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

3. Makra 2

verze z 19. května 2019

Minule jsme zavedli makra a ukázali, jak fungují ve vyhodnocovacím procesu. Dnes se soustředíme na komplikovanější příklady a na problémy, na které programátor během práce s makry narazí.

1 λ-seznamy maker

Víme, že λ-seznamy funkcí mohou obsahovat mimo jiné klíčová slova `&rest`, `&key` a `&optional`. λ-seznamy maker mohou navíc obsahovat (mimo jiné) klíčové slovo `&body`.

Klíčové slovo `&body` má stejný význam, jako klíčové slovo `&rest`, ale indikuje, že seznam parametrů, který je jím uvozen, tvoří tělo, tedy výrazy, které se budou vyhodnocovat. To ovlivňuje automatické formátování výrazů s makrem, jak ukazuje **příklad**:

Při definici makra `my-when` z minulé přednášky (tedy s λ-seznamem `(condition &rest expressions)`) bude formátování (např. pomocí tabelátoru) vypadat takto:

```
(my-when x
         (print 'jedna)
         'dvě)
```

Pokud použijeme λ-seznam `(condition &body expressions)`), bude formátování lepší:

```
(my-when x
  (print 'jedna)
  'dvě)
```

To je jediný rozdíl mezi klíčovými slovy `&rest` a `&body`.

λ-seznamy maker mají ještě další zvláštnosti, ale o nich tady mluvit nebudeme.

2 Problém vícenásobného vyhodnocení

Uvažme následující variantu makra `when`: makro `whenv` má pracovat stejně jako makro `when`, ale s tím rozdílem, že jako výsledek vrátí hodnotu testované podmínky:

```
CL-USER 3 > (whenv (cdr '(1 2))
              (print "Ano"))

"Ano"
(2)
```

Nejjednodušší možnost (a nesprávná, jak uvidíme za chvíli), jak makro napsat, je tato:

```
(defmacro whenv (condition &body body)
  `(when ,condition
     ,@body
     ,condition))
```

Při testování makra nejspíš nenarazíme na žádný problém. Alespoň test, který jsem ukázal před chvílí, proběhne podle očekávání. Určité podezření bychom ale měli pojmout při expanzi makra. Výraz napsaný před chvílí expanduje takto:

```
(when (cdr '(1 2))
  (print "Ano")
  (cdr '(1 2))))
```

Vidíme, že při vyhodnocení se výraz `(cdr '(1 2))` vyhodnotí dvakrát, což jistě není to, co by uživatel čekal. Vidět je to při vedlejším efektu:

```
CL-USER 3 > (whenv (cdr (print '(1 2)))
              (print "Ano"))

(1 2)
"Ano"
(1 2)
(2)
```

Zde jistě čekal, že se seznam `(1 2)` vytiskne jen jednou. (Funkce `print` vytiskne svůj argument a pak ho ještě vrátí jako výsledek.)

Narazili jsme na první problém s makry, na který je třeba si dát pozor: **problém vícenásobného vyhodnocení**.

Problém demonstrujeme ještě na jednom příkladě. Chceme napsat makro `test-number`, které realizuje větvení na tři větve podle znaménka zadaného čísla:

```
(defmacro test-number (number minus zero plus)
  `(cond ((< ,number 0) ,minus)
         ((= ,number 0) ,zero)
         ((> ,number 0) ,plus)))
```

S makrem je zdánlivě všechno v pořádku, ale my už tušíme, že zadaný výraz bude vyhodnocovat vícekrát. Můžete si sami vyzkoušet, k čemu povede toto:

```
(let ((n 10))
  (test-number (print n)
    'minus
    'zero
    'plus))
```

3 Problém zabrání symbolu

Problém s vícenásobným vyhodnocováním u makra `whenv` se můžeme také pokusit vyřešit pomocí nové vazby, na kterou navážeme vyhodnocený argument:

```
(defmacro whenv (condition &body body)
  `(let ((result ,condition))
     (when result
       ,@body
       result)))
```

Test:

```
CL-USER 16 > (whenv (cdr (print '(1 2)))
                (print "Ano"))

(1 2)
"Ano"
(2)
```

Vidíme, že ke dvojímu vyhodnocení testovaného výrazu nedošlo.

Tím jsme vyřešili problém vícenásobného vyhodnocení, ale bohužel vytvořili problém nový, a to **problém zabrání symbolu** (*symbol capture*). Zkusme tento test:

```
CL-USER 18 > (let ((result '(1 2)))
                (whenv (cdr result)
                  (print "Neprázdné cdr v seznamu:")
                  (print result)))

"Neprázdné cdr v seznamu:"
(2)
(2)
```

Na předposledním řádku vidíme, co vytiskla funkce `print`: seznam `(2)`. My jsme ovšem čekali, že to bude seznam `(2 3)`.

Abychom zjistili, v čem je problém, nahradíme makro-výraz v testu jeho expanzí:

```
(let ((result '(1 2)))
  (let ((result (cdr result)))
    (when result
      (print "Neprázdné cdr v seznamu:")
      (print result))))
```

Teď už by to mělo být jasné. Došlo k zabrání symbolu `result`.

