KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI
www.inf.upol.cz

Michal Krupka
krupka.inf.upol.cz
michal.krupka@upol.cz
17. listopadu 12, 771 46 Olomouc

Paradigmata programování 1◇ poznámky k přednášce

6. Seznamy 2

verze z 12. listopadu 2019

1 Páry a seznamy: opakování a něco navíc

Nejprve zopakujeme základní informace o párech a seznamech.

Tečkový pár (stručně *pár*, také *cons*) je datová struktura, která se skládá ze dvou složek, nazývaných *car* a *cdr*. Tečkové páry se zapisují do kulatých závorek, složky jsou odděleny tečkou.

Funkce `cons` (konstruktor tečkových párů):

```
CL-USER 1 > (cons 1 2)
(1 . 2)
```

Funkce `car` a `cdr` (selektory tečkových párů):

```
CL-USER 2 > (setf c (cons 1 2))
(1 . 2)

CL-USER 3 > (car c)
1

CL-USER 4 > (cdr c)
2
```

Typový predikát `consp` (test na tečkové páry):

```
CL-USER 5 > (consp 1)
NIL

CL-USER 6 > (consp t)
NIL

CL-USER 7 > (consp c)
T
```

Symbol `NIL` se používá jako **prázdný seznam**. V této souvislosti ho můžeme psát i jako `()`. **Predikát `null`** (test na prázdný seznam):

```
CL-USER 9 > (null 1)
NIL

CL-USER 10 > (null (cons 1 2))
NIL

CL-USER 11 > (null nil)
T
```

Může být napsán takto:

```
(defun null (e)
  (eql e nil))
```

Čistý seznam délky n (stručně *seznam*, anglicky *proper list*) je pro $n = 0$ prázdný seznam (tj. symbol `nil`) a pro $n > 0$ tečkový pár, jehož *cdr* je seznam délky $n - 1$.

Zápis seznamu:

```
CL-USER 12 > (cons 1 (cons 2 (cons 3 (cons 4 nil))))
(1 2 3 4)
```

`1, 2, 3, 4`: *prvky seznamu.*

Seznam je délky 4.

Nová funkce na vytvoření seznamu:

```
CL-USER 13 > (list 1 2 3 4)
(1 2 3 4)
```

2 Základy práce se seznamy

Některé funkce, které tady uvádím, jsou v Lispu k dispozici. Abych se jednoduše vyhnul konfliktu názvů, dávám jim předponu `my-`. Tady jsou uvedeny na ukázku, jak by mohly být napsány.

Typový predikát.
Predikát `proper-list-p` zjišťuje, zda je jeho argument čistý seznam. Tady je několik možných variant, jak by mohl být napsán (první je vyloženě začátečnická, poslední je nejlepší).

```
(defun proper-list-p (x)
  (if (null x)
      t
    (if (consp x)
        (if (proper-list-p (cdr x))
            t
          nil)
      nil)))
```

```
(defun proper-list-p (x)
  (if (null x)
      t
    (if (consp x)
        (proper-list-p (cdr x))
      nil)))
```

```
(defun proper-list-p (x)
  (if (null x)
      t
    (and (consp x)
         (proper-list-p (cdr x)))))
```

```
(defun proper-list-p (x)
  (or (null x)
      (and (consp x)
           (proper-list-p (cdr x)))))
```

Poslední dvě verze fungují díky **zkrácenému vyhodnocování** logických spojek.

n-tý prvek.

Funkce `my-nth` vrací prvek seznamu o daném indexu (indexuje se od nuly).

```
(defun my-nth (n list)
  (if (= n 0)
      (car list)
    (my-nth (- n 1) (cdr list))))
```

n-tý zbytek.

Funkce `my-nthcdr` vrací pár o daném indexu v seznamu (indexuje se od nuly).

```
(defun my-nthcdr (n list)
  (if (= n 0)
      list
    (my-nthcdr (- n 1) (cdr list))))
```

n-tý prvek (znovu).

