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

Makra 2

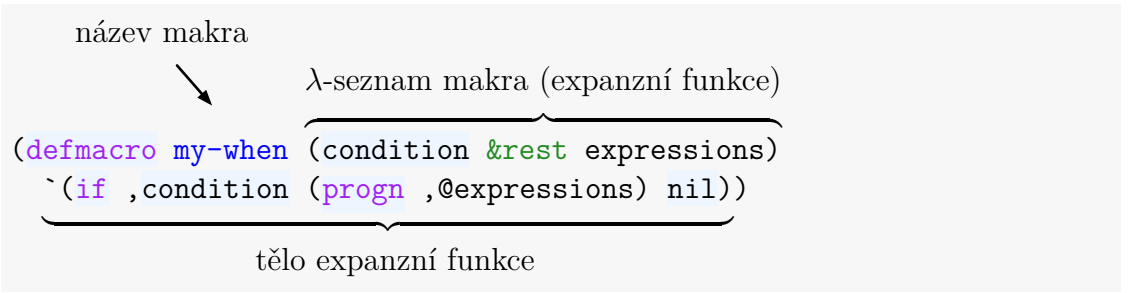
verze z 28. února 2020

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 Co dosud víme o makrech

Makro je nástroj umožňující abstrakci na úrovni zdrojového kódu. Pomocí maker lze definovat operátory s vlastními pravidly vyhodnocování, podobně jako je tomu u speciálních operátorů. Pravidla vyhodnocování se zadají pomocí funkce, tzv. **expanzní funkce makra**, která převede zadaný makro-výraz na jiný výraz.

Nové makro se definuje makrem `defmacro`.



Expanze:

```
(my-when t 1) ⟶ (if t (progn 1) nil)
(my-when (> x y) 1) ⟶ (if (> x y) (progn 1) nil)
(my-when (> x y) (print z) (+ x y))
    ⟶ (if (> x y) (progn (print z) (+ x y)) nil)
```

Expandovaný výraz se pak, jak víme, znovu předloží vyhodnocovacímu procesu k vyhodnocení.

Vyhodnocení makro-výrazu se tedy skládá ze dvou kroků. Při ladění je dobré zabývat se každým krokem zvlášť.

2 λ-seznamy maker

O λ-seznamu expanzní funkce hovoříme jen pro zjednodušení a představu. Přesnější je mluvit o λ-seznamech maker. Uvidíme, že λ-seznamy maker mají proti λ-seznamům funkcí zvláštnosti.

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` 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, a to klíčové slovo `&whole` a destrukturalizaci.

Klíčové slovo `&whole` může být na začátku λ-seznamu makra, následováno jedním parametrem. V tomto parametru bude uložen celý expandovaný výraz:

```
(defmacro my-when (&whole whole condition &rest expressions)
  (format t "~%Expanduje se výraz ~s " whole)
  `(if ,condition (progn ,@expressions) nil))
```

Test:

```
CL-USER 1 > (my-when (< 0 1) 'ano)

Expanduje se výraz (MY-WHEN (< 0 1) (QUOTE ANO))
ANO
```

Další zvláštností λ-seznamů maker je možnost tzv. *destrukturalizace*. Místo libovolného parametru je možno použít opět λ-seznam. Např. makro `dolist` by mohlo mít následující λ-seznam:

```
((var list &optional result) &body body)
```

3 Problém vícenásobného vyhodnocení

Uvažme následující testovací 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` svůj argument tiskne a pak ho ještě vrací 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))
```

4 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)
      result)))
```

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

K **zabrání symbolu** může dojít, 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`. Používá také výhodnou strategii psaní maker: neřešit vše v expanzi výrazu, ale expandovat výraz na co nejjednodušší aplikaci pomocné funkce. Pracovně tomu říkáme **pravidlo co nejrychlejšího úniku z makra do funkce**.

