Paradigmata programování 1

Tečkové páry, symbolická data a kvotování

Vilém Vychodil

Katedra informatiky, PřF, UP Olomouc

Přednáška 4

Přednáška 4: Přehled

1. Hierarchická data:
   - potřeba složených dat,
   - tečkové páry, konstruktory, selektory,
   - implementace tečkových párů.
2. Symbolická data a kvotování:
   - speciální forma `quote`,
   - symboly jako data.
3. Vytváření abstrakcí pomocí dat:
   - procedury vs. data,
   - role abstrakčních bariér.

Vytváření abstrakcí pomocí dat

Probrané prostředky vytváření abstrakcí:

- *pojmenování hodnot* (`define`, ...)
- *vytváření procedur* (`lambda`, prostředí, ...)

Reprezentace dat v jazyku Scheme

- *jednoduchá* (*atomická*) *data* (čísla, pravdivostní hodnoty, ...)
- **složená** (**hierarchická**) **data** – nový fenomén
  - neformálně: „data obsahující (jiná) data“
  - pracujeme s nimi nepřímo pomocí *konstruktorů* a *selektorů*

Konstruktory: vytváří složená data z jednodušších dat

Selektory: vrací jednotlivé složky složených dat

Příklad (Motivační příklad)

Napište proceduru pro výpočet kořenů kvadratické rovnice $ax^2 + bx + c = 0$ pomocí:

$$\frac{-b \pm \sqrt{D}}{2a}, \qquad \text{kde } D = b^2 - 4ac.$$

Představa řešení:

```
(define koreny
  (lambda (a b c)
    (let* ((diskr (- (* b b) (* 4 a c)))
           (koren1 (/ (+ (- b) (sqrt diskr)) (* 2 a)))
           (koren2 (/ (- (- b) (sqrt diskr)) (* 2 a))))
      teď bychom chtěli vrátit dvě hodnoty současně
     )))
```

Zbývá dořešit:

- jak vracet „dvě hodnoty současně“, . . .

Příklad (Nepříliš uspokojivé řešení)

Můžeme „obejít“ následovně:

```
(define koreny
  (lambda (a b c p)
    (let ((diskr ((- (* b b) (* 4 a c)))))
       (/ ((if p + -) (- b) (sqrt diskr)) 2 a))))
```

Role parametru `p`:

- příznak (pravdivostní hodnota)
- na základě příznaku se vrací první nebo druhý kořen (proč?)

Proč nám koncepčně vadí:

- *de facto* neřeší problém
- zbytečný (a nepřehledný) parametr navíc
- opakování výpočtu D

Příklad (Jiný motivační příklad – racionální aritmetika)

Předpokládejme, že potřebujeme implementovat sčítání racionálních čísel:

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + cb}{bd}.$$

Představa řešení:

```
(define r+
  (lambda (cit1 jmen1 cit2 jmen2)
            chceme předávat jen dva argumenty = dvě racionální čísla
    (let* ((cit (+ (* cit1 jmen2) (* cit2 jmen1)))
           (jmen (* jmen1 jmen2))))
      teď bychom chtěli vrátit dvě hodnoty současně: cit a jmen
    )))
```

Dva problémy: *vracení* dvou hodnot / *argumenty* by měly být *složené hodnoty*

Příklad (Práce (třeba) s geometrickými objekty . . .)

Uvažujme: objekty typu

- *bod*,
- *úsečka*,
- *kružnice*, . . .

Společné rysy:

- jsou *geometrické objekty*,
- mohou být reprezentovány jako složená data, . . .

