第30章

实践：HTML生成库，解释器部分

在本章和下一章里，你将审视过去几章里用到的HTML生成器FOO的底层实现。FOO是Common Lisp中相当普遍但在非Lisp语言，即“面向语言”的编程里相对罕见的编程类型的一个示例。相比在函数、类和宏的基础上构建出来的API，FOO提供了一个可以嵌入到你的Common Lisp程序中的领域相关语言。

FOO提供了用于同样S-表达式语言的两个语言处理器。一个是解释器，它接受作为数据的一个FOO“程序”并解释它来生成HTML。另一个是编译器，它将可能嵌入在Common Lisp代码中的FOO表达式编译成生成HTML的Common Lisp代码，并执行这些嵌入的代码。解释器是以函数`emit-html`的形式出现的，而编译器是作为宏`html`提供的，你在前面的章节里用到过它们。

在本章，你将首先认识一些在解释器和编译器之间共享的基础设施，然后了解解释器的实现。在下一章里，我将向你展示编译器是如何工作的。

30.1 设计一个领域相关语言

设计嵌入式语言需要两个步骤：首先，设计一门可以让你表达想要表达的事物的语言；其次，实现一个或几个处理器，接受一个以该语言表达的“程序”，然后要么进行该语言所表达的操作，要么将该程序转译成具有等价行为的Common Lisp代码。

因此，第一步是设计HTML生成语言。设计好的领域相关语言的关键是在表达能力和简洁性之间找好平衡。举个例子，一个用来生成HTML的表达性强但不太简洁的“语言”是字面HTML字符串的语言。该语言的合法“形式”是那些含有字面HTML的字符串。该“语言”的语言处理器可以通过简单地原样输出它们来处理这些形式。


```
(defvar *html-output* *standard-output*)

(defun emit-html (html)
  "An interpreter for the literal HTML language."
  (write-sequence html *html-output*))

(defmacro html (html)
```

```
  "A compiler for the literal HTML language."
  `(write-sequence ,html *html-output*))
```

这个“语言”表达性很强，因为它可以表达“任何”你可能生成的HTML。[1]另一方面，该语言没有什么简洁性可言，因为它带给你的是零压缩率，它的输入就是输出。

为了设计一个可以给你一些有用的压缩但又不会牺牲太多表达性的语言，你需要识别输出的细节中那些重复和无关的部分。然后可以让输出的这些方面隐含在该语言的语义中。

例如，由于HTML的结构，每个开放的标签都有一个配对的闭合标签。[2]当你手工编写HTML时，就不得不书写这些闭合标签，但可以通过隐式地生成这些闭合标签从而改进你的HTML生成语言的简洁性。

另一种以牺牲一些表达性为代价来提高简洁性的方式是，让语言处理器负责在元素之间添加适当的空白，包括空行和缩进。当通过编程生成HTML时，你通常不太关心元素前后的换行，以及不同的元素相对于它们的父元素是否正确缩进了。让语言处理器根据一些规则来插入空白就意味着你不需要担心它们了。FOO事实上支持两种模式：一种使用最少量的空白，生成极其高效和紧凑的HTML；另一种可以生成格式良好的HTML，其中不同的元素根据它们的角色相对其他元素缩进或分离开。

另一个最好可以交给语言处理器来做的细节是对特定字符的转义，诸如<、>和&这些字符在HTML中有特殊的含义。显然，如果只是通过将字符串打印到一个流中来生成HTML的话，那么将由你来把字符串中出现的任何这类特殊字符替换成对应的转义序列“<”、“>”和“&”。但如果语言处理器知道哪些字符串将被输出成元素数据的话，那么它就可以帮你自动地转义这些字符。

