Makra v CL

Starý text k makrům, používejte na vlastní nebezpečí.

Princip abstrakce v programování

Abstrakce

Myšlenkový proces, kterým předmět zbavujeme zvláštností a odlišností a zjišťujeme jeho obecné a podstatné vlastnosti.

V programování

U různých jevů (kódu, dat) vytipujeme několik základních vlastností a jejich souhrn označíme jménem.

Výhody abstrakce

- v daný moment řešíme pouze omezené množství problémů,
- neřešíme problémy, které zrovna nejsou podstatné,
- zvyšuje se možnost znovupoužitelnosti kódu,
- zvyšuje se čitelnost kódu,
- snižuje se chybovost v kódu.

Programová abstrakce

Objevila se v strukturovaném (procedurálním) programování, kdy se opakující se kód začal zapouzdřovat do jedné procedury:

```
;; výpočet obsahu trojúhelníka na dvou místech programu:
(/ (* 2 3) 2)
...
(/ (* 3 4) 2)

;; abstrakce pomocí procedury (funkce):
(defun triangle-area (base height)
  (/ (* base height) 2))

;; použití:
(triangle-area 2 3)
...
(triangle-area 3 4)
```

Výhody:

- zvyšuje čitelnost a přehlednost kódu (`(triangle-area 2 3)` je srozumitelnější než `(/ (* 2 3) 2)`),
- znovupoužitelnost (jednou napsanou funkci lze použít jindy),
- snadnější úprava kódu (kód upravujeme jen na jednom místě)
- a z předchozího plynoucí menší chybovost.

Otázka: Jak se ve výše uvedeném kódu uplatňují rysy a výhody abstrakce? A konkrétně programové abstrakce?

Datová abstrakce

Souvisí s abstraktními bariérami (a zapouzdřením). Nemusíme vědět, jak jsou data implementována, stačí, že s nimi umíme zacházet — abstraktní datové typy.

Výhody:

- zejména snadnost změny implementace (najdou se i další).

Otázka: Jak se u abstraktních datových typů uplatňují rysy a výhody abstrakce? A konkrétně datové abstrakce?

Problém s programovou abstrakcí

Existují příklady opakování kódu, které se nedá vyřešit použitím funkce.

Příklad

```
(defun test-string (string &key length< length> newlines no-newlines)
  (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)))
```

Opakující se kód `(if podmínka výraz t)` nelze rozumně abstrahovat funkcí. Nelze vytvořit abstrakci *logická implikace*.

Příklad

Chceme, aby objekt po změně vnitřního stavu změnu nahlásil. Abychom se vyhnuli zbytečnému hlášení změn při rekurzi, použijeme počítadlo:

```
(defun set-color (object color)
  (incf (counter object))
  (set-color (subobject-1 object) color)
  (set-color (subobject-2 object) color)
  ...
  (decf (counter object))
  (when (zerop (counter object))
    (report-change object)))
```

Na takové případy je v Common Lispu připraven systém maker.

Makra: první pohled

Makra jsou nástroj na transformaci zdrojového kódu.

Pojmy k naučení

- Expanzní funkce
- Expanze makra

Vyjasnit si, za jakých okolností dojde během vyhodnocovacího procesu k expanzi makra.

Příklad s logickou implikací

Expanzní funkce provede následující expanzi:

```
(impl a b) --> (if a b t)
```

Seznam parametrů pro expanzní funkci neobsahuje název makra. Jak by tedy funkce vypadala?

```
(defun impl-expansion (expr-1 expr-2)
  (list 'if expr-1 expr-2 't))
```

Neboli:

```
(defun impl-expansion (expr-1 expr-2)
  `(if ,expr-1 ,expr-2 t))
```

Definice makra:

```
(defmacro impl (expr-1 expr-2)
  `(if ,expr-1 ,expr-2 t))
```

Úprava funkce **test-string**:

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

Makro **impl** poskytuje výše zmíněné výhody abstrakce.

Naučit se

- Vysvětlit, co dělá operátor **defmacro**,
- zkoušet expanzi maker ve vývojovém prostředí LispWorks.

λ-seznamy maker

Víme, že λ-seznamy funkcí mohou obsahovat následující klíčová slova: **&rest**, **&key**, **&allow-other-keys**, **&optional**. λ-seznamy maker mohou navíc obsahovat klíčová slova **&body** a **&whole**.

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, viz příklad níže.

V λ-seznamech maker navíc je možno na každém místě, kde je očekáván parametr, použít další λ-seznam.

Příklad. Při definici makra **my-case** s λ-seznamem **(val &rest options)** bude automatické formátování vypadat takto:

```
(my-case x
         (1 'jedna)
         (2 'dvě))
```

zatímco s λ-seznamem **(val &body options)** takto:

```
(my-case x
  (1 'jedna)
  (2 'dvě))
```

Klíčové slovo **&whole** se může vyskytovat jako první prvek λ-seznamu a musí být následováno symbolem, na který bude při expanzi navázán celý expandovaný makrovýraz.

