13

Computation at Compile-Time

The preceding chapters described several types of operators which have to be implemented by macros. This one describes a class of problems which could be solved by functions, but where macros are more efficient. Section 8.2 listed the pros and cons of using macros in a given situation. Among the pros was "computation at compile-time." By defining an operator as a macro, you can sometimes make it do some of its work when it is expanded. This chapter looks at macros which take advantage of this possibility.

13.1 New Utilities

Section 8.2 raised the possibility of using macros to shift computation to compile-time. There we had as an example the macro `avg`, which returns the average of its arguments:

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

Figure 13.1 shows `avg` defined first as a function and then as a macro. When `avg` is defined as a macro, the call to `length` can be made at compile-time. In the macro version we also avoid the expense of manipulating the `&rest` parameter at runtime. In many implementations, `avg` will be faster written as a macro.

The kind of savings which comes from knowing the number of arguments at expansion-time can be combined with the kind we get from `in` (page 152), where it was possible to avoid even evaluating some of the arguments. Figure 13.2 contains two versions of `most-of`, which returns true if most of its arguments do:

```
(defun avg (&rest args)
  (/ (apply #'+ args) (length args)))

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

Figure 13.1: Shifting computation when finding averages.

```
(defun most-of (&rest args)
  (let ((all 0)
        (hits 0))
    (dolist (a args)
      (incf all)
      (if a (incf hits)))
    (> hits (/ all 2))))

(defmacro most-of (&rest args)
  (let ((need (floor (/ (length args) 2)))
        (hits (gensym)))
    `(let ((,hits 0))
       (or ,@(mapcar #'(lambda (a)
                         `(and ,a (> (incf ,hits) ,need)))
                     args)))))
```

Figure 13.2: Shifting and avoiding computation.

```
> (most-of t t t nil)
T
```

The macro version expands into code which, like `in`, only evaluates as many of the arguments as it needs to. For example, `(most-of (a) (b) (c))` expands into the equivalent of:

```
(let ((count 0))
  (or (and (a) (> (incf count) 1))
      (and (b) (> (incf count) 1))
      (and (c) (> (incf count) 1))))
```

In the best case, just over half the arguments will be evaluated.

```
(defun nthmost (n lst)
  (nth n (sort (copy-list lst) #'>)))

(defmacro nthmost (n lst)
  (if (and (integerp n) (< n 20))
      (with-gensyms (glst gi)
        (let ((syms (map0-n #'(lambda (x) (gensym)) n)))
          `(let ((,glst ,lst))
             (unless (< (length ,glst) ,(1+ n))
               ,@(gen-start glst syms)
               (dolist (,gi ,glst)
                 ,(nthmost-gen gi syms t))
               ,(car (last syms))))))
      `(nth ,n (sort (copy-list ,lst) #'>))))

(defun gen-start (glst syms)
  (reverse
    (maplist #'(lambda (syms)
                 (let ((var (gensym)))
                   `(let ((,var (pop ,glst)))
                      ,(nthmost-gen var (reverse syms)))))
             (reverse syms))))

(defun nthmost-gen (var vars &optional long?)
  (if (null vars)
      nil
      (let ((else (nthmost-gen var (cdr vars) long?)))
        (if (and (not long?) (null else))
            `(setq ,(car vars) ,var)
            `(if (> ,var ,(car vars))
                 (setq ,@(mapcan #'list
                                 (reverse vars)
                                 (cdr (reverse vars)))
                       ,(car vars) ,var)
                 ,else)))))
```

Figure 13.3: Use of arguments known at compile-time.

A macro may also be able to shift computation to compile-time if the values of particular arguments are known. Figure 13.3 contains an example of such a macro. The function `nthmost` takes a number *n* and a list of numbers, and returns the *n*th largest among them; like other sequence functions, it is zero-indexed:

```
> (nthmost 2 '(2 6 1 5 3 4))
4
```

The function version is written very simply. It sorts the list and calls `nth` on the result. Since `sort` is destructive, `nthmost` copies the list before sorting it. Written thus, `nthmost` is inefficient is two respects: it conses, and it sorts the entire list of arguments, though all we care about are the top *n*.

If we know *n* at compile-time, we can approach the problem differently. The rest of the code in Figure 13.3 defines a macro version of `nthmost`. The first thing this macro does is look at its first argument. If the first argument is not a literal number, it expands into the same code we saw above. If the first argument is a number, we can follow a different course. If you wanted to find, say, the third biggest cookie on a plate, you could do it by looking at each cookie in turn, always keeping in your hand the three biggest found so far. When you have looked at all the cookies, the smallest cookie in your hand is the one you are looking for. If *n* is a small constant, not proportional to the number of cookies, then this technique gets you a given cookie with less effort that it would take to sort all of them first.

This is the strategy followed when *n* is known at expansion-time. In its expansion, the macro creates *n* variables, then calls `nthmost-gen` to generate the code which has to be evaluated upon looking at each cookie. Figure 13.4 shows a sample macroexpansion. The macro `nthmost` behaves just like the original function, except that it can't be passed as an argument to `apply`. The justification for using a macro is purely one of efficiency: the macro version does not cons at runtime, and if *n* is a small constant, performs fewer comparisons.

To have efficient programs, must one then take the trouble to write such huge macros? In this case, probably not. The two versions of `nthmost` are intended as an example of a general principle: when some arguments are known at compile-time, you can use a macro to generate more efficient code. Whether or not you take advantage of this possibility will depend on how much you stand to gain, and how much more effort it will take to write an efficient macro version. Since the macro version of `nthmost` is long and complicated, it would only be worth writing in extreme cases. However, information known at compile-time is always a factor worth considering, even if you choose not to take advantage of it.