30.2 FOO语言

理论已经讲得足够多了。下面我们来快速浏览一下FOO所实现的语言，然后你将看到两个FOO语言处理器的实现：本章是解释器，第31章是编译器。

和Lisp本身一样，FOO语言的基本语法由构成Lisp对象的Lisp形式定义而成。该语言定义了合法的FOO形式是如何被转化成HTML的。

最简单的FOO形式是诸如字符串、数字和关键字符号[3]这样的自求值Lisp对象。你将需要函数`self-evaluating-p`来测试一个给定对象在FOO的意义下是否是自求值的。

```
(defun self-evaluating-p (form)
  (and (atom form) (if (symbolp form) (keywordp form) t)))
```

---

① 事实上，它可能是表达性太强了，甚至可以生成那些不太合法的HTML输出。当然，如果你需要生成不十分正确的HTML用来补偿有bug的Web浏览器的话，这也可以成为一个特性。另外，语言处理器通常也会接受那些词法严谨并且形态良好，但运行起来以后却不可避免地产生未定义行为的程序。

② 好吧，其实是几乎所有标签。诸如IMG和BR等特定标签不是这样的。你将在30.7节里处理它们。

③ 在Common Lisp标准所描述的严格语言里，关键字符号不是自求值的，尽管它们在事实上确实求值到它们自身。参见语言标准的3.1.2.1.3节或HyperSpec里的简要讨论。

通过使用**PRINC-TO-STRING**，满足该谓词的对象将把它们转换成字符串，然后在转义任何诸如<、>或&这类保留字符后输出。当值作为属性输出时，字符“"”和“'”也需要转义。这样，你可以在一个自求值对象上调用html宏从而将其输出到*html-output*（初始绑定到***STANDARD-OUTPUT***）。表30-1给出了一些不同的自求值对象是如何被输出的。

表30-1 自求值对象的FOO输出

FOO 形式	生成的 HTML
"foo"	foo
10	10
:foo	FOO
"foo & bar"	foo & bar

当然，多数HTML都由带有标签的元素构成。用来描述每个元素的三部分信息分别是标签、属性集合以及含有文本和/或更多HTML元素的主体。这样，你需要一种方式将这三部分信息表示成Lisp对象，最好是Lisp读取器已经知道如何读取的对象。[①]如果你暂时不考虑属性的话，那么在Lisp列表和HTML元素之间存在一个明显的映射：任何HTML元素均可被表示成一个列表，其**FIRST**部分是一个名字与该元素的标签名相同的符号，而**REST**部分是一个代表其他HTML元素的由自求值对象或列表组成的列表。这样：

```
<p>Foo</p> ↔ (:p "Foo")

<p><i>Now</i> is the time</p> ↔ (:p (:i "Now") " is the time")
```

现在唯一的问题是在哪里插入属性。由于多数元素都没有属性，如果可以继续使用前面用于无属性元素的语法就最好了。FOO提供了两种方式来表示带有属性的元素。第一种是简单地将属性包含在列表中紧随符号之后的位置上，其中命名了属性的关键字符号和代表属性值形式的对象交替出现。元素的主体开始于列表中第一个在属性名的位置上却并非关键字符号的那一项。因此：

```
HTML> (html (:p "foo"))
<p>foo</p>
NIL
HTML> (html (:p "foo " (:i "bar") " baz"))
<p>foo <i>bar</i> baz</p>
NIL
HTML> (html (:p :style "foo" "Foo"))
<p style='foo'>Foo</p>
NIL
HTML> (html (:p :id "x" :style "foo" "Foo"))
<p id='x' style='foo'>Foo</p>
NIL
```

对于那些喜欢在元素的属性和主体间有更明确界限的人们，FOO还支持另一种语法：如果列表的第一个元素本身是一个以关键字符号为其首元素的列表，那么外层的列表就代表一个以该关

① 使用那些Lisp读取器知道如何读取的对象，这并不是一个十分严格的要求。由于Lisp读取器本身是可定制的，你还可以定义一个新的读取器层面的语法来处理新的对象类型。但这样做不值得，并且会带来更多麻烦。

