第31章

实践：HTML生成库，编译器部分

现在你即将看到FOO编译器是如何工作的。编译器和解释器之间的主要区别在于，解释器处理程序并直接产生某些行为，这对于FOO解释器来说就是生成HTML；但编译器处理同样的程序却可以产生以其他语言表示的实现相同行为的代码。在FOO中，编译器是将FOO转译成Common Lisp代码的Common Lisp宏，因此它可以直接嵌入到Common Lisp程序中。编译器通常比解释器更有优势，这是因为编译过程是预先完成的，因此它们可以多花一点儿时间来优化它们生成的代码，使其更加高效。FOO编译器能够做到这点是因为它通过将字面文本尽可能地合并在一起，从而使用比解释器更少量的写入操作来输出同样的HTML。当编译器是一个Common Lisp宏时，你还可以获得额外的优势：让含有嵌入式Common Lisp代码的语言更容易被编译器理解，编译器只需将这些代码识别出来，并直接嵌入到其所生成代码的正确位置上就可以了。FOO编译器将利用这种能力。

31.1 编译器

编译器的基本架构包括3层。首先你将实现一个类`html-compiler`，它带有一个用来保存可调整的向量的槽，这种向量用来集聚那些代表在执行`process`函数的过程中调用后台接口广义函数的那些操作码（op）。

随后你将实现后台接口中广义函数上的方法，在向量中保存操作序列。每一个op被表示成一个列表，包括一个命名了该操作的关键字和传递给产生该op的函数的参数。函数`sexp->ops`实现了编译器的第一阶段，在一个FOO形式列表的每个形式上调用带有`html-compiler`实例的`process`函数来编译该列表。

这个编译器保存的op向量随后被传给一个用来优化它的函数，将相邻的`raw-string` op合并成一次性输出的字符串组合的单个op。该优化函数也可以将那些只用于美化打印的op提取出来，这在多数时候会很重要，因为它使你得以合并更多的`raw-string` op。

最后，优化后的op向量被传递给第3个函数`generate-code`，它返回一个实际输出HTML的Common Lisp表达式的列表。当`*pretty*`为真时，`generate-code`生成使用特化在`html-pretty-printer`上的方法的代码来输出美化的HTML；而当`*pretty*`为`NIL`时，它生成

直接向流*html-output*写入的代码。

宏html实际上会生成一个含有两个展开式的主体，一个是在*pretty*绑定到T时生成的，而另一个是在*pretty*绑定到NIL时生成的。究竟使用哪个展开式取决于*pretty*在运行期的值。这样，每个含有对html宏调用的函数都将同时含有生成美化和紧凑输出的代码。

编译器和解释器之间的另一个明显区别是，编译器可以在它生成的代码中嵌入Lisp形式。为了利用这点，你需要修改process函数，使其在处理一个并非FOO形式的表达式时可以调用embed-code和embed-value函数。由于所有的自求值对象都是合法的FOO形式，不需要传递给process-sexp-html的形式只有那些不匹配FOO点对形式语法的列表和非关键字的符号，即唯一的非自求值原子。你可以假设任何非FOO的点对都是用来内联运行的代码，而所有的符号都是其值打算嵌入的变量。

```
(defun process (processor form)
  (cond
    ((sexp-html-p form) (process-sexp-html processor form))
    ((consp form)       (embed-code processor form))
    (t                  (embed-value processor form))))
```

现在让我们查看编译器的代码。首先你应当创建两个函数，它们略微地抽象了将在编译的前两阶段用来保存op的向量。

```
(defun make-op-buffer () (make-array 10 :adjustable t :fill-pointer 0))

(defun push-op (op ops-buffer) (vector-push-extend op ops-buffer))
```

接下来你可以定义html-compiler类和特化在其上的方法，以实现后台接口。

```
(defclass html-compiler ()
  ((ops :accessor ops :initform (make-op-buffer))))

(defmethod raw-string ((compiler html-compiler) string &optional newlines-p)
  (push-op `(:raw-string ,string ,newlines-p) (ops compiler)))

(defmethod newline ((compiler html-compiler))
  (push-op '(:newline) (ops compiler)))

(defmethod freshline ((compiler html-compiler))
  (push-op '(:freshline) (ops compiler)))

(defmethod indent ((compiler html-compiler))
  (push-op `(:indent) (ops compiler)))

(defmethod unindent ((compiler html-compiler))
  (push-op `(:unindent) (ops compiler)))

(defmethod toggle-indenting ((compiler html-compiler))
  (push-op `(:toggle-indenting) (ops compiler)))

(defmethod embed-value ((compiler html-compiler) value)
  (push-op `(:embed-value ,value ,*escapes*) (ops compiler)))

(defmethod embed-code ((compiler html-compiler) code)
  (push-op `(:embed-code ,code) (ops compiler)))
```