K zabrání symbolu dochází, pokud je v expanzi makra vytvořeno prostředí, které zastiňuje prostředí vytvořené uživatelem, a symboly v uživatelově kódu tak mají jiné než předpokládané aktuální vazby.

Problém lze řešit vygenerováním jedinečného symbolu pomocí funkce `gensym`. Funkci lze volat bez argumentu nebo s jedním argumentem. Pokud je argument použit, měl by to být řetězec a dostane se do názvu symbolu. Symboly vytvořené funkcí `gensym` začínají znaky „`#:`“.

```
CL-USER 21 > (gensym)
#:G1144

CL-USER 22 > (gensym "SYMBOL")
#:SYMBOL1145
```

Třetí (a konečně správná) verze makra vypadá takto:

```
(defmacro whenv (condition &body body)
  (let ((res-symbol (gensym)))
    `(let ((,res-symbol ,condition))
       (when ,res-symbol
         ,@body
         ,res-symbol))))
```

Výraz

```
(whenv (cdr result)
  (print "Neprázdné cdr v seznamu:")
  (print result))
```

nyní expanduje na

```
(LET ((#:G1151 (CDR RESULT)))
  (WHEN #:G1151
    (PRINT "Neprázdné cdr v seznamu:")
    (PRINT RESULT)
    #:G1151))
```

a symbol `result` už není zabrán.

Správná implementace makra `test-number`:

```
(defmacro test-number (number minus zero plus)
  (let ((value (gensym "VALUE")))
    `(let ((,value ,number))
       (cond ((< ,value 0) ,minus)
             ((= ,value 0) ,zero)
             ((> ,value 0) ,plus)))))
```

Mimochodem, makro `whenv` lze napsat i takto:

```
(defun whenv-help (cond-val body-fun)
  (when cond-val
    (funcall body-fun)
    cond-val))

(defmacro whenv (condition &body body)
  `(whenv-help ,condition (lambda () ,@body)))
```

Toto řešení má jistě své výhody: nevytváří problém vícenásobného vyhodnocení a zabrání symbolu a nepoužívá funkci `gensym`.

Na závěr ještě poznamenejme, že makro `whenb` z minulé přednášky také zabírá symbol, ale v tomto případě jde o jeho zamýšlený efekt.

4 Rekurze v makrech

Složitější makra ve své expanzi obsahují sama sebe. Takovým makrům se říká *rekurzivní makra*. Příkladem může být tato verze makra `and` pro libovolný počet argumentů:

```
(defmacro my-and (&rest forms)
  (if forms
      `(when ,(car forms) (my-and ,@(cdr forms)))
    t))
```

Rekurzivní makro musí testovat ukončení rekurze během expanze. Následující makro tuto podmínku nesplňuje:

```
(defmacro while (condition &body body)
  `(when ,condition
     ,@body
     (while ,condition ,@body)))
```

Proto se při kompilaci výrazu obsahujícího toto makro kompilátor zacyklí. Můžete to vyzkoušet třeba zkompilováním této funkce:

```
(defun test ()
  (let ((n 10))
    (while (> n 0)
      (print n)
      (setf n (- n 1)))))
```

Je několik způsobů, jak makro `while` napsat správně. Jeden z nich je tento:

```
(defmacro while (condition &body body)
  `(do-while (lambda () ,condition)
             (lambda () ,@body)))

(defun do-while (cond-fun body-fun)
  (when (funcall cond-fun)
    (funcall body-fun)
    (do-while cond-fun body-fun)))
```

Makro teď používá pomocnou funkci `do-while`. Ta je rekurzivní, což ovšem u funkce samozřejmě nevadí.

Nyní můžete zkusit znovu zkompilovat a pak i otestovat funkci `test`.

Otázky a úkoly na cvičení

1. Napište makro `whenbb`, které je obdobou makra `when`, ale v jeho těle je proměnná `it` navázána na hodnotu podmínky:

```
> (whenbb (cdr '(a b))
    (print it)
    nil)

(B)
NIL
```

2. Odhadněte a pak vyzkoušejte, na co expandují následující výrazy:

```
(my-and)
(my-and a)
(my-and a b)
(my-and a b c)
```

3. Napište makro `my-or` pro libovolný počet argumentů.

4. Vysvětlete poslední uvedenou verzi makra `whenv`.

5. Přepište podobným způsobem makro `test-number`.

6. Napište makro `swap`, které vymění hodnoty uložené ve dvou proměnných:

```
CL-USER 1 > (setf a 1 b 2)
2

CL-USER 2 > (swap a b)
2

CL-USER 3 > (list a b)
(2 1)
```

   K nastavení hodnoty proměnné použijte makro `setf`.

7. Napište makro `all-cond`, které pracuje podobně jako makro `cond`, ale vyhodnotí všechny větve, jejichž podmínka je splněna. Následující výraz:

```
(all-cond ((< 1 2) (print 1))
          ((> 1 2) (print 2))
          ((= 1 1) (print 3))
          ((= 1 2) (print 4)))
```

   tedy vytiskne postupně čísla `1` a `3`.