Paradigmata programování 1

Seznamy

Vilém Vychodil

Katedra informatiky, PřF, UP Olomouc

Přednáška 5

Přednáška 5: Přehled

1. Seznamy jako datové struktury:
   - konstrukce seznamů pomocí párů,
   - boxová notace s ukazateli,
   - seznamy jako S-výrazy a jako data.
2. Procedury pro manipulaci se seznamy a související témata:
   - konstruktory seznamů, délka seznamu, zřetězení, mapování,
   - páry uchovávající délku seznamu,
   - abstraktní interpret Scheme a správa paměti.
3. Typové systémy:
   - datové typy v jazyku Scheme,
   - vlastnosti typových systémů v programovacích jazycích.

Příklad (Motivační příklad: zapouzdření více hodnot)

Způsoby zapouzdření 4 hodnot:

```
(cons 10 (cons 20 (cons 30 40))) ⟼ (10 . (20 . (30 . 40)))
(cons (cons (cons 10 20) 30) 40) ⟼ (((10 . 20) . 30) . 40)
(cons (cons 10 20) (cons 30 40)) ⟼ ((10 . 20) . (30 . 40))
(cons 10 (cons (cons 20 30) 40)) ⟼ (10 . ((20 . 30) . 40))
```

⋮

Úmluva:

- páry se zanořují „zleva doprava“
- místo posledního prvku je „indikátor konce“

```
(cons 10 (cons 20 (cons 30 (cons 40 'KONEC))))
   ⟼ (10 . (20 . (30 . (40 . KONEC))))
```

Prázdný seznam

- nový element jazyka,
- slouží k „ukončování seznamů“,
- vyhodnocuje se sám na sebe (podle bodu (D), viz Eval):

```
()          ⟼  ()   v některých interpretech Scheme končí chybou
(quote ())  ⟼  ()
'()         ⟼  ()
```

Poznámky:

- v některých dialektech LISPu `()` označuje symbol `NIL` vyhodnocující se na `NIL`
- prázdný seznam „je pouze jeden“

Definice (Seznam)

Seznam je každý element L splňující právě jednu z následujících podmínek:

1. L je prázdný seznam (to jest L je element vzniklý vyhodnocením `'()`), nebo
2. L je pár ve tvaru $(E \,.\, L')$, kde E je libovolný element a L' je seznam. V tomto případě se element E nazývá **hlava seznamu** L a seznam L' se nazývá **tělo seznamu** L (řidčeji též *ocas seznamu* L).

Předpokládejme, že seznam L je ve tvaru $(E \,.\, L')$.

Pod pojmem **prvek seznamu** L rozumíme element E (*první prvek seznamu* L) a dále všechny prvky seznamu L'.

Počet všech prvků seznamu se nazývá **délka seznamu**.

Prázdný seznam *nemá žádný prvek*.

Důsledek: prázdný seznam má *délku* 0.

Externí reprezentace seznamů

Zkrácená notace (vychází z externí reprezentace párů):

1. pokud je druhým prvkem páru opět pár, odstraníme tečku náležející reprezentaci vnějšího páru a závorky náležejících vnitřnímu páru,
2. pokud je druhým prvkem páru prázdný seznam, odstraníme tečku z reprezentace celého páru a obě závorky z reprezentace prázdného seznamu.

Příklad (Příklady seznamů)

`()`	`= ()`	prázdný seznam
`(a . ())`	`= (a)`	jednoprvkový seznam
`(a . (b . ()))`	`= (a b)`	dvouprvkový seznam
`(1 . (2 . (3 . ())))`	`= (1 2 3)`	tříprvkový seznam
`(1 . ((20 . 30) . ()))`	`= (1 (20 . 30))`	dvouprvkový seznam
`((1 . ()) . ((2 . ()) . ()))`	`= ((1) (2))`	dvouprvkový seznam

Zajímavé důsledky

Konstruktory a selektory seznamů:

- `cons` – vytvoří nový seznam připojením prvku na začátek seznamu
- `car` – vrací první prvek seznamu (hlavu seznamu)
- `cdr` – vrací seznam bez prvního prvku (tělo seznamu)

Pozor:

`(car '())` $\Longmapsto$ „CHYBA: Argument není pár“
`(cdr '())` $\Longmapsto$ „CHYBA: Argument není pár“

Vztah program $\times$ data:

- dva pojmy: seznam jako data $\times$ seznam jako S-výraz
- externí reprezentace seznamu L je čitelná readerem právě tehdy, když je externí reprezentace seznamu L symbolický výraz
- externí reprezentace není vždy čitelná: `(cons + '())` $\Longmapsto$ („procedura“)

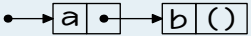
Příklad (Příklady odvozených konstruktorů a selektorů)

```
(define first car)
(define second cadr)
(define third caddr)

(define jednoprvkovy
  (lambda (x)
    (cons x '())))

(define dvouprvkovy
  (lambda (x y)
    (cons x (cons y '()))))

(define triprvkovy
  (lambda (x y z)
    (cons x (cons y (cons z '())))))
```

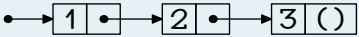
Boxová notace s ukazateli

(a . ()) = (a)

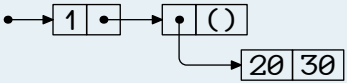
(a . (b . ())) = (a b)



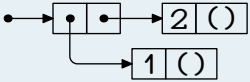
(1 . (2 . (3 . ()))) = (1 2 3)



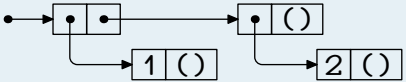
(1 . ((20 . 30) . ())) = (1 (20 . 30))



((1 . ()) . (2 . ())) = ((1) 2)



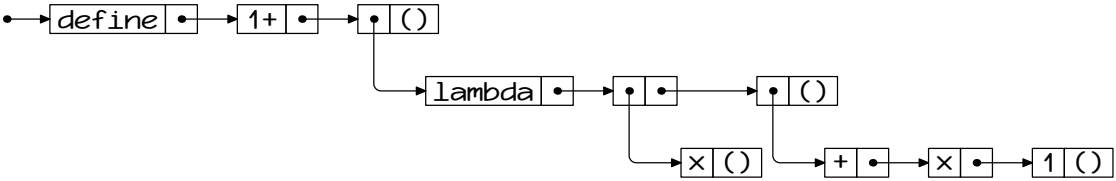
((1 . ()) . ((2 . ()) . ())) = ((1) (2))



Program jako data

Symbolický výraz a jeho interní reprezentace:

```
(define 1+ (lambda (x) (+ x 1)))
```



Úvaha:

- $\mathrm{Eval}[E, \mathcal{P}]$ je definován nad elementy jazyka, nikoliv S-výrazy,
- vyhodnocovací proces formulujeme pro páry.

Definice (vyhodnocení elementu E v prostředí $\mathcal{P}$)

Výsledek vyhodnocení elementu E v prostředí $\mathcal{P}$, značeno $\mathrm{Eval}[E, \mathcal{P}]$, je definován:

(A) Pokud je E *číslo*, ...

(B) Pokud je E *symbol*, ...

(C) Pokud je E *tečkový pár* tvaru $(E_1 \ . \ E_a)$, pak mohou nastat dvě situace: pak $F_1 := \mathrm{Eval}[E_1, \mathcal{P}]$ a rozlišujeme:

(C.α) E_a je seznam $(E_2 \ E_3 \cdots E_n)$; postupujeme následovně:

(C.1) Pokud F_1 je *procedura*, pak pro $F_2 := \mathrm{Eval}[E_2, \mathcal{P}], \ldots, F_n := \mathrm{Eval}[E_n, \mathcal{P}]$ položíme $\mathrm{Eval}[E, \mathcal{P}] := \mathrm{Apply}[F_1, F_2, \ldots, F_n]$.