Vidíme, že funkci `my-nth` můžeme zjednodušit použitím funkce `my-nthcdr`:

```
(defun my-nth (n list)
  (car (my-nthcdr n list)))
```

Odstranění zbytku seznamu.

Funkce `my-ldiff` vrací k danému seznamu a jeho zbytku seznam bez zbytku:

```
(defun my-ldiff (list tail)
  (if (or (null list) (eql list tail))
      ()
    (cons (car list)
          (my-ldiff (cdr list) tail))))
```

Spojení dvou seznamů.

Funkce `append-2` spojuje dva seznamy. Později ji zobecníme, aby pracovala s libovolným počtem seznamů.

```
(defun append-2 (list1 list2)
  (if (null list1)
      list2
    (cons (car list1)
          (append-2 (cdr list1) list2))))
```

Test:

```
CL-USER 1 > (append-2 (list 1 2 3) (list 4 5 6))
(1 2 3 4 5 6)
```

Iterativní verze dělá něco jinak.

Pokus vytvořit iterativní verzi vede k jinému výsledku:

```
(defun my-revappend (list1 list2)
  (if (null list1)
      list2
    (my-revappend (cdr list1)
                  (cons (car list1) list2))))
```

Test:

```
CL-USER 2 > (my-revappend (list 1 2 3) (list 4 5 6))
(3 2 1 4 5 6)
```

Otočení seznamu.
Pomocí funkce `my-revappend` můžeme vytvořit užitečnou funkci `my-reverse`:

```
(defun my-reverse (list)
  (my-revappend list ()))
```

Test:

```
CL-USER 3 > (my-reverse (list 1 2 3 4 5 6))
(6 5 4 3 2 1)
```

3 Reprezentace matematických objektů seznamy

Seznamy lze použít k reprezentaci *vektorů*, které můžeme zjednodušeně chápat jako k-tice čísel (matematická definice je jiná). Ty můžeme násobit čísly a sčítat mezi sebou (to je ale nutné, aby sčítanci měli stejnou délku).

Násobek vektoru.
Funkce `scale-list` vynásobí všechny prvky daného seznamu (musí to tedy být čísla) daným číslem.

```
(defun scale-list (list factor)
  (if (null list)
      ()
    (cons (* factor (car list))
          (scale-list (cdr list) factor))))
```

Součet dvou vektorů.
Funkce `sum-lists-2` sečte dva seznamy jako vektory (tedy po složkách). Později ji zobecníme na libovolný počet argumentů.

```
(defun sum-lists-2 (list1 list2)
  (if (null list1)
      ()
      (cons (+ (car list1) (car list2))
            (sum-lists-2 (cdr list1) (cdr list2)))))
```

Seznamy lze také chápat jako množiny. V takovém případě nám nezáleží na pořadí prvků. Uvedeme některé základní množinové relace a operace.

Test na prvek.

Predikát `elementp` testuje, zda je daná hodnota prvkem seznamu. K porovnávání používá predikát `eql`.

```
(defun elementp (x list)
  (and (not (null list))
       (or (eql x (car list))
           (elementp x (cdr list)))))
```

Přidání prvku k množině.

Funkce `my-adjoin` přidá prvek k množině, ale jen pokud tam ještě není.

```
(defun my-adjoin (x list)
  (if (elementp x list)
      list
    (cons x list)))
```

Podmnožinovost.

Predikát `my-subsetp` testuje, zda je první argument podmnožinou druhého. Argumenty jsou seznamy, ovšem chápány jako množiny. K testování příslušnosti jednotlivých prvků do množiny používá už napsaný predikát `elementp`.

```
(defun my-subsetp (list1 list2)
  (if (null list1)
      t
      (and (elementp (car list1) list2)
           (my-subsetp (cdr list1) list2))))
```

Lepší verze.

```
(defun my-subsetp (list1 list2)
  (or (null list1)
      (and (elementp (car list1) list2)
           (my-subsetp (cdr list1) list2))))
```