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

Hlavní problémy s makry

- vícenásobné vyhodnocení
- zabrání symbolu
- jsou to *neprominutelné chyby*

5 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))
```

Další příklad, jak uvidíme nesprávný:

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

Abychom pochopili, co není na příkladu správně, uvědomíme si, že programy v Lispu lze i kompilovat. Kompilátor má následující vlastnosti:

- Převádí program napsaný v daném jazyce (v našem případě v Lispu) na ekvivalentní program napsaný v jiném jazyce.
- Aby to mohl udělat, **expanduje všechna makra**.

Proto se při kompilaci výrazu obsahujícího makro `while` 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. (Všimněte si, že je opět použité pravidlo co nejrychlejšího úniku z makra do funkce.)

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

6 Čas expanze a čas běhu

Makro `case` je podobné příkazu `switch` v jazyce C. Jak pracuje, pochopíme z příkladu:

```
(case x
  (1 'jedna)
```

```
  (2 'dvě)
  (3 'tři)
  (otherwise 'moc))
```

Chceme napsat jeho variantu `random-case`, která nebude obsahovat testovanou hodnotu, ale větev, která se vykoná, vybere náhodně. Například výraz

```
(random-case
  ('jedna)
  ('dvě)
  ('tři))
```

by měl náhodně vrátit jeden ze symbolů `jedna`, `dvě`, nebo `tři`. (Symboly jsou v závorkách, aby větve mohly obsahovat víc výrazů.)

Zde jsou tři možnosti, jak makro napsat. Jedna z nich je nesprávná, jedna nešikovná a jedna optimální:

```
(defun random-case-help (body)
  (if (null body)
      '()
    (cons (cons (- (length body) 1) (car body))
          (random-case-help (cdr body)))))

(defmacro random-case (&body body)
  `(case (random (length ',body))
     ,@(random-case-help body)))

(defmacro random-case (&body body)
  `(case (random ,(length body))
     ,@(random-case-help body)))

(defmacro random-case (&body body)
  `(case ,(random (length body))
     ,@(random-case-help body)))
```

Rozdíl mezi těmito možnostmi je samozřejmě v tom, za jakých okolností se které výrazy vyhodnocují: zda **v čase expanze makra** (tj. jsou vyhodnocované expanzní funkcí), nebo **v čase běhu** (tj. jsou pouze expanzní funkcí vytvářené a vracené jako její výsledek).

Otázky a úkoly na cvičení

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

```
> (awhen (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. Vylepšete makro `my-and` tak, aby místo `t` vracelo hodnotu posledního výrazu. Pro výraz `(my-and)` by mělo stále vracet `t`. (Lze to udělat velmi jednoduše pomocí `&optional`-parametru.)
4. Napište makro `my-or` pro libovolný počet argumentů.
5. Vysvětlete poslední uvedenou verzi makra `whenv`.
6. Přepište podobným způsobem makro `test-number`.
7. Rozhodněte, zda v následující implementaci makra `prog1` dochází k problému zabrání symbolu. Pokud ano, uveďte příklad, pokud ne, zdůvodněte.

```
(defmacro my-prog1 (form1 &body forms)
  `(let ((result ,form1)
         (fun (lambda () ,@forms)))
     (funcall fun)
     result))
```

8. Rozhodněte, zda v následující implementaci makra `prog1` dochází k problému zabrání symbolu. Pokud ano, uveďte příklad, pokud ne, zdůvodněte.

```
(defmacro my-prog1 (first &body body)
  `(let ((result ,first)
         (bresult (progn ,@body)))
     result))
```

9. Operátor `test` by měl vytisknout svůj nevyhodnocený argument i jeho hodnotu a tuto hodnotu vrátit i jako výsledek. Například:

```
> (* 2 (test (+ 3 4)))
Vyhodnocovaný výraz: (+ 3 4)
Hodnota výrazu: 7
14
```

Pokud je to možné, definujte operátor jako funkci. Pokud to možné není, zdůvodněte to a definujte jej jako makro.

10. Jak víme, makro `dolist` prochází prvky seznamu. Napište makro `doatoms`, které podobně prochází všechny atomy ve struktuře tečkových párů. Příklad:

```
> (doatoms (x '(1 (2 . (3)) (4)))
    (print x))

1
2
3
4
```

Makro by se mělo obejít bez vytváření pomocných struktur.

11. 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`. Návratovou hodnotou makra se nejprve nezabývejte a pak řešení vylepšete, aby vracelo hodnotu posledního vyhodnoceného výrazu.

12. Napište makro `alias` sloužící k definici nového jména zadaného operátoru (např. kvůli zkrácení dlouhého názvu). Například po vyhodnocení výrazu

```
(alias funcall fc)
```

bychom mohli místo operátoru `funcall` v libovolném výrazu použít `fc`:

```
> (fc #'list 1 2 3)
(1 2 3)
```

Makro musí fungovat správně na všechny typy operátorů (tj. funkce makra i speciální operátory).