有了这些方法，你就可以实现编译器的第一阶段sexp->ops了。

```
defun sexp->ops (body)
  (loop with compiler = (make-instance 'html-compiler)
     for form in body do (process compiler form)
     finally (return (ops compiler))))
```

在目前的阶段里你不需要担心*pretty*的值，只需记录下被process调用的所有函数即可。下面是sexp->ops从一个简单的FOO形式中产生的结果：

```
HTML> (sexp->ops '((:p "Foo")))
#((:FRESHLINE) (:RAW-STRING "<p" NIL) (:RAW-STRING ">" NIL)
  (:RAW-STRING "Foo" T) (:RAW-STRING "</p>" NIL) (:FRESHLINE))
```

下一个阶段optimize-static-output接受一个op的向量并返回一个含有优化后版本的新向量。对于每个:raw-string op，算法是简单的，它将字符串写入到一个临时字符串缓冲区里。这样，相邻的:raw-string op将把需输出的字符串拼接成单个字符串。一旦遇到了一个不是:raw-string的op，你就将目前位置构造的字符串通过助手函数compile-buffer转化成一个交替出现:raw-string和:newline两个op的序列，然后再添加下一个op。这个函数也负责在*pretty*为**NIL**时清除美化打印的op。

```
(defun optimize-static-output (ops)
  (let ((new-ops (make-op-buffer)))
    (with-output-to-string (buf)
      (flet ((add-op (op)
               (compile-buffer buf new-ops)
               (push-op op new-ops)))
        (loop for op across ops do
             (ecase (first op)
               (:raw-string (write-sequence (second op) buf))
               ((:newline :embed-value :embed-code) (add-op op))
               ((:indent :unindent :freshline :toggle-indenting)
                (when *pretty* (add-op op)))))
        (compile-buffer buf new-ops)))
    new-ops))
(defun compile-buffer (buf ops)
  (loop with str = (get-output-stream-string buf)
     for start = 0 then (1+ pos)
     for pos = (position #\Newline str :start start)
     when (< start (length str))
     do (push-op `(:raw-string ,(subseq str start pos) nil) ops)
     when pos do (push-op '(:newline) ops)
     while pos))
```

最后一步是将这些op转化成相应的Common Lisp代码。这一阶段也会关注*pretty*的值。当*pretty*为真时，它生成在*html-pretty-printer*上调用后台广义函数的代码，该变量绑定在html-pretty-printer上。当*pretty*为**NIL**时，它生成的代码将直接写入*html-output*，后者就是美化打印器用来发送输出的那个流。

实际的函数generate-code很简单。

```
(defun generate-code (ops)
  (loop for op across ops collect (apply #'op->code op)))
```

所有的工作都是由广义函数op->code上的方法来完成的，它们全部使用op名的EQL特化符特化在了参数op上。

```
(defgeneric op->code (op &rest operands))

(defmethod op->code ((op (eql :raw-string)) &rest operands)
  (destructuring-bind (string check-for-newlines) operands
    (if *pretty*
      `(raw-string *html-pretty-printer* ,string ,check-for-newlines)
      `(write-sequence ,string *html-output*))))

(defmethod op->code ((op (eql :newline)) &rest operands)
  (if *pretty*
    `(newline *html-pretty-printer*)
    `(write-char #\Newline *html-output*)))

(defmethod op->code ((op (eql :freshline)) &rest operands)
  (if *pretty*
    `(freshline *html-pretty-printer*)
    (error "Bad op when not pretty-printing: ~a" op)))

(defmethod op->code ((op (eql :indent)) &rest operands)
  (if *pretty*
    `(indent *html-pretty-printer*)
    (error "Bad op when not pretty-printing: ~a" op)))

(defmethod op->code ((op (eql :unindent)) &rest operands)
  (if *pretty*
    `(unindent *html-pretty-printer*)
    (error "Bad op when not pretty-printing: ~a" op)))

(defmethod op->code ((op (eql :toggle-indenting)) &rest operands)
  (if *pretty*
    `(toggle-indenting *html-pretty-printer*)
    (error "Bad op when not pretty-printing: ~a" op)))
```

