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

10. Pokročilá práce se seznamy

1 Potlačení vyhodnocení operátorem `quote`

```
> (quote (1 2 3))
(1 2 3)
> (quote symbol)
symbol
```

Zkráceně:

```
> '(1 2 3)
(1 2 3)
> 'symbol
symbol
```

Symboly jsou tedy prvky prvního řádu.

Pomocí `quote` lze jednoduše pracovat se seznamy. Ale nikoliv je vytvářet (to teď není podstatné, ale bude, až začneme pracovat s vedlejším efektem):

```
(define (quote-test-1)
  (list 1 2 3))

(define (quote-test-2)
  '(1 2 3))
```

```
> (eq? (quote-test-1) (quote-test-1))
#f
> (eq? (quote-test-2) (quote-test-2))
#t
```

2 Procedury s nepovinnými parametry

Pokud ve zřetězených tečkových párech není poslední *cdr* prázdný seznam, ale libovolná jiná hodnota, hovoříme někdy o *tečkovém seznamu*. **Není to seznam** (nesplňuje podmínky definice seznamu), ale zapisuje se podobně:

```
> (define dl (cons 1 (cons 2 (cons 3 4))))
> dl
(1 2 3 . 4)
> (cdr dl)
(2 3 . 4)
> (cdr (cdr dl))
(3 . 4)
> (cdr (cdr (cdr dl)))
4
```

O hodnotách v *car* jednotlivých párů stále říkáme, že jsou to *prvky* tečkového seznamu.

Krajní případ: libovolnou hodnotu, která není pár, chápeme jako tečkový seznam, který nemá žádný prvek a jehož posledním *cdr* je ona hodnota.

Jiný zápis vyhodnocovaného seznamu

Aplikace procedury na seznam argumentů:

```
> (+ . (1 2 3))
6
> (+ 1 . (2 3))
6
> (+ 1 2 . (3))
6
```

Ve všech případech se vyhodnocoval seznam `(+ 1 2 3)`.

Toho využívá Scheme u definice procedur s nepovinnými parametry. Například procedura

```
(define proc-1 (lambda (x y . r)
                 (display x)
                 (newline)
                 (display y)
                 (newline)
                 (display r)))
```

se chová takto:

```
> (proc-1 1 2)
1
2
()
> (proc-1 1 2 3)
1
2
(3)
> (proc-1 1 2 3 4)
1
2
(3 4)
```

Parametry `x` a `y` hrají podobnou roli jako dříve: při aplikaci procedury se na ně navážou první dva argumenty. Můžeme říci, že jsou to její *povinné parametry*. Symbol `r` je rovněž parametr a při aplikaci procedury obsahuje seznam dalších argumentů, na které byla procedura aplikována.

Tečka v zápisu tečkového seznamu `(x y . r)` odděluje poviné parametry od parametru `r`. Tento tečkový seznam by mohl být ručně vytvořen takto:

```
> (cons 'x (cons 'y 'r))
(x y . r)
```

Seznam parametrů procedury v λ-výrazu může být tečkový seznam. V takovém případě symboly před tečkou jsou povinné parametry. Uživatel musí proceduru aplikovat nejméně s tolika argumenty, kolik je těchto parametrů. Při aplikaci procedury budou tyto symboly navázány na argumenty jako dříve.

Symbol za tečkou je symbol pro ostatní argumenty. Při aplikaci procedury bude symbol navázán na seznam ostatních (nepovinných) argumentů.

Procedura `proc-1` musí tedy být aplikována na nejméně dva argumenty. Příklady, kdy aplikace procedury vede k chybě kvůli nesprávnému počtu argumentů:

```
> (proc-1 1)
chyba
> (proc-1)
chyba
```

Proceduru, která přijímá libovolný počet argumentů (včetně žádného), lze definovat takto:

```
(define proc-2 (lambda r
                 (display r)))
```

V tomto případě bude při aplikaci procedury parametr `r` navázán na seznam všech argumentů (tedy případně i na prázdný seznam):

```
> (proc-2 1)
(1)
> (proc-2 1 2 3 4)
(1 2 3 4)
> (proc-2)
()
```

Procedury `proc-1` a `proc-2` lze definovat i bez pomoci λ-výrazu:

```
(define (proc-1 x y . r)
  (display x)
  (newline)
  (display y)
  (newline)
  (display r))

(define (proc-2 . r)
  (display r))
```

Proceduru `list` na vytvoření seznamu, kterou známe z minulé přednášky, lze tedy napsat takto jednoduše:

```
(define (list . x)
  x)
```

Test:

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

Příklad. Procedura `inc` vrátí k danému číslu číslo o `1` větší. Jako nepovinný argument můžeme zadat jiný přírůstek:

```
(define (inc x . inc-lst)
  (let ((increment (if (null? inc-lst) 1 (car inc-lst))))
    (+ x increment)))
```

Testy:

```
> (inc 5)
6
> (define a 10)
> (inc a)
11
> (inc a 10)
20
> (inc 5 a)
15
```

3 Procedury vyššího řádu pro seznamy

Z následujících procedur je většina ve Scheme k dispozici. Tady si ukazujeme, jak by mohly být napsány. (Názvy procedur se někdy liší od těch uvedených na přednášce.)

