PROGRAMOVÉ STRUKTURY: FUNKCIONÁLNÍ PROGRAMOVÁNÍ

Čisté výrazy, referenční transparentnost, normální forma výrazu, lambda kalkul, LISP, s-výrazy, seznamy, funkcionály

Funkcionální programování

- Probereme základy jazyka LISP
- Plný popis jazyka najdete např.
  - http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/html/cltl/cltl2.html
- Imperativní jazyky jsou založeny na von Neumann architektuře
  - primárním kritériem je efektivita výpočtu
  - modelem je Turingův stroj
  - základní konstrukcí je příkaz
  - příkazy mění stavový prostor programu

Funkcionální programování (2)

- Funkcionální jazyky jsou založeny na matematických funkcích
  - program definuje funkci
  - výsledkem je funkční hodnota
  - modelem je Lambda kalkul
    - Church – 30 léta
  - základní konstrukcí je výraz
    - definuje algoritmus i vstup
  - výrazy jsou
    - čisté
      - nemění stavový prostor programu
    - s vedlejším efektem
      - mění stavový prostor

Vlastnosti čistých výrazů

- Hodnota výsledku nezávisí na pořadí vyhodnocování
  - tzv. *Church-Roserova* vlastnost
- Výraz lze vyhodnocovat paralelně
  - např. dělence i dělitele ve výrazu `(x*2+3)/(fce(y)*x)`
  - pokud `fce(y)` bude mít vedlejší efekt a změní hodnotu `x`
    - nebude to čistý výraz
- 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

Nalezení největšího čísla

1. Ze dvou čísel
   - označme symbolem `def` definici funkce

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

2. Ze čtyř čísel

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

3. Z N čísel

```
def maxN(N) jestliže délka(N) = 2
    pak max2(první-z(N), druhý-z(N))
    jinak max2(první-z(N), maxN(zbytek(N)))
```

- Prostředky funkcionálního programování jsou
  - kompozice složitějších funkcí z jednodušších
  - rekurze

Základní pojmy

Definice: **Matematická funkce** je zobrazení prvků jedné množiny, nazývané definiční obor funkce, do množiny druhé, 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
- Příklad: aritmetický výraz

  `E = (4+7+10)*(5-2)=(11+10)*(5-2)=21*(5-2)=21*3=63`
  - přepisovací pravidla jsou určena tabulkami pro `+`, `*`, ...
  - smysl výrazu je redukcemi zachován
  - **vlastnost referenční transparentnosti** je vlastností vyhodnocování funkcionálního programu

Základní pojmy (2)

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 funkce F na argumenty A, psáno FA
  - používá operaci abstrakce ve tvaru $\lambda(x)\ \ M[x]$
  - definuje funkci (zobrazení) $x \rightarrow M[x]$

Lambda výrazy

- Př. λ(x) x*x*x — takto to píší matematici

  v programu je to uzávorkováno

forma = výraz

definuje bezejmennou funkci x*x*x lambda výrazem

- Lambda výrazy popisují bezejmenné funkce
- Lambda výrazy jsou aplikovány na parametry, např.

(λ(x) x*x*x) 5 = 5*5*5=125

popis funkce — argumenty

aplikace (vyvolání funkce)

Lambda výrazy (2)

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

```
((lambda (x) (* x x)) 5)
((lambda (y) ((lambda (x) (+ (* x x) y)) 2)) 3)
⇒ 7
```

  - přesvědčíme se v Lispu

Lambda výrazy (3)

- V předchozím příkladu výraz

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

  - obsahuje vázanou proměnnou `x` na `2` a volnou proměnnou `y`
  - ta je pak vázána na 3 ve výrazu

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

- V LISPu lze zapisovat také

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

  - dá to obojí 7?

- Ta upovídanost s lambda má důvod v přesném určení pořadí, jaký skutečný argument odpovídá jakému formálnímu

Lambda výrazy (4)

- Pořadí vyhodnocování argumentů lze provádět
  - všechny se vyhodnotí před aplikací funkce
    - *eager evaluation* (dychtivé vyhodnocení)
  - argument se vyhodnotí těsně před jeho použitím v aplikaci funkce
    - *lazy evaluation* (líné vyhodnocení)
- Pochopit význam pojmu
  - volná proměnná
  - vázaná proměnná

Lisp (dříve LISP)

- Oficiálně
  - LISt Processing
    - zpracování seznamů
- Neoficiálně
  - Lost in Stupid Parentheses
  - Lost In a Sea of Parentheses
  - Lots of Isolated Silly Parentheses
  - Lots of Irritating Single Parentheses
  - Lisp Is Simply Perfect
  - Language Intended for Smart People