键字为标签的HTML元素，嵌套列表的REST部分作为其属性，外层列表的REST部分作为其主体。这样，可以像下面这样书写前面两个表达式：

```
HTML> (html ((:p :style "foo") "Foo"))
<p style='foo'>Foo</p>
NIL
HTML> (html ((:p :id "x" :style "foo") "Foo"))
<p id='x' style='foo'>Foo</p>
NIL
```

下面的函数测试一个给定对象是否匹配这两种语法：

```
(defun cons-form-p (form &optional (test #'keywordp))
  (and (consp form)
       (or (funcall test (car form))
           (and (consp (car form)) (funcall test (caar form))))))
```

应当将test函数参数化，因为以后你需要在该名字上使用稍微不同的谓词来测试相同的两种语法。

为了完全地抽象掉两种语法变体之间的差异，你可以定义函数parse-cons-form，它接受一个形式并将其解析成3个元素：标签、属性列表和主体列表，然后以多值的形式返回它们。实际求值点对形式的代码将使用该函数而不担心它所采用的语法。

```
(defun parse-cons-form (sexp)
  (if (consp (first sexp))
    (parse-explicit-attributes-sexp sexp)
    (parse-implicit-attributes-sexp sexp)))

(defun parse-explicit-attributes-sexp (sexp)
  (destructuring-bind ((tag &rest attributes) &body body) sexp
    (values tag attributes body)))

(defun parse-implicit-attributes-sexp (sexp)
  (loop with tag = (first sexp)
     for rest on (rest sexp) by #'cddr
     while (and (keywordp (first rest)) (second rest))
     when (second rest)
       collect (first rest) into attributes and
       collect (second rest) into attributes
     end
     finally (return (values tag attributes rest))))
```

现在已经基本规范了语言，你可以考虑如何实际来实现语言的处理器了。怎样才能将一系列的FOO形式转化成你想要的HTML呢？前面提到，需要实现FOO的两个语言处理器：一个解释器负责遍历一棵FOO形式树并直接输出对应的HTML，一个编译器遍历一棵树并将其转化成可以输出同样HTML的Common Lisp代码。无论是解释器还是编译器都将构建在一个共同的代码基础之上，它会支持诸如转义保留字符和生成美观的缩进输出之类的特性，因此我们有理由从这里开始。

30.3 字符转义

需要依赖的首要基础设施是知道如何转义那些HTML中带有特殊含义字符的代码。有三个这样的字符一定不能出现在元素或属性值的文本中，它们是<、>和&。在元素文本或属性值中，这些字符必须被替换成字符引用项“<”、“>”和“&”。类似地，在属性值中，用来给值定界的引号必须被转义，把“'”变成“'”把“"”变成“"”。此外，任何字符都可被表示成一个数值的字符引用项，后者依次由&、#、一个以十进制数表示的数值代码，以及一个分号组成。这些数值转义项有时用来在HTML中嵌入非ASCII的字符。

包 定 义

由于FOO是一个底层库，你开发的这个包并不依赖很多外部代码。只有通常依赖的来自COMMON-LISP包的名字以及几乎同样经常依赖的来自COM.GIGAMONKEYS.MACRO-UTILITIES的宏生成宏的名字。另一方面，该包需要导出使用FOO的代码所需要的所有名字。下面是来自本书Web站点上下载的源代码中的**DEFPACKAGE**定义：

```
(defpackage :com.gigamonkeys.html
  (:use :common-lisp :com.gigamonkeys.macro-utilities)
  (:export :with-html-output
           :in-html-style
           :define-html-macro
           :html
           :emit-html
           :&attributes))
```

下面的函数接受单个字符并返回一个含有该字符的字符引用项的字符串：

```
(defun escape-char (char)
  (case char
    (#\& "&amp;")
    (#\< "&lt;")
    (#\> "&gt;")
    (#\' "&apos;")
    (#\" "&quot;")
    (t (format nil "&#~d;" (char-code char)))))
```