(C.2) Pokud F_1 je *speciální forma*, pak $\mathrm{Eval}[E, \mathcal{P}] := \mathrm{Apply}[F_1, E_2, \ldots, E_n]$.

(C.e) Pokud F_1 není procedura ani speciální forma: „CHYBA: Nelze provést aplikaci: E_1 se nevyhodnotil na proceduru ani na speciální formu.“.

(C.β) E_a není seznam; skončíme vyhodnocování „CHYBA: Nelze provést aplikaci: E_a není seznam argumentů.“.

(D) Ve všech ostatních případech klademe $\mathrm{Eval}[E, \mathcal{P}] := E$.

Příklad (vyhodnocování párů)

Důsledky:

```
(+ (* 7 3) 5)                                  ⟼  26
(+ (* 7 3) . (5))                              ⟼  26
(+ (* 7 3) . (5 . ()))                         ⟼  26
(+ (* 7 . (3)) 5)                              ⟼  26
(+ . ((* . (7 . (3 . ()))) . (5 . ())))        ⟼  26
```

Možnost explicitně volat Read**:**

```
(+ 1 (read))  ⟼  ···

(list 'zacatek (read) 'konec)  ⟼  ···

(let ((x (read)))
  (cons '+ (cons 1 x)))  ⟼  ···
```

Procedury pro práci se seznamy: `list`

- (odvozený) konstruktor seznamu
- libovolný počet prvků, včetně žádného

Příklad (použití `list`)

```
(list)                          ⟼  ()
(list 1 2 3)                    ⟼  (1 2 3)
(list + 1 2)                    ⟼  („procedura sčítání“ 2 3)
(list '+ 1 2)                   ⟼  (+ 1 2)
(list (+ 1 2))                  ⟼  (3)
(list (list 1) (list 2))        ⟼  ((1) (2))

(list 1 2 #t ())        ⟼  (1 2 #t ())
(quote (1 2 #t ()))     ⟼  (1 2 #t ())
```

Procedury pro práci se seznamy: `length`

- procedura vracející délku seznamu
- pozor: počet atomických elementů $\times$ délka seznamu

Příklad (použití `length`)

```
(length '(a b c d))      ⟹  4
(length '(a (b c) d))    ⟹  3
(length '(a (b c d)))    ⟹  2
(length '((a b c d)))    ⟹  1
```

Indexace prvků:

- indexace $0, \ldots, n-1$, kde n je délka seznamu
- pozor: seznam není pole (**!!**)

Procedury pro práci se seznamy: `build-list`

- konstruktor: bere délku a proceduru F jako argument
- zkonstruuje seznam $(\mathrm{Apply}[F,0]\ \ \mathrm{Apply}[F,1]\ \cdots\ \mathrm{Apply}[F,n-1])$

Příklad (použití `build-list`)

```
(build-list 5 (lambda (x) x))   ⟹  (0 1 2 3 4)
(build-list 5 (lambda (x) 2))   ⟹  (2 2 2 2 2)
(build-list 0 (lambda (x) x))   ⟹  ()
(build-list 5 -)                ⟹  (0 -1 -2 -3 -4)
(build-list 5 list)             ⟹  ((0) (1) (2) (3) (4))
```

- `build-list` je procedura vyššího řádu
- *je nejobecnější konstruktor seznamů, jejichž délka je předem známá a ve kterých hodnoty obsažených prvků závisejí výhradně na pozici prvků v seznamu*

Procedury pro práci se seznamy: `list-ref`

- procedura vracející prvek n-prvkového seznamu na pozici $i \in \{0, \ldots, n-1\}$
- pozor: přistupování k prvků s velkým indexem je neefektivní (**!!**)

Příklad (použití `list-ref`)

```
(list-ref '(a (b c) d) 0)   ⟹  a
(list-ref '(a (b c) d) 1)   ⟹  (b c)
(list-ref '(a (b c) d) 2)   ⟹  d
(list-ref '(a (b c) d) 3)   ⟹  „CHYBA: prvek na pozici 3 neexistuje“
(list-ref '() 0)            ⟹  „CHYBA: prvek na pozici 0 neexistuje“
```