其中两个最有趣的op->code方法是用来为:embed-value和:embed-code这两个op生成代码的方法。在:embed-value方法中，可以根据escapes操作数的值来生成稍有不同的代码，因为当escapes为NIL时你不需要生成对escape的调用。而当*pretty*和escapes均为NIL时，你可以生成使用PRINC来直接向流中输出值的代码。

```
(defmethod op->code ((op (eql :embed-value)) &rest operands)
  (destructuring-bind (value escapes) operands
    (if *pretty*
      (if escapes
        `(raw-string
          *html-pretty-printer* (escape (princ-to-string ,value) ,escapes) t)
        `(raw-string *html-pretty-printer* (princ-to-string ,value) t))
      (if escapes
        `(write-sequence (escape (princ-to-string ,value) ,escapes) *html-output*)
        `(princ ,value *html-output*)))))
```

这样，类似下面的代码

```
HTML> (let ((x 10)) (html (:p x)))
<p>10</p>
NIL
```

就能够正常工作了，因为html将(:p x)转译成了类似下面的形式：

```
(progn
  (write-sequence "<p>" *html-output*)
  (write-sequence (escape (princ-to-string x) "<>&") *html-output*)
  (write-sequence "</p>" *html-output*))
```

当上述代码在**LET**上下文中代替了对html的调用时，你可以得到下面的代码：

```
(let ((x 10))
  (progn
    (write-sequence "<p>" *html-output*)
    (write-sequence (escape (princ-to-string x) "<>&") *html-output*)
    (write-sequence "</p>" *html-output*)))
```

并且在生成的代码中，对x的引用将变成一个对来自html形式外围**LET**的词法变量的引用。

另一方面，:embed-code方法有趣是因为它是如此地简单。process传递形式到embed-code，后者又将其插入到:embed-code op中，因此你只需再把它拉出来并返回。

```
(defmethod op->code ((op (eql :embed-code)) &rest operands)
  (first operands))
```

这让类似下面的代码得以工作：

```
HTML> (html (:ul (dolist (x '(foo bar baz)) (html (:li x)))))
<ul>
  <li>FOO</li>
  <li>BAR</li>
  <li>BAZ</li>
</ul>
NIL
```

其中外层的html调用可以展开成类似下面的形式：

```
(progn
  (write-sequence "<ul>" *html-output*)
  (dolist (x '(foo bar baz)) (html (:li x)))
  (write-sequence "</ul>" *html-output*)))
```

如果你在**DOLIST**的主体中展开对html的调用，那么你将得到类似下面的东西：

```
(progn
  (write-sequence "<ul>" *html-output*)
  (dolist (x '(foo bar baz))
    (progn
      (write-sequence "<li>" *html-output*)
      (write-sequence (escape (princ-to-string x) "<>&") *html-output*)
      (write-sequence "</li>" *html-output*)))
  (write-sequence "</ul>" *html-output*))
```

事实上，这些代码将会生成你所见过的输出。

31.2 FOO特殊操作符

你可以就此打住，FOO语言的表达性已经足够用来生成你所关心的几乎任何HTML了。尽管如此，你还可以为该语言添加两个特性，只需要再写一点儿代码就可以使其更加强大：特殊操作符和宏。

FOO中的特殊操作符类似于Common Lisp中的特殊操作符。特殊操作符可用来在语言中表达那些无法通过语言的基本求值规则来表达的事物。或者从另一个角度来看，特殊操作符提供了访问语言求值器所使用的底层机制的能力。[①]

举一个简单的例子，在FOO编译器中，语言求值器使用embed-value函数来生成代码，把变量的值嵌入到输出的HTML中。不过，由于传递给embed-value的只是符号，无法在目前所描述的语言里嵌入任意Common Lisp表达式的值。process函数将点对单元传给了embed-code而非embed-value，因此返回的值被忽略了。通常这就是你所需要的东西，因为在FOO程序中嵌入Lisp代码的主要原因是使用Lisp的控制构造。不过，有时你也希望在生成的HTML中嵌入计算后的值。例如，你可能希望下面的FOO程序可以生成一个含有随机数的段落标签：

```
(:p (random 10))
```

但这样不行，因为代码会运行，然后值被丢掉了。

```
HTML> (html (:p (random 10)))
<p></p>
NIL
```

在目前你所实现的语言中，可以通过在html的外面计算该值，然后再通过一个变量嵌入它，从而绕过这个限制。

```
HTML> (let ((x (random 10))) (html (:p x)))
<p>1</p>
NIL
```