你可以使用该函数来作为函数escape的基础，escape函数接受一个字符串和一个字符序列，然后返回第一个参数的一个副本，其中所有在第二个参数中出现过的字符都被替换成由escape-char返回的对应的字符项。

```
(defun escape (in to-escape)
  (flet ((needs-escape-p (char) (find char to-escape)))
    (with-output-to-string (out)
      (loop for start = 0 then (1+ pos)
            for pos = (position-if #'needs-escape-p in :start start)
            do (write-sequence in out :start start :end pos)
            when pos do (write-sequence (escape-char (char in pos)) out)
            while pos))))
```

你还可以定义如下两个参数：*element-escapes*，它含有需要在正常元素数据中转义的所有字符；*attribute-escapes*，它含有在属性值中需要转义的字符集。

```
(defparameter *element-escapes* "<>&")
(defparameter *attribute-escapes* "<>&\"'")
```

下面是一些例子：

```
HTML> (escape "foo & bar" *element-escapes*)
"foo &amp; bar"
HTML> (escape "foo & 'bar'" *element-escapes*)
"foo &amp; 'bar'"
HTML> (escape "foo & 'bar'" *attribute-escapes*)
"foo &amp; &apos;bar&apos;"
```

最后，还需要一个变量*escapes*，它绑定到需要进行转义的字符集上。其初值设置成*element-escapes*的值，但是当生成属性时，它会被重新绑定在*attribute-escapes*的值上。

```
(defvar *escapes* *element-escapes*)
```

30.4　缩进打印器

为了生成精美缩进输出，你可以定义一个类indenting-printer，它包装在一个输出流的外围，再定义一些函数在使用该类的一个实例来将字符串输出到该流的同时跟踪何时开始新的一行。该类如下所示：

```
(defclass indenting-printer ()
  ((out                  :accessor out                  :initarg :out)
   (beginning-of-line-p :accessor beginning-of-line-p :initform t)
   (indentation          :accessor indentation          :initform 0)
   (indenting-p          :accessor indenting-p          :initform t)))
```

操作在indenting-printers上的主函数是emit，它接受该打印器和一个字符串，并将该字符串输出到打印器的输出流上，同时跟踪它何时输出一个换行以重置beginning-of-line-p槽。

```
(defun emit (ip string)
  (loop for start = 0 then (1+ pos)
     for pos = (position #\Newline string :start start)
     do (emit/no-newlines ip string :start start :end pos)
     when pos do (emit-newline ip)
     while pos))
```

为了实际输出该字符串，它用到了函数emit/no-newlines，后者通过助手函数indent-if-necessary来输出任何需要的缩进，然后再将字符串写入该流。该函数也会被其他用来输出一个确定不需要换行的字符串的代码所直接调用。

```
(defun emit/no-newlines (ip string &key (start 0) end)
  (indent-if-necessary ip)
  (write-sequence string (out ip) :start start :end end)
  (unless (zerop (- (or end (length string)) start))
    (setf (beginning-of-line-p ip) nil)))
```

助手函数indent-if-necessary通过检测beginning-of-line-p和indenting-p来决定是否需要输出缩进，以及当两者均为真时输出由indentation的值所指定数量的空格。使用indenting-printer的代码可以通过操作indentation和indenting-p槽来控制缩进。递增和递减indentation可以改变前导空格的数量，而将indenting-p设置为NIL可以临时关闭缩进。

```
(defun indent-if-necessary (ip)
  (when (and (beginning-of-line-p ip) (indenting-p ip))
    (loop repeat (indentation ip) do (write-char #\Space (out ip)))
    (setf (beginning-of-line-p ip) nil)))
```

indenting-printer API中的最后两个函数是emit-newline和emit-freshline，两者都用来输出一个换行符，类似于**FORMAT**指令~%和~&。这就是说，唯一的区别在于emit-newline总是输出一个换行，而emit-freshline则只有在beginning-of-line-p为假时才输出。这样，中间没有任何emit的多个emit-freshline调用将不会产生一个空行。在一些代码希望生成以换行结束的输出，而另一些代码希望生成以换行开始的输出，但你不希望两者之间产生空行时，这非常有用。

```
(defun emit-newline (ip)
  (write-char #\Newline (out ip))
  (setf (beginning-of-line-p ip) t))

(defun emit-freshline (ip)
  (unless (beginning-of-line-p ip) (emit-newline ip)))
```

