Metaprogramowanie

Techniki pisania makr

Aleksander Balicki

Instytut Informatyki

1 grudnia 2010

```
> (defmacro for ((var start stop) &body body)
  `(do ((,var ,start (1+ ,var))
        (limit ,stop))
       ((> ,var limit))
     ,@body))
```

```
> (defmacro for ((var start stop) &body body)
  `(do ((,var ,start (1+ ,var))
        (limit ,stop))
       ((> ,var limit))
     ,@body))

> (for (x 1 5)
    (princ x))
12345
NIL
```

```
> (defmacro for ((var start stop) &body body)
  `(do ((,var ,start (1+ ,var))
        (limit ,stop))
       ((> ,var limit))
     ,@body))

> (for (x 1 5)
    (princ x))
12345
NIL
```

Tu przechwytujemy zmienną limit. Jeżeli była ona użyta w kontekscie wywołania tego makra, to nie zadziala ono w zamierzony sposob.

```
> (for (limit 1 5)
    (princ limit))
```

Co robi to polecenie?

```
> (for (limit 1 5)
    (princ limit))
```

Co robi to polecenie?
Generuje Matrix.

```
> (for (limit 1 5)
    (princ limit))
```

Co robi to polecenie?
Generuje Matrix.
Zobaczmy co się dzieje po rozwinięciu tego makra:

```
> (do ((limit 1 (1+ limit))
    (limit 5))
      ((> limit limit))
        (princ limit))
```

```
> (let ((limit 5))
    (for (i 1 10)
      (when (> i limit)
        (princ i))))
NIL
```

```
> (let ((limit 5))
    (for (i 1 10)
      (when (> i limit)
        (princ i))))
NIL
```

Tutaj przykład jak nasz błąd nie daje o sobie znać, po prostu cicho zwraca NIL. Jest bardzo trudno znaleść taki błąd, jeśli jest zaszyty głęboko w infrastrukturze makr naszego programu.

```
> (defun fn (x) (+ x 1))
FN
> (defmacro mac (x) `(fn ,x))
MAC
> (mac 10)
11
> (labels ((fn (y) (- y 1)))
    (mac 10))
9
```

```
> (defun fn (x) (+ x 1))
FN
> (defmacro mac (x) `(fn ,x))
MAC
> (mac 10)
11
> (labels ((fn (y) (- y 1)))
    (mac 10))
9
```

Zaskakująco funkcję można też scapture'ować, tak jak każdy symbol.

```
> (defmacro for ((var start stop) &body body)
    (let ((gstop (gensym)))
        `(do ((,var ,start (1+ ,var))
              (,gstop ,stop))
             ((> ,var ,gstop))
            ,@body)))
```

```
> (defmacro for ((var start stop) &body body)
    (let ((gstop (gensym)))
        `(do ((,var ,start (1+ ,var))
              (,gstop ,stop))
             ((> ,var ,gstop))
            ,@body)))
```

To jest poprawna wersja z użyciem funkcji gensym.

```
> (defmacro for ((var start stop) &body body)
    (let ((gstop (gensym)))
        `(do ((,var ,start (1+ ,var))
              (,gstop ,stop))
             ((> ,var ,gstop))
            ,@body)))
```

To jest poprawna wersja z użyciem funkcji gensym.

```
> (setq x (gensym))
#:G48
> (setq *gensym-counter* 48 y (gensym))
#:G48
> (eq x y)
NIL
```

Włożenie makra do osobnej paczki - wtedy zmienne nie skolidują. Nie jest to zalecany sposób, bo makra specyficzne dla danego projektu wylądują w innej paczce.

```
> (let ((x 2))
    (for (i 1 (incf x))
         (princ i)))
12345678910111213...
```

```
> (let ((x 2))
    (for (i 1 (incf x))
         (princ i)))
12345678910111213...
```

Ilość ewaluacji wyrażenia ma znaczenie, nie licząc nieefektywności, wyrażenie może wywoływać skutki uboczne, takie jak zwiększenie wartości zmiennej.

