Paradigmata programování 1

Vytváření abstrakcí pomocí procedur

Vilém Vychodil

Katedra informatiky, PřF, UP Olomouc

Přednáška 2

Přednáška 2: Přehled

1. Uživatelsky definované procedury:
   - procedury a λ-výrazy,
   - speciální forma `lambda` a vznik procedur
   - aplikace procedur, prostředí a jejich hierarchie,
   - rozšíření vyhodnocovacího procesu,
   - lexikální a dynamický rozsah platnosti symbolů.
2. Procedury vyšších řádů:
   - procedury jako elementy prvního řádu,
   - procedury versus (matematické) funkce.
3. Jazyk Scheme:
   - další podmíněné výrazy, speciální formy `cond`, `and` a `or`.

Opakování

Syntax a sémantika jazyka

- **syntax** = tvar (jak se program zapisuje)
- **sémantika** = význam (co program znamená)

Jazyk Scheme

- program = posloupnost S-výrazů,
- interpretace programu – daná popisem vyhodnocování elementů jazyka
- během vyhodnocování dochází k *postupné aplikaci procedur*

Abstrakce pojmenováním hodnot

- definice vazeb symbolů v počátečním prostředí
- nutné zavést *speciální formy*
- vyšší míra abstrakce = vyšší programátorský komfort

Nový typ abstrakce: vytváření nových procedur

Motivace

- redundance kódu – snaha eliminovat (čistota kódu / efektivita),
- vytvoření nové procedury (musíme zajistit),
- pojmenování nové procedury (známe `define`).

Příklad (Výpočet délky přepony)

Místo:

```
(sqrt (+ (* 3 3) (* 4 4)))  ⟼  5
```

Chceme:

```
(prepona 3 4)  ⟼  5
```

λ-výrazy a vznik procedur

Vytvoření procedury:

```
(lambda (x) (* x x))  ⟼  #<procedure 8221f10>
```

„procedura, která pro dané x vrací násobek x s x“

Přímá aplikace procedury:

```
((lambda (x) (* x x)) 8)  ⟼  64
```

Pojmenování a následná aplikace procedury:

```
(define na2
  (lambda (x) (* x x)))

(na2 2)          ⟼  4
(+ 1 (na2 4))    ⟼  17
(na2 (na2 8))    ⟼  4096
```

Mantra k zapamatování: procedury vznikají vyhodnocením λ-výrazů (!!)

Syntaxe λ-výrazů

Definice (λ-výraz)

Každý seznam ve tvaru

`(lambda (`$\langle param_1 \rangle\ \langle param_2 \rangle \cdots \langle param_n \rangle$`) `$\langle tělo \rangle$`)`, kde

- n je nezáporné číslo,
- $\langle param_1 \rangle, \langle param_2 \rangle, \ldots, \langle param_n \rangle$ jsou vzájemně různé symboly,
- $\langle tělo \rangle$ je libovolný symbolický výraz,

se nazývá **λ-výraz**.

- λ-výraz je seznam (ve speciálním tvaru),
- symboly $\langle param_1 \rangle, \ldots, \langle param_n \rangle$ jsou **formální argumenty** (**parametry**),
- n je *počet formálních argumentů* (*parametrů*)

Poznámky k λ-výrazům

(lambda (⟨$param_1$⟩ ⟨$param_2$⟩ ⋯ ⟨$param_n$⟩) ⟨$tělo$⟩)

Formální argumenty:

- účel = *pojmenování hodnot* (se kterými je vytvořená proc. aplikována),
- připouští se i *nula formálních argumentů* (prázdný seznam **!!**).

Tělo procedury:

- představuje „předpis procedury“,
- je vykonáváno při aplikaci procedury,
- (obvykle) obsahuje symboly ⟨$param_i$⟩.