但这样做很麻烦，尤其是当你考虑把(random 10)传递给embed-value而非embed-code时，最初的形式刚好可以完成你想要做的事。因此，你可以定义一个特殊操作符:print，它被FOO语言处理器按照一个和正常FOO表达式不同的规则来处理。确切地说，不是输出一个<print>元素，而是将其主体中的形式传给embed-value。这样，你就可以像下面这样生成一个含有随机数的段落：

```
HTML> (html (:p (:print (random 10))))
<p>9</p>
NIL
```

很明显，这个特殊操作符只在编译的FOO代码中才有用，因为embed-value不能工作在解释器中。另一个可以同时用在解释和编译的FOO代码中的特殊操作符是:format，它让你使用**FORMAT**函数来产生输出。特殊操作符:format的参数是一个用作格式控制字符串的字符串和任

① 对于FOO特殊操作符和宏之间的相似之处，我将在31.3节里讨论到，这与Lisp本身的情况是一样的。事实上，理解了FOO特殊操作符和宏的工作方式，也有助于让你理解Common Lisp的有关设计理念。

何需要插入的参数。当所有:format的参数都是自求值对象时，通过将它们传递给**FORMAT**而生成出一个字符串，并且随后像其他任何字符串那样输出该字符串。这允许:format形式被用在传递给emit-html的FOO中。在编译过的FOO中，:format的参数可以是任意Lisp表达式。

其他特殊操作符控制了哪些字符被自动转义以及显式输出换行：:noescape特殊操作符使其主体中的所有形式作为正常FOO形式来求值，除了*escapes*绑定到**NIL**，:attribute可以在*escapes*绑定为*attribute-escapes*的情况下求值其主体中的Lisp形式。而:newline则被转译成输出一个显式换行的代码。

那么，你怎样才能定义特殊操作符呢？对特殊操作符的处理有两个方面：语言处理器如何识别那些使用了特殊操作符的形式，以及在处理每个特殊操作符的时候如何得知需要运行哪些代码？

你可以修改process-sexp-html来识别每一个特殊操作符，并用适当的方法来处理它们——特殊操作符在逻辑上是语言实现的一部分，并且它们不会有太多。不过，如果有更加模块化的方法来添加新的特殊操作符就更好了，这并不是为了方便FOO的用户而只是为了方便你自己。

我们将"特殊形式"定义为任何这样的列表，其**CAR**是一个特殊操作符名字的符号。你可以通过将一个非NIL值添加到该符号属性表中关键字html-special-operator下以标记特殊操作符的名字。因此，可以像下面这样定义函数来测试一个给定形式是否为特殊形式：

```
(defun special-form-p (form)
  (and (consp form) (symbolp (car form)) (get (car form) 'html-special-operator)))
```

实现每个特殊操作符的代码负责按照它们需要的方式提取该列表的其余部分，并按照特殊操作符所要求的语义来做事。假设你定义一个函数process-special-form，它接受语言处理器和一个特殊形式，并运行适当的代码来生成处理器对象上的一个调用序列，那么你可以修订顶层的process函数，像下面这样来处理特殊形式：

```
(defun process (processor form)
  (cond
    ((special-form-p form) (process-special-form processor form))
    ((sexp-html-p form)    (process-sexp-html processor form))
    ((consp form)          (embed-code processor form))
    (t                     (embed-value processor form))))
```

必须将special-form-p子句放在最前面，因为特殊形式可能看起来在词法上跟正常的FOO表达式一样，就好像Common Lisp的特殊形式也可能看起来像正常函数调用一样。

现在你只需实现process-special-form就好了。不用去定义实现了所有特殊操作符的单个函数，而应当转而定义一个宏，允许你像正规函数一样地定义特殊操作符，并负责在特殊操作符的名字的属性表中添加html-special-operator项。事实上，你保存在属性表中的值可以是一个实现该特殊操作符的函数。下面就是这个宏：

```
(defmacro define-html-special-operator (name (processor &rest other-parameters)
                                        &body body)
  `(eval-when (:compile-toplevel :load-toplevel :execute)
     (setf (get ',name 'html-special-operator)
           (lambda (,processor ,@other-parameters) ,@body))))
```

这是一个相当高级的宏类型，但如果你逐行地观察它就会发现，其实也没有什么特别的。为

了了解它的工作方式，我们取该宏的一个简单用例，即特殊操作符:noescape的定义，然后查看其宏展开式。如果你写出了下面的定义：

```
(define-html-special-operator :noescape (processor &rest body)
  (let ((*escapes* nil))
    (loop for exp in body do (process processor exp))))
```