W Common Lispie przyjęto konwencje ewaluacji argumentów od lewej do prawej, dobrze jest zachowywać tą konwencję.

```
> (setq x 10)
10
> (+ (setq x 3) x)
6
```

Zła kolejność ewaluacji:

```
(defmacro for ((var start stop) &body body)
  (let ((gstop (gensym)))
       `(do ((,gstop ,stop)
                     (,var ,start (1+ ,var)))
                    ((> ,var ,gstop))
             ,@body)))
```

Zła kolejność ewaluacji:

```
(defmacro for ((var start stop) &body body)
  (let ((gstop (gensym)))
       `(do ((,gstop ,stop)
                     (,var ,start (1+ ,var)))
                    ((> ,var ,gstop))
            ,@body)))

> (let ((x 1))
    (for (i x (setq x 13))
         (princ i)))
13
NIL
```

Dobra kolejność ewaluacji:

```
(defmacro for ((var start stop) &body body)
  (let ((gstop (gensym)))
       `(do ((,var ,start (1+ ,var))
                   (,gstop ,stop))
                  ((> ,var ,gstop))
             ,@body)))
```

Dobra kolejność ewaluacji:

```
(defmacro for ((var start stop) &body body)
  (let ((gstop (gensym)))
       `(do ((,var ,start (1+ ,var))
                   (,gstop ,stop))
                  ((> ,var ,gstop))
             ,@body)))

> (let ((x 1))
    (for (i x (setq x 13))
         (princ i)))
12345678910111213
NIL
```

Wywoływanie skutków ubocznych w trakcie expandowania makra

```
(defmacro nil! (x)
  (incf *nil!s*)
    `(setf ,x nil))
```

Wywoływanie skutków ubocznych w trakcie expandowania makra

```
(defmacro nil! (x)
  (incf *nil!s*)
    `(setf ,x nil))
```

jest niezalecane, bo nie wiadomo ile razy makro będzie expandowane. Zmienna *nil!s* nie będzie miała zapisanej ilości użyć makra, tylko ilości jego expansji.

Można wywoływać makra rekurencyjnie, oby tylko expand kiedys się zakończył. Zły przykład:

```
(defmacro nthb (n lst)
  `(if (= ,n 0)
       (car ,lst)
      (nthb (- ,n 1) (cdr ,lst))))
```

Można wywoływać makra rekurencyjnie, oby tylko expand kiedys się zakończył. Zły przykład:

```
(defmacro nthb (n lst)
  `(if (= ,n 0)
       (car ,lst)
      (nthb (- ,n 1) (cdr ,lst))))
```

Expanduje się do:

Można wywoływać makra rekurencyjnie, oby tylko expand kiedys się zakończył. Zły przykład:

```
(defmacro nthb (n lst)
  `(if (= ,n 0)
       (car ,lst)
       (nthb (- ,n 1) (cdr ,lst))))
```

Expanduje się do:

```
(if (= x 0)
    (car y)
        (if (= (- x 1) 0)
        (car (cdr y))
        (nthb (- (- x 1) 1) (cdr (cdr y)))))
```

Można to rozwiązać wywołując funkcję rekurencyjnie z makra

```
(defmacro nthd (n lst)
  `(nth-fn ,n ,lst))
(defun nth-fn (n lst)
  (if (= n 0)
      (car lst)
      (nth-fn (- n 1) (cdr lst))))
```

Można to rozwiązać wywołując funkcję rekurencyjnie z makra

```
(defmacro nthd (n lst)
  `(nth-fn ,n ,lst))
(defun nth-fn (n lst)
  (if (= n 0)
      (car lst)
      (nth-fn (- n 1) (cdr lst))))
```

Jeśli rekursja jest ogonowa, to można to zamienić na while'a

Przykład or-a z rekurencyjnym wywołaniem makra

```
(defmacro orb (&rest args)
  (if (null args)
      nil
      (let ((sym (gensym)))
        `(let ((,sym ,(car args)))
           (if ,sym
               ,sym
               (orb ,@(cdr args)))))))
```

Proste przykłady:

```
(defmacro avg (&rest args)
  `(/ (+ ,@args) ,(length args)))

> (avg pi 4 5)
4.047...
```

Proste przykłady:

```
(defmacro avg (&rest args)
  `(/ (+ ,@args) ,(length args)))
```

```
> (avg pi 4 5)
4.047...
```

Liczenie długości listy argumentów w czasie kompilacji.

Proste przykłady:

```
(defmacro avg (&rest args)
  `(/ (+ ,@args) ,(length args)))
```

```
> (avg pi 4 5)
4.047...
```

Liczenie długości listy argumentów w czasie kompilacji. Robimy pracę, którą za nas powinien wykonywać jakiś statyczny optymalizator kodu w kompilatorze.