Vyhodnocení λ-výrazů

- lambda nemůže být procedura,
- lambda je speciální forma, nutno specifikovat, jak se vyhodnocuje (**!!**)

Volné a vázané výskyty symbolů

Symboly vyskytující se v těle λ-výrazu dělíme na:

1. **vázané symboly** – symboly, které jsou formálními argumenty λ-výrazu
2. **volné symboly** – všechny ostatní symboly v těle λ-výrazu

Příklad

```
(lambda (x y nepouzity)
  (* (+ 1 x) y))
```

- nemají výskyt v těle: `nepouzity`
- vázané: `x`, `y`
- volné: `+`, `*`

- volné a vázané symboly – různé role
- konzistentní přejmenování vázaných symbolů nemění význam λ-výrazu

Zjednodušený model aplikace procedur

Definice (aplikace uživatelsky definovaných procedur)

Při aplikaci procedury vzniklé vyhodnocením λ-výrazu

$$\texttt{(lambda (}\langle param_1\rangle\ \langle param_2\rangle \cdots \langle param_n\rangle\texttt{)}\ \langle tělo\rangle\texttt{)}$$

dojde k vytvoření **lokálního prostředí,** ve kterém:

- na symboly formálních argumentů $\langle param_1\rangle, \ldots, \langle param_n\rangle$ jsou **navázány hodnoty**, se kterými je procedura *aplikována*
- požadavek: stejný počet argumentů a formálních argumentů
- v lokálním prostředí je *vyhodnoceno* $\langle tělo\rangle$ *procedury*
- během vyhodnocení těla procedury se hledají vazby symbolů následovně:
  - vazby **vázaných symbolů** se hledají v **lokálním prostředí**,
  - vazby **volných symbolů** se hledají **v počátečním prostředí**.
- výsledek aplikace procedury = hodnota vzniklá jako výsledek vyhodnocení těla

Příklad (Příklady uživatelsky definovaných procedur)

```
(define na3 (lambda (x) (* x (na2 x))))
(define na4 (lambda (x) (na2 (na2 x))))
(define na5 (lambda (x) (* (na2 x) (na3 x))))
(define na6 (lambda (x) (na2 (na3 x))))
  ⋮

(define abs
  (lambda (x)
    (if (>= x 0)
        x
        (- x))))
```

Příklad (Identita)

```
((lambda (x) x) 20)  ⟼  20

(define id
  (lambda (x) x))

(id (* 2 20))        ⟼  40
(id (+ 1 (id 20)))   ⟼  21
(id #f)              ⟼  #f
((id -) 10 20)       ⟼  -10
```

Příklad (Projekce)

```
(lambda (x y z) x)  ⟼  první projekce
(lambda (x y z) y)  ⟼  druhá projekce
(lambda (x y z) z)  ⟼  třetí projekce
```

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

```
(1-z-3 10 20 30)                          ⟼  10
(2-z-3 10 (+ 1 (3-z-3 2 4 6)) 20)         ⟼  7
((3-z-3 #f - +) 13)                       ⟼  13
((2-z-3 1-z-3 2-z-3 3-z-3) 10 20 30)      ⟼  20
```

Příklad (Konstantní procedury a procedury bez argumentu)

```
(lambda (x) 10)   ⟹   procedura „vrať číslo 10“
(lambda (x) #f)   ⟹   procedura: „vrať pravdivostní hodnotu #f“
(lambda (x) +)    ⟹   procedura: „vrať hodnotu navázanou na symbol +“
  ⋮
(define c (lambda (x) 10))

(+ 1 (c 20))                      ⟹   11
(((lambda (x) -) 10) 20 30)       ⟹   -10

(define const-proc (lambda (x) 10))
(define noarg-proc (lambda () 10))

(const-proc 20)   ⟹   10
(noarg-proc)      ⟹   10
```

Nutné rozšíření (abstraktního) interpretu Scheme

Aplikace uživatelsky definovaných procedur vyžaduje:

1. **obecný pojem prostředí**
   - prostředí již není jen jedno (globální / počáteční)
   - každé aplikace vyžaduje lokální prostředí pro uchování hodnot argumentů
2. **rozšíření Eval**
   - předchozí chápání vyhodnocování nestačí (vyhodnocení symbolu je problém)
   - vyhodnocování elementů musí být definováno relativně vzhledem k prostředí
3. **přesný popis sémantiky speciální formy `lambda`**
   - z čeho se skládá procedura vzniklá vyhodnocením λ-výrazu
4. **popis Apply pro uživatelsky definované procedury**
   - jak probíhá aplikace uživatelsky definované procedury

Prostředí a jejich hierarchie

Definice (prostředí)

Prostředí $\mathcal{P}$ obsahuje:

- tabulka vazeb mezi symboly a elementy (jako doposud),
- ukazatel na svého **předka** (ukazatel na jiné prostředí).