那么它相当于下面的写法：

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (setf (get ':noescape 'html-special-operator)
        (lambda (processor &rest body)
          (let ((*escapes* nil))
            (loop for exp in body do (process processor exp))))))
```

在第20章里讨论过，**EVAL-WHEN**特殊操作符确保了当你用**COMPILE-FILE**编译时，其主体中的代码产生的效果在编译期可见。如果你想要在一个文件中使用define-html-special-operator，然后在同样的文件中使用刚刚定义的特殊操作符的话，那么这个EVAL-WHEN将是必要的。

随后的**SETF**表达式将符号:noescape的属性html-special-operator设置成一个匿名函数，其参数列表与define-html-special-operator中指定的参数列表相同。通过将define-html-special-operator定义成参数列表分成两部分的形式，即processor和其余的部分，你可以确保所有的特殊操作符都至少接受一个参数。

该匿名函数的主体就是提供给define-html-special-operator的主体。该匿名函数的任务是通过在后台接口上作适当的调用来生成正确的HTML，或生成可以生成它们的代码，从而实现该特殊操作符。它还可以使用process来求值一个作为FOO形式的表达式。

特殊操作符:noescape尤其简单，它所做的全部就是在*escapes*绑定到**NIL**的情况下，将其主体中的Lisp形式传递给process。换句话说，这个特殊操作符禁止了由process-sexp-html进行的正常字符转义。

有了以这种方式定义的特殊操作符，process-special-form所要做的就是查询特殊操作符名字的属性表中的匿名函数，并将其应用在处理器和Lisp形式的其余部分上。

```
(defun process-special-form (processor form)
  (apply (get (car form) 'html-special-operator) processor (rest form)))
```

现在你可以开始定义其余的5个FOO特殊操作符了。与:noescape类似的是:attribute，它在*escapes*绑定到*attribute-escapes*的情况下对其主体中的Lisp形式求值。这个特殊操作符在你想要编写用来输出属性值的助手函数时将会很有用。如果你编写一个类似下面的函数：

```
(defun foo-value (something)
  (html (:print (frob something))))
```

其中的html宏用来生成在*element-escapes*中转义字符的代码。但如果你正在计划像下面这样来使用foo-value：

```
(html (:p :style (foo-value 42) "Foo"))
```

那么你会希望它可以生成使用*attribute-escapes*的代码。因此，你可以另行像下面这样来编写该函数：[1]

```
(defun foo-value (something)
  (html (:attribute (:print (frob something)))))
```

:attribute的定义如下所示：

```
(define-html-special-operator :attribute (processor &rest body)
  (let ((*escapes* *attribute-escapes*))
    (loop for exp in body do (process processor exp))))
```

下面两个特殊操作符:print和:format用来输出值。早先讨论过，特殊操作符:print用于在编译过的FOO程序里嵌入任意Lisp表达式的值，差不多等价于先用(format nil ...)生成一个字符串然后再嵌入它。将:format定义为特殊操作符主要是出于方便的考虑。

```
(:format "Foo: ~d" x)
```

比下面这个更好一些：

```
(:print (format nil "Foo: ~d" x))
```

它还有另外一点优势，如果你将:format与全部是自求值的参数一起使用，那么FOO可以在编译期求值:format而无需等到运行期再做。:print和:format的定义如下所示：

```
(define-html-special-operator :print (processor form)
  (cond
    ((self-evaluating-p form)
     (warn "Redundant :print of self-evaluating form ~s" form)
     (process-sexp-html processor form))
    (t
     (embed-value processor form))))

(define-html-special-operator :format (processor &rest args)
  (if (every #'self-evaluating-p args)
    (process-sexp-html processor (apply #'format nil args))
    (embed-value processor `(format nil ,@args))))
```

特殊操作符:newline强制输出一个字面换行，这有时是有用的。

```
(define-html-special-operator :newline (processor)
  (newline processor))
```

最后，特殊操作符:progn和Common Lisp中的PROGN特殊操作符相似。它简单地按顺序处理其主体中的Lisp形式。

```
(define-html-special-operator :progn (processor &rest body)
  (loop for exp in body do (process processor exp)))
```

换句话说，下面的形式

