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. Páry a seznamy 2

verze z 30. října 2024

1 Opakování

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):

```
> (cons 1 2)
(1 . 2)
```

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

```
> (bind c (cons 1 2))
(1 . 2)

> (car c)
1

> (cdr c)
2
```

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

```
> (consp 1)
NIL

> (consp t)
NIL

> (consp c)
T
```

Predikáty `null` a `atom`

```
> (null 1)
NIL

> (atom 1)
T

> (null (cons 1 2))
NIL

> (atom (cons 1 2))
NIL

> (null nil)
T

> (atom nil)
T
```

Implementace:

```
(defun null (elem)
  (eql elem nil))

(defun atom (elem)
  (not (consp elem)))
```

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:

```
> (cons 1 (cons 2 (cons 3 (cons 4 nil))))
(1 2 3 4)
```

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

Seznam je délky 4.

Nová vestavěná funkce na vytvoření seznamu:

```
> (list 1 2 3 4)
(1 2 3 4)

> (car (list 1 2 3 4))
1
```

```
> (cdr (list 1 2 3 4))
(2 3 4)
```

2 Další základní funkce

Minule jsme implementovali funkce, které se seznamy pracují, ale nové seznamy nevytvářejí. Takové funkce si ukážeme dnes.

K napsání většiny funkcí se opět hodí uvažovat **deklarativně**.

Predikát `proper-list-p` podrobněji.

Nejprve se ale podrobněji vrátíme k predikátu `proper-list-p`, který 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. Poslední je snadné napsat, pokud uvažujeme **deklarativně**.

Spojení dvou seznamů.

Funkce `append-2` spojuje dva seznamy. Později si řekneme, jak ji lze zobecnit, 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)
```

Verze s koncovou rekurzí dělá něco jinak.

Pokus vytvořit verzi s koncovou rekurzí vede k jinému výsledku:

```
(defun revappend (list1 list2)
  (if (null list1)
      list2
    (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 reverse (list)
  (revappend list nil))
```

Test:

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

Každý druhý prvek seznamu.

```
(defun even-conses (list)
  (if (null list)
      nil
    (cons (car list) (odd-conses (cdr list)))))

(defun odd-conses (list)
  (if (null list)
      nil
    (even-conses (cdr list))))
```

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)
      nil
    (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)
      nil
    (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 adjoin (x list)
  (if (elementp x list)
      list
    (cons x list)))
```

Podmnožinovost.

Predikát `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 subsetp (list1 list2)
  (if (null list1)
      t
    (and (elementp (car list1) list2)
         (subsetp (cdr list1) list2))))
```

Lepší verze.

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

Průnik.

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

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

Potenční množina.

Funkce `subsets` vrací k dané množině (reprezentované seznamem) množinu všech jejích podmnožin. Lze ji napsat i jinak.

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

4 Třídění seznamů sléváním

Následuje složitější příklad: třídění seznamu algoritmem *merge sort*. Stručný popis algoritmu:

Třídění seznamu l algoritmem *merge sort*

Algoritmus *merge sort*

Vstup: seznam l
Výstup: seznam vzniklý setříděním seznamu l

1. Je-li l prázdný nebo jednoprvkový, výsledkem je l.
2. Není-li, rozdělíme l na seznamy l_1 a l_2 délky lišící se nejvýše o 1.
3. Seznamy l_1 a l_2 setřídíme algoritmem *merge sort*.
4. Setříděné seznamy slijeme a výsledek vrátíme.

Algoritmus slévání

Vstup: setříděné seznamy l_1 a l_2
Výstup: setříděný seznam s prvky seznamů l_1 a l_2

1. Je-li l_1 nebo l_2 prázdný, vrátíme ten druhý.
2. Jinak, je-li první prvek $l_1 \leq$ prvnímu prvku l_2, slijeme $cdr\, l_1$ a l_2, dopředu přidáme $car\, l_1$ a výsledek vrátíme.
3. Jinak slijeme l_1 a $cdr\, l_2$, dopředu přidáme $car\, l_2$ a výsledek vrátíme.

Další podrobnosti byly vysvětleny na přednášce.

