Paradigmata programování 1

Lokální vazby a definice

Vilém Vychodil

Katedra informatiky, PřF, UP Olomouc

Přednáška 3

Přednáška 3: Přehled

1. Lokální vazby:
   - motivační příklady – proč lokální vazby,
   - prostředky pro vytváření nových prostředí a vazeb,
   - speciální formy `let` a `let*`.
2. Lokální definice:
   - modifikace lokálního prostředí pomocí `define`,
   - rozšíření těla λ-výrazů, vytváření procedur a jejich aplikace,
   - interní definice procedur.
3. Nějaké další pojmy:
   - vytváření abstrakčních bariér (pomocí procedur),
   - metody vývoje softwaru: top-down a bottom-up.

Opakování

Prostředí (tabulky vazeb symbolů)

- **globální prostředí**
  - je pouze jedno,
  - nemá (lexikálního) předka,
  - vazby v něm jsou dány na počítku činnosti interpretu.
- **lokální prostředí**
  - je jich obecně mnoho,
  - každé z nich má svého (lexikálního) předka,
  - vazby v nich vznikají během aplikace procedur.

Vznik procedur

- procedury vznikají vyhodnocováním λ-výrazů
- procedura se skládá z: seznamu formálních argumentů, těla, prostředí vzniku

Aplikace procedur

- vznik prostředí, navázání parametrů na argumenty, vyhodnocení těla

Příklad (Motivační příklad)

Napište proceduru pro výpočet obsahu trojúhelníka využívající Heronův vzorec:

$$S_\Delta = \sqrt{s(s-a)(s-b)(s-c)}, \text{ kde } s = \frac{a+b+c}{2}.$$

Řešení:

```
(define heron
  (lambda (a b c)
    (sqrt (* (/ (+ a b c) 2)
             (- (/ (+ a b c) 2) a)
             (- (/ (+ a b c) 2) b)
             (- (/ (+ a b c) 2) c)))))
```

Odporné: obrovská redundance kódu + neefektivita. (**!!**)

Příklad (Rešení pomocí pomocné procedury I.)

$$S_\Delta = \sqrt{s(s-a)(s-b)(s-c)}, \text{ kde } s = \frac{a+b+c}{2}.$$

```
(define s-
  (lambda (a b c x)
    (- (/ (+ a b c) 2) x)))

(define heron
  (lambda (a b c)
    (sqrt (* (s- a b c 0)
             (s- a b c a)
             (s- a b c b)
             (s- a b c c)))))
```

Lepší, ale ne ideální (líp čitelné, redundance v podstatě zůstává).

Příklad (Rešení pomocí pomocné procedury II.)

$$S_\Delta = \sqrt{s(s-a)(s-b)(s-c)}, \text{ kde } s = \frac{a+b+c}{2}.$$

```
(define %heron
  (lambda (a b c s)
    (sqrt (* s (- s a) (- s b) (- s c)))))

(define heron
  (lambda (a b c)
    (%heron a b c (/ (+ a b c) 2))))
```

Prakticky ideální řešení:

- vyšší efektivita: při aplikaci `heron` se hodnota s počítá jen jednou
- přehledný kód, žádná redundance

Příklad (Rešení pomocí pomocné procedury III.)

$$S_\Delta = \sqrt{s(s-a)(s-b)(s-c)}, \text{ kde } s = \frac{a+b+c}{2}.$$

Modifikace předchozího:

```
(define heron
  (lambda (a b c)
    ((lambda (s)
       (sqrt (* s (- s a) (- s b) (- s c))))
     (/ (+ a b c) 2))))
```

Ještě lepší než předchozí:

- není zbytečně vytvořena pojmenovaná procedura `%heron`
- žádná redundance (srovnej s předchozím)
- ukázat: jak vypadají prostředí během aplikace

Motivace pro lokální vazby

Místo předchozího:

```
(define heron
  (lambda (a b c)
    ((lambda (s)
       (sqrt (* s (- s a) (- s b) (- s c))))
     (/ (+ a b c) 2))))
```

Přehledněji:

```
(define heron
  (lambda (a b c)
    (let ((s (/ (+ a b c) 2)))
      (sqrt (* s (- s a) (- s b) (- s c))))))
```

$\Longleftarrow$ lokální vazba symbolu `s`

Definice (Syntaxe speciální formy `let`)