Procedury pracující s daty tohoto typu:

- najdi průsečík(y) přímky a kružnice (různé výsledky: bod, dva body, nic)
- najdi průsečík(y) dvou přímek (různé výsledky: bod, úsečka, přímka, nic)

⋮

Tečkové páry

Tečkový pár – nový element jazyka

- zapouzdřuje v sobě dva elementy (dvě hodnoty)
- základní reprezentant složených dat v jazyku Scheme

Konstruktor páru: procedura `cons` (angl. *construct*)

- dva argumenty, *první* a *druhá složka* páru

Selektory páru: procedury `car` a `cdr`

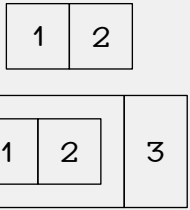
- `car` vrací první složku páru
- `cdr` vrací druhou složku páru

Poznámka (Původ jmen)

Původní implementace LISPu na počítači IBM 704.
„car“ = „Contents of Address part of Register“
„cdr“ = „Contents of Decrement part of Register“

Příklad (Použití konstruktorů a selektorů párů)

```
(define par1 (cons 1 2))
(define par2 (cons par1 3))
```




```
(car par1)  ⟼  1
(cdr par2)  ⟼  2

(car (car par2))  ⟼  1
(car (cdr par2))  ⟼  „CHYBA: Argument pro car není pár“
```

Chování `car` a `cdr`:

- argumenty musejí být vždy páry,
- v ostatních případech skončí aplikace selektorů chybou.

Příklad (Odvozené selektory párů)

```
(define caar (lambda (p) (car (car p))))
(define cadr (lambda (p) (car (cdr p))))
(define cdar (lambda (p) (cdr (car p))))
(define cddr (lambda (p) (cdr (cdr p))))
(define caaar (lambda (p) (car (caar p))))
  ⋮
(caar (cons (cons 10 20) (cons 30 40)))  ⟼  10
(cdar (cons (cons 10 20) (cons 30 40)))  ⟼  20
(cadr (cons (cons 10 20) (cons 30 40)))  ⟼  30
(cddr (cons (cons 10 20) (cons 30 40)))  ⟼  40
```

Poznámka:

- caar, cadr, . . . jsou předdefinované až po kombinace čtyř „a“ a „d“.

Externí reprezentace párů

Otázka: Jak printer vypisuje výsledek `(cons 1 2)`?

Tečkové pár skládající se ze složek $\langle A \rangle$ a $\langle B \rangle$ vypisuje reader jako $(\langle A \rangle \,.\, \langle B \rangle)$.

`(cons 1 2)` $\quad\Longmapsto\quad$ `(1 . 2)`

Zkracující konvence: v případě, že druhý prvek páru je opět pár:

- *vynechávají se závorky* náležející *vnitřnímu páru*
- *vynechává se tečka* náležející *vnějšímu.*

Úmluva (o jednoznačnosti externí reprezentace párů a seznamů)

Tečku „ . “ v externí reprezentaci párů **nepovažujeme za symbol**.

Důsledek: páry $(\langle A \rangle \;.\; \langle B \rangle)$ nejsou trojprvkové seznamy.

Příklad (Externí reprezentace párů ve zkrácené formě)

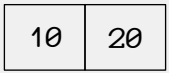
```
(cons 10 20)                ⟼  (10 . 20)
(cons (cons 10 20) 30)      ⟼  ((10 . 20) . 30)
(cons 10 (cons 20 30))      ⟼  (10 . (20 . 30)) = (10 20 . 30)

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

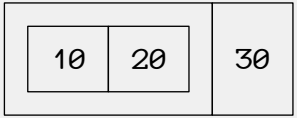
Příklad (Co nejsou externí reprezentace párů)

