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 2 ⋄ poznámky k přednášce**

# 2. Makra 1

verze z 19. května 2019

## 1 Meze procedurální abstrakce

V minulém semestru jsme se setkali s programovou abstrakcí vytvářenou pomocí funkcí (procedur). Takové abstrakci můžeme říkat *procedurální abstrakce*. Pokud si potřebujete zopakovat, co to programová abstrakce pomocí procedur je a jaké má výhody, podívejte se na druhou přednášku minulého semestru.

V této části si ukážeme jedno omezení procedurální abstrakce a jak toto omezení řeší Common Lisp pomocí tzv. maker. Makra pak budeme používat v dalších přednáškách jako nástroj na experimentování s jazykem.

### Problém

Chceme napsat funkci `test-string`, která otestuje pro zadaný řetězec zadanou sadu podmínek:

```
(defun test-string (string &key length< length> newlines
                                 no-newlines char char-not)
  (and (if length< (< (length string) length<) t)
       (if length> (> (length string) length>) t)
       (if newlines (find #\newline string) t)
       (if no-newlines (not (find #\newline string)) t)
       (if char (find char string) t)
       (if char-not (not (find char string)) t)))
```

Poznámky:

- Na textové řetězce fungují některé funkce na práci se seznamy, například `length` a `find`. Řetězce jsou speciální typ jednorozměrných polí, neboli *vektorů*. Vektory a seznamy se v Lispu souhrnně nazývají *posloupnosti*. Posloupnostem je věnována jedna kapitola v dokumentaci (CL HyperSpec) a obsahuje hodně funkcí, které pracují obecně s posloupnostmi.
- Znaky v Common Lispu se zadávají syntaxí `#\`. Např. `#\A` je velké písmeno `A`. Některé znaky se dají v LispWorks zadat i názvem, např. `#\space` nebo v příkladu použitý `#\newline`. To je samozřejmě výhodné zejména pro neviditelné znaky, jako třeba mezeru, kterou ovšem můžeme zadat i takto: `#\ `.

- Textový řetězec je jednorozměrné pole znaků:

```
CL-USER 1 > (aref "abc"1)
#\b

CL-USER 2 > (aref "ab c"2)
#\Space
```

### Testy:

```
CL-USER 1 > (test-string "abc" :length< 10 :length> 2)
T

CL-USER 2 > (test-string "abc" :length< 10 :length> 5)
NIL

CL-USER 3 > (test-string "abc" :length< 10 :no-newlines t)
T

CL-USER 4 > (test-string "a
bc" :length< 10 :no-newlines t)
NIL

CL-USER 5 > (test-string "abc" :length< 10 :char #\b)
T

CL-USER 6 > (test-string "abc" :length< 10 :char #\d)
NIL
```

Poznámka:

- Řetězec ve čtvrtém testu obsahuje jako svůj druhý znak (tj. znak s indexem 1) znak pro nový řádek:

```
CL-USER 3 > (aref "a
bc"1)
#\Newline
```

Problém s opakovaným kódem, který nelze řešit procedurální abstrakcí:

```
(defun test-string (string &key length< length> newlines
                                 no-newlines char char-not)
  (and (if length< (< (length string) length<) t)
```

```
        (if length> (> (length string) length>) t)
        (if newlines (find #\newline string) t)
        (if no-newlines (not (find #\newline string)) t)
        (if char (find char string) t)
        (if char-not (not (find char string)) t)))
```

Ve skutečnosti používáme **logickou implikaci**. Programátor při psaní funkce totiž uvažoval takto:

1. Aby řetězec prošel testem, musí splňovat *všechny* podmínky dané jednotlivými nepovinnými parametry, tedy podmínky pro parametry `length<`, `length>`, `newlines`, `no-newlines`, `char`, `char-not`.
2. Každá podmínka říká: „Jestliže uživatel zadal hodnotu parametru, pak řetězec musí mít vlastnost danou touto hodnotou“. (Pokud uživatel hodnotu nezadal, je podmínka automaticky splněna a nic se nemusí testovat.) Podmínka má tvar logické implikace.

Logickou implikaci ovšem nemáme v jazyce (máme jen konjunkci, disjunkci, negaci). Bylo by užitečné si ji naprogramovat, tj. udělat příslušnou abstrakci, tak aby šlo funkci napsat takto:

```
(defun test-string (string &key length< length> newlines
                                no-newlines char char-not)
  (and (impl length< (< (length string) length<))
       (impl length> (> (length string) length>))
       (impl newlines (find #\newline string))
       (impl no-newlines (not (find #\newline string)))
       (impl char (find char string))
       (impl char-not (not (find char string)))))
```

Tento tvar by měl tu výhodu, že **přesněji odráží způsob, jakým programátor o funkci uvažuje**. (To je ostatně podstatná vlastnost programovacích jazyků: zda umějí jednoduše vyjádřit programátorovy myšlenky.)