```
(defun merge-sort (list)
  (if (or (null list)
          (null (cdr list)))
      list
    (merge-lists (merge-sort (even-conses list))
                 (merge-sort (odd-conses list)))))

(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))))))
```

Nové symboly

funkce: `list`

Otázky a úkoly na cvičení

1. Napište funkci `make-ar-seq-list`, která vytvoří seznam členů aritmetické posloupnosti se zadaným prvním členem, diferencí a počtem členů:

```
> (make-ar-seq-list 10 2 6)
(10 12 14 16 18 20)
```

2. Napište funkci `make-ar-seq-list` pomocí koncové rekurze.

3. Napište funkci `make-geom-seq-list` která vytvoří seznam členů geometrické posloupnosti s daným prvním členem, kvocientem a počtem členů:

```
> (make-geom-seq-list 5 2 10)
(5 10 20 40 80 160 320 640 1280 2560)
```

4. Napište ji pomocí koncové rekurze.

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

```
> (bind l (list 1 2 3))
(1 2 3)

> (bind copy (copy-list l))
(1 2 3)

> (eql l copy)
NIL

> (eql (cdr l) (cdr copy))
NIL

> (eql (cdr (cdr l)) (cdr (cdr copy)))
NIL
```

(do jazyka)

6. Napište funkci `remove`, která ze zadaného seznamu vypustí všechny výskyty zadaného prvku:

```
> (remove 1 (list 1 2 1 3 1 4 1 5 1 6))
(2 3 4 5 6)
```

(do jazyka)

7. Napište predikát `tailp`, který pro dva zadané seznamy zjistí, zda první je totožný (pomocí `eql`) s několikátým *Cdr* druhého:

```
> (bind list (list 1 2 3 4))
(1 2 3 4)
```

```
> (tailp (cdr (cdr list)) list)
T

> (tailp (list 3 4) list)
NIL
```

(do jazyka)

8. Napište funkci `ldiff`, která k daným dvěma seznamům vrátí nový seznam podle toho, zda druhý z argumentů je, nebo není několikátým *Cdr* prvního: pokud je, bude nový seznam obsahovat všechny prvky prvního ze seznamů až po druhý, pokud není, bude výsledkem kopie prvního seznamu (proměnná `list` obsahuje seznam z předchozího příkladu):

```
> (ldiff list (cdr (cdr list)))
(1 2)

> (ldiff list (list 3 4))
(1 2 3 4)
```

(do jazyka)

9. Napište znovu funkci `copy-list`, tentokrát pomocí funkce `ldiff`.

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

```
> (factorials 6)
(1 1 2 6 24 120)
```

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

11. 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.

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

```
> (list-tails (list 1 2 3))
((1 2 3) (2 3) (3) NIL)
```

13. Funkci `my-sum` definujeme takto:

```
(defun my-sum-help (list n len)
  (if (>= n len)
      0
    (+ (nth n list) (my-sum-help list (+ n 1) len)))))

(defun my-sum (list)
  (my-sum-help list 0 (length list))))
```

Kolikrát se během vyhodnocování výrazu `(my-sum (list 1 2 3 4 5 6 7 8 9 10))` zavolá funkce `cdr`?

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

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

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

```
> (scalar-product (list 1 -1 2) (list 2 -1 -2))
-1
```

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

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

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

```
> (remove-duplicates (list 5 6 1 5 5 6 4 2 1))
(5 6 4 2 1)
```

(do jazyka)

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

```
> (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.) *(do jazyka)*

19. 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.

20. Vylepšete reprezentaci množin seznamy pro množiny čísel tím, že budete seznamy stále udržovat setříděné. To by mělo zrychlit hledání i přidávání prvků.

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

```
> (bind l (list (list (list 1) 2 3 4) 5 (list 6 7)))
(((1) 2 3 4) 5 (6 7))

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

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

```
> (bind lst (list 1 (list 2 (list 3 4) 5) (list 6 7)))
(1 (2 (3 4) 5) (6 7))

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