Výjimka: **globální** (**počíteční**) **prostředí** – nemá předka (pouze tabulka vazeb).

Značení

- $\mathcal{P}_1 \prec \mathcal{P}_2$ znamená: $\mathcal{P}_1$ je předkem prostředí $\mathcal{P}_2$
  (také: **$\mathcal{P}_1$ je nadřazeno $\mathcal{P}_2$**)
- $s \mapsto_{\mathcal{P}} E$ znamená: na symbol s je v prostředí $\mathcal{P}$ navázán element E
  (také: **E je aktuální vazba symbolu s v prostředí $\mathcal{P}$**)
- zkrácené značení: $s \mapsto E$ pokud je $\mathcal{P}$ patrné z kontextu

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*, pak $\mathrm{Eval}[E, \mathcal{P}] := E$.

(B) Pokud je E *symbol*, mohou nastat tři situace:

- (B.1) Pokud $E \mapsto_{\mathcal{P}} F$, pak $\mathrm{Eval}[E, \mathcal{P}] := F$.
- (B.2) Pokud E *nemá vazbu v* $\mathcal{P}$ a pokud $\mathcal{P}' \prec \mathcal{P}$, pak $\mathrm{Eval}[E, \mathcal{P}] := \mathrm{Eval}[E, \mathcal{P}']$.
- (B.e) Pokud E *nemá vazbu v* $\mathcal{P}$ a pokud $\mathcal{P} = \mathcal{P}_G$, pak „CHYBA: E nemá vazbu“.

(C) Pokud je E ve tvaru $(E_1\ \ E_2 \cdots E_n)$, pak $F_1 := \mathrm{Eval}[E_1, \mathcal{P}]$ a rozlišujeme:

- (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: první prvek seznamu E se nevyhodnotil na proceduru ani na speciální formu.“.

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

Anatomie uživatelsky definovaných procedur

Co jsou uživatelsky definované proedury:

- elementy jazyka Scheme,
- nemají čitelnou reprezentaci,
- skládají se z několika složek (komponent), které jsou potřeba pro jejich aplikaci.

Definice (uživatelsky definovaná procedura)

Každá trojice ve tvaru

$\langle\langle parametry\rangle, \langle tělo\rangle, \mathcal{P}\rangle$, kde

- $\langle parametry\rangle$ je **seznam formálních argumentů**,
- $\langle tělo\rangle$ je **libovolný element**,
- $\mathcal{P}$ je **prostředí,**

se nazývá **uživatelsky definovaná procedura.**

Sémantika speciální formy `lambda`

Definice (speciální forma `lambda`)

Při aplikaci speciální formy `lambda` vyvolané vyhodnocením λ-výrazu

$$\texttt{(lambda (}\langle param_1\rangle\ \langle param_2\rangle \cdots \langle param_n\rangle\texttt{)}\ \langle tělo\rangle\texttt{)}$$

v prostředí $\mathcal{P}$ vznikne procedura $\langle(\langle param_1\rangle\ \langle param_2\rangle \cdots \langle param_n\rangle), \langle tělo\rangle, \mathcal{P}\rangle$.

Poznámky:

- při aplikaci speciální formy `lambda` se nic nevyhodnocuje (**!!**)
- vytvoření nové procedury je „levná záležitost“
- procedura: parametry, tělo, **prostředí vzniku** (prostředí aplikace `lambda`)
- `lambda` je „překvapivě jednoduchá“

Aktuální prostředí = prostředí, v němž byla vyvolána aplikace speciální formy

Definice (aplikace procedury)

Mějme dánu proceduru E a nechť $E_1, \ldots, E_n$ jsou libovolné elementy jazyka.