Lze implikaci naprogramovat jako funkci? Bylo by to jednoduché:

```
(defun impl (antecedent consequent)
  (if antecedent consequent t))
```

Ne, protože to by nepoužívala **zkrácené vyhodnocování**. Vyhodnocení výrazu `(impl a b)` by vedlo za všech okolností i k vyhodnocení podvýrazu `b`, což není u logických operací vhodné (logické operace ve většině programovacích jazyků nevyhodnocují nutně všechny své argumenty).

Jak ji naprogramovat jinak? V Common Lispu lze na řešení tohoto problému použít **makra**.

# 2 Jak pracují makra

*Makra* jsou nástroj na transformaci zdrojového kódu. V Lispu lze například napsat makro `impl`, které transformuje zdrojový kód takto:

```
(impl a b) ⟶ (if a b t)
```

Tomu se říká *expanze makra*. Transformovaný (expandovaný) výraz se pak znovu předloží vyhodnocovacímu procesu.

Některé operátory, o kterých jsme pro jednoduchost říkali, že jsou to speciální operátory, jsou ve skutečnosti makra. Například `and` a `or` by se mohly expandovat takto:

```
(and a b) ⟶ (if a b nil)
(or a b) ⟶ (if a a b)
(and a b c d) ⟶ (if a (and b c d) nil)
```

Jak se výrazy expandují, lze zjistit pomocí funkce `macroexpand-1`. Např. v LispWorks:

```
CL-USER 12 > (macroexpand-1 '(and a b c d))
(IF A (AND B C D) NIL)
T
```

(Druhou hodnotu `T` můžeme ignorovat.) Expanze maker můžeme zkoušet i ve vývojovém prostředí vyvoláním z menu.

Operátor `cond` je rovněž makro. Výraz

```
(cond ((< a 0) -1)
      ((= a 0) 0)
      (t 1))
```

by mohl expandovat na výraz

```
(if (< a 0)
    -1
  (if (= a 0)
      0
    1))
```

(Podívejte se, jak je to v LispWorks).

Operátor `if` ovšem není makro, je to opravdu speciální operátor.

Jak vidíme, vyhodnocení složeného výrazu, jehož operátorem je název makra, probíhá ve dvou krocích:

1. výraz se expanduje,
2. výsledek expanze se vyhodnotí.

**Úplný vyhodnocovací proces v Lispu** (včetně maker) tedy pracuje takto:

**Vyhodnocení výrazu $E$ v prostředí** *env*

**Je-li $E$ symbol,** výsledkem je hodnota symbolu $E$ v prostředí *env*.

**Je-li $E$ jiný atom než symbol,** výsledkem je $E$.

**Je-li $E$ seznam s operátorem $o$ a argumenty $a_1, \ldots, a_n$,** pak

> **Jestliže $o$ je speciální operátor,** seznam se vyhodnotí podle pravidel tohoto speciálního operátoru.
>
> **Jestliže $o$ je název makra,** $E$ se expanduje podle pravidel pro toto makro a výsledný výraz se vyhodnotí v prostředí *env*.
>
> **Jinak** $o$ musí být název funkce.
>
> 1. Vyhodnocením `(function` $o$`)` (neboli `#'`$o$) v prostředí *env* se zjistí funkce $f$,
> 2. zjistí se hodnoty $v_1, \ldots, v_n$ argumentů $a_1, \ldots, a_n$ v prostředí *env* (opět vyhodnocovacím procesem).
> 3. Výsledkem vyhodnocení je výsledek aplikace funkce $f$ na hodnoty $v_1$, $\ldots$, $v_n$.

# 3 Jak se definují makra

Napsat makro znamená popsat způsob, jakým se složené výrazy, jejichž operátorem je název makra, expandují. Zde můžeme těžit z toho, že složené výrazy jsou seznamy a expanzi popsat jako funkci v Lispu. Argumenty takové funkce by byly (nevyhodnocené!) argumenty expandovaného výrazu. Například expanze makra `impl` by mohla být vykonaná touto funkcí:

```
(defun impl-expansion (ant-expr cons-expr)
  (list 'if ant-expr cons-expr 't))
```

Definovat makro znamená stanovit jeho název a popsat jeho expanzní funkci. To se dělá pomocí operátoru `defmacro` (který je sám makrem, ale to není důležité). Například makro `impl` můžeme definovat takto:

```
(defmacro impl (ant-expr cons-expr)
  (list 'if ant-expr cons-expr 't))
```

Vidíme tedy, že definicí makra sdělujeme tři údaje:

1. název makra,
2. seznam parametrů expanzní funkce,
3. tělo expanzní funkce.

Po napsání makra bychom měli nejprve otestovat, že se správně expanduje (tj. otestovat jeho expanzní funkci). Například:

```
CL-USER 14 > (macroexpand-1 '(impl x y))
(IF X Y T)
T
```

(U složitějších maker samozřejmě uděláme testů víc.)

