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

9. Seznamy 2

1 Páry a seznamy: opakování a něco navíc

Nejprve zopakujeme základní informace o párech a seznamech.

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

Procedura `cons` (konstruktor tečkových párů):

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

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

```
> (define p (cons 3 4))
> (car p)
3
> (cdr p)
4
```

Typový predikát `pair?` (test na tečkové páry):

```
> (pair? 1)
#f
> (pair? (cons 1 1))
#t
```

Prázdný seznam je zvláštní hodnota (atom) `()`, vyhodnocuje se sám na sebe:

```
> ()
()
```

Predikát `null?` (test na prázdný seznam):

```
> (null? ())
#t
> (null? (cons 1 2))
#f
> (null? 12)
#f
```

Může být napsán takto:

```
(define (null? x)
  (eq? x ()))
```

Seznam je buď prázdný seznam, nebo tečkový pár, jehož *cdr* je seznam. Zápis seznamu:

```
> (cons 0 (cons 1 (cons 2 (cons 3 ()))))
(0 1 2 3)
```

`0, 1, 2, 3`: *prvky seznamu.*

Délka seznamu: počet prvků neboli počet párů.

Procedura na vytvoření seznamu:

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

2 Základy práce se seznamy

Některé uvedené procedury jsou ve Scheme k dispozici. Tady jsou uvedeny na ukázku, jak by mohly být napsány. Ve Scheme nejsou procedury `remove-tail`, `append-2` (je tam ale obecnější procedura `append`) a `revapend`.

Typový predikát. Predikát `list?` zjišťuje, zda je jeho argument seznam. Tady je několik možných variant, jak by mohl být napsán (poslední je nejlepší).

```
(define (list? x)
  (if (null? x)
      #t
      (if (pair? x)
          (if (list? (cdr x))
              #t
              #f)
          #f)))
```

```
(define (list? x)
  (if (null? x)
      #t
      (if (pair? x)
          (list? (cdr x))
          #f)))
```

```
(define (list? x)
  (if (null? x)
      #t
      (and (pair? x) (list? (cdr x)))))
```

```
(define (list? x)
  (or (null? x)
      (and (pair? x) (list? (cdr x)))))
```

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

n-tý prvek. Procedura `list-ref` vrací prvek seznamu o daném indexu (indexuje se od nuly).

```
(define (list-ref lst k)
  (if (= k 0)
      (car lst)
      (list-ref (cdr lst) (- k 1))))
```

n-tý zbytek. Procedura `list-tail` vrací pár o daném indexu v seznamu (indexuje se od nuly).

```
(define (list-tail lst k)
  (if (= k 0)
      lst
      (list-tail (cdr lst) (- k 1))))
```

n-tý prvek (znovu). Vidíme, že proceduru `list-ref` můžeme zjednodušit použitím procedury `list-tail`:

```
(define (list-ref lst k)
  (car (list-tail lst k)))
```

Odstranění zbytku seznamu. Procedura `remove-tail` vrací k danému seznamu a jeho zbytku seznam bez zbytku:

```
(define (remove-tail lst tail)
  (if (or (null? lst) (eq? lst tail))
      ()
      (cons (car lst) (remove-tail (cdr lst) tail))))
```

Spojení dvou seznamů. Proceduru `append-2` později zobecníme, aby pracovala s libovolným počtem seznamů.

```
(define (append-2 lst1 lst2)
  (if (null? lst1)
      lst2
      (cons (car lst1) (append-2 (cdr lst1) lst2))))
```

Test:

```
> (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 překvapivému výsledku:

```
(define (revappend lst1 lst2)
  (if (null? lst1)
      lst2
      (revappend (cdr lst1) (cons (car lst1) lst2))))
```

Test:

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

Otočení seznamu. Pomocí procedury `revappend` můžeme vytvořit užitečnou proceduru `reverse`:

```
(define (reverse lst)
  (revappend lst ()))
```

Test:

```
> (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. Procedura `scale-list` vynásobí všechny prvky daného seznamu (musí to tedy být čísla) daným číslem.

```
(define (scale-list lst factor)
  (if (null? lst)
      ()
      (cons (* factor (car lst))
            (scale-list (cdr lst) factor))))
```

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

```
(define (sum-lists-2 lst1 lst2)
  (if (null? lst1)
      ()
      (cons (+ (car lst1) (car lst2))
            (sum-lists-2 (cdr lst1) (cdr lst2)))))
```

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 `element?` testuje, zda je daná hodnota prvkem seznamu. K porovnávání používá predikát `eq?`.

```
(define (element? x lst)
  (and (not (empty? lst))
       (or (eq? x (car lst))
           (element? x (cdr lst)))))
```

Podmnožinovost. Predikát `subset?` 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 `element?`.

```
(define (subset? lst1 lst2)
  (if (null? lst1)
      #t
      (and (element? (car lst1) lst2)
           (subset? (cdr lst1) lst2))))
```

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

```
(define (intersection lst1 lst2)
  (cond ((null? lst1) ())
        ((element? (car lst1) lst2) (cons (car lst1)
                                          (intersection (cdr lst1)
                                                        lst2)))
        (#t (intersection (cdr lst1) lst2))))
```

4 Třídění seznamů

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

```
(define (merge-sort lst)
  (let* ((len (length lst))
         (len/2 (floor (/ len 2)))
         (lst2 (list-tail lst len/2))
         (lst1 (remove-tail lst lst2)))
    (if (<= len 1)
        lst
        (merge-lists (merge-sort lst1) (merge-sort lst2)))))

(define (merge-lists lst1 lst2)
  (cond ((null? lst1) lst2)
        ((null? lst2) lst1)
        ((<= (car lst1) (car lst2)) (cons (car lst1)
                                          (merge-lists (cdr lst1) lst2)))
        (#t (cons (car lst2)
                  (merge-lists lst1 (cdr lst2))))))
```

Otázky a úkoly na cvičení

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

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

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

```
> (define lst (list 1 2 3))
> (define lst-copy (copy-list lst))
> lst-copy
(1 2 3)
> (eq? lst lst-copy)
#f
> (eq? (cdr lst) (cdr lst-copy))
#f
... atd.
```

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

```
> (equal-lists? lst lst-copy)
#t
> (equal-lists? lst (reverse lst-copy))
#f
> (equal-lists? lst (list 1 2))
#f
```

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

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

5. Napište proceduru `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) ())
```

6. Napište proceduru `list-sum`, která vrátí součet všech prvků daného seznamu:

```
> (list-sum (list 1 2 3))
6
```

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

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

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

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

9. Napište proceduru `vector-length`, která vrátí délku vektoru zadaného seznamem:

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

10. Napište proceduru `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)
```

11. Napište proceduru `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.)

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

```
(define (equal-sets? lst1 lst2)
  (and (subset? lst1) (subset? lst2)))
```

Navrhněte rychlejší verzi tohoto predikátu.

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

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

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

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