Aplikace procedury E na argumenty $E_1, \ldots, E_n$ (v tomto pořadí), značená $\mathrm{Apply}[E, E_1, \ldots, E_n]$ je definována následovně:

- Pokud je E primitivní procedura, ...
- Pokud je E uživatelsky definovaná procedura ve tvaru $\langle (\langle param_1 \rangle \cdots \langle param_m \rangle), \langle tělo \rangle, \mathcal{P} \rangle$, pak:
  1. Pokud se $m \neq n$, pak „CHYBA: Chybný počet argumentů“.
  2. Vytvoří se *nové prázdné prostředí* $\mathcal{P}_l$, které nazýváme **lokální prostředí procedury.**
  3. Nastavíme *předka prostředí* $\mathcal{P}_l$ na hodnotu $\mathcal{P}$.
  4. V prostředí $\mathcal{P}_l$ se zavedou vazby $\langle param_i \rangle \mapsto E_i$ pro $i = 1, \ldots, n$.
  5. Položíme $\mathrm{Apply}[E, E_1, \ldots, E_n] := \mathrm{Eval}[\langle tělo \rangle, \mathcal{P}_l]$.

Příklad (Výpočet délky přepony)

```
(define na2
  (lambda (x)
    (* x x)))

(define soucet-ctvercu
  (lambda (a b)
    (+ (na2 a) (na2 b))))

(define prepona
  (lambda (odvesna-a odvesna-b)
    (sqrt (soucet-ctvercu odvesna-a odvesna-b))))

(prepona 3 4)
```

$\Longmapsto$ 5 (jak vypadají prostředí)

Procedury vyšších řádů

Pojmenované / anonymní procedury

- ve většině PJ vznikají procedury jako **pojmenované** (mají vždy jméno),
- ve Scheme vznikají jako **anonymní** (bezejmenné).

Procedury vyšších řádů (pojem pochází z matematické logiky)

- procedury, kterým předáváme jiné procedury jako argumenty
- procedury, které vracejí jiné procedury jako výsledky aplikace

Poznámka (nikoli okrajová):

- ve vetšině programocích jazyků jsou procedury vyšších řádů „černá magie“
- ve Scheme (a jiných funkcionálních jazycích) jsou „zadarmo“
  – všechny procedury ve Scheme jsou *de facto* procedury vyšších řádů

Příklad (Procedury jako argumenty)

```
(define infix
  (lambda (x operace y)
    (operace x y)))
```

```
(infix 10 + 20)                          ⟼  30
(infix 10 - (infix 2 + 5))               ⟼  3
(infix 10 (lambda (x y) x) 20)           ⟼  10
(infix 10 (lambda (x y) 66) 20)          ⟼  66

(infix 10 (lambda (x) 66) 20)            ⟼  CHYBA!
(infix 10 20 30)                         ⟼  CHYBA!
```

Příklad (Procedury jako návratové hodnoty)

```
(define curry+
  (lambda (c)
    (lambda (x)
      (+ x c))))

(define f (curry+ 10))
```

`f` $\longmapsto$ `#<PROCEDURE ...>`

`(f 20)` $\longmapsto$ `30` (jak vypadají prostředí)

- role symbolů: v těle `f` je `x` vázaný a `c` volný
- obecný princip rozkladu zvaný „currying“ (H. Curry)
- lze aplikovat na procedury jiné než `+`
- analogicky pro tři a více argumentů

Procedury (ve Scheme) $\times$ funkce (matematické)

Procedury ve Scheme (speciální elementy jazyka)

- mají vstupní argumenty, po aplikaci produkují výstupní hodnoty

Matematické funkce (zobrazení, více v kursu *Úvod do informatiky*)

- **zobrazení** z množin $A_1, \ldots, A_n$ do množiny B je relace $f \subseteq A_1 \times \cdots \times A_n \times B$, pro kterou platí: pro každé $a_1 \in A_1, \ldots, a_n \in A_n$ existuje právě jedno $b \in B$ tak, že $\langle a_1, \ldots, a_n, b \rangle \in f$.
- označujeme $f : A_1 \times \cdots \times A_n \to B$
- $f(a_1, \ldots, a_n) = b$ místo $\langle a_1, \ldots, a_n, b \rangle \in f$

Různé pojmy:

- některé procedury se nechovají jako zobrazení,
- některá zobrazení mohou být reprezentována procedurami,
- některá zobrazení nemohou být reprezentována procedurami.