Příklad. Makro **my-case**, definované takto:

```
(defmacro my-case (&whole whole val &body options)
  (format nil "~%my-case expanduje výraz~%~s" whole)
  ...)
```

při expanzi výše uvedeného výrazu vytiskne

```
my-case expanduje výraz
(MY-CASE X (1 (QUOTE JEDNA)) (2 (QUOTE DVě)))
```

Další zvláštností λ-seznamů maker je možnost tzv. destructuringu. Místo libovolného parametru je možno použít opět λ-seznam. Např. makro **my-dolist** tak může mít následující λ-seznam:

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

Dvě nebezpečí

Při psaní maker hrozí dvě základní nebezpečí, se kterými je třeba počítat: zabrání symbolu a vícenásobné vyhodnocení. Oba problémy je třeba pochopit a při psaní maker se jim vyhnout (neprominutelné chyby).

Problém: zabrání symbolu (variable capture)

Definice operátoru **prog1** pomocí makra:

```
(defmacro my-prog1 (first-expr &body body)
  `(let ((value ,first-expr))
     ,@body
     value))
```

Chceme vrátit **(+ value 10)** a vytisknout **(* value 10)**:

```
(let ((value 10))
  (my-prog1 (+ value 10)
    (print (* value 10))))
```

Co se stane? Proč?

Dojde k zabrání symbolu **value**.

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.

Řešení pomocí jedinečného symbolu vygenerovaného funkcí **gensym**.:

```
(defmacro my-prog1 (first-expr &body body)
  (let ((symbol (gensym "VALUE")))
    `(let ((,symbol ,first-expr))
       ,@body
       ,symbol)))
```

Otázka: dochází k zabrání symbolu v následujícím příkladu?

```
(defmacro my-prog1 (expr1 &body body)
   `(let ((result ,expr1)
          (body-fun (lambda () ,@body)))
      (funcall body-fun)
      result))
```

Ke správnému pochopení příkladu je třeba přesně rozumět lexikálním uzávěrům a speciálnímu operátoru **`let`**.

Makro **`prog1`** je ovšem možno napsat jako funkci!

```
(defun my-prog1 (val1 &rest rest)
  val1)
```

Ve variantě Lispu, která nemá specifikované pořadí vyhodnocování argumentů funkce (například Scheme), by to ovšem nešlo.

Chtěné zabrání symbolu. Některá makra mají zabrání symbolu jako svůj hlavní efekt (např. makro **`dolist`** nebo makro **`whenb`** z minulého cvičení). U těchto maker programátor ví, že k zabrání symbolu dochází (je to v dokumentaci makra), a využívá toho.

Problém: vícenásobné vyhodnocení

Makro **`test-number`** na větvení podle znaménka zadaného čísla:

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

Pokus:

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

Co se stane? Proč?

Nesprávné řešení (zabrání symbolu!):

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

Správné řešení:

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

Při psaní maker je třeba dávat pozor na dva problémy:

- Zabrání symbolu
- Vícenásobné vyhodnocení

Je třeba znát způsob, jak se jim vyhnout.

Další záludnosti maker

Odlišení fáze expanze od fáze běhu

V případě, že je zdrojový kód programu kompilován, nedochází k expanzi makra opakovaně, ale pouze jednou, při jeho kompilaci.

Příklad. Makro **`random-case`** má vykonat náhodně vybraný výraz ze svého těla. Okomentujte následující tři implementace tohoto makra.

```
(defun random-case-help (body)
  "Pomocná funkce pro random-case"
  (let ((n -1))
    (mapcar (lambda (e) (list (incf n) e))
            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)))
```

Použití funkce `eval`

Víme z dřívějška, že používat funkci **`eval`** není vhodné (umíte vyjmenovat důvody?). Systém maker v Common Lispu je vymyšlen tak, že (téměř) nikdy není nutné funkci **`eval`** použít. V našich příkladech je použití této funkce zakázáno.

Rekurzivní makra

V expanzi makrovýrazu se může opět vyskytovat totéž makro. Takovým makrům říkáme rekurzivní makra. Ukončující podmínka rekurze by se měla vyhodnocovat během expanze:

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

Zde je všechno v pořádku, ukončovací podmínka rekurze se vyhodnocuje v čase expanze.

Testovací funkce na makro **`while`**:

```
(defun test ()
  (let ((x 10))
    (while (plusp x)
      (print x)
      (decf x 3))))
```

Makro **`while`** pomocí cyklu (tedy nikoli rekurzivně):

```
(defmacro while (condition &body body)
  `(loop (unless ,condition (return))
         ,@body))
```

Rekurzivní verze makra **`while`**. Je nesprávně, ukončující podmínka rekurze se vyhodnocuje až za běhu, expanze tedy vždy opět obsahuje volání **`while`**:

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

Kdy se problém projeví? Při kompilaci. Připomeňme, že při kompilaci funkce se nejprve expandují všechna makra. U našeho makra **while** expanze nikdy neskončí. Problém lze demonstrovat tak, že vyhodnotíme definici makra a pak zkusíme zkompilovat funkci **test**.