Vývoj Lispu

- LISP 1.0 (od 1958)
- MacLisp
- InterLisp
- Franz Lisp
- Scheme
- Common Lisp
- ANSI Common Lisp – ten budeme používat
- Arc
- Clojure

Použití a vlastnosti Lispu

- Použití
  - umělá inteligence
    - expertní systémy
    - symbolická manipulace
    - robotika
    - strojové vidění
    - práce s přirozeným jazykem
  - návrh VLSI (Very Large Scale Integration)
  - CAD systémy
- Základní vlastnosti
  - vše je seznamem (program i data)
  - jazyk není case-sensitive

Datové typy

- S-výraz = symbolický výraz
  - jednoduchý = atom
    - čísla
      - celá `125`
      - reálná `-5.25`
    - symboly
      - znaky `+  T  NIL`
      - identifikátory `atom`
      - řetězce `"dlouhý řetězec"`
  - strukturovaný
    - seznamy
      - jednoduché `(e1 e2 e3 … en)`
      - vnořované `(sqrt (+ 3 8.1))`

Postup vyhodnocování s-výrazu

- Často interpretační
- Cyklus prováděný funkcí `EVAL`
  1. výpis promptu
  2. uživatel zadá lispovský výraz (zápis funkce)
  3. provede se vyhodnocení argumentů
  4. aplikuje funkci na vyhodnocené argumenty
  5. vypíše výsledek (funkční hodnota)
- Pár příkladů s aritmetickými funkcemi

```
(+ 1111111111111111 2222222222222222)
```

  - má nekonečnou aritmetiku

```
(sqrt (* 4 pi))
```

  - zná matematické funkce a konstanty

```
2 * 2
```

  - způsobí chybu

Typový systém

- Funkce pracují s určitými typy hodnot
- Typování je prostředkem zvyšující spolehlivost programů
- Typová kontrola
  - statická
    - znáte např. z Javy
  - dynamická
    - nevyžaduje deklaraci typů argumentů a funkčních hodnot
    - Lisp, Prolog, Python

Zajímavosti Lispu

- Komplexní čísla
  - výraz `(* #c(2 2) #c(1.1 2.1))`
  - vyhodnotí jako `#c(-1.9999998 6.3999996)`
- Operátory jsou n-nární
  - výraz `(+ 1 2 3 4 5)`
  - dá výsledek `15`
- Nekonečná aritmetika pro `integer`

Elementární funkce

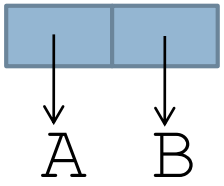
- Teoreticky lze s nimi vystačit pro zápis jakéhokoliv algoritmu
  - je to ale stejně neohrabané jako *Turingův stroj*
- `CAR` **alias** `FIRST`
  - selektor – výběr prvého prvku
- `CDR` **alias** `REST`
  - selektor – výběr zbytku seznamu
- `CONS`
  - konstruktor – vytvoří dvojici z argumentů
- `ATOM`
  - test, zda argument je atomický
- `EQUAL`
  - test rovnosti argumentů

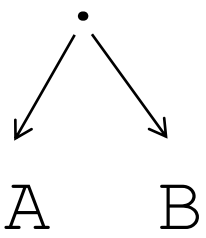
Elementární funkce (2)

- Ostatní funkce lze odvodit z elementárních
  - test prázdného seznamu `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 tomu zabránit
    - funkce `QUOTE` nebo lépe symbol `'` (apostrof)
      - Např. `(FIRST '(REST (1 2 3)))`
      - vs. `(FIRST (REST '(1 2 3)))`
      - první výraz vrátí `REST`, druhý hodnotu `2`

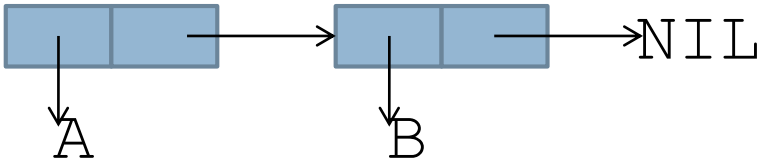
Konstrukce Lispovských seznamů

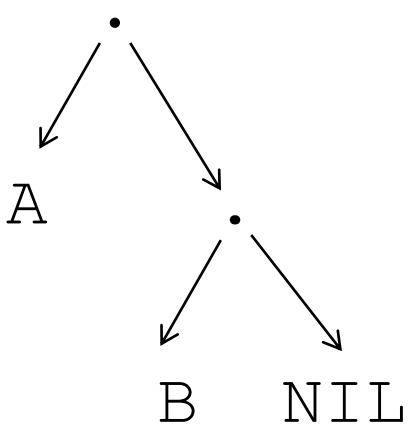
- CONS dvou atomů je tzv. *tečka dvojice*
  - `(CONS 'a 'b)`
  - `(A . B)`