Příklad (Procedury, které se nechovají jako matematické funkce)

Nekončící série aplikací:

```
(define f (lambda (x) (x x))
(f f)  ⟼  ...
```

Aplikace procedury vracející pro stejné argumenty různé hodnoty:

```
(random 5)  ⟼  3
(random 5)  ⟼  2
(random 5)  ⟼  1
(random 5)  ⟼  3
  ⋮
(define f (lambda (x) (random 5)))
(f 10)  ⟼  2
(f 10)  ⟼  3
  ⋮
```

Matematické funkce reprezentovatelné procedurami

Příklad

Funkce $f : \mathbb{R} \to \mathbb{R}$ dané předpisy

$$f(x) = x^2,\ f(x) = \frac{x+1}{2},\ f(x) = |x|, \ldots$$

lze reprezentovat procedurami vzniklými vyhodnocením:

```
(lambda (x) (* x x))
(lambda (x) (/ (+ x 1) 2))
(lambda (x) (if (>= x 0) x (- x)))
  ⋮
```

V informatice (a matematice) běžná praxe:

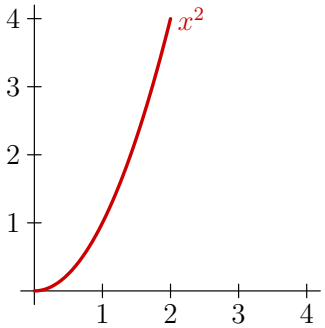
- funkce (procedury) se vyjadřují pomocí jiných funkcí (procedur) (např. posunutím, škálováním, ...)
- funkce (procedury) lze skládat, ...

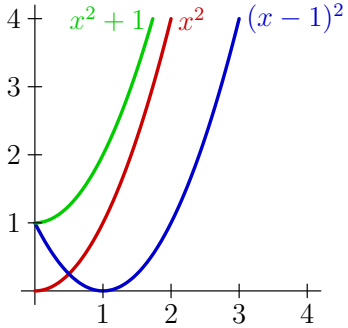
Funkce vzniklé posunem f po osách x a y

Definice (Funkce vzniklé posunutím)

Mějme funkci $f\colon \mathbb{R} \to \mathbb{R}$. Pak pro každé $k \in \mathbb{R}$, zavedeme funkci $f_{X,k}\colon \mathbb{R} \to \mathbb{R}$ a $f_{Y,k}\colon \mathbb{R} \to \mathbb{R}$ tak, že položíme

$$f_{X,k}(x) = f(x-k), \qquad f_{Y,k}(x) = f(x) + k.$$





Příklad (Vyjádření ve Scheme)

```
(define x-shift
  (lambda (f k)
    (lambda (x)
      (f (- x k)))))

(define y-shift
  (lambda (f k)
    (lambda (x)
      (+ k (f x)))))
```

```
(x-shift na2 1)       ⟹  „druhá mocnina posunutá o 1 na ose x“
(y-shift na2 1)       ⟹  „druhá mocnina posunutá o 1 na ose y“
((x-shift na2 1) 1)   ⟹  0
((x-shift na2 1) 2)   ⟹  1
```

Příklad (Procedura generující polynomické funkce)

Funkce $f : \mathbb{R} \to \mathbb{R}$ daná

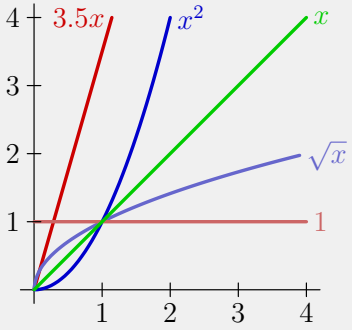
$$f(x) = a \cdot x^n \,,$$

kde $a \in \mathbb{R}$ a $n \in \mathbb{R}$.