```
(nthmost 2 nums)
```

expands into:

```
(let ((#:g7 nums))
  (unless (< (length #:g7) 3)
    (let ((#:g6 (pop #:g7)))
      (setq #:g1 #:g6))
    (let ((#:g5 (pop #:g7)))
      (if (> #:g5 #:g1)
          (setq #:g2 #:g1 #:g1 #:g5)
          (setq #:g2 #:g5)))
    (let ((#:g4 (pop #:g7)))
      (if (> #:g4 #:g1)
          (setq #:g3 #:g2 #:g2 #:g1 #:g1 #:g4)
          (if (> #:g4 #:g2)
              (setq #:g3 #:g2 #:g2 #:g4)
              (setq #:g3 #:g4))))
    (dolist (#:g8 #:g7)
      (if (> #:g8 #:g1)
          (setq #:g3 #:g2 #:g2 #:g1 #:g1 #:g8)
          (if (> #:g8 #:g2)
              (setq #:g3 #:g2 #:g2 #:g8)
              (if (> #:g8 #:g3)
                  (setq #:g3 #:g8)
                  nil))))
    #:g3))
```

Figure 13.4: Expansion of `nthmost`.

13.2 Example: Bezier Curves

Like the `with-` macro (Section 11.2), the macro for computation at compile-time is more likely to be written for a specific application than as a general-purpose utility. How much can a general-purpose utility know at compile-time? The number of arguments it has been given, and perhaps some of their values. If we want to use other constraints, they will probably have to be ones imposed by individual programs.

As an example, this section shows how macros can speed up the generation of Bezier curves. Curves must be generated fast if they are being manipulated interactively. It turns out that if the number of segments in the curve is known

beforehand, most of the computation can be done at compile-time. By writing our curve-generator as a macro, we can weave precomputed values right into code. This should be even faster than the more usual optimization of storing them in an array.

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:

$$x = (x_3 - 3x_2 + 3x_1 - x_0)u^3 + (3x_2 - 6x_1 + 3x_0)u^2 + (3x_1 - 3x_0)u + x_0$$

$$y = (y_3 - 3y_2 + 3y_1 - y_0)u^3 + (3y_2 - 6y_1 + 3y_0)u^2 + (3y_1 - 3y_0)u + y_0$$

If we evaluate these equations for n values of u between 0 and 1, we get n points on the curve. For example, if we want to draw the curve as 20 segments, then we would evaluate the equations for u = .05, .1,. . ., .95. There is no need to evaluate them for u of 0 or 1, because if u = 0 they will yield the first endpoint (x_0, y_0), and if u = 1 they will yield the second endpoint (x_3, y_3).

An obvious optimization is to make n fixed, calculate the powers of u beforehand, and store them in an $(n-1) \times 3$ array. By defining the curve-generator as a macro, we can do even better. If n is known at expansion-time, the program could simply expand into n line-drawing commands. The precomputed powers of u, instead of being stored in an array, could be inserted as literal values right into the macro expansion.

Figure 13.5 contains a curve-generating macro which implements this strategy. Instead of drawing lines immediately, it dumps the generated points into an array. When a curve is moving interactively, each instance has to be drawn twice: once to show it, and again to erase it before drawing the next. In the meantime, the points have to be saved somewhere.

With n = 20, `genbez` expands into 21 `setf`s. Since the powers of u appear directly in the code, we save the cost of looking them up at runtime, and the cost of computing them at startup. Like the powers of u, the array indices appear as constants in the expansion, so the bounds-checking for the `(setf aref)`s could also be done at compile-time.

13.3 Applications

Later chapters contain several other macros which use information available at compile-time. A good example is `if-match` (page 242). Pattern-matchers compare two sequences, possibly containing variables, to see if there is some way of assigning values to the variables which will make the two sequences equal. The

```
(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)))
           (setf (aref *pts* 0 0) ,gx0
                 (aref *pts* 0 1) ,gy0)
           ,@(map1-n #'(lambda (n)
                         (let* ((u (* n *du*))
                                (u^2 (* u u))
                                (u^3 (expt u 3)))
                           `(setf (aref *pts* ,n 0)
                                  (+ (* ax ,u^3)
                                     (* bx ,u^2)
                                     (* cx ,u)
                                     ,gx0)
                                  (aref *pts* ,n 1)
                                  (+ (* ay ,u^3)
                                     (* by ,u^2)
                                     (* cy ,u)
                                     ,gy0))))
                     (1- *segs*))
           (setf (aref *pts* *segs* 0) ,gx3
                 (aref *pts* *segs* 1) ,gy3))))))
```

Figure 13.5: Macro for generating Bezier curves.

design of `if-match` shows that if one of the sequences is known at compile-time, and only that one contains variables, then matching can be done more efficiently. Instead of comparing the two sequences at runtime and consing up lists to hold the variable bindings established in the process, we can have a macro generate code to perform the exact comparisons dictated by the known sequence, and can store the bindings in real Lisp variables.

The embedded languages described in Chapters 19–24 also, for the most part, take advantage of information available at compile-time. Since an embedded language is a compiler of sorts, it's only natural that it should use such information. As a general rule, the more elaborate the macro, the more constraints it imposes on its arguments, and the better your chances of using these constraints to generate efficient code.