- CONS atomu a seznamu je seznam
  - `(CONS 'a '(b))`
  - `(A B)`





Převod tečka dvojice na seznam

- Vyskytuje-li se " . " před " ("
  - lze vynechat " . " i " (" včetně odpovídající ") "
- Při výskytu " . NIL" lze " . NIL" vynechat
- Příklady

  `((A . NIL) . NIL)` je seznam `((A))`
  - je třeba psát mezera tečka mezera

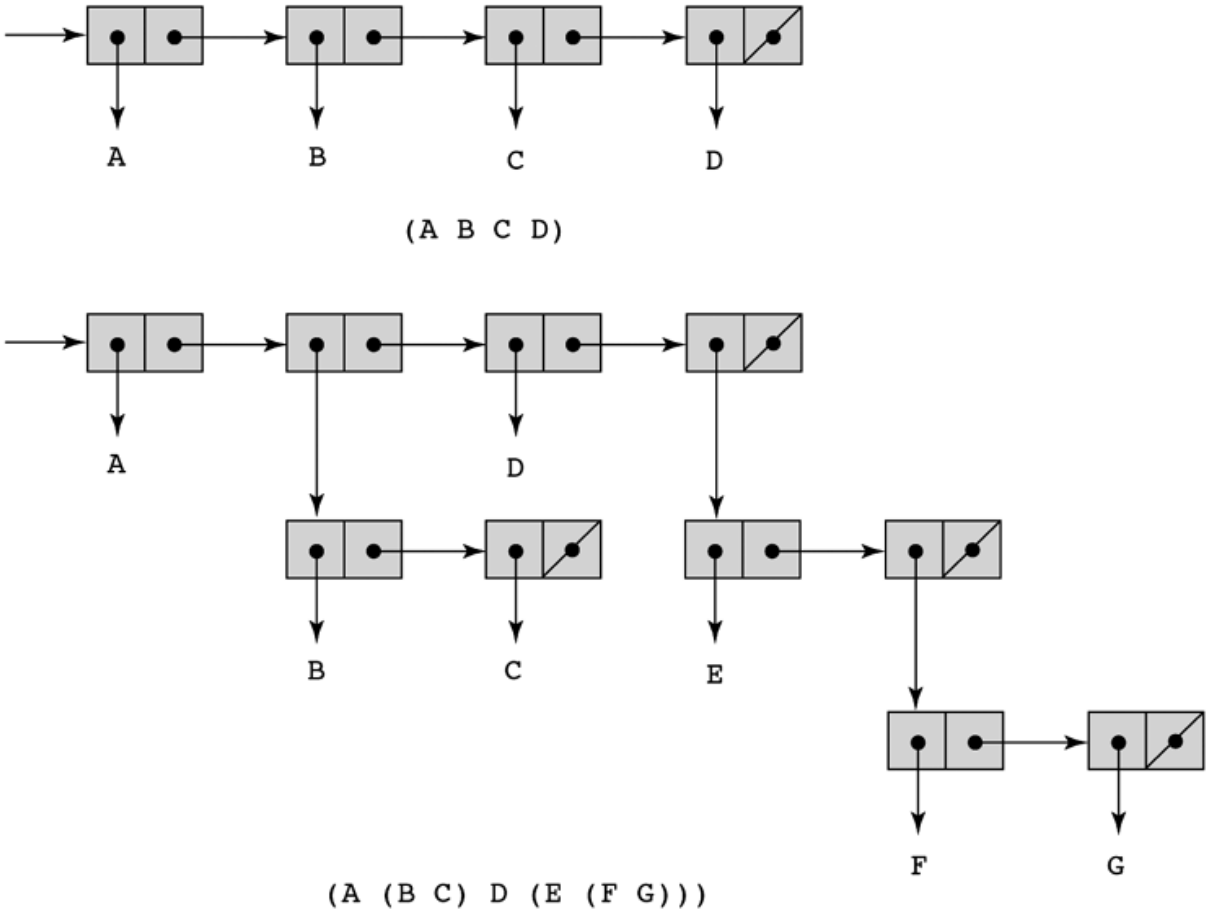
  `(a . NIL)` je seznam `(A)`

  `((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 je potřebný editor hlídající párování závorek