Odpovídající procedura:

```
(define make-poly-f
  (lambda (a n)
    (lambda (x)
      (* a (expt x n)))))
```

```
(make-poly-f 3.5 1) ⟹ f(x) = 3.5x
(make-poly-f 1 2)   ⟹ f(x) = x^2
(make-poly-f 1 1)   ⟹ f(x) = x
(make-poly-f 1 1/2) ⟹ f(x) = √x
(make-poly-f 1 0)   ⟹ f(x) = 1
```



Příklad (Složení dvou funkcí / procedur)

Mějme funkce $f\colon X \to Y$ a $g\colon Y \to Z$. Pak složenou funkcí $(f \circ g)\colon X \to Z$ nazveme funkci, jejíž hodnota je definovaná předpisem

$$(f \circ g)(x) = g\big(f(x)\big) \quad \text{pro každé } x \in X.$$

Vlastnosti operace skládání:

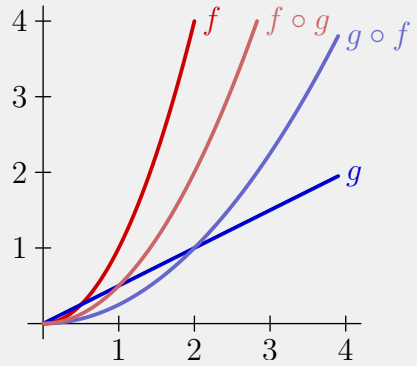
$$f \circ (g \circ h) = (f \circ g) \circ h, \qquad \text{asociativita,}$$
$$\iota \circ f = f \circ \iota = f, \qquad \text{neutralita vzhledem k } \iota(x) = x.$$

Ve Scheme:

```
(define compose2
  (lambda (f g)
    (lambda (x)
      (g (f x)))))
```

Příklad (Skládání procedur – příklady)

```
(define f na2)
(define g (make-poly-f 1/2 1))
(define f*g (compose2 f g))
(define g*f (compose2 g f))
```



$$\texttt{f*g} \quad \Longmapsto \quad (f \circ g)(x) = \frac{1}{2} \cdot x^2$$

$$\texttt{g*f} \quad \Longmapsto \quad (g \circ f)(x) = \left(\frac{x}{2}\right)^2$$

Funkcionální přístup

Výhodné: chápat procedury jako funkce

- snadné ladění (debugging / odbroukování)
- výsledky aplikace závisejí pouze na argumentech procedur (idealizace; pozor na symboly, které se vyskytují jako volné!)

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*.

Kouzlo Scheme: vše je (může být) element prvního řádu. (**!!**)

Lexikální a dynamický rozsah platnosti

Lexikální rozsah platnosti (symbolů / proměnných)

- vazby symbolů v těle procedury, jejichž vazby nejsou nalezeny v lokálním prostředí se hledají v **prostředí vzniku procedury**
- používá: většina programovacích jazyků včetně Scheme, C, Pascal, ...
- plusy: strukturu prostředí lze vyčíst z programu
- někdy se nazývá: **statický rozsah platnosti**

Dynamický rozsah platnosti (symbolů / proměnných)

- vazby symbolů v těle procedury, jejichž vazby nejsou nalezeny v lokálním prostředí se hledají v **prostředí aplikace procedury**
- používá: prakticky nikdo (FoxPro)
- minusy: vazby symbolů lze vyčíst až za běhu programu / obtížné ladění

Příklad (Statický vs. dynamický rozsah platnosti)

```
(define curry+
  (lambda (c)
    (lambda (x)
      (+ x c))))

(define c 100)

(define f (curry+ 10))
(f 20)                    ⟹  ???

((lambda (c)
   (f 20))
 1000)                    ⟹  ???
```

Scheme: negace pravdivostních hodnot

Příklad (negace zobecněných pravdivostních hodnot)

```
(not #t)                          ⟼  #f
(not #f)                          ⟼  #t
(not 0)                           ⟼  #f
(not -12.5)                       ⟼  #f
(not (lambda (x) (+ x 1)))        ⟼  #f
(not (<= 1 2))                    ⟼  #f
(not (> 1 3))                     ⟼  #t
```

Procedura `not` je definovatelná:

```
(define not
  (lambda (x)
    (if x #f #t)))
```

Scheme: vytváření podmínek pomocí konjunkce

Příklad (konjunkce zobecněných pravdivostních hodnot)

```
(and (= 0 0) (odd? 1) (even? 2))          ⟼  #t
(and (= 0 0) (odd? 1) (even? 2) 666)      ⟼  666
(and 1 #t 3 #t 4)                         ⟼  4
(and 10)                                  ⟼  10
(and +)                                   ⟼  „procedura sčítání“
(and)                                     ⟼  #t
(and (= 0 0) (odd? 2) (even? 2))          ⟼  #f
(and 1 2 #f 3 4 5)                        ⟼  #f
```