Procedury pro práci se seznamy: reverse

- převrácení seznamu

Příklad (použití reverse)

```
(reverse '(a b c d))  ⟹  (d c b a)
(reverse '())         ⟹  ()
```

Umíme naprogramovat:

```
(define reverse
  (lambda (l)
    (let ((len (length l)))
      (build-list len
                  (lambda (i)
                    (list-ref l (- len i 1)))))))
```

Procedury pro práci se seznamy: append

- vytvoří nový seznam spojením libovolného množství seznamů

Příklad (použití append)

```
(append '(a b c) '(1 2))          ⟹  (a b c 1 2)
(append '(a (b c)) '(1 2))        ⟹  (a (b c) 1 2)
(append '(a (b) c) '((1 2)))      ⟹  (a (b) c (1 2))
(append '() '(1 2))               ⟹  (1 2)
(append '(a b c) '())             ⟹  (a b c)
(append '() '())                  ⟹  ()

(append '(a b c) '(1 2) '(#t #f))  ⟹  (a b c 1 2 #t #f)
(append '(a b c) '() '(#t #f))     ⟹  (a b c #t #f)
(append '() '() '())               ⟹  ()
(append '(a b c))                  ⟹  (a b c)
(append)                           ⟹  ()
```

Příklad (spojení dvou seznamů: procedura append2)

Umíme naprogramovat (je hrubě neefektivní **!!**):

```
(define append2
  (lambda (l1 l2)
    (let ((len1 (length l1))
          (len2 (length l2)))
      (build-list (+ len1 len2)
                  (lambda (i)
                    (if (< i len1)
                        (list-ref l1 i)
                        (list-ref l2 (- i len1))))))))
```

Poznámka:

```
(append 'a b c) '(1 2))                    ⟹  (a b c 1 2)
(cons 'a (cons 'b (cons 'c '(1 2))))       ⟹  (a b c 1 2)
```

Mapování přes seznamy a procedura map

- dva argumenty: procedura F a seznam $\langle prvek_1 \rangle, \langle prvek_2 \rangle, \ldots, \langle prvek_n \rangle$
- výsledek: $(\mathrm{Apply}(F, \langle prvek_1 \rangle)\ \ \mathrm{Apply}(F, \langle prvek_2 \rangle) \cdots \mathrm{Apply}(F, \langle prvek_n \rangle))$

Příklad (použití map)

```
(map (lambda (x) (+ x 1)) '(1 2 3 4))   ⟹  (2 3 4 5)
(map - '(1 2 3 4))                      ⟹  (-1 -2 -3 -4)
(map list '(1 2 3 4))                   ⟹  ((1) (2) (3) (4))
(map (lambda (x) x) '(a (b c) d))       ⟹  (a (b c) d)
(map (lambda (x) #f) '(1 2 3))          ⟹  (#f #f #f)
(map (lambda (x) #f) '())               ⟹  ()
(map (lambda (x) (<= x 3)) '(1 2 3 4))  ⟹  (#t #t #t #f)
(map even? '(1 2 3 4))                  ⟹  (#f #t #f #t)
```

Mapování přes více seznamů a procedura map

- tři a více argumentů:
  - první argument: procedura F, která bere k argumentů
  - další argumenty: seznamy $L_1, \ldots, L_k$

Příklad (použití map pro více seznamů)

```
(map + '(1 2 3) '(10 20 30))      ⟹  (11 22 33)
(map cons '(a b) '(x y))          ⟹  ((a . x) (b . y))
(map cons '(a b #t) '(x y #f))    ⟹  ((a . x) (b . y) (#t . #f))

(map <= '(1 2 3 4 5) '(5 4 3 2 1))
   ⟹  (#t #t #t #f #f)
(map (lambda (x y z)
       (list x (+ y z))) '(a b) '(1 2) '(10 20))
   ⟹  ((a 11) (b 22))
```

Příklad (mapování přes jeden seznam: procedura map1)