Interní reprezentace seznamů

Figure 15.1

Internal representation of two LISP lists



Výsledky funkce CONS

Figure 15.2

The result of several CONS operations

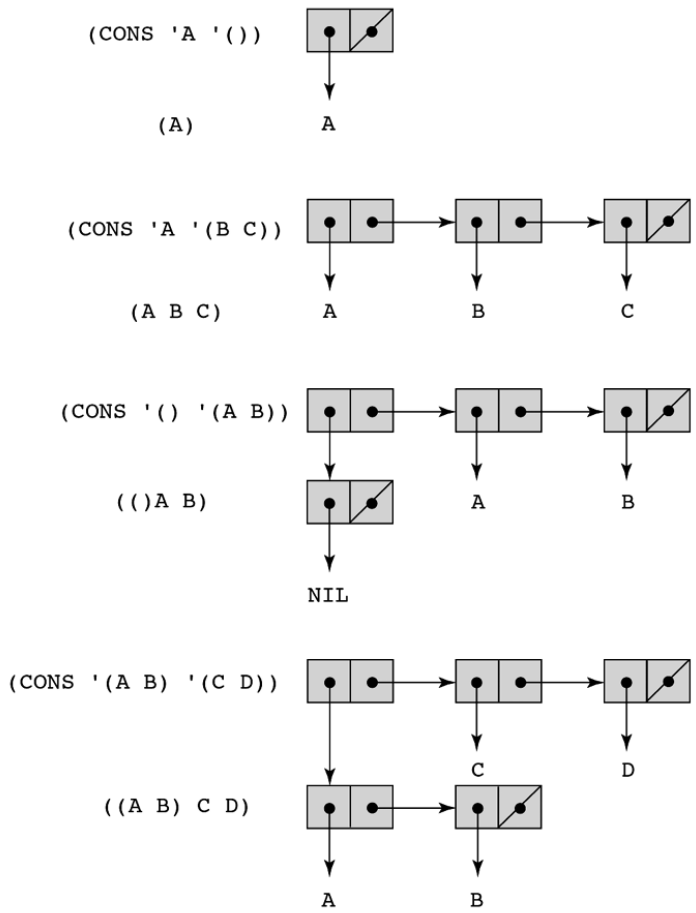
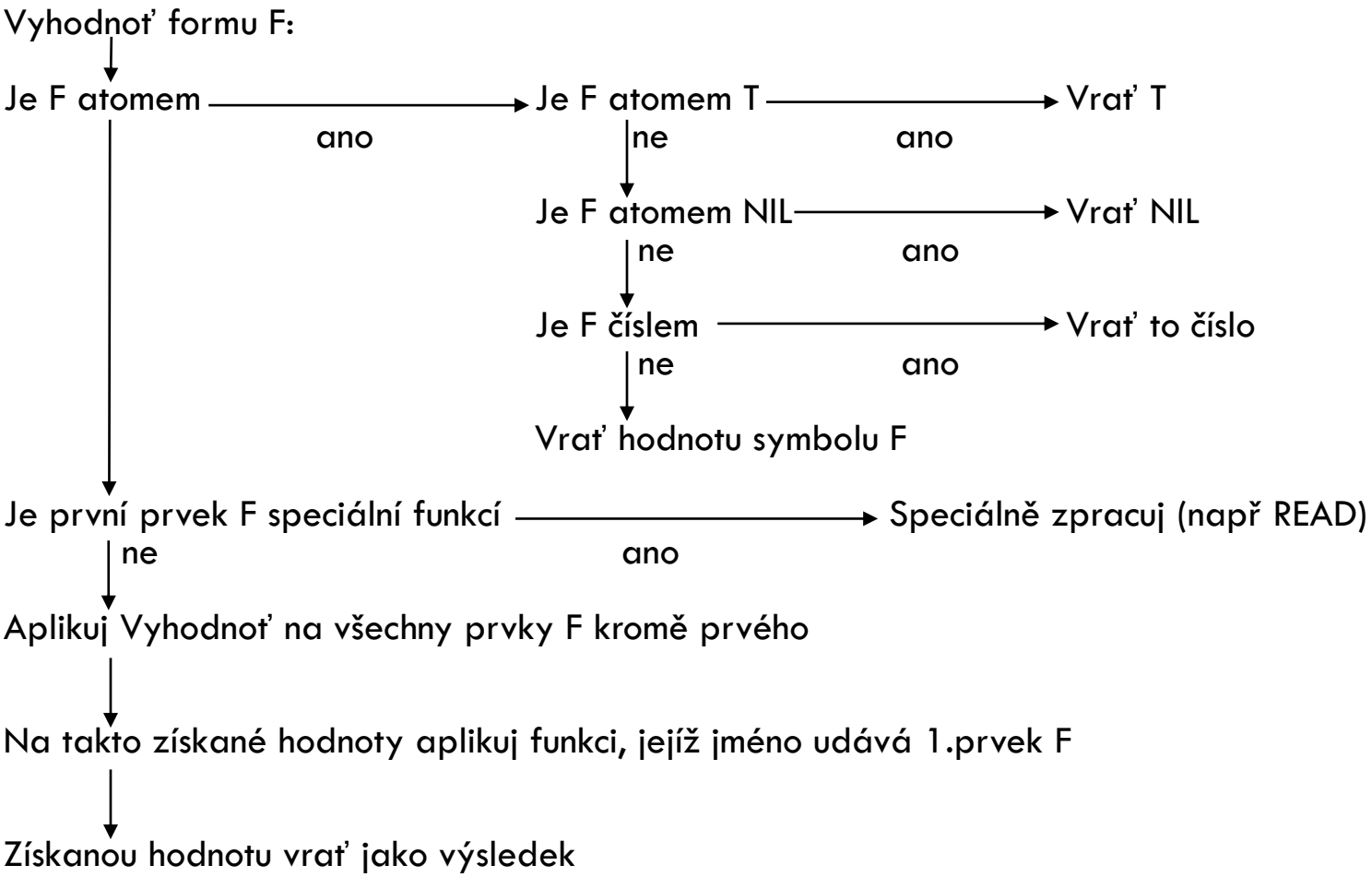


