméno-symbolu argument)

vrátí hodnotu argumentu a naváže hodnotu argumentu na nevyhodnocené jméno symbolu

SET je obdobná, ale vyhodnotí i jméno symbolu

LISP - Přiřazování

Př.

(SETQ X 1) → 1

(SETQ X (+ 1 X) → 2

X → 2

>(SETQ LETADLO ‘BOING)

BOING

>LETADLO

BOING

>(SETQ BOING ‘JUMBO)

JUMBO

>(SETQ LETADLO BOING)

JUMBO

>LETADLO

JUMBO

>(SET LETADLO ‘AEROBUS)

AEROBUS

>LETADLO

JUMBO

>JUMBO

AEROBUS

j1

LISP – Přiřazování (jen pro informaci)

```
(LET (
        (var-name-1 expression-1)
        (var-name-2 expression-2)
        ...
        (var-name-n expression-n))
        body
   )
```

- Vyhodnotí všechny výrazy, provede vazbu hodnot na jména a pak vyhodnotí tělo (v prostředí LET).
- Proměnné jsou lokální v LET příkazu, mimo neexistují
- Hodnotou LET je hodnota poslední formy těla LET.

Př.
```
(LET ((a 3) (b 4))
        (SQRT (+ (* a a) (* b b)))
)
5
```

Snímek 27

j1 jezek_ka; 3.4.2008

LISP – Definice funkcí

Definujeme zápisem: (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 lispovského symbolu: (jen pro informaci)

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, nejčastěji jen jedna. Při vyvolání fce se všechny vyhodnotí. Funkční hodnotou je hodnota poslední z forem

Př

(defun fce(x) (+ x x) (* x x)) odpoví fce a vytvoří funkční vazbu pro FCE

Když ji pak vyvoláme např. (fce 5) odpoví nám 25

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 vzhledem k závorkám ⇒ lepší možnost load ze souboru

(LOAD “jmeno-souboru”)

(LOAD “D:\\PGS\\LISP\\soubor.lsp”)

LISP – Definice funkcí

Př. 1max.lsp

```
(defun max2(x y) (if (> x y) x y))
(defun ma(n) (if
;;;        (equal (list-length n) 2)  list-lenth je standardní fce
;;;        naprogramujeme si ji sami pojmenovanou delka
             (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))) ))

;;;        vyvolání např  (ma '(1 8 3 5))
```

LISP – Definice funkcí

Př. 2sude-poradi.lsp

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

Význam zápisu:

Pokud má x více než jeden prvek, dej do výsledku druhý prvek (tj car z cdr x) s výsledkem rekurzivně vyvolané fce sude s argumentem, kterým je zbytek ze zbytku x (tj cdr z cdr x, tj část x od tretího prvku dál). Pokud má seznam x jen jeden prvek, je výsledkem prázdný seznam

LISP – Definice funkcí

Př.3NSD-Fakt.lsp

```
(defun nsd (x y)
   (cond ((zerop (- x y)) y)       ;;; je-li rozdil x y nula, vysledek je y
         ((> y x) (nsd x (- y x))) ;;je-li y vetsi x vyvolej nsd x a y-x
         (t (nsd y (- x y)))       ;;jinak vyvolej nsd y a x-y
))

(defun fakt (x)
   (cond        ((= x  0) 1)       ;;faktorial nuly je jedna
                (T (* x  (fakt (- x 1)))) ;;jinak je x krat faktorial x-1
))
```

LISP – Definice funkcí

P5 4AppendMember.lsp

Redefinice append a member musíme explicitně povolit. Po load hlasi, že funkce je zamknutá. Pokud odpovíme :c ignorujeme zamknutí makra a funkce se předefinuje

```
(DEFUN APPEND (X  Y)   ;;;pro dva argumenty
   (IF (NULL  X)
                  Y         ;;je-li X prazdny je vysledkem Y
                  (CONS  (CAR  X) (APPEND (CDR  X)  Y))
) )    ;;jinak je vysledkem seznam zacinajici X a za nim APPEND…

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

Př volání (MEMBER ‘X ‘(A B X Z)) tato dá T standardní dá (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

Výběr posledního prvku, je také ve standardních

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

(LAST '(1 3 2 8 (4 5))) → ((4 5))

LISP – Další standardní funkce (pro informaci)

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, takze povolit předef    :c
    (cond ((null x) 0)
          ((atom x) 1)
          (t (+ 1 (count (cdr x))))))

(defun avrg ()          ;;;hlavni program je posloupnost forem
    (print "napis seznam cisel")
    (setq x (read))
    (setq avg (/ (sum x) (count x)))
    (princ "prumer je ")
    (print avg))    ;;je-li real a prvky jsou celociselne, vypise zlomkem
```

Př. pro zabavení 6hanoi.lsp * výstup příkazem format tvaru:

(FORMAT cíl řídící řetězec argumenty)

na obrazovku	t
netiskne ale vrátí	nil
Na soubor	proud

~%	odřádkování
~a	řetězcový argument
~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 (pro informaci)

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) **(pro informaci)**

```
(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))
))
```

Př 8DOpriklad.lsp **(pro informaci)**

Co se tiskne?