Ale nie wszystkie rzeczy kompilator potrafi zoptymalizować.
Skomplikowany przypadek:

Definition

A Bezier curve is defined in terms of four points—two endpoints and two control points. When we are working in two dimensions, these points define parametric equations for the x and y coordinates of points on the curve. If the two endpoints are (x_0, y_0) and (x_3, y_3) and the two control points are (x_1, y_1) and (x_2, y_2), then the equations defining points on the curve are:
$x = (x^3 - 3x^2 + 3x^1 - x^0)u^3 + (3x^2 - 6x^1 + 3x^0)u^2 + (3x^1 - 3x^0)u + x^0$
$y = (y^3 - 3y^2 + 3y^1 - y^0)u^3 + (3y^2 - 6y^1 + 3y^0)u^2 + (3y^1 - 3y^0)u + y^0$
for $0 < u < 1$

```
(defconstant *segs* 20)
(defconstant *du*   (/ 1.0 *segs*))
(defconstant *pts* (make-array (list (1+ *segs*) 2)))
(defmacro genbez (x0 y0 x1 y1 x2 y2 x3 y3)
  (with-gensyms (gx0 gx1 gy0 gy1 gx3 gy3)
    `(let ((,gx0 ,x0) (,gy0 ,y0)
           (,gx1 ,x1) (,gy1 ,y1)
           (,gx3 ,x3) (,gy3 ,y3))
      (let ((cx (* (- ,gx1 ,gx0) 3))
            (cy (* (- ,gy1 ,gy0) 3))
            (px (* (- ,x2 ,gx1) 3))
            (py (* (- ,y2 ,gy1) 3)))
        (let ((bx (- px cx))
              (by (- py cy))
              (ax (- ,gx3 px ,gx0))
              (ay (- ,gy3 py ,gy0)))
 *cośtam cośtam*
```

```
(let ((result (big-long-calculation)))
  (if result
      (foo result)))
```

```
(let ((result (big-long-calculation)))
  (if result
      (foo result)))
```

Czy nie byłoby łatwiej napisać:

```
(if (big-long-calculation)
    (foo it))
```

Anaforyczne makra to umożliwiają.

```
(defmacro aif (test-form then-form &optional else-form)
  `(let ((it ,test-form))
     (if it ,then-form ,else-form)))
```

```
(defmacro aif (test-form then-form &optional else-form)
  `(let ((it ,test-form))
     (if it ,then-form ,else-form)))
```

Można analogicznie napisać awhile, awhen, acond...

```
(defmacro aif (test-form then-form &optional else-form)
  `(let ((it ,test-form))
     (if it ,then-form ,else-form)))
```

Można analogicznie napisać awhile, awhen, acond...

```
(defmacro aand (&rest args)
  (cond ((null args) t)
        ((null (cdr args)) (car args))
        (t `(aif ,(car args) (aand ,@(cdr args))))))
```

```
(defmacro aif (test-form then-form &optional else-form)
  `(let ((it ,test-form))
     (if it ,then-form ,else-form)))
```

Można analogicznie napisać awhile, awhen, acond...

```
(defmacro aand (&rest args)
  (cond ((null args) t)
        ((null (cdr args)) (car args))
        (t `(aif ,(car args) (aand ,@(cdr args))))))
```

Tu lekka modyfikacja sprawia, że it zawsze przyjmuje wartość wcześniej sprawdzanego argumentu.

Rekurencyjna lambda:

```
(defmacro alambda (parms &body body)
  `(labels ((self ,parms ,@body))
     #'self))

(alambda (x) (if (= x 0) 1 (* x (self (1- x)))))
```

Rekursja na listach the old fashion way:

```
(defun our-every (fn lst)
  (if (null lst)
      t
(and (funcall fn (car lst))
(our-every fn (cdr lst)))))
```

Rekursja na listach with a slightly better approach:

```
(defun lrec (rec &optional base)
  (labels ((self (lst)
             (if (null lst)
                 (if (functionp base)
                     (funcall base)
                     base)
                 (funcall rec (car lst)
                              #'(lambda ()
                                  (self (cdr lst)))))))
    #'self))

(lrec #'(lambda (x f) (and (oddp x) (funcall f)))
      t)
```