```
(html (:p (:progn "Foo " (:i "bar") " baz")))
```

---

① :noescape和:attribute特殊操作符必须被定义成特殊操作符，这是因为FOO是在编译期决定如何转义的，而不是运行期。这允许FOO在编译期转义字面值，从而比在运行期扫描所有输出要高效得多。

将生成与下面形式相同的代码：

```
(html (:p "Foo " (:i "bar") " baz"))
```

这可能看起来是个奇怪的需要，因为正常的FOO表达式可以在其主体中有任意多个形式。不过，这个特殊操作符将在一种情形里变得非常有用——当编写FOO宏的时候，这将把你带到你需要实现的最后一个语言特性上。

31.3 FOO宏

FOO宏类似于Common Lisp宏。FOO宏是一点儿代码，它接受FOO表达式作为参数并返回一个新的FOO表达式作为结果，后者随后按照正常的FOO求值规则来求值。实际的实现与特殊操作符的实现非常相似。

和特殊操作符一样，你可以定义一个谓词来测试给定Lisp形式是否是一个宏形式。

```
(defun macro-form-p (form)
  (cons-form-p form #'(lambda (x) (and (symbolp x) (get x 'html-macro)))))
```

使用前面定义的函数cons-form-p，是因为你想要允许宏被用在所有非宏的FOO点对形式语法中。不过，你需要传递一个不同的谓词函数，它用来测试形式名是否是一个带有非NIL的html-macro属性的符号。另外，和特殊操作符的实现一样，你将定义一个宏用来定义FOO宏，它负责将一个函数保存在宏名的属性表中，位于关键字html-macro之下。不过，定义一个宏的过程会更加复杂一些，因为FOO支持两种类型的宏。一些你将定义的宏类似于正常的HTML元素，并且可能想要容易地访问一个属性列表。其他的宏只是简单地想要直接访问它们主体的元素。

你可以使这两种类型的宏的区别变得隐性：当你定义一个FOO宏时，参数列表可以包含一个&attributes参数。如果有这个参数的话，那么宏形式将按照正常点对形式来解析，并且宏函数将被传递两个值，一个属性的plist和一个构成宏形式体的表达式列表。没有&attributes参数的宏形式将不会解析属性，并且宏函数将被应用在单一参数上，即一个含有主体表达式的列表。前者对于本质上的HTML模板很有用。例如：

```
(define-html-macro :mytag (&attributes attrs &body body)
  `((:div :class "mytag" ,@attrs) ,@body))

HTML> (html (:mytag "Foo"))
<div class='mytag'>Foo</div>
NIL
HTML> (html (:mytag :id "bar" "Foo"))
<div class='mytag' id='bar'>Foo</div>
NIL
HTML> (html ((:mytag :id "bar") "Foo"))
<div class='mytag' id='bar'>Foo</div>
NIL
```

后一种类型的宏对于编写管理其主体中Lisp形式的宏更加有用。这个类型的宏可以作为一种HTML控制构造类型来使用。作为一个简单的例子，考虑下面这个实现了:if构造的宏：

```
(define-html-macro :if (test then else)
  `(if ,test (html ,then) (html ,else)))
```

该宏允许你写出下面的代码：

```
(:p (:if (zerop (random 2)) "Heads" "Tails"))
```

而不必写成下面这个较长的版本：

```
(:p (if (zerop (random 2)) (html "Heads") (html "Tails")))
```

为了决定你应该生成哪种类型的宏，你需要一个函数来解析define-html-macro的参数列表。该函数返回两个值，&attributes参数的名字或者其不存在时为**NIL**，以及一个含有args中去掉&attributes标记及其后续列表元素后剩下的所有元素的列表。[①]

```
(defun parse-html-macro-lambda-list (args)
  (let ((attr-cons (member '&attributes args)))
    (values
     (cadr attr-cons)
     (nconc (ldiff args attr-cons) (cddr attr-cons)))))

HTML> (parse-html-macro-lambda-list '(a b c))
NIL
(A B C)
HTML> (parse-html-macro-lambda-list '(&attributes attrs a b c))
ATTRS
(A B C)
HTML> (parse-html-macro-lambda-list '(a b c &attributes attrs))
ATTRS
(A B C)
```

参数列表中&attributes后面跟着的元素也可以是一个解构参数列表。

```
HTML> (parse-html-macro-lambda-list '(&attributes (&key x y) a b c))
(&KEY X Y)
(A B C)
```

现在你可以开始定义define-html-macro了。根据是否指定了&attributes参数，你需要生成HTML宏的两种形式之一，因此主宏简单地检测其正在定义哪种类型的HTML宏，并随后调用一个助手函数来生成正确类型的代码。