Hledání. Procedura `find` najde první prvek seznamu, který se rovná zadanému prvku. V případě neúspěchu vrací `#f`. Jako test rovnosti je použit predikát `eq?`. Je možné zadat vlastní test rovnosti jako třetí nepovinný argument.

```
(define (find x lst . pred-lst)
  (let ((pred (if (null? pred-lst) eq? (car pred-lst))))
    (cond ((null? lst) #f)
          ((pred x (car lst)) (car lst))
          (#t (find x (cdr lst) pred)))))
```

Testy:

```
> (find 2 '(1 2 3))
2
> (find 5 '(1 2 3))
#f
> (find 'b '(a b c))
b
> (find 2 '(1 2 3) >)
1
> (find 2 '(1 2 3) <)
3
> (find 'b
        '((a . 1) (b . 2) (c . 3))
        (lambda (x y) (eq? x (car y))))
(b . 2)
```

Procedura `findf` najde první prvek seznamu splňující daný predikát. V případě neúspěchu vrací `#f`.

```
(define (findf proc? lst)
  (cond ((null? lst) #f)
        ((proc? (car lst)) (car lst))
        (#t (findf proc? (cdr lst)))))
```

Testy:

```
> (findf (lambda (x) (> x 10))
         '(2 4 6 7 8 12 5))
12
> (findf odd? '(2 4 6 7 8 12 5))
7
> (findf null? '(2 4 6 7 8 12 5))
#f
```

Procedury `find` a `findf` nám nepomohou, pokud hledáme prvek `#f`:

```
> (find #f '(1 2 3))
#f
> (find #f '(1 2 3 #f))
#f
> (findf (lambda (x) (not (eq? x 0)))
         '(0 0 0 #f 0 0 0))
#f
```

Procedury `member` a `memf` pracují stejně jako procedury `find` a `findf`, ale v případě úspěchu nevrací hledaný prvek, ale celý zbytek seznamu, který tímto prvkem začíná. Například pro proceduru `member`:

```
> (member 2 '(1 2 3))
(2 3)
> (member 5 '(1 2 3))
#f
> (member 2 '(1 2 3) >)
(1 2 3)
> (member 2 '(1 2 3) <)
(3)
> (member 'b
          '((a . 1) (b . 2) (c . 3))
          (lambda (x y) (eq? x (car y))))
((b . 2) (c . 3))
```

Tyto procedury nám tedy umožní odlišit nalezení `#f` od neúspěchu:

```
> (member #f '(1 2 3))
#f
> (member #f '(1 2 3 #f))
(#f)
> (memf (lambda (x) (not (eq? x 0)))
        '(0 0 0 #f 0 0 0))
(#f 0 0 0)
```

Filtrace. Procedura `filter` vrátí seznam prvků daného seznamu, které splňují daný predikát:

```
(define (filter pred? lst)
  (cond ((null? lst) ())
        ((pred? (car lst)) (cons (car lst)
                                 (filter pred? (cdr lst))))
        (#t (filter pred? (cdr lst)))))
```

Příklad:

```
> (filter odd? '(1 2 3 4 5 6 7 8 9))
(1 3 5 7 9)
```

Mapování. Procedura `map-1` aplikuje zadanou proceduru na každý prvek zadaného seznamu a výsledky vrátí v seznamu.

```
(define (map-1 proc lst)
  (if (null? lst)
      ()
      (cons (proc (car lst)) (map-1 proc (cdr lst)))))
```

Procedura má rozmanité použití:

```
> (map-1 - '(1 2 3))
(-1 -2 -3)
> (map-1 - '(1 2 3 4 5 6))
(-1 -2 -3 -4 -5 -6)
> (map-1 even? '(1 2 3 4 5 6))
(#f #t #f #t #f #t)
> (map-1 car '((1) (1 2) (2 3 4) (4 5 6 7) (7 8 9 10 11)))
(1 1 2 4 7)
```

```
> (map-1 cdr '((1) (1 2) (2 3 4) (4 5 6 7) (7 8 9 10 11)))
(() (2) (3 4) (5 6 7) (8 9 10 11))
> (map-1 length '((1) (1 2) (2 3 4) (4 5 6 7) (7 8 9 10 11)))
(1 2 3 4 5)
> (map-1 (lambda (x) (if (pair? x) (car x) #f))
         '(() (2) (3 4) (5 6 7) (8 9 10 11)))
(#f 2 3 5 8)
```