The new macro way:

```
(defmacro alrec (rec &optional base)
    (let ((gfn (gensym)))
        `(lrec #'(lambda (it ,gfn)
                   (symbol-macrolet ((rec (funcall ,gfn)))
                                    ,rec))
           ,base)))

(defmacro on-cdrs (rec base &rest lsts)
  `(funcall (alrec ,rec #'(lambda () ,base)) ,@lsts))
```

Przykłady on-cdrs:

```
(defun our-copy-list (lst)
  (on-cdrs (cons it rec) nil lst))
(defun our-remove-duplicates (lst)
  (on-cdrs (adjoin it rec) nil lst))
(defun our-find-if (fn lst)
  (on-cdrs (if (funcall fn it) it rec) nil lst))
(defun our-some (fn lst)
  (on-cdrs (or (funcall fn it) rec) nil lst))
```

Analogicznie można zdefiniować rekursory na drzewach.

Leniwą ewaluację uzyskujemy w lispie przez użycie obiektow typu delay. Scheme ma wbudowany system obsługujący delaye. Jest to funkcja, która jest placeholderem dla naszego argumentu i w odpowiednim momencie możemy ją zmusić do zwrócenia wartości.

```
> (let ((x 2))
    (setq d (delay (1+ x))))
#S(DELAY ...)
```

Leniwą ewaluację uzyskujemy w lispie przez użycie obiektow typu delay. Scheme ma wbudowany system obsługujący delaye. Jest to funkcja, która jest placeholderem dla naszego argumentu i w odpowiednim momencie możemy ją zmusić do zwrócenia wartości.

```
> (let ((x 2))
    (setq d (delay (1+ x))))
#S(DELAY ...)
```

Do ewaluacji delaya używamy funkcji force, która zmusza delay do obliczenia wartości, a w innych przypadkach działa jak identyczność.

```
> (force 'a)
A
> (force d)
3
```

```
(let ((count 0))
  (lambda (msg)
    (case msg
      ((:inc)
        (incf count))
      ((:dec)
        (decf count)))))
```

Definiujemy funkcję, która kontroluje nam licznik.

```
(let ((count 0))
  (lambda (msg)
    (case msg
      ((:inc)
        (incf count))
      ((:dec)
        (decf count)))))
```

Definiujemy funkcję, która kontroluje nam licznik.

Możemy chcieć stworzyć abstrakcję do tego typu lambd.

```
(defmacro! dlambda (&rest ds)
  `(lambda (&rest ,g!args)
     (case (car ,g!args)
       ,@(mapcar
           (lambda (d)
             `(,(if (eq t (car d))
                    t
                    (list (car d)))
               (apply (lambda ,@(cdr d))
                      ,(if (eq t (car d))
                           g!args
                           `(cdr ,g!args)))))
           ds))))
```

Przykład dlambdy:

```
(setf (symbol-function 'count-test)
    (let ((count 0))
      (dlambda
        (:reset () (setf count 0))
        (:inc (n) (incf count n))
        (:dec (n) (decf count n))
        (:bound (lo hi)
          (setf count
            (min hi
                 (max lo
                      count)))))))
```

Funkcje z długimi nazwami są niewygodne:

```
(destructuring-bind ...)
(multiple-value-bind ...)
```

Pomocne makra:

```
(defmacro abbrev (short long)
  `(defmacro ,short (&rest args)
       `(,',long ,@args)))
(defmacro abbrevs (&rest names)
  `(progn
       ,@(mapcar #'(lambda (pair)
                     `(abbrev ,@pair))
                 (group names 2))))
```

Przykład użycia:

```
(abbrevs dbind destructuring-bind
mvbind multiple-value-bind
mvsetq multiple-value-setq)
```

Gettery i settery pola obiektu:

```
(defmacro propmacro (propname)
  `(defmacro ,propname (obj)
       `(get ,obj ',',propname)))

(defmacro propmacros (&rest props)
  `(progn
       ,@(mapcar #'(lambda (p) `(propmacro ,p))
                 props)))

(setf (color 'ball1) 'green)
```

Gettery i settery pola obiektu:

```
(defmacro propmacro (propname)
  `(defmacro ,propname (obj)
       `(get ,obj ',',propname)))

(defmacro propmacros (&rest props)
  `(progn
       ,@(mapcar #'(lambda (p) `(propmacro ,p))
                 props)))

(setf (color 'ball1) 'green)
```

Łatwa zmiana backendu przechowywania pól.