Umíme naprogramovat (opět neefektivní):

```
(define map1
  (lambda (f l)
    (build-list (length l)
                (lambda (i)
                  (f (list-ref l i))))))
```

Příklad (projekce datové tabulky – příklad použití map)

```
(define projection
  (lambda (table id-list)
    (map (lambda (row)
           (map (lambda (n)
                  (list-ref row n))
                id-list))
         table)))

(define mesta
  '((Olomouc 120 3)
    (Prostejov 50 2)
    (Praha 1200 8)))

(projection mesta '(0 2))
   ⟼  ((Olomouc 3) (Prostejov 2) (Praha 8))
```

Problém: páry uchovávající délku seznamu

Jak se počítá délka seznamu:

- `length` prochází seznamem od začátku do konce
- inkrementuje čítač
- pro zjištění délky seznamu jej celý projdeme – pomalé

Možnost urychlení:

- reimplementace páru
- každý pár v sobě obsahuje:
  - první složku – `car`
  - druhou složku – `cdr`
  - čítač – délka seznamu začínajícího aktuálním párem
- nové verze `cons`, `car` a `cdr`

Úskalí: konstruktivní / destruktivní manipulace se seznamem (**!!**)

Příklad (Implementace párů uchovávajících délku seznamu)

```
(define cons
  (lambda (hlava telo)
    (define delka (+ 1 (fast-length telo)))
    (lambda (dispatch-f)
      (dispatch-f hlava telo delka))))

(define 1-ze-3 (lambda (x y z) x))
(define 2-ze-3 (lambda (x y z) y))
(define 3-ze-3 (lambda (x y z) z))

(define car (lambda (p) (p 1-ze-3)))
(define cdr (lambda (p) (p 2-ze-3)))

(define fast-length
  (lambda (p)
    (if (null? p) 0 (p 3-ze-3))))
```

Správa paměti

Příklad (nedosažitelnost párů v paměti)

```
(define r '(x y))
(define s (cons 100 r))
r                          ⟹   (x y)
s                          ⟹   (100 x y)
(define s #f)
r                          ⟹   (x y)
s                          ⟹   #f
```

- některé páry během života interpretu „přestanou být dosažitelné“
- abstraktní interpret neřeší – reálný interpret musí řešit

Algoritmus „Mark & Sweep“

Reálné interprety:

- nově vytvářené páry dávají na *haldu*
- podprogram: *garbage collector* periodicky uklízí haldu

Dvě fáze algoritmu:

Mark
- prochází se dosažitelné elementy
- začíná se v globálním prostředí
- páry, procedury + prostředí, označují se obsažené elementy

Sweep
- projdi haldu
- smaž vše, co nemá značku
- odstraň všechny značky (příprava pro příští Mark)

Různá rozšíření, . . . inkrementální varianty, a podobně.

Datové typy jazyka Scheme

Zatím známé elementy jazyka: *čísla*, *pravdivostní hodnoty*, *symboly*, *tečkové páry*, *prázdný seznam*, *procedury* (*primitivní procedury* a *uživatelsky definované procedury*); *nedefinovaná hodnota*; *seznam* (odvozený typ)

Predikáty pro detekci typů elementů:

`boolean?`	*pravdivostní hodnota,*
`list?`	*seznam,*
`null?`	*prázdný seznam,*
`number?`	*číslo,*
`pair?`	*tečkový pár,*
`procedure?`	*procedura* (*primitivní* nebo *uživatelská*),
`symbol?`	*symbol.*

Scheme: datové typy jsou disjunktní (až na seznamy)

- pokud `list?` vrací pro nějaký element pravdu, pak vrací pravdu i právě jeden z predikátů `pair?` nebo `null?`

Definice (predikát rovnosti `equal?`)

`(equal?` $\langle element_1 \rangle$ $\langle element_2 \rangle$`)` vrací `#t`, právě když platí:

- oba $\langle element_1 \rangle$ a $\langle element_2 \rangle$ jsou buď `#t` nebo `#f`;
- oba $\langle element_1 \rangle$ a $\langle element_2 \rangle$ jsou stejné symboly;
- oba $\langle element_1 \rangle$ a $\langle element_2 \rangle$ jsou čísla, jsou obě buď v přesné nebo v nepřesné reprezentaci a obě čísla jsou si numericky rovny, to jest aplikací `=` na $\langle element_1 \rangle$ a $\langle element_2 \rangle$ bychom získali `#t`;
- oba $\langle element_1 \rangle$ a $\langle element_2 \rangle$ jsou prázdné seznamy;
- oba $\langle element_1 \rangle$ a $\langle element_2 \rangle$ jsou nedefinované hodnoty;
- oba $\langle element_1 \rangle$ a $\langle element_2 \rangle$ jsou stejné primitivní procedury nebo stejné uživatelsky definované procedury, nebo stejné speciální formy;
- oba $\langle element_1 \rangle$ a $\langle element_2 \rangle$ jsou páry ve tvarech $\langle element_1 \rangle =$ `(`E_1 `.` F_1`)` a $\langle element_2 \rangle =$ `(`E_2 `.` F_2`)` a platí:
  - výsledek aplikace `equal?` na E_1 a E_2 je `#t`
  - výsledek aplikace `equal?` na F_1 a F_2 je `#t`;

a vrací `#f` ve všech ostatních případech.

Příklad (použití equal?)

```
(equal? 2 2.0)   ⟹ #f
(= 2 2.0)        ⟹ #t
(equal? 2 2)     ⟹ #t
(equal? 2.0 2.0) ⟹ #t

(equal? (list 1 (cons 'a 'b)) '(1 (a . b)))    ⟹ #t
(equal? (list 1.0 (cons 'a 'b)) '(1 (a . b)))  ⟹ #f

(equal? (append '(a b) '(1 2 3))
        (append '(a b 1) (list 2 3)))  ⟹ #t

(= '(1 2 3) (list 1 2 3))        ⟹ CHYBA
(equal? '(1 2 3) (list 1 2 3))   ⟹ #t
```

Typové systémy: síla typového systému

1. **silně typované jazyky**
   - pro každou operaci je přesně vymezen přípustný typ argumentů
   - provedení operace s jinými než povolenými argumenty selže
   - zástupci: C, Scheme, ML,... (většina PJ)
2. **slabě typované jazyky**
   - před provedením operace překladač/interpret provádí přetypování tak, aby bylo možné operaci provést
   - konverzní pravidla – možnost přetypování argumentů
   - zástupci: Perl
   - příklad: `print "10" + 20;` $\Longmapsto$ `30`
   - nebezpečí vzniku chyb (**!!**)

Typové systémy: typ proměnné × typ hodnoty

1 staticky typované jazyky

- datové typy je možné úplně určit statickou analýzou zdrojového kódu
- během analýzy má každá použitá proměnná identifikován svůj typ
- někdy je *potřeba typ deklarovat*
- mluvíme o **typu proměnné**
- zástupci: C, ML,... (většina PJ)
- výhoda: lze lépe optimalizovat překlad
- výhoda (na baterky): špatní programátoři (většina) to mají snazší

2 dynamicky typované jazyky

- datové typy lze určit až za běhu programu
- jedna proměnná (symbol) může nabývat hodnot různých typů
- mluvíme o **typu hodnoty**
- zástupci: Scheme, Perl, Python
- výhoda: flexibilita jazyka (+ nezbytnost při objektové orientaci)

Typové systémy: bezpečnost

1. **bezpečně typované jazyky**
   - výsledek operace (pokud je proveditelná), nesmí skončit havárií
   - nemožnost dostat výpočetní proces do nekonzistentního tvaru
   - zástupci: Scheme, ML,...
2. **nebezpečně typované jazyky**
   - opak bezpečnosti
   - zástupci: C, C++

```
int main (int argc, char **argv)
{
    int *a = 0;
    *a = 100;
    return 0;
}
```