Průnik.
Funkce `my-intersection` vrací k daným dvěma seznamům (chápaným jako množiny) jejich průnik.

```
(defun my-intersection (list1 list2)
  (cond ((null list1) ())
        ((elementp (car list1) list2)
         (cons (car list1)
               (my-intersection (cdr list1) list2)))
        (t (my-intersection (cdr list1) list2))))
```

Potenční množina.
Funkce `subsets` vrací k dané množině (reprezentované seznamem) množinu všech jejích podmnožin.

```
(defun subsets (list)
  (if (null list)
      (list ())
    (let ((cdr-subsets (subsets (cdr list))))
      (append-2 cdr-subsets
                (add-to-all (car list) cdr-subsets)))))

(defun add-to-all (elem list)
  (if (null list)
      ()
    (cons (cons elem (car list)) (add-to-all elem (cdr list)))))
```

4 Odbočka: operátor `let*`

Speciální operátor `let*` vyžaduje argumenty ve stejném tvaru jako operátor `let` a pracuje velmi podobně.

```
                popis prostředí
(let* ( (a 2)   (b (+ a 2)) )
       popis vazby  popis vazby
  (* a b) )
   tělo
```

Popis prostředí Seznam libovolné délky. Jeho prvky jsou *popisy vazeb*.

Popis vazby Dvouprvkový seznam. Na prvním místě má symbol, na druhém libovolný výraz.

Tělo Libovolný výraz.

Vyhodnocení

Výraz

```
(let* ((a 2)
       (b (+ a 2)))
  (* a b)))
```

Speciální operátor `let*` pracuje jako několik vnořených výrazů se speciálním operátorem `let`, každý vždy jen s jedním popisem vazby. Výraz nahoře je tedy ekvivalentní výrazu

```
(let ((a 2))
  (let ((b (+ a 2)))
    (* a b)))
```

5 Třídění seznamů

Následuje složitější příklad na třídění seznamu algoritmem *merge sort*. Podrobnosti byly vysvětleny na přednášce.

```
(defun merge-sort (list)
  (let* ((len (length list))
         (len/2 (floor (/ len 2)))
         (list2 (nthcdr len/2 list))
         (list1 (ldiff list list2)))
    (if (<= len 1)
        list
      (merge-lists (merge-sort list1) (merge-sort list2)))))

(defun merge-lists (l1 l2)
  (cond ((null l1) l2)
        ((null l2) l1)
        ((<= (car l1) (car l2))
         (cons (car l1) (merge-lists (cdr l1) l2)))
        (t (cons (car l2) (merge-lists l1 (cdr l2))))))
```

Otázky a úkoly na cvičení

1. Napište funkci `last-pair`, která k danému neprázdnému seznamu vrátí jeho poslední pár:

```
CL-USER 1 > (last-pair (list 1 2 3))
(3)
```

2. Napište funkci `my-copy-list`, která k danému seznamu vrátí jeho kopii, tedy seznam sestavený z nově vytvořených párů, ale obsahující tytéž prvky:

```
CL-USER 2 > (setf l (list 1 2 3))
(1 2 3)

CL-USER 3 > (setf copy (my-copy-list l))
(1 2 3)

CL-USER 4 > (eql l copy)
NIL

CL-USER 5 > (eql (cdr l) (cdr copy))
NIL

CL-USER 6 > (eql (cdr (cdr l)) (cdr (cdr copy)))
NIL
```

3. Napište predikát `equal-lists-p`, který zjistí, zda dané dva seznamy obsahují tytéž prvky (porovnávané predikátem `eql`) ve stejném pořadí. Například (s použitím seznamů `l` a `copy` z předchozího příkladu)

```
CL-USER 7 > (equal-lists l copy)
T

CL-USER 8 > (equal-lists l (list 1 2))
NIL

CL-USER 9 > (equal-lists l (reverse l))
NIL
```

4. Napište funkci `my-remove`, která ze zadaného seznamu vypustí všechny výskyty zadaného prvku:

```
CL-USER 10 > (remove 1 (list 1 2 1 3 1 4 1 5 1 6))
(2 3 4 5 6)
```

5. Napište funkci `remove-nthcdr`, která k danému seznamu vrátí seznam vzniklý nahrazením jeho n-tého *cdr* prázdným seznamem.