```
(10 . )     ( . 20)     (10 . 20 . 30)     (10 . 20 30)
```

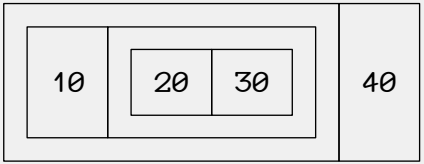
Příklad (Páry v boxové notaci)



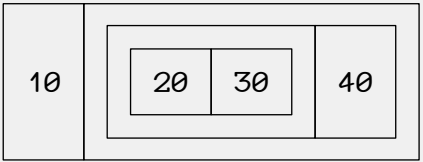
(10 . 20)



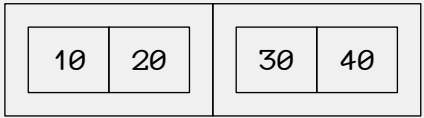
((10 . 20) . 30)



((10 20 . 30) . 40)



(10 (20 . 30) . 40)



((10 . 20) 30 . 40)

Páry jako elementy prvního řádu

Definice (element prvního řádu)

Element prvního řádu je každý element jazyka, pro který platí:

1. element může být *pojmenován*,
2. element může být *předán proceduře jako argument*,
3. element může *vzniknout aplikací* (*voláním*) *procedury*,
4. element může být *obsažen v hierarchických datových strukturách*.

Příklad (Procedura vracející pár; pár obsahující proceduru)

```
(define a (lambda () (cons a #f)))
```

`(a)`	$\Longmapsto$	(„procedura“ . `#f`)
`(car (a))`	$\Longmapsto$	„procedura“
`((car (a)))`	$\Longmapsto$	(„procedura“ . `#f`)
`(car ((car (a))))`	$\Longmapsto$	„procedura“

Rozšíření readeru

Definice (Rozšíření S-výrazů)

- Jsou-li $e, f, e_1, e_2, \ldots, e_n$ symbolické výrazy $(n \geq 0)$, pak

  $$(e \;\; e_1 \;\; e_2 \;\; \cdots \;\; e_n \;.\; f)$$

  je symbolický výraz. Pokud je $n = 0$, symbolický výraz $(e \;.\; f)$ nazveme **pár.**

Dva pojmy, nutno rozlišovat:

- pár jako **element jazyka** (hierarchická data),
- pár jako **symbolický výraz**.

Důsledky:

Element pár je interní reprezentace symbolického výrazu pár.

Externí reprezentace (pouze) některých elementů pár jsou symbolické výrazy (pár).

- viz předchozí příklad jako „protipříklad“

Implementace párů pomocí procedur vyšších řádů

Problém:

- *Lze reprezentovat hierarchická data i bez existence párů?*
- **Ano**: pomocí procedur vyšších řádů + lexikálního rozsahu platnosti

Příklad (Kořeny kvadratické rovnice bez použití párů)

```
(define koreny
  (lambda (a b c)
      (let* ((diskr (- (* b b) (* 4 a c)))
             (koren1 (/ (+ (- b) (sqrt diskr)) (* 2 a)))
             (koren2 (/ (- (- b) (sqrt diskr)) (* 2 a))))
        (lambda (prvni-nebo-druhy)
          (if prvni-nebo-druhy koren1 koren2)))))
```

- vrácená hodnota je **procedura** (jednoho argumentu), využití `let*`-o-λ

Příklad (Získání kořenů)

```
(define koreny
  (lambda (a b c)
      (let* ((diskr (- (* b b) (* 4 a c)))
             (koren1 (/ (+ (- b) (sqrt diskr)) (* 2 a)))
             (koren2 (/ (- (- b) (sqrt diskr)) (* 2 a))))
        (lambda (prvni-nebo-druhy)
          (if prvni-nebo-druhy koren1 koren2)))))

(koreny 1 -2 2)  ⟹  procedura

(define oba-koreny (koreny 1 -2 2))

(oba-koreny #t)  ⟹  1+i
(oba-koreny #f)  ⟹  1-i
```

Příklad (Aplikace postupu pro vytvoření reprezentace párů)

```
(define cons
  (lambda (x y)
     (lambda (k)
        (if k x y))))
```

```
((cons 1 2) #t)  ⟼  1
((cons 1 2) #f)  ⟼  2
```

```
(define car (lambda (p) (p #t)))
(define cdr (lambda (p) (p #f)))
```

```
(define p (cons 2 3))
```

p ⟼ „procedura“

```
(car p)  ⟼  2
(cdr p)  ⟼  3
```

Definitivní implementace párů pomocí procedur vyšších řádů

Čistší řešení pomcí projekcí:

```
(define cons
  (lambda (x y)
     (lambda (proj)
       (proj x y))))

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

(define car (lambda (p) (p 1-z-2)))
(define cdr (lambda (p) (p 2-z-2)))
```

- procedury vyšších řádů + lexikální rozsah platnosti = hierarchická data
- *kde jsou data vlastně ukrytá?*
- *jak vypadaji prostředí behem aplikace?*

Speciální forma quote

Speciální forma quote vrací svůj argument v nevyhodnocené podobě:

Definice (speciální forma quote)

Speciální forma quote se používá s jedním argumentem

(quote $\langle arg \rangle$).

Výsledkem aplikace této speciální formy je přímo $\langle arg \rangle$ (bez vyhodnocení).

Příklad (Použití quote)

```
(quote 10)          ⟹  10
(quote yellow)      ⟹  yellow
(quote (a . b))     ⟹  (a . b)
```

Kvotování, zkrácená forma použití quote

Zkrácená forma: metasyntaktický znak apostrof „ ′ “

Příklad (Stejný efekt jako v předchozím příkladě)

```
'10          ⟼  10
'yellow      ⟼  yellow
'(a . b)     ⟼  (a . b)
```

Poznámka:

- je „zbytečné“ kvotovat čísla (obecně: elementy vyhodnocující se na sebe sama),
- `quote` nemůže být procedura,
- reader expanduje výrazy ′$\langle A \rangle$ na výrazy (`quote` $\langle A \rangle$)
- apostrof „ ′ “ *není symbol* stejně jako tečka „ . “
- *terminus technicus* **syntaktický cukr**

Příklad (Příklady kvotovaných výrazů)

Symbol × procedura

```
(define plus +)
(plus 1 2)   ⟼   3
plus         ⟼   „procedura sčítání čísel“
```

```
(define plus '+)
(plus 1 2)   ⟼   „CHYBA: + není procedura ani speciální forma.“
plus         ⟼   +
```

Symbol vyhodnocující se na sebe sama (často používané v Common LISPu)