Schéma lispovského vyhodnocování



Konstruktor APPEND

- Vytvoří seznam z argumentů
  - připouští libovolné množství argumentů
  - vynechá jejich vnější závorky
- Příklady

```
(APPEND '(A B) '(B A))  dá   (A B B A)
(APPEND NIL '(A) NIL)   dá   (A)
(APPEND () '(A) ())     dá   (A)
(APPEND)                dá   NIL
(APPEND '(A))           dá   (A)
(APPEND '((B A)) '(A) '(B A))
  dá      ((B A) A B A)
```

Konstruktor LIST

- Vytvoří seznam ze zadaných argumentů
- Příklady

```
(LIST 'A '(A B) 'C)    dá    (A (A B) C)
(LIST 'A)              dá    (A)
(LIST)                 dá    NIL
```

- Porovnejte

```
(LIST '(X (Y Z)) '(X Y))      dá    ((X (Y Z)) (X Y))
(APPEND '(X (Y Z)) '(X Y))    dá    (X (Y Z) X Y)
(CONS '(X (Y Z)) '(X Y))      dá    ((X (Y Z)) X Y)
```

Vnořování selektorů

- Selektory `CAR` a `CDR` lze vnořovat zkráceně
  - `CxxR: CAAR, CADR, CDAR, CDDR`
  - `CxxR: CAAAR, CAADR, …`
- Příklady

```
(CAADR '(X (Y Z))) = (CAR (CAR (CDR '(X (Y Z)))))
                                    ((Y) Z)
                                      (Y)
                                       Y
```

```
(CADAR '((1 2 3) (4 5)))
2
```

Testy typů argumentů

- Je argument
  - atomický `ATOM`
    - `(ATOM 3)` dá `T`
  - prázdný seznam `NULL`
  - číslem `NUMBERP`
  - symbolem `SYMBOLP`
  - seznamem `LISTP`
- Příklady
  - `(NULL NIL)` a `(NULL ())`
  - `(LISTP NIL)` a `(LISTP ())`
  - `(SYMBOLP NIL)` a `(SYMBOLP ())`
    - všechny dají `T`
  - `(NUMBERP '(22))` dá `NIL`
  - `(NUMBERP '22)` a `(NUMBERP 22)` dají `T`
  - `(SYMBOLP 'A)`
    - dá `T`, pokud nemá `A` přiřazenu hodnotu
    - dá `NIL`, pokud `A` je pomocí `SETQ` přiřazena hodnota

Testy rovnosti

- Jsou hodnoty argumentů (může jich být více) stejná
  - čísla `=` `(= 1 1 1)`
  - atomy `EQ` `(EQ NIL ())`
  - s-výrazy `EQUAL` `(EQUAL (+ 1 1) (-3 1))`
- Jsou hodnoty argumentů (může jich být více)
  - v sestupném pořadí `>`
  - ve stoupajícím pořadí `<`
  - v nestoupajícím pořadí `>=`
  - v neklesajícím pořadí `<=`

Testy rovnosti – příklady


```
(EQ 3.0 3)                                  NIL
(= 3.0 3)                                   T
(EQ '(A) '(A))                              NIL
(EQ 'A 'A)                                  T
(EQUAL '(A) '(A))                           T
(EQUAL (+ 2 3 3) (CAR '(8)))                T
(< 2 3 4 (* 4 4) 20)                        T
(= 1.0 1 1.00 (- 3 1 1.00))                 T
```