有了这些先决条件，现在就可以开始进入FOO处理器的核心地带了。

30.5 HTML 处理器接口

现在可以定义将被FOO语言处理器用来输出HTML的接口了。你可以将该接口定义成一组广义函数，因为需要两个实现：一个实际输出HTML，而另一个可被html宏用来收集一个需要执行的操作的列表，其可被优化并编译成以更高效方式生成同样输出的代码。我把这些广义函数称为后台接口。它由下面8个广义函数构成：

```
(defgeneric raw-string (processor string &optional newlines-p))

(defgeneric newline (processor))

(defgeneric freshline (processor))

(defgeneric indent (processor))

(defgeneric unindent (processor))

(defgeneric toggle-indenting (processor))

(defgeneric embed-value (processor value))

(defgeneric embed-code (processor code))
```

这些函数中有一些明显地有其对应的indenting-printer系列函数，但重要的是理解这些广义函数定义了FOO语言处理器所使用的抽象操作，并且它们并不总是通过对indenting-printer系列函数的调用来实现的。

但也许理解这些抽象操作语义的最简单方式是去查看特化在html-pretty-printer类上的方法的具体实现，该类被用来生成人类可读的HTML。

30.6　美化打印器后台

你可以从定义带有两个槽的类开始：一个槽用来保存indenting-printer的实例，另一个槽用来保存制表符宽度，即对于HTML元素的每一层嵌套缩进你想要增加的空格数。

```
(defclass html-pretty-printer ()
  ((printer   :accessor printer   :initarg :printer)
   (tab-width :accessor tab-width :initarg :tab-width :initform 2)))
```

现在你可以实现特化在html-pretty-printer上的构成后台接口的8个广义函数上的方法了。

FOO处理器使用raw-string函数来输出不需要字符转义的字符串，这要么是因为你实际想要输出一个正常保留的字符，要么是所有的保留字都已经被转义了。通常raw-string以不含有换行的字符串来调用，因此默认的行为是使用emit/no-newlines，除非调用者指定了一个非NIL的newlines-p参数。

```
(defmethod raw-string ((pp html-pretty-printer) string &optional newlines-p)
  (if newlines-p
    (emit (printer pp) string)
    (emit/no-newlines (printer pp) string)))
```

函数newline、freshline、indent、unindent和toggle-indenting实现了对于底层indenting-printer的相当直接的管理。唯一的亮点是只有当动态变量*pretty*为真时，HTML美化打印器才会生成美化的输出。当它是NIL时，你应当生成没有不必要空白的紧凑HTML。因此除了newline之外，下面的方法全部都会在做任何事之前检查*pretty*：[①]

```
(defmethod newline ((pp html-pretty-printer))
  (emit-newline (printer pp)))

(defmethod freshline ((pp html-pretty-printer))
  (when *pretty* (emit-freshline (printer pp))))

(defmethod indent ((pp html-pretty-printer))
  (when *pretty*
    (incf (indentation (printer pp)) (tab-width pp))))
```

① 另一个更纯粹的面向对象的方法是定义两个类，也许是html-pretty-printer和html-raw-printer，然后对于那些只有在*pretty*为真时才发挥作用的方法定义特化在html-raw-printer上的空操作（no-op）方法。不过，在本例中，在定义了所有空操作方法以后，你会得到更多的代码，并且随后还需要确保在正确的时候创建了正确类的实例。但一般来讲，使用多态来替换条件语句是一个好的策略。

```
(defmethod unindent ((pp html-pretty-printer))
  (when *pretty*
    (decf (indentation (printer pp)) (tab-width pp))))

(defmethod toggle-indenting ((pp html-pretty-printer))
  (when *pretty*
    (with-slots (indenting-p) (printer pp)
      (setf indenting-p (not indenting-p)))))
```