Verze makra **while** s pomocnou funkcí. Makro pouze expanduje na volání funkce. Zde samozřejmě není žádný problém:

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

Pomocná funkce již (samozřejmě) může být rekurzivní:

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

Příklad

Demonstrujeme několik problémů s makry na makru **at-least**. Toto makro má pro zadanou hodnotu parametru **n** zjistit, zda alespoň **n** ze zadaných výrazů má hodnotu ***pravda***. Makro má používat zkrácené vyhodnocování parametrů, tj. vyhodnocovat pouze tolik parametrů, kolik je třeba.

Každá následující verze makra odstraňuje nějaký problém verze předchozí.

```
(defmacro at-least (count &rest conditions)
  `(cond ((null ',conditions) (zerop ,count))
         ((zerop ,count) t)
         (,(first conditions)
           (at-least (1- ,count) ,@(rest conditions)))
         (t (at-least ,count ,@(rest conditions)))))
```

Odstranění problému rekurze. Ukončující podmínka se nyní řeší během expanze, makro se tedy při kompilaci nezacyklí:

```
(defmacro at-least (count &rest conditions)
  (if (null conditions)
      `(zerop ,count)
    `(cond ((zerop ,count) t)
           (,(first conditions)
             (at-least (1- ,count) ,@(rest conditions)))
           (t (at-least ,count ,@(rest conditions))))))
```

Tato verze ovšem zabírá symbol. Řešení:

```
(defmacro at-least (count &rest conditions)
  (if (null conditions)
      `(zerop ,count)
    (let ((count-sym (gensym)))
      `(let ((,count-sym ,count))
         (cond ((zerop ,count-sym) t)
```

```
                (,(first conditions)
                  (at-least (1- ,count-sym) ,@(rest conditions)))
                (t (at-least ,count-sym ,@(rest conditions))))))))
```

Jeden podstatný problém ještě zůstal. Expanze má exponenciální velikost vzhledem k délce seznamu `conditions`. Řešení:

```
(defmacro at-least (count &rest conditions)
  (if (null conditions)
      `(zerop ,count)
    (let ((count-sym (gensym)))
      `(let ((,count-sym ,count))
         (if (zerop ,count-sym)
             t
           (at-least (if ,(first conditions)
                         (1- ,count-sym)
                       ,count-sym)
                     ,@(rest conditions)))))))
```

Tady skončíme, i když řešení stále není ideální (není úplně splněna podmínka zkráceného vyhodnocování).

Příklady použití maker

```
;; na začátku soubor otevře, na konci zavře
(with-open-file (s "~/my-file.txt" :direction :input)
  (read-line s)
  ...)

;; práce s více soubory, zajištění správného chování při chybě
(with-open-files ((s1 "~/my-file-1.txt") (s2 "~/my-file-2.txt"))
  (write-line (read-line s1) s2)
  ...)

;; na začátku se připojí k databázi, na konci odpojí
(with-database-connection (db)
  ...)

;; na začátku otevře soubor, zapíše "Začátek stahování, čas"
;; na konci zapíše "Konec stahování, čas, chybová zpráva",
;; zavře soubor
(with-log ("stahování" &optional (log-file *my-log-file*))
  ...)

;; kritická sekce i s ošetřením chyb
(with-lock (lock :read t :write nil)
  ...)

;; zákaz zápisu do daných proměnných
(read-only (a b c)
  ...)

;; vytvoření html souboru
(html (:ul (dolist (item stuff)
             (html (:li item)))))

;; průchod strukturou
(do-tree (x tree)
  ...)
```

```
;; projde všechny podmnožiny dané množiny:
(do-subsets (set '(1 2 3))
  (print set))

;; projde všechny webové stránky, na něž se odkazuje daná stránka,
;; do dané hloubky, s vyloučením duplicit
(do-web-pages (p start-page depth)
  (print-page p))

;; vyhodnocení výrazu s určitou pravděpodobností
(random-case (0.8 (do-alive))
             (0.2 (do-dead)))

;; paralelní vyhodnocení výrazů, každý v jiném vlákně
(co-dotimes (x 10)
  (worker x))

;; funkce, která si pamatuje jednou vypočítané výsledky
(defmemfun fib (n)
  (if (<= n 1)
      1
    (+ (fib (- n 1)) (fib (- n 2)))))

;; definice testu
(deftest (nth-prime 10) 23)

;; webový server: odpověď na http://myserver/random-number
(define-url-function random-number (request limit)
  (html
    (:head (:title "Random"))
    (:body
      (:p "Random number: " (:print (random limit))))))
```