Speciální forma `let` se používá ve tvaru:

$$\begin{array}{l}\texttt{(let ((}\langle symbol_1\rangle\ \langle hodnota_1\rangle\texttt{)}\\ \qquad\texttt{(}\langle symbol_2\rangle\ \langle hodnota_2\rangle\texttt{)}\\ \qquad\quad\vdots\qquad\quad\vdots\\ \qquad\texttt{(}\langle symbol_n\rangle\ \langle hodnota_n\rangle\texttt{)))}\\ \quad\langle tělo\rangle\texttt{)},\end{array}$$

kde

- n je nezáporné celé číslo,
- $\langle symbol_1\rangle$, $\langle symbol_2\rangle$, ..., $\langle symbol_n\rangle$ jsou vzájemně různé symboly,
- $\langle hodnota_1\rangle$, $\langle hodnota_2\rangle$, ..., $\langle hodnota_n\rangle$ a $\langle tělo\rangle$ jsou libovolné S-výrazy.

Terminologie: `let`-blok (nebo `let`-výraz)

Sémantika `let`-bloku

Vyhodnocení

$$\begin{array}{l}\texttt{(let ((}\langle symbol_1\rangle\ \ \langle hodnota_1\rangle\texttt{)}\\ \qquad\texttt{(}\langle symbol_2\rangle\ \ \langle hodnota_2\rangle\texttt{)}\\ \qquad\quad\vdots\qquad\quad\vdots\\ \qquad\texttt{(}\langle symbol_n\rangle\ \ \langle hodnota_n\rangle\texttt{)))}\\ \quad\langle tělo\rangle\texttt{)}\end{array}$$

je ekvivalentní vyhodnocení

$$\begin{array}{l}\texttt{((lambda (}\langle symbol_1\rangle\ \ \langle symbol_2\rangle\cdots\langle symbol_n\rangle\texttt{)}\\ \quad\langle tělo\rangle\texttt{)}\\ \quad\langle hodnota_1\rangle\ \ \langle hodnota_2\rangle\cdots\langle hodnota_n\rangle\texttt{)}\end{array}$$

Důsledek: vyhodnocení

(let ((⟨$symbol_1$⟩ ⟨$hodnota_1$⟩)
(⟨$symbol_2$⟩ ⟨$hodnota_2$⟩)
⋮ ⋮
(⟨$symbol_n$⟩ ⟨$hodnota_n$⟩))
⟨$tělo$⟩)

probíhá následovně:

1. S-výrazy ⟨$hodnota_1$⟩,...,⟨$hodnota_n$⟩ jsou vyhodnoceny v aktuálním prostředí $\mathcal{P}$ v nespecifickém pořadí. Výsledky jejich vyhodnocení označme E_i.
2. Vytvoří se nové prázdné prostředí $\mathcal{P}_l$.
3. Předek prostředí $\mathcal{P}_l$ je nastaven na $\mathcal{P}$.
4. V prostředí $\mathcal{P}_l$ se zavedou vazby ⟨$symbol_i$⟩ $\mapsto E_i$.
5. Výsledek vyhodnocení je pak roven $\mathrm{Eval}[\langle tělo \rangle, \mathcal{P}_l]$.

Poznámky k `let`-bloku

Syntax vs. sémantika

- `let`-blok je seznam (ve speciálním tvaru),
- význam `let`-bloku je dán přepisem pomocí λ-výrazů.

Speciální forma:

- `let` je speciální forma, nemůže být procedura
- protipříklad: `(let ((x 10)) (* x x))`

Pozor:

- `let` není „příklaz přiřazení“,
- `let` definuje nové vazby v novém prostředí (**!!**),
- vzniklé vazby mohou **překrýt vazby v nadřazeném prostředí**.

Příklad (Překrývání vazeb)

Důsledky způsobu zavedení `let`:

```
(define x 100)

(let ((x 200)
      (y 500))
   (* x y 10))
```

Fenomén 1:

- **překrytí vazby** symbolu `x`,
- v těle `let`-bloku není globální vazba `x` dosažitelná.

Fenomén 2:

- **zapouzdření vazby** symbolu `y`,
- lokální vazba symbolu `y` není vidět z vnějšku `let`-bloku.