最后，函数embed-value和embed-code只被FOO编译器使用。embed-value用来生成将会输出Common Lisp表达式值的代码，而embed-code用来嵌入一点代码运行并丢弃其结果。在解释器中，你无法有目的地求值嵌入的Lisp代码，因此这些函数上的相应方法将总是报错。

```
(defmethod embed-value ((pp html-pretty-printer) value)
  (error "Can't embed values when  interpreting. Value: ~s" value))

(defmethod embed-code ((pp html-pretty-printer) code)
  (error "Can't embed code when interpreting. Code: ~s" code))
```

使用状况系统来解决问题

一个替代的方法是使用EVAL来求值解释器中的Lisp表达式。这种方法的问题在于EVAL无法访问词法环境。因此，无法让类似下面的代码正常工作：

```
(let ((x 10)) (emit-html '(:p x)))
```

其中x是一个词法变量。在运行期传递给emit-html的符号x与同名的词法变量没有特别的关联。Lisp编译器将代码中对x的引用指向该变量，但在代码被编译以后，名字x和该变量之间就不再有必要的关联了。这就是当你认为EVAL是一个解决方案时，可能会判断错误的主要原因。

不过，如果x是一个以**DEFVAR**或**DEFPARAMETER**声明的动态变量（从而可能会被命名为*x*以代替x），那么**EVAL**就可以得到它的值。这样，允许FOO解释器在某些情况下使用**EVAL**将是有用的，但总是使用**EVAL**显然不是个好主意，所以你可以将**EVAL**跟状况系统一起使用，将两种思想组合在一起从而博采众长。

首先定义当embed-value和embed-code在解释器中调用时你将抛出的一些错误类。

```
(define-condition embedded-lisp-in-interpreter (error)
  ((form :initarg :form :reader form)))

(define-condition value-in-interpreter (embedded-lisp-in-interpreter) ()
  (:report
   (lambda (c s)
     (format s "Can't embed values when interpreting. Value: ~s" (form c)))))

(define-condition code-in-interpreter (embedded-lisp-in-interpreter) ()
  (:report
   (lambda (c s)
     (format s "Can't embed code when interpreting. Code: ~s" (form c)))))
```

现在你可以实现embed-value和embed-code来抛出这些错误，并提供一个将使用**EVAL**来求值Lisp形式的再启动函数。

```
(defmethod embed-value ((pp html-pretty-printer) value)
  (restart-case (error 'value-in-interpreter :form value)
    (evaluate ()
      :report (lambda (s)
                (format s "EVAL ~s in null lexical environment." value))
      (raw-string pp (escape (princ-to-string (eval value)) *escapes*) t))))

(defmethod embed-code ((pp html-pretty-printer) code)
  (restart-case (error 'code-in-interpreter :form code)
    (evaluate ()
      :report (lambda (s)
                (format s "EVAL ~s in null lexical environment." code))
      (eval code))))
```

现在你可以做类似下面的事情：

```
HTML> (defvar *x* 10)
*X*
HTML> (emit-html '(:p *x*))
```

然后你将以下列信息进入调试器：

```
Can't embed values when interpreting. Value: *X*
   [Condition of type VALUE-IN-INTERPRETER]
Restarts:
  0: [EVALUATE] EVAL *X* in null lexical environment.
  1: [ABORT] Abort handling SLIME request.
  2: [ABORT] Abort entirely from this process.
```

如果你调用evaluate再启动，那么embed-value将EVAL *x*，得到值10，然后生成下面的HTML：

```
<p>10</p>
```

然后，出于方便的考虑，你可以在特定的情形下提供再启动函数，即可以调用evaluate再启动的函数。evaluate再启动函数无条件地调用同名的再启动，而eval-dynamic-variables和eval-code仅当其状况中的Lisp形式是一个动态变量或潜在的代码时才调用再启动。

```
(defun evaluate (&optional condition)
  (declare (ignore condition))
  (invoke-restart 'evaluate))

(defun eval-dynamic-variables (&optional condition)
  (when (and (symbolp (form condition)) (boundp (form condition)))
    (evaluate)))

(defun eval-code (&optional condition)
  (when (consp (form condition))
    (evaluate)))
```