Promenna1 init1 Promenna2 init2

```
(DO (( x  1 (+  x  1))  (y  10  (*  y  0.5))) ;soucasna inicializace
```

Test konce step1 step1

```
    ((>  x  4)  y)
    (print  y)
    (print 'pocitam)
)
```

koncova forma

formy provadene pri iteracich

```
10
POCITAM
5.0
POCITAM
2.5
POCITAM
1.25
POCITAM
0.625
```

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

```
(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))            ; pocitame poradi od 0
                (T (nty (cdr S) (- x 1)))
    ))
```

LISP – Další standardní funkce (pro informaci)

(EVAL a) vyhodnotí výsledek vyhodnocení argumentu

Př.

```
>(EVAL (LIST ‘REST (LIST ‘QUOTE ‘(2  3  4))))
(3  4)
```

(QUOTE ‘(2 3 4)

(REST ‘(2 3 4))

(EVAL ‘(REST ‘(2 3 4)))

```
>(EVAL (READ))
(+  3  2)
```

to napiseme pro READ

```
5
>(EVAL (CONS ‘+  ‘(2  3  5)))
10
(SET ‘A  2)
(EVAL  (FIRST  ‘(A  B)))
2
```

* LISP – Další standardní funkce (pro informaci)

(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))) ;;podvyraz pv1=x+y  a  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 (pro informaci)

Komu nevoní funkcionální může využít sekvenční vyhodnocování forem pomocí formy PROG. Ta sekvenčně vyhodnocuje formy a může obsahovat i příkaz skoku **(GO návěští).** Návrat z PROG je (**RETURN forma),** kde **forma** udává hodnotu celé PROG

Tvar je:

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

Př. (DEFUN Iteracne-MEMBER (atom S)

```
        (PROG ( )
          opakuj
           (COND ((NULL S) (RETURN NIL))
                         ((EQUAL atom (CAR S)) (RETURN T))
           )
           (SETQ S (CDR S))
           (GO opakuj)
  ))
```

LISP – Další standardní funkce (pro informaci)

```
(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 (pro informaci)

```
(DEFUN co-vraci (Z)
    (LIST (FIRST Z) (posledni-prvek))
)
(DEFUN  poslední-prvek ( );;ta fce pracuje s nelokalnim Z, ale co to bude?
    (FIRST (LAST  Z))
)
>(SETQ  Z  '(1  2  3  4))
(1  2  3  4)
>(co-vraci '(A  B  C  D))
```

u dynamickeho

u statickeho

(A 4) u statickeho rozsahu platnosti, platnost jmen je dána lexikálním tvarem progranu-CLISP

(A D) u dynamickeho rozsahu platnosti, platnost jmen je dána vnořením určeným exekucí volání funkcí -GCLISP

Shrnutí zásad

- **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.**

Máte-li chuť, zkuste vyřešit

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 průhledněji?

```
(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 (pro informaci)

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) *Pamatujte si alespo+n ten pojem*

(pro informaci)

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 (pro informaci)

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 (pro informaci)

- 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.? (pro informaci)

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)) )    )
```

?LISP

-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)) )    )
```

?tak to nejde, chtěl by vyhodnotit car jako proměnnou

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

zabráníme vyhodnocení car

a to pak bude vysledek

Zde stačilo použít quote, pokud ale ve funkci pracujeme s volnými proměnnými bude muset LISP zjistit funkční vazbu (najít kod funkce), takže jméno funkce bude vyhodnocovat, ale jinak než normální s-výraz. Aby to udělal, musíme mu dát vědět funkcí FUNCTION

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 **(pro informaci)**

(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 (to si prohlédneme)

MAPCAR aplikuje fci na prvky seznamů, které jsou dalšími argumenty, až do vyčerpání kratšího ze seznamů. 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 **(pro informaci)**

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)

Př.95DERIV.LSP (pro pobavení)

```
(defun deriv (e x);; derivuje podle x vyraz e zapsany v LISP notaci
  (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)) ))
```

Např: (deriv ‘(* (sin (* x x)) (+ 5 x)) ‘x)

Výsledný výraz oproti prologovskému programu není zjednodušený