Příklad (Souběžné vyhodnocování)

Výrazy definující hodnoty vazeb se vyhodnocují souběžně:

```
(define x 10)

(let ((x (+ x 1))
      (y (+ x 2)))
  y)  ⟼  12
```

Další příklad:

```
(define x 100)

(let ((x 10)
      (y (* 2 x)))
  (+ x y))  ⟼  210
```

Příklad (Záměna vazeb)

Použití `let` pro záměnu vazeb dvou symbolů:

```
(let ((+ *)
      (* +))
  (+ (* 10 10) 20))  ⟹  400
```

Předchozí je elegantnější řešení než:

```
(define plus +)
(define + *)
(define * plus)
```

Důvody:

- druhé řešení modifikuje globální/aktuální prostředí,
- zbytečná vazba pomocného symbolu.

Programátorský trik „let over lambda“

Jak vypadá:

- procedura je vytvořena v lokálním prostředí
- obvykle výsledek aplikace jiné procedury

$$
\begin{array}{l}
\texttt{(let (}\texttt{(}\langle symbol_1\rangle\ \langle hodnota_1\rangle\texttt{)} \\
\qquad\texttt{(}\langle symbol_2\rangle\ \langle hodnota_2\rangle\texttt{)} \\
\qquad\quad\vdots\qquad\quad\vdots \\
\qquad\texttt{(}\langle symbol_n\rangle\ \langle hodnota_n\rangle\texttt{))} \\
\texttt{(lambda (}\langle param_1\rangle\ \langle param_2\rangle\cdots\langle param_k\rangle\texttt{)} \\
\quad\langle tělo\rangle\texttt{))}
\end{array}
$$

Hoyte D.: *Let Over Lambda*, ISBN 978–1435712751 ($29 GBP).
Amazon: L-O-L is one of the most hardcore computer programming books out there.

Příklad (Jednoduché použití „let over lambda“)

Porovnání absolutních hodnot:

```
(define <=
  (let ((<= <=))
    (lambda (x y)
      (<= (abs x) (abs y)))))
```

```
(define <=
  (let ((f <=))
    (lambda (x y)
      (f (abs x) (abs y)))))
```

Jak vypadají prostředí?

Nesprávné řešení:

```
(define <=
  (lambda (x y)
    (<= (abs x) (abs y))))
```

Příklad (Poloha bodu a přímky v rovině: pokročilé „let over lambda“)

```
(define intersection
  (lambda (x1 y1 x2 y2)
    (let ((a (- y2 y1))
          (b (- x1 x2))
          (c (- (* x2 y1) (* x1 y2))))
      (lambda (x y delta)
        (<= (abs (+ (* a x) (* b y) c))
            delta)))))

(define moje (intersection 30 10 -30 -20))
(moje 10 0 1/100)  ⟹  #t
(moje 0 -5 1/100)  ⟹  #t
(moje 5 0 1/100)   ⟹  #f
```

Vnořování `let`-bloků

Vnořené `let`-bloky:

```
(let ((x 10))
  (let ((y (* x x)))
    (let ((z (- y x)))
      (/ z y))))  ⟹  9/10
```

Ekvivalentně pomocí nové speciální formy `let*`:

```
(let* ((x 10)
       (y (* x x))
       (z (- y x)))
  (/ z y))  ⟹  9/10
```

Definice (Syntaxe speciální formy `let*`)

Speciální forma `let*` se používá ve tvaru:

$$\begin{array}{l}\texttt{(let* ((}\langle symbol_1\rangle\ \langle hodnota_1\rangle\texttt{)}\\ \qquad\texttt{(}\langle symbol_2\rangle\ \langle hodnota_2\rangle\texttt{)}\\ \qquad\quad\vdots\qquad\quad\vdots\\ \qquad\texttt{(}\langle symbol_n\rangle\ \langle hodnota_n\rangle\texttt{))}\\ \quad\langle tělo\rangle\texttt{)},\end{array}$$

kde

- n je nezáporné celé číslo,
- $\langle symbol_1\rangle$, $\langle symbol_2\rangle$, ..., $\langle symbol_n\rangle$ jsou symboly (**nyní mohou být stejné**),
- $\langle hodnota_1\rangle$, $\langle hodnota_2\rangle$, ..., $\langle hodnota_n\rangle$ a $\langle tělo\rangle$ jsou libovolné S-výrazy.