Testy logických podmínek

- `AND` a `OR` mají libovolný počet argumentů
- `NOT` má jen jeden argument
- Všechny hodnoty různé od `NIL` považují za pravdivé
- Argumenty vyhodnocují zleva doprava
- Hodnota funkce je hodnota naposledy vyhodnoceného argumentu
- Používá zkrácený výpočet argumentů
  - pokud při `AND` vyhodnotí argument jako `NIL`
  - obdobně pro `OR`, vyhodnotí-li argument jako `T`
    - další argumenty již nevyhodnocuje
- Příklady

```
(AND (NULL NIL) (ATOM 5) (+ 4 3)) vrátí 7
(OR NIL (= 2 (CAR '(2)) (+ 1 1) (- 3.0 1.0) 0) 9)
vrátí 9
```

Větvení programu – IF

- `IF` forma

```
(IF podmínka then-část else-část)

(IF podmínka
  then-část  ;vždy jen jeden s-výraz
  else-část  ;nepovinná část
)
```

- Větvení se potřebuje k vytváření uživatelových funkcí

```
((lambda (x) (if (< x 0) (* x x (- x)) (* x x x))) -5)
```

  - vrátí výsledek 125

Větvení programu – COND

- Forma `COND` je speciální funkcí s proměnným počtem argumentů

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

- Popis
  1. Postupně vyhodnocuje podmínky, dokud nenarazí na prvou, která je pravdivá
  2. Pak vyhodnotí formy patřící pravdivé podmínce
  3. Hodnotou `COND` je hodnota z poslední vyhodnocované formy
  4. Při nesplnění žádné z podmínek není hodnota `COND` definována

Větvení programu – COND (2)

- Pseudozápis pomocí IF

```
COND if podm1 then
       {forma11 forma12 … forma1n}
     else
       if podm2 then
         {forma21 forma22 … forma1m}
       else
       …
         if podmk then
           {formak1 formak2 … formako}
         else
           NIL
```

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
  - vstup a výstup
  - pojmenování uživatelských funkcí
  - pojmenování hodnot symbolů (`SET`, `SETQ`)

```
(SETQ jmeno-symbolu argument)
```

  - vrátí hodnotu argumentu a naváže hodnotu argumentu na nevyhodnocené jméno symbolu
  - `SET` je obdobná, ale vyhodnotí i jméno symbolu

Příklady přiřazení


```
> (SETQ X 1)
1
> (SETQ X (+ 1 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
```

Definice funkcí

- Definujeme zápisem

  `(DEFUN jméno-funkce (argumenty) tělo-funkce)`

  - přiřadí jménu-funkce lambda výraz definovaný tělem-funkce
    - tj. `(LAMBDA (argumenty) tělo-funkce)`
  - vytvoří funkční vazbu na jméno-funkce
- Argumenty jsou ve funkci lokální
- `DEFUN` nevyhodnocuje svoje argumenty
- Hodnotou formy `DEFUN` je nevyhodnocené jméno-funkce

Definice funkcí (2)

- Tělo funkce je posloupností forem, nejčastěji jen jedna
  - při volání funkce se všechny vyhodnotí
  - funkční hodnotou je hodnota poslední z forem
- Příklad
  - `(DEFUN fce(x) (+ x x) (* x x))`
  - odpoví `FCE` a vytvoří funkční vazbu pro FCE
  - jejím voláním, např. `(fce 5)` odpoví `25`

Definice funkcí (3)


```
> (DEFUN max2(x y) (IF (> x y) x y))
MAX2
> (max2 10 20)
20
> (DEFUN max4(u v x y) (max2 (max2 u v) (max2 x y)))
MAX4
> (max4 5 9 12 1)
12
```

- Interaktivní psaní lispovských programů způsobuje postupnou demenci vzhledem k závorkám
  - lepší možnost load ze souboru

```
(LOAD "soubor.lsp")
```

Program maximum.lsp


```
(DEFUN max2(x y)
  (IF (> x y) x y)
)

(DEFUN max4(u v x y)
  (max2 (max2 u v) (max2 x y))
)

(DEFUN maxn(n)
  (IF (EQUAL (delka n) 2)
    (max2 (CAR n) (CAR (CDR n)))
    (max2 (CAR n) (maxn (CDR n)))
  )
)

(DEFUN delka(n)
  (IF (EQUAL n NIL)
    0
    (+ 1 (delka (CDR n)))
  )
)
; zkusíme dotaz (maxn '(1 8 3 5))
```

Program sude-poradi.lsp


```
(DEFUN sude (x)
   (COND
      ((NOT (NULL (CDR x)))
       (CONS (CAR (CDR x))(sude (CDR (CDR x)))))
      (T NIL)
   )
)
```