Když se ubezpečíme, že expanzní funkce funguje, otestujeme i samotný expandovaný výraz. U nás bychom třeba mohli otestovat funkci `test-string` napsanou pomocí nového makra.

# 4 Zpětný apostrof

Aby byl z definice makra zřejmější tvar expandovaného výrazu, můžeme použít syntax se zpětným apostrofem (*backquote*). Například místo

```
(list 'if a b 't)
```

můžeme napsat

```
`(if ,a ,b t)
```

Zpětný apostrof funguje podobně jako klasický apostrof, tj. potlačí vyhodnocování výrazu za ním. Vyhodnocení podvýrazů složených výrazů lze pak zařídit čárkou:

```
CL-USER 4 > `((+ 1 1) ,(+ 1 1))
((+ 1 1) 2)

CL-USER 5 > `((+ 1 (+ 1 1)) (+ 1 ,(+ 1 1)) ,(+ 1 (+ 1 1)))
((+ 1 (+ 1 1)) (+ 1 2) 3)
```

Čárku lze také použít v kombinaci se zavináčem. Dosáhneme tak rozpuštění seznamu vzniklého vyhodnocením podvýrazu:

```
CL-USER 13 > `((list 1 2) ,(list 1 2) ,@(list 1 2))
((LIST 1 2) (1 2) 1 2)
```

Pokud by v proměnné `rest` byl uložen seznam `(c d)`, pak dostaneme:

```
CL-USER 21 > `(a b ,@rest e)
(A B C D E)
```

### Příklad

V Lispu je k dispozici makro `when`, které nejprve otestuje danou podmínku a pokud je splněna, vyhodnotí postupně zadané výrazy a výsledek posledního vrátí. Není-li splněna, vrátí jen `nil`:

```
CL-USER 14 > (when (> 1 0) 'ano)
ANO

CL-USER 15 > (when (> 1 0) (print "Tisk") 'ano)

"Tisk"
ANO

CL-USER 16 > (when (< 1 0) (print "Tisk") 'ano)
NIL
```

Obdobu tohoto makra můžeme napsat takto:

```
(defmacro my-when (condition &rest expressions)
  `(if ,condition (progn ,@expressions)))
```

Test expanze:

```
CL-USER 19 > (macroexpand-1 '(my-when (> 1 0) (print "Tisk") 'ano))
(IF (> 1 0) (PROGN (PRINT "Tisk") (QUOTE ANO)))
```

# 5 Interpret Scheme s makry

Zdrojový kód k této přednášce obsahuje náš interpret Scheme s přidanými makry. Podrobnosti najdete tam.

# Otázky a úkoly na cvičení

Makra implementujte nejprve bez použití zpětného apostrofu a teprve potom s ním. Při testování vždy nejprve vyzkoušejte expanzi.

1. Definujte makra `and-2` a `or-2`, která implementují logickou konjunkci a disjunkci pro dva argumenty se zkráceným vyhodnocováním.

2. Napište makro `if-zero`, které má stejnou syntax jako `if`, ale místo podmínky očekává číslo. Pokud je číslo nula, vyhodnotí se první větev, jinak se vyhodnotí druhá:

```
CL-USER 32 > (if-zero (+ 1 1) (+ 1 2) (+ 1 3))
4

CL-USER 33 > (if-zero (- 1 1) (- 1 2) (- 1 3))
-1
```

3. Makro `unless` v Lispu pracuje stejně jako makro `when`, ale výrazy vyhodnotí, když daná podmínka není splněna:

```
CL-USER 14 > (unless (> 1 0) 'ano)
NIL

CL-USER 15 > (unless (> 1 0) (print "Tisk") 'ano)

NIL

CL-USER 16 > (unless (< 1 0) (print "Tisk") 'ano)

"Tisk"
ANO
```

   Napište svou verzi tohoto makra s názvem `my-unless`.

4. Napište následující verzi makra `when`: Makro `whenb` akceptuje navíc proti makru `when` jako svůj první argument symbol. Na něj bude v těle makra navázán výsledek vyhodnocení podmínky. Příklad:

```
CL-USER 4 > (setf l (list 1 2))
(1 2)

CL-USER 5 > (whenb x (second l) (+ x 1))
3

CL-USER 6 > (whenb x (third l) (+ x 1))
NIL
```

V definici makra můžete použít makro `when`.

5. Napište makro `reverse-progn`, které pracuje stejně jako speciální operátor `progn` (obdoba schemovského `begin`), ale výrazy vyhodnocuje v opačném pořadí:

```
CL-USER 35 > (reverse-progn 1 (print 2) (print 3) (print 4))

4
3
2
1
```

   Na obrácení seznamu můžete použít funkci `reverse`.

6. Zodpovězte otázky na konci zdrojového souboru interpretu Scheme s makry.