现在你使用**HANDLER-BIND**来设置一个处理器，它自动为你调用evaluate再启动。

```
HTML> (handler-bind ((value-in-interpreter #'evaluate)) (emit-html '(:p *x*)))
<p>10</p>
T
```

最后，你可以定义一个宏来提供一个漂亮的语法绑定两种类型的错误的处理器。

```
(defmacro with-dynamic-evaluation ((&key values code) &body body)
  `(handler-bind (
        ,@(if values `((value-in-interpreter #'evaluate)))
        ,@(if code `((code-in-interpreter #'evaluate))))
     ,@body))
```

一旦定义了这个宏，就可以写出下面的代码：

```
HTML> (with-dynamic-evaluation (:values t) (emit-html '(:p *x*)))
<p>10</p>
T
```

30.7 基本求值规则

现在将FOO语言与它的处理器接口连接起来，你所需要的全部就是一个函数，它接受一个对象并处理它，其中调用适当的处理器函数来生成HTML。例如，当给定类似

```
(:p "Foo")
```

的简单形式时，该函数可以在处理器上执行下面的调用序列：

```
(freshline processor)
(raw-string processor "<p" nil)
(raw-string processor ">" nil)
(raw-string processor "Foo" nil)
(raw-string processor "</p>" nil)
(freshline processor)
```

目前你可以定义一个简单的函数，它只是检查一个Lisp形式是否是合法的FOO形式，并在是的情况下将其交给process-sexp-html来处理。在下一章里，你将为该函数添加一些额外的代码来允许其处理宏和特殊操作符。但目前它看起来像下面这样：

```
(defun process (processor form)
  (if (sexp-html-p form)
    (process-sexp-html processor form)
    (error "Malformed FOO form: ~s" form)))
```

函数sexp-html-p检查一个给定对象是否是合法的FOO表达式，它要么是一个自求值形式，要么是一个正确格式化了的点对。

```
(defun sexp-html-p (form)
  (or (self-evaluating-p form) (cons-form-p form)))
```

自求值形式很容易处理，只需用PRINC-TO-STRING将其转化成一个字符串，并转义其中出现在变量*escapes*中的字符，该变量前面说过是初始绑定到*element-escapes*的值上的。

点对形式则传递给process-cons-sexp-html来处理。

```
(defun process-sexp-html (processor form)
  (if (self-evaluating-p form)
    (raw-string processor (escape (princ-to-string form) *escapes*) t)
    (process-cons-sexp-html processor form)))
```

函数process-cons-sexp-html随后负责输出开放的标签、任何属性、主体以及闭合的标签。这里主要的复杂之处在于为了生成美化的HTML，就需要根据输出的元素类型来输出空行并调整缩进。你可以将HTML中定义的所有元素分为三类：块、段落和内联元素。当输出块元素（例如body和ul）时在它们的开放标签之前和闭合标签之后都要换行，并且它们的内容需要缩进一层。当输出段落元素（例如p、li和blockquote）时在开放标签之前和闭合标签之后都要换行。内联元素只是简单地在行内输出。下面3个参数列出了每种类型的元素：

```
(defparameter *block-elements*
  '(:body :colgroup :dl :fieldset :form :head :html :map :noscript :object
    :ol :optgroup :pre :script :select :style :table :tbody :tfoot :thead
    :tr :ul))

(defparameter *paragraph-elements*
  '(:area :base :blockquote :br :button :caption :col :dd :div :dt :h1
    :h2 :h3 :h4 :h5 :h6 :hr :input :li :link :meta :option :p :param
    :td :textarea :th :title))

(defparameter *inline-elements*
  '(:a :abbr :acronym :address :b :bdo :big :cite :code :del :dfn :em
    :i :img :ins :kbd :label :legend :q :samp :small :span :strong :sub
    :sup :tt :var))
```