- Pokud má x více než jeden prvek, dej do výsledku druhý prvek (tj. `CAR` z `CDR` x) s výsledkem rekurzivně volané funkce sude s argumentem, kterým je zbytek ze zbytku x (tj. `CDR` z `CDR` x, tj. část x od třetího prvku dál).
- Pokud má seznam x jen jeden prvek, je výsledkem prázdný seznam

Program nsd.lsp


```
(DEFUN nsd (x y)
   (COND
      ((ZEROP (- x y)) y)
      ((> y x)(nsd x (- y x)))
      (T       (nsd y (- x y)))
   )
)
```

Program fakt.lsp


```
(DEFUN fakt (x)
   (COND
      ((= x 0) 1)
      (T       (* x (fakt (- x 1))))
   )
)
```

Program append.lsp


```
(DEFUN myappend (x  y)
  (IF (NULL  x)
    y
     (CONS (CAR x)
           (myappend (CDR x)  y)
      )
  )
)
```

Program member.lsp


```
(DEFUN mymember (x  s)
   (COND
      ((NULL s)              NIL)
      ((EQUAL x (CAR s)) T)
      (T      (mymember x (CDR s)))
   )
)
```

Další standardní funkce

- Aritmetické

```
> (- 10 1 2 3 4)
0
> (/ 100 5 4 3)
5/3
```

- Operace na seznamech

```
> (LIST-LENGTH '(1 2 3 4 5))
5
```

Vstupy a výstupy

`(OPEN soubor :DIRECTION smer)`

- otevře soubor a spojí ho s novým proudem, který vrátí jako hodnotu
- hodnotou je jméno proudu (= souboru)
- příklad

```
(SETQ s (OPEN "d:\\data.txt" :DIRECTION :output))
```

  - standardně je vstup z klávesnice, výstup obrazovka

`(CLOSE proud)`

- zapíše hodnoty na disk a uzavře daný proud
  - přepne na standardní
- příklad

```
(CLOSE s)
```

`(READ proud)`

`(PRINT a proud)`

Program prumer.lsp


```
(DEFUN sum(x)
  (COND
        ((NULL x) 0)
        ((ATOM x) x)
        (T        (+ (CAR x) (sum (CDR x))))
  )
)

(DEFUN mycount(x)
  (COND
        ((NULL x) 0)
        ((ATOM x) 1)
        (T        (+ 1 (mycount (CDR x))))
  )
)

(DEFUN avrg() ;;;main program
  (PRINT "napis seznam cisel")
  (SETQ x (READ))
  (SETQ avg (/ (sum x) (mycount x)))
  (PRINC "prumer je ")
  (PRINT avg)
)
```

Funkce FORMAT

- Formátování řetězce s argumenty
  `(FORMAT cíl řídící-řetězec argumenty)`
- Hodnoty cíle
  - `T` – tiskne na obrazovku
  - `NIL` – netiskne, ale vrátí
  - proud – zapíše do souboru
- Symboly v řídícím řetězci
  - `~%` – odřádkuje
  - `~a` – řetězcový argument
  - `~s` – symbolický výraz
  - `~d` – desítkové číslo

Program hanoi.lsp


```
(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)
)
```

Funkce řízení výpočtu

`(WHEN test formy)`

- je-li test T, je hodnotou hodnota poslední formy

`(DOLIST (proměnná seznam) forma)`

- váže `proměnná` na prvky až do vyčerpání seznamu a vyhodnocuje formu
- příklad

```
> (DOLIST (x '(a b c)) (PRINT x))
A
B
C
NIL
```

Funkce řízení výpočtu (2)

(LOOP formy)

- opakovaně vyhodnocuje formy až se provede forma RETURN
- příklad

```
>(SETQ a 4)
4
>(LOOP (SETQ a (+ a 1)) (WHEN (> a 7) (RETURN a)))
8
```

```
(DO ((var1 init1 step1) … (varn initn stepn))
  (testkonce forma1 forma2 ...formam)
  formy-prováděné-při-každé-iteraci
)
```

Program fibonacci.lsp


```
(DEFUN fibon(N)
  (COND
       ((EQUAL N 0) 0)
       ((EQUAL N 1) 1)
       ((EQUAL N 2) 1)
       (T           (foo (- N 2)))
))

(DEFUN foo(N)
  (SETQ F1 1)
  (SETQ F2 0)
  (LOOP
    (SETQ F (+ F1 F2))
    (SETQ F2 F1)
    (SETQ F1 F)
    (SETQ N (- N 1))
    (WHEN (EQUAL N 0) (RETURN F))
  )
)
```