```
(define blah 'blah)
blah  ⟼  blah
```

Vytváření abstrakčních bariér pomocí dat

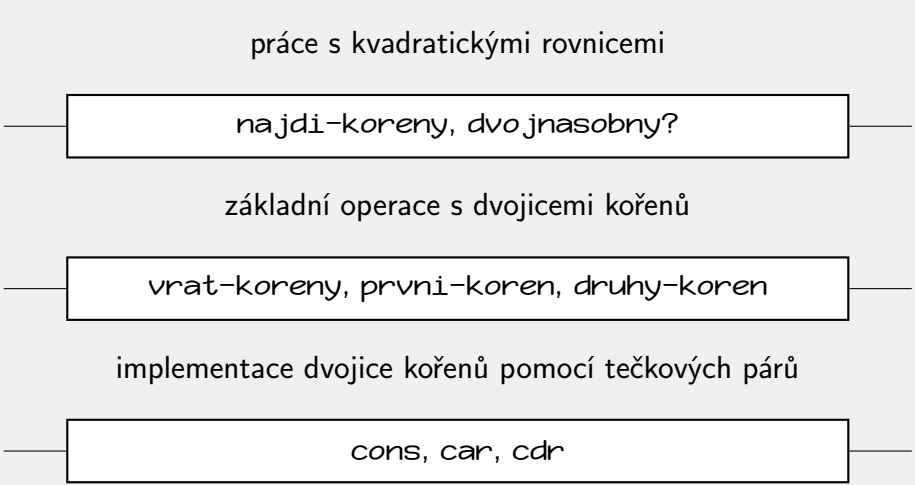
Připomenutí: **bottom-up** metoda vývoje programu:

- postupně obohacujeme jazyk přidáváním nových procedur (a hierarchických datových struktur),
- rozvrstvení do několika (nezávislých) vrstev (možnost reimplementace),
- snaha: vytvořit bohatý jazyk pro snadné vyřešení výchozího problému.

Pojmy spojené s metodou bottom-up a datovou abstrakcí:

- **černá skříňka** (black box)
  - vychází z toho, že *reprezentace dat* jsou implementovány po vrstvách,
  - z vyšší vrstvy se díváme na data v nižších vrstvách jako na černé skříňky,
  - „nezajímá“ nás *fyzická organizace dat* na nižších vrstvách,
  - zajímá nás pouze: jak vypadají *konstruktory* a *selektory* dat,
  - snadná možnost *záměny jedné vrstvy za druhou*.
- **abstrakční bariéra**
  - pomyslný mezník mezi dvěma vrstvami programu,

Příklad (Abstrakční bariéry v příkladu s kořeny kvadratické rovnice)



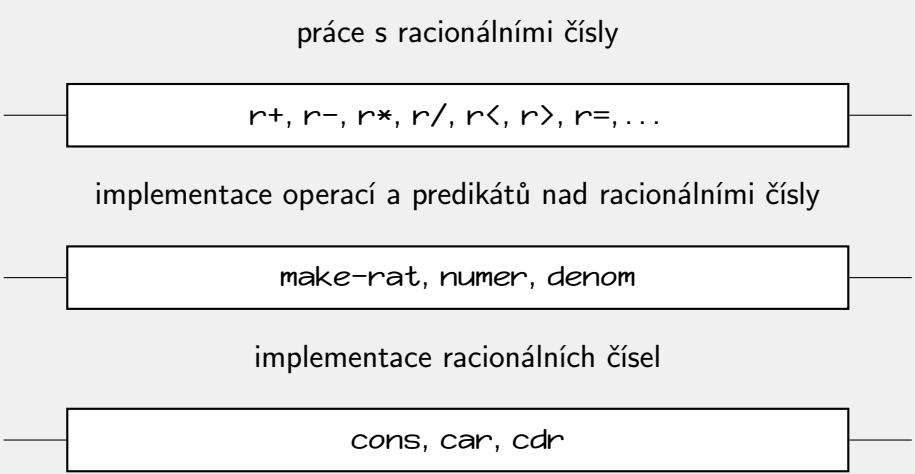
Příklad (Implementace bariér v příkladu s kořeny kvadratické rovnice)

```
(define vrat-koreny cons)
(define prvni-koren car)
(define druhy-koren cdr)

(define najdi-koreny
  (lambda (a b c)
     (let ((diskr (- (* b b) (* 4 a c))))
       (vrat-koreny (/ (- (- b) (sqrt diskr)) 2 a)
                    (/ (+ (- b) (sqrt diskr)) 2 a)))))

(define dvojnasobny?
  (lambda (koreny)
    (= (prvni-koren koreny)
       (druhy-koren koreny))))
```

Příklad (Abstrakční bariéry v příkladu s racionální aritmetikou)



Příklad (Fragment počáteční implementace)

```
(define make-r (lambda (x y) (cons x y)))

(define numer (lambda (x) (car x)))
(define denom (lambda (x) (cdr x)))

(define r+
  (lambda (x y)
    (make-r (+ (* (numer x) (denom y))
               (* (denom x) (numer y)))
            (* (denom x) (denom y)))))
```

$\vdots$

Příklad (Příklad reimplementace vrstvy)

```
(define make-r
  (lambda (x y)
    (let ((g (gcd x y)))
      (cons (/ x g) (/ y g)))))
```

Přidaná hodnota:

- kvalitativní změna programu malým zásahem do jedné vrstvy
- (patologicky) neovlivňuje procedury na dalších vrstvách