Akumulace. Proceduru `foldr-1` pochopíte z její definice a z příkladů.

```
(define (foldr-1 proc init lst)
  (if (null? lst)
      init
      (proc (car lst) (foldr proc init (cdr lst)))))
```

```
> (foldr-1 + 0 '(1 2 3 4))
10
> (foldr-1 - 0 '(1 2 3 4))
-2
> (foldr-1 cons () '(1 2 3 4))
(1 2 3 4)
> (foldr-1 cons '(5 6 7 8) '(1 2 3 4))
(1 2 3 4 5 6 7 8)
> (foldr-1 list '() '(1 2 3 4))
(1 (2 (3 (4 ()))))
> (foldr-1 (lambda (x y) (cons (- x) y))
           ()
           '(1 2 3))
(-1 -2 -3)
> (foldr-1 (lambda (x y) (if (odd? x) (cons x y) y))
           ()
           '(1 2 3 4 5 6 7 8 9))
(1 3 5 7 9)
```

Vidíme, že procedura `foldr-1` je hodně obecná, s její pomocí můžeme se seznamy provádět operace, na které jsme používali dříve uvedené procedury, např. mapování a filtraci.

4 Aplikace pomocí procedury `apply`

Během vyhodnocování seznamu, jehož první prvek se vyhodnotí na proceduru, dochází k aplikaci této procedury na seznam argumentů, které vzniknou vyhodno-

cením prvků *cdr* vyhodnocovaného seznamu. Procedura `apply` umožňuje aplikovat zadanou proceduru na seznam argumentů vzniklý až během práce programu:

```
> (apply + '(1 2 3))
6
> (apply cons '(1 2))
(1 . 2)
```

Procedura akceptuje i více než dva argumenty. Pokud je aplikována na více argumentů, tak argumenty předcházející poslednímu (kromě prvního) s posledním argumentem spojí do seznamu, takže např. výrazy `(apply + '(1 2 3))`, `(apply + 1 '(2 3))`, `(apply + 1 2 '(3))`, `(apply + 1 2 3 ())` povedou ke stejnému výsledku.

Příklady:

```
> (apply min 5 2 '(6 1 7))
1
> (apply * 10 (map (lambda (x) (+ 1 x))
                   (list 0 1 2))
60
```

Procedura `apply` je tedy aplikována na následující argumenty: proceduru, libovolný počet hodnot a seznam. Výsledkem aplikace je výsledek aplikace zadané procedury na uvedené hodnoty a prvky uvedeného seznamu.

Příklad (porovnání seznamů). Zobecníme proceduru `equal-lists?` z minulého cvičení, aby akceptovala libovolný počet argumentů. Původní proceduru přejmenujeme na `equal-lists-2?`. Nová procedura bude pracovat tak, že pomocí procedury `equal-lists-2?` porovná první seznam se všemi ostatními.

```
(define (equal-lists? (lst1 . lists))
  (if (null? lists)
      #t
      (and (equal-lists-2? lst1 (car lists))
           (apply 'equal-lists? lst1 (cdr lists)))))
```

Otázky a úkoly na cvičení

1. Napište proceduru `power`, která umocní zadané číslo na druhou. Jako nepovinný argument akceptuje jiný exponent:

```
> (power 5)
25
> (power 5 3)
125
```

2. Napište proceduru `seconds`, která vrátí počet sekund od začátku (nepřestupného) roku. Jako argumenty procedura akceptuje měsíc, den, hodinu, minutu a sekundu, všechny počítány od 0. Kromě měsíce jsou další argumenty nepovinné a mají výchozí hodnotu 0.

3. Napište procedury `member` a `memf`. Pak pomocí nich přepište procedury `find` a `findf`.

4. Navrhněte iterativní varianty procedur `find`, `findf`. Všimněte si, že se (pokud je napíšete jednoduše), chovají jinak, než původní procedury.

5. Napište některou z procedur pro hledání, filtraci a mapování pomocí procedury `foldr`.

6. Napište proceduru `foldl`, která pracuje stejně jako `foldr` až na to, že prochází seznam zleva doprava. Nepoužívejte přitom proceduru `reverse`.

7. Zobecněte proceduru `sum-lists-2` z minulé přednášky na proceduru `sum-lists` tak, aby akceptovala libovolný počet seznamů. Kolik by měla procedura mít povinných parametrů? Vždy můžete předpokládat, že argumenty procedury jsou seznamy stejné délky.

8. Napište proceduru `map`, která bude zobecněním procedury `map-1` na libovolný počet seznamů:

```
> (map + '(1 2 3) '(4 5 6))
(5 7 9)
> (map - '(4 5 6) '(1 2 3))
(3 3 3)
> (map cons '(a b c) '(1 2 3))
((a . 1) (b . 2) (c . 3))
> (map + '(1 2 3) '(4 5 6) '(7 8 9))
(12 15 18)
> (map list '(a b c d) '(1 2 3 4) '(#t #f #t #f))
((a 1 #t) (b 2 #f) (c 3 #t) (d 4 #f))
```

   Můžete předpokládat, že všechny seznamy, které jsou argumenty procedury, mají stejnou délku.