```
(defmacro define-html-macro (name (&rest args) &body body)
  (multiple-value-bind (attribute-var args)
      (parse-html-macro-lambda-list args)
    (if attribute-var
      (generate-macro-with-attributes name attribute-var args body)
      (generate-macro-no-attributes name args body))))
```

实际生成展开式的函数如下所示：

```
(defun generate-macro-with-attributes (name attribute-args args body)
  (with-gensyms (attributes form-body)
    (if (symbolp attribute-args) (setf attribute-args `(&rest ,attribute-args)))
```

① 注意&attributes只不过是另一个符号罢了，以“&”开头的名字本质上没有什么特别之处。

```
    `(eval-when (:compile-toplevel :load-toplevel :execute)
       (setf (get ',name 'html-macro-wants-attributes) t)
       (setf (get ',name 'html-macro)
             (lambda (,attributes ,form-body)
               (destructuring-bind (,@attribute-args) ,attributes
                 (destructuring-bind (,@args) ,form-body
                   ,@body)))))))

(defun generate-macro-no-attributes (name args body)
  (with-gensyms (form-body)
    `(eval-when (:compile-toplevel :load-toplevel :execute)
       (setf (get ',name 'html-macro-wants-attributes) nil)
       (setf (get ',name 'html-macro)
             (lambda (,form-body)
               (destructuring-bind (,@args) ,form-body ,@body))))))
```

你将定义的宏函数接受一个或两个参数，然后使用**DESTRUCTURING-BIND**来将参数提取出来并绑定到在对define-html-macro的调用中所定义的参数上。在两个展开式中，你都需要将宏函数保存在其名字的属性表中的html-macro之下，并且在属性html-macro-wants-attributes下保存一个布尔值，以指示该宏是否接受&attributes参数。你在下面的函数expand-macro-form中使用该属性来决定宏函数被调用的方式：

```
(defun expand-macro-form (form)
  (if (or (consp (first form))
          (get (first form) 'html-macro-wants-attributes))
    (multiple-value-bind (tag attributes body) (parse-cons-form form)
      (funcall (get tag 'html-macro) attributes body))
    (destructuring-bind (tag &body body) form
      (funcall (get tag 'html-macro) body))))
```

最后一步是通过在顶层process函数的派发**COND**语句里添加一个子句来集成对宏的支持。

```
(defun process (processor form)
  (cond
    ((special-form-p form) (process-special-form processor form))
    ((macro-form-p form)   (process processor (expand-macro-form form)))
    ((sexp-html-p form)    (process-sexp-html processor form))
    ((consp form)          (embed-code processor form))
    (t                     (embed-value processor form))))
```

这就是process的最终版本。

31.4 公共 API

现在，你终于完成了对html宏的实现，它就是FOO编译器的主入口点。FOO公共API的其余部分还包括我在上一章里讨论过的emit-html和with-html-ouput，以及上一节里讨论过的define-html-macro。define-html-maco之所以需要成为公共API的一部分，是因为FOO的用户也希望编写自己的HTML宏。另一方面，define-html-special-operator不是公共API一部分的理由，是它需要太多的关于FOO内部的知识来定义一个新的特殊操作符。并且应该几乎

没有什么功能是无法使用已有的语言和特殊操作符来完成的。[1]

在我到达html之前，公共API的最后一个元素是另一个宏in-html-style。该宏通过设置*xhtml*变量来控制FOO生成XHTML还是正常的HTML。将其定义为宏的原因是你希望把设置*xhtml*的代码包装在**EVAL-WHEN**中，这样就可以在一个文件里设置它，并让其影响同一个文件中的所有对html宏后续的使用。

```
(defmacro in-html-style (syntax)
  (eval-when (:compile-toplevel :load-toplevel :execute)
    (case syntax
      (:html (setf *xhtml* nil))
      (:xhtml (setf *xhtml* t)))))
```

最后让我们来查看html宏本身。实现html的唯一难点是必须生成那种可同时生成美观和紧凑输出的代码，具体取决于变量*pretty*在运行期的值。这样，html需要生成一个含有**IF**表达式和两个版本代码的展开式，一个在*pretty*绑定为真时编译，另一个在它绑定为**NIL**时编译。更复杂的是，html调用经常会含有嵌入的html调用，就像这样：