Pozor: and ve Scheme je speciální forma (**!!**)

Definovatelnost `and` pomocí `if`

(and $\langle test_1 \rangle \cdots \langle test_n \rangle$)

lze nahradit vnořenými `if`-výrazy:

```
(if ⟨test_1⟩
    (if ⟨test_2⟩
        (if ···
            (if ⟨test_{n-1}⟩
                ⟨test_n⟩
                #f)
              . . .
            #f)
        #f)
    #f)
```

Scheme: vytváření podmínek pomocí disjunkce

Příklad (konjunkce zobecněných pravdivostních hodnot)

```
(or (even? 1) (= 1 2) (odd? 1))   ⟼  #t
(or (= 1 2) (= 3 4) 666)          ⟼  666
(or 1 #f 2 #f 3 4)                ⟼  1
(or (+ 10 20))                    ⟼  30
(or)                              ⟼  #f
(or #f)                           ⟼  #f
(or #f (= 1 2) #f)                ⟼  #f
```

Pozor: *or* ve Scheme je speciální forma (**!!**)

Definovatelnost `or` pomocí `if`

(`or` $\langle test_1 \rangle \cdots \langle test_n \rangle$)

lze (*zatím bez újmy*) nahradit vnořenými `if`-výrazy:

```
(if ⟨test_1⟩
    ⟨test_1⟩
    (if ⟨test_2⟩
        ⟨test_2⟩
        (if ···
            (if ⟨test_n⟩
                ⟨test_n⟩
                #f) ··· )))
```

Příklad (příklady použití **and** a **or**)

```
(define within?
  (lambda (x a b)
    (and (>= x a) (<= x b))))

(define overlap?
  (lambda (a b c d)
    (or (within? a c d)
        (within? b c d)
        (within? c a b)
        (within? d a b))))
```

Definice (Speciální forma cond)

Speciální forma cond se používá ve tvaru:

(cond (⟨$test_1$⟩ ⟨$důsledek_1$⟩)
(⟨$test_2$⟩ ⟨$důsledek_2$⟩)
⋮
(⟨$test_n$⟩ ⟨$důsledek_n$⟩)
(else ⟨$náhradník$⟩)), kde $n \geq 0$ a náhradník je *nepovinný*

Aplikace cond probíhá:

- cond vyhodnocuje (v aktuálním prostředí) výrazy $\langle test_1 \rangle, \ldots, \langle test_n \rangle$ do toho okamžiku, až narazí na první $\langle test_i \rangle$, který se vyhodnotil na pravdu
- vyhodnocování dalších $\langle test_{i+1} \rangle, \ldots, \langle test_n \rangle$ se neprovádí a výsledkem je hodnota vzniklá vyhodnocením výrazu $\langle důsledek_i \rangle$ v aktuálním prostředí
- jinak: vyhodnotí se $\langle náhradník \rangle$ nebo je hodnota nedfinovaná

Příklad (Použití speciální formy cond)

Funkce signum:

$$\operatorname{sgn} x = \begin{cases} -1 & \text{pokud } x < 0, \\ 0 & \text{pokud } x = 0, \\ 1 & \text{pokud } x > 0. \end{cases}$$

Implementace ve Scheme:

```
(define sgn
  (lambda (x)
    (cond ((= x 0) 0)
          ((> x 0) 1)
          (else -1))))
```

Vztah cond a if

```
(cond (⟨test_1⟩ ⟨důsledek_1⟩)
      (⟨test_2⟩ ⟨důsledek_2⟩)
        ⋮
      (⟨test_n⟩ ⟨důsledek_n⟩)
      (else ⟨náhradník⟩))
```

$\Longmapsto$

```
(if ⟨test_1⟩
    ⟨důsledek_1⟩
    (if ⟨test_2⟩
        ⟨důsledek_2⟩
        (if ···
            (if ⟨test_n⟩
                ⟨důsledek_n⟩
                ⟨náhradník⟩) ··· )))
```

`if` pomocí `cond` ještě jednodušší . . .