6. Napište funkci `each-other`, která k danému seznamu vrátí seznam prvků se sudým indexem. Například:

```
CL-USER 11 > (each-other (list 1 2 3 4 5 6 7))
(1 3 5 7)
```

7. Napište funkci `factorials`, která vrátí seznam zadané délky, jehož n-tý prvek bude mít hodnotu $n!$:

```
CL-USER 3 > (factorials 6)
(1 1 2 6 24 120)
```

   Funkce by měla počítat hodnoty prvků pomocí již vypočítaných hodnot předchozích prvků.

8. Napište funkci `fib-list`, která vrátí seznam zadané délky, jehož n-tý prvek bude roven n-tému Fibonacciho číslu. Funkce by měla počítat hodnoty prvků pomocí již vypočítaných hodnot předchozích prvků.

9. Napište funkci `list-tails`, která k danému seznamu vrátí seznam všech jeho konců včetně vstupního seznamu a prázdného seznamu:

```
CL-USER 12 > (list-tails (list 1 2 3))
((1 2 3) (2 3) (3) NIL)
```

10. Napište funkci `list-sum`, která vrátí součet všech prvků daného seznamu:

```
CL-USER 13 > (list-sum (list 1 2 3))
6
```

11. Napište funkci `subtract-lists-2`, která vypočítá rozdíl dvou stejně dlouhých seznamů chápaných jako vektory:

```
CL-USER 14 > (subtract-lists (list 1 -1 2) (list 2 -1 -2))
(-1 0 -4)
```

12. Napište funkci `scalar-product`, která vrátí skalární součin dvou stejně dlouhých seznamů chápaných jako vektory:

```
CL-USER 15 > (scalar-product (list 1 -1 2) (list 2 -1 -2))
-1
```

13. Napište funkci `vector-length`, která vrátí délku vektoru zadaného seznamem:

```
CL-USER 16 > (vector-length (list 3 0 -4))
5
```

14. Napište funkci `my-remove-duplicates`, která z daného seznamu vypustí duplicitní prvky. Na pořadí prvků výsledku nezáleží, takže toto je jen jedna možnost:

```
CL-USER 17 > (my-remove-duplicates (list 5 6 1 5 5 6 4 2 1))
(5 6 4 2 1)
```

15. Napište funkci `my-union`, která ke dvěma seznamům představujícím množiny vrátí jejich sjednocení:

```
CL-USER 18 > (my-union (list 2 3 7) (list 1 3 5 7 9))
(2 3 7 1 5 9)
```

    (Prvky výsledného seznamu mohou být v jiném pořadí.)

16. Následující predikát zjišťuje, zda jsou si dvě množiny rovny:

```
(defun equal-sets-p (list1 list2)
  (and (my-subsetp list1 list2) (my-subsetp list2 list1)))
```

    Navrhněte rychlejší verzi tohoto predikátu.

17. Napište funkci `flatten`, která „rozpustí“ podseznamy daného seznamu (ze zápisu seznamu vymaže všechny závorky kromě první a poslední):

```
CL-USER 1 > (setf l (list (list (list 1) 2 3 4) 5 (list 6 7)))
(((1) 2 3 4) 5 (6 7))

CL-USER 2 > (flatten l)
(1 2 3 4 5 6 7)
```

18. Napište funkci `deep-reverse`, která otočí daný seznam a všechny jeho podseznamy, pod-podseznamy atd.:

```
CL-USER 3 > (setf lst (list 1
                            (list 2 (list 3 4) 5)
                            (list 6 7)))
(1 (2 (3 4) 5) (6 7))

CL-USER 4 > (deep-reverse lst)
((7 6) (5 (4 3) 2) 1)
```