Terminologie: `let*`-blok (nebo `let*`-výraz)

Sémantika `let*`-bloku

Vyhodnocení

(let* ((⟨$symbol_1$⟩ ⟨$hodnota_1$⟩)
(⟨$symbol_2$⟩ ⟨$hodnota_2$⟩)
⋮ ⋮
(⟨$symbol_n$⟩ ⟨$hodnota_n$⟩))
⟨$tělo$⟩)

definujeme následovně:

- Jestliže $n = 0$ nebo $n = 1$, je vyhodnocení stejné, jako u speciální formy `let`.
- Jinak je vyhodnocení stejné jako vyhodnocení následujícího výrazu:

(let ((⟨$symbol_1$⟩ ⟨$hodnota_1$⟩))
(let* ((⟨$symbol_2$⟩ ⟨$hodnota_2$⟩))
⋮
((⟨$symbol_n$⟩ ⟨$hodnota_n$⟩))
⟨$tělo$⟩))

Příklad (Vyjádření `let*`-bloku pomocí λ-výrazů)

`let*`-blok

```
(let* ((x 10)
       (y (* x x))
       (z (- y x)))
  (/ z y))  ⟹  9/10
```

je ekvivalentní

```
((lambda (x)
   ((lambda (y)
      ((lambda (z)
         (/ z y))
       (- y x)))
    (* x x)))
 10)  ⟹  9/10
```

Příklad (Použití symoblů se stejnými jmény)

```
(let* ((x 10)          ⟵ x má hodnotu 10
       (x (* 2 x))     ⟵ x má hodnotu 20
       (x (* 2 x))     ⟵ x má hodnotu 40
       (x (* 2 x)))    ⟵ x má hodnotu 80
   x)  ⟹  80

(let ((x 10)
      (x (* 2 x))
      (x (* 2 x))
      (x (* 2 x)))
   x)  ⟹ „CHYBA: Jména vazeb v let-bloku musejí být různá“
```

Modifikace lokálního prostředí

Co mají `let` a `let*` společné:

- je vždy vytvořeno (aspoň jedno) nové prostředí,
- aktuální prostředí není změněno,
- dochází pouze k překrytí vazeb.

Přímá modifikace (lokálního) prostředí:

- použití `define` v globálním prostředí (známe),
- použití `define` v lokálním prostředí aplikace procedury:
  - Je možné provést?
  - Ano, pro snazší práci navíc rozšíříme λ-výrazy.

Příklad (Použití `define` v těle procedury)

Použití `define` bez významu:

```
(lambda (x) (define y 20))
```

Využití speciální formy `and`:

```
(lambda (x)
  (and (define y (- x 2))
       (define z (/ x y))
       (+ x y z)))
```

Poznámky:

- při aplikaci vznikne pouze jedno prostředí se třemi vazbami
- využívá faktu, že „nedefinované hodnota“ je „pravda“ a vyhodnocení `and`

Syntaxe λ-výrazů (s rozšířeným chápáním těla)

Definice (λ-výraz)

Každý seznam ve tvaru

`(lambda (`$\langle param_1 \rangle \langle param_2 \rangle \cdots \langle param_n \rangle$`)` $\langle výraz_1 \rangle \langle výraz_2 \rangle \cdots \langle výraz_m \rangle$`)`, kde

- n je nezáporné číslo,
- m je kladné číslo,
- $\langle param_1 \rangle, \langle param_2 \rangle, \ldots, \langle param_n \rangle$ jsou vzájemně různé symboly,
- $\langle výraz_1 \rangle, \langle výraz_2 \rangle, \ldots, \langle výraz_m \rangle$ jsou **symbolické výrazy, tvořící tělo**,

se nazývá **λ-výraz**.

- tělo obsahuje víc S-výrazů, jinak vše při starém

Sémantika λ-výrazů (s rozšířeným chápáním těla)

Rozšíření λ-výrazů a vznik procedur:

Procedura vzniklá vyhodnocením:

(lambda $(\langle param_1 \rangle \langle param_2 \rangle \cdots \langle param_n \rangle)$ $\langle výraz_1 \rangle \langle výraz_2 \rangle \cdots \langle výraz_m \rangle$)

v aktuálním prostředí $\mathcal{P}$ je následující:

$\langle (\langle param_1 \rangle \langle param_2 \rangle \cdots \langle param_n \rangle); \langle výraz_1 \rangle, \langle výraz_2 \rangle, \ldots, \langle výraz_m \rangle; \mathcal{P} \rangle$

Aplikace vzniklé procedury:

- jako doposud, vznikne nové lokální prostředí $\mathcal{P}_l$
- předek $\mathcal{P}_l$ je nastaven na $\mathcal{P}$
- vazby v $\mathcal{P}_l$ odpovídají parametrům navázaným na argumenty
- v $\mathcal{P}_l$ se postupně vyhodnotí $\langle výraz_1 \rangle, \langle výraz_2 \rangle, \ldots, \langle výraz_m \rangle$
- **výsledek aplikace:** hodnota $\langle výraz_m \rangle$ (předchozí zapomenuty; vedlejší efekt)

Příklad (Definice nových vazev v lokálním prostředí)

Obsah trojúhelníka:

$$S_\Delta = \sqrt{s(s-a)(s-b)(s-c)}, \text{ kde } s = \frac{a+b+c}{2}.$$

```
(define heron
  (lambda (a b c)
    (define s (/ (+ a b c) 2))
    (sqrt (* s (- s a) (- s b) (- s c)))))
```

Vytvoření vazeb v prázdném lokálním prostředí:

```
(let ()
  (define x (lambda () y))
  (define y 10)
  (x))
```
$\quad \Longmapsto \quad$ `10` ... Proč funguje?

Interní definice (procedur)

Interní definice = *lokální navázání procedry na symbol*

- Scheme od počátku koncipováno tak, aby byly interní definice možné
- Algol 60 – první PJ s interními definicemi (procedura v proceduře)
- například C nebo C++ neumožňují (GNU C umožňuje, nestandardní)

Schematicky:

```
(define f
  (lambda (···)

    (define g          ⟵ procedura navázaná na g je vytvořena jako interní
      (lambda (···)
        ···))

    (··· (g ···) ···)))
```

Příklad (Funkce přibližné derivace ve Scheme)

Derivace funkce f v bodě x je číslo $f'(x)$ definované vztahem

$$f'(x) = \lim_{\delta \to 0} \left(\frac{f(x+\delta) - f(x)}{\delta} \right).$$

Význam:

- $f'(x)$ je směrnice tečny k f v bodě x (pokud je definované),
- $f' \colon S \to \mathbb{R}$ je funkce, kde $S \subseteq \mathbb{R}$.

Jak stanovit funkci f' na základě f?

- Numerická aproximace:
  - odstraníme limitní přechod,
  - stanovíme směrnici sečny pro dost malé $\delta > 0$.
- Ve Scheme: procedura vyššího řádu s argumenty f a δ vracející přibližnou f'.

Příklad (Funkce přibližné derivace: řešení pomocí interních definic)

```
(define derivace
  (lambda (f delta)

    (define smernice
      (lambda (a b)
        (/ (- (f b) (f a))
           (- b a))))

    (lambda (x)
      (smernice x (+ x delta)))))

(define f-deriv (derivace sqrt 0.001))
(f-deriv 0.01)   ⟼   4.8808848170152
(f-deriv 0.5)    ⟼   0.70675358090799
  ⋮
```

Metody vývoje programů

Problém vývoje velkého programu

- velké programové celky přestávají být přehledné,
- nutnost **vhodného strukturování** a organizace programu,
- fenomén, který se objevil s nástupem vyšších PJ.

Základní metody granularizace programových celků

1. **top-down** (postup typický pro procedurální jazyky)
   - pohled na program jako na „celek“,
   - rozdělení celku na menší části,
   - opětovné rozdělení na menší části, dokud není dozaženo požadované granularity.
2. **bottom-up** (postup typický pro Scheme a ostatní dialekty LISPu)
   - postupně obohacujeme jazyk přidáváním nových procedur,
   - rozvrstvení do několika (nezávislých) vrstev (možnost reimplementace),
   - snaha: vytvořit bohatý jazyk pro snadné vyřešení výchozího problému.

Černé skříňky a abstrakční bariéry

Pojmy spojené s metodou bottom-up:

- **černá skříňka** (black box)
  - vychází z toho, že procedury jsou implementovány po vrstvách,
  - z vyšší vrstvy se díváme na procedury v nižších vrstvách jako na černé skříňky,
  - „nezajímá“ nás implementace procedur na nižších vrstvách,
  - zajímá nás pouze: **vstupní argumenty** a co jsou **výsledky aplikace**,
  - snadná možnost *záměny jedné vrstvy za druhou*.
- **abstrakční bariéra**
  - pomyslný mezník mezi dvěma vrstvami programu,
  - není *absolutní pojem* – bariéry mohu vznikat/zanikat během vývoje programu,

Zatím není markantní, efekt uvidíme na 12. přednášce.