```
(html (:ul (dolist (item stuff)) (html (:li item))))
```

如果外层的html展开成一个带有两个版本代码的**IF**表达式，一个用在*pretty*为真时，另一个用在其为假时，那么再把内嵌的html形式也展开成两个版本的话就太傻了。事实上，这将导致代码量呈指数级增长，因为内嵌的html也将展开两次——一次是在*pretty*为真的分支里，另一次是在*pretty*为假的分支里。如果每个展开式都生成两个版本，那么你总共有4个版本。而如果这个内嵌的html形式还含有另一个内嵌的html形式，那么你最后将得到该代码的8个版本。如果编译器足够聪明，它最终会意识到其生成的多数代码都是没用的，并将清除它们，但即便找出这点也需要相当多的时间，从而减慢了任何使用嵌套html调用的函数的编译时间。

幸运的是，你可以通过生成一个用**MACROLET**局部重定义html宏的展开式，让其只生成正确类型的代码，从而避免无用代码的爆炸。首先你定义一个助手函数，它接受由sexp->ops返回的op向量并在*pretty*绑定为指定值的情况下对其应用optimize-static-output和generate-code，即受*pretty*影响的两个阶段，然后再将得到的结果插入到**PROGN**中。（**PROGN**返回**NIL**是为了让输出更简洁。）

```
(defun codegen-html (ops pretty)
  (let ((*pretty* pretty))
    `(progn ,@(generate-code (optimize-static-output ops)) nil)))
```

有了这个函数，你随后就可以像下面这样来定义html：

```
(defmacro html (&whole whole &body body)
  (declare (ignore body))
  `(if *pretty*
     (macrolet ((html (&body body) (codegen-html (sexp->ops body) t)))
```

① 在底层的语言处理基础设施里，到目前为止，还没有充分地通过特殊操作符暴露出来的一个元素是对缩进的处理。如果你想要令FOO更加灵活（尽管这要以增加其API的复杂度为代价），那么你可以添加用于管理底层缩进打印器的特殊操作符。不过看起来解释额外的特殊操作符的代价将远远超出在语言表达性上获得的微小提升。

```
    (let ((*html-pretty-printer* (get-pretty-printer)))
      ,whole))
  (macrolet ((html (&body body) (codegen-html (sexp->ops body) nil)))
    ,whole)))
```

其中的**&whole**参数代表最初的html形式，而由于它被插入到两个**MACROLET**主体的展开式中，它将使用每个html的新定义重新处理，一个生成美化打印的代码，另一个生成非美化打印的代码。注意变量*pretty*被同时用于宏展开期和结果代码运行期。在宏展开期，它被codegen-html用来使generate-code生成一种或另一种类型的代码。而在运行期，它被顶层html宏生成的IF表达式用来决定实际上应该运行美化打印还是非美化打印的代码。

31.5 结束语

和往常一样，你可以继续研究这些代码，以不同的方式来增强它。一个有趣的方向是使用底层的输出框架来产生其他类型的输出。从本书Web站点上下载的FOO版本中，你将找到一些实现了CSS输出的代码，它们可在解释器和编译器中与HTML输出集成在一起。这是一个有趣的案例，因为CSS的语法不能像HTML那样简单地映射到S-表达式上。不过，如果你实际查看那些代码，将看到定义一种S-表达式语法来表示CSS中的多种结构仍然是可能的。

一项更具雄心的底层处理是添加对生成嵌入式JavaScript的支持。如果做法得当，那么为FOO添加JavaScript支持可以得到两大好处。一个好处是，在定义出一种可映射到JavaScript语法的S-表达式语法以后，你就可以开始编写Common Lisp的宏，为你用来编写客户端代码的语言添加新的构造，然后再将它们编译成JavaScript。另一个好处是，作为从FOO的S-表达式JavaScript到正常JavaScript转换的一部分，你可以轻松处理不同浏览器的JavaScript实现中的那些细微但令人讨厌的区别。这就是说，FOO生成的JavaScript代码要么可以含有适当的条件代码，在不同的浏览器里做不同的事，要么直接根据你想要支持的浏览器来生成不同的代码。然后如果你把FOO用在动态生成的页面中，它可以使用来自User-Agent中的信息来让请求生成针对该浏览器的JavaScript代码。

如果这些事情激发了你的兴趣，那么你应当自行去实现它们，因为这已经是本书实践性内容的最后一章了。在下一章里我将做个总结，同时简要讨论一些我尚未在本书涉及的主题，包括如何查找第三方库，如何优化Common Lisp代码，以及如何交付Lisp应用程序。