Program cyklus-do.lsp


```
(DEFUN foo()
  (DO (( x  1 (+  x  1)) (y 10 (*  y  0.5)))
    ((>  x  4)  y)
      (PRINT  y)
      (PRINT 'pocitam)
  )
)

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

Program n-ty.lsp


```
(SETQ v "vysledek je ")

(DEFUN nta(S x)
  (DO ((i 1 (+ i 1)))
    ((= i x) (PRINC v) (CAR S))
    (SETQ S (CDR S))
  )
)

(DEFUN delej()
  (nta (READ) (READ))
)

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

Funkce EVAL

(EVAL A)

- vyhodnotí výsledek vyhodnocení argumentu
- příklady

```
> (EVAL (LIST 'REST (LIST 'QUOTE '(2 3 4))))
                          '(2 3 4)
                 (REST (2 3 4))
(3 4)
> (EVAL (READ))
(+ 2 3)                ; toto zadáme z klávesnice
5
> (EVAL (CONS '+ '(2 3 5)))
10
> (SET 'A 2)
> (EVAL (FIRST '(A B)))
2
```

Rozsah platnosti proměnných


```
(DEFUN co-vraci(z)
   (LIST (FIRST z) (posledni-prvek))
)
(DEFUN  posledni-prvek( )   ; nelokální z
   (FIRST (LAST z))         ; jakou má hodnotu
)
> (SETQ z '(1  2  3  4))    ; statický rozsah
> (CO-VRACI '(a  b  c  d)) ; dynamický rozsah
```

- `(A 4)` u statického rozsahu platnosti, platnost jmen je dána lexikálním tvarem programu – CLISP
- `(A D)` u dynamického 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` a `LISTP` testují typy dat
- =, `EQ` a `EQUAL` testují rovnost
- >, <, <=, >= testují pořadí
- `SETQ` a `SET` přiřazují symbolům globální hodnoty
- `DEFUN` definuje funkci, parametry v ní jsou lokální

Shrnutí zásad (2)

- `COND` umožňuje výběr alternativy
- `AND`, `OR` a `NOT` jsou logické funkce
- Proud je zdrojem nebo konzumentem dat
  - `OPEN` je otevře, `CLOSE` je zruší
- `PRINT`, `PRIN1`, `PRINC` a `WRITE` zajišťují výstup
- `READ` zabezpečuje vstup
- `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

Zkuste vyřešit, co to počítá


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

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

Schéma rekurzivního výpočtu

- S jednoduchým testem

```
(DEFUN fce (parametry)
  (COND (test-konce koncová-hodnota)   ; primitivní případ
        (test       rekurzivní-volání) ; redukce úlohy
  )
)
```

- S násobným testem

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

Program rekurze.lsp

- Odstraní výskyty prvků `e` v nejvyšší úrovni seznamu `s`

```
(DEFUN mydelete(e s)
  (COND
    ((NULL s)           NIL)
    ((EQUAL e (CAR s)) (mydelete e (CDR s)))
    (T   (CONS (CAR s) (mydelete e (CDR s))))
  )
)
```

Program rekurze.lsp (2)

- Zjistí maximální hloubku vnoření seznamu
  - funkce `MAX` je standardní funkcí

```
(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))
                      )
     )
  )
)
```

Program rekurze.lsp (3)

- Najde prvek s největší hodnotou ve vnořovaném seznamu

```
(DEFUN max-prvek(s)
  (COND
    ((ATOM s)          s)
    ((NULL (CDR s)) (max-prvek (CAR s)))
    (T   (MAX (max-prvek (CAR s))
              (max-prvek (CDR s))
         )
     )
  )
)
```

Funkcionály

- Funkce, jejichž argumentem je funkce nebo vrací funkci jako svoji hodnotu
- Vytváří programová schémata, použitelná pro různé aplikace
- *Higher Order Functions*
- Příklad
  - pro každý prvek s seznam S proveď funkci f(s)
    - to je schéma
  - programové schéma pro zobrazení

    `f: (s1, s2, …, sn)` → `(f(s1), f(s2), …, f(sn))`

Programové schéma pro zobrazení


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

Programové schéma filtru


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

Další programová schémata

- 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)))
   )
)
```

Další programová schémata (2)

- Programové schéma pro zjištění, zda všechny prvky splňují predikát

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

Funkcionály (2)

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

Ukázkový příklad

- 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)
                     )
                )
    )
  )
)
```

Ukázkový příklad (2)

- 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))
                )

    )
  )
)
```

Ukázkový příklad (3)

- Zavolání funkce

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

  - takto to nelze, chtěl by vyhodnotit `CAR` jako proměnnou

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

  - takto zabráníme vyhodnocení `CAR` a to pak bude výsledek

```
(a c)
```