函数block-element-p和paragraph-element-p测试一个给定标签是否是对应列表的一个成员。[①]

```
(defun block-element-p (tag) (find tag *block-elements*))

(defun paragraph-element-p (tag) (find tag *paragraph-elements*))
```

其他两个带有它们自己的谓词的分类是那些总是空的元素，例如br和hr，以及空白需要保留的三个元素pre、style和script。前者在生成正规HTML（换句话说，不是XHTML）时需要特别处理，因为它们没有对应的闭合标签。而在输出那三个内部空白需要保留的标签时，你可以临时关闭缩进，这样美化打印器就不会添加任何不属于元素实际内容的空白了。

```
(defparameter *empty-elements*
  '(:area :base :br :col :hr :img :input :link :meta :param))

(defparameter *preserve-whitespace-elements* '(:pre :script :style))
```

① 你不需要用于*inline-elements*的谓词，因为你只可能测试块和段落元素。在这里，我只是出于完备性的考虑才包括了该参数。

```
(defun empty-element-p (tag) (find tag *empty-elements*))

(defun preserve-whitespace-p (tag) (find tag *preserve-whitespace-elements*))
```

当生成HTML时你需要的最后一点儿信息是，你是否在生成XHTML，因为这将影响到你输出空元素的方式。

```
(defparameter *xhtml* nil)
```

有了全部这些信息，现在就可以开始处理一个FOO的点对形式了。你使用parse-cons-form来将列表解析成3个部分：标签符号，一个可能为空的属性键值对以及一个可能为空的主体形式。然后，你用助手函数emit-open-tag、emit-element-body和emit-close-tag来分别输出开放的标签、主体以及闭合的标签。

```
(defun process-cons-sexp-html (processor form)
  (when (string= *escapes* *attribute-escapes*)
    (error "Can't use cons forms in attributes: ~a" form))
  (multiple-value-bind (tag attributes body) (parse-cons-form form)
    (emit-open-tag     processor tag body attributes)
    (emit-element-body processor tag body)
    (emit-close-tag    processor tag body)))
```

在emit-open-tag中你需要在适当的时候调用freshline，然后用emit-attributes来输出属性。你需要将元素的主体传递给emit-open-tag，这样它在输出XHTML时，就知道究竟是用“/>”还是“>”来结束标签。

```
(defun emit-open-tag (processor tag body-p attributes)
  (when (or (paragraph-element-p tag) (block-element-p tag))
    (freshline processor))
  (raw-string processor (format nil "<~(~a~)" tag))
  (emit-attributes processor attributes)
  (raw-string processor (if (and *xhtml* (not body-p)) "/>" ">")))
```

在emit-attributes中，属性名必须是关键字符号，因此它不会被求值，但你应当调用顶层process函数来求值那些属性值，同时将*escapes*绑定到*attribute-escapes*。为了给指定的布尔属性带来方便，这时属性的值就是属性名本身，如果一个属性的值为T——只是T而非任何其他的真值，那么就将其值替换成该属性的名字。[①]

```
(defun emit-attributes (processor attributes)
  (loop for (k v) on attributes by #'cddr do
        (raw-string processor (format nil " ~(~a~)='" k))
        (let ((*escapes* *attribute-escapes*))
          (process processor (if (eql v t) (string-downcase k) v)))
        (raw-string processor "'")))
```

元素主体的输出过程与属性值的输出类似：你可以在主体上循环调用process来求值其中的

① 尽管XHTML要求布尔属性必须用其名字作为值以表示一个真值，但在HTML中简单地包含一个没有值的属性名也是合法的，例如使用<option selected>而非<option selected='selected'>。所有兼容HTML 4.0的浏览器都应当同时理解两种形式，但一些有bug的浏览器对于特定属性可能只理解没有值的那种形式。如果需要为这样的浏览器生成HTML，那么你将修改emit-attributes从而以不同的方式输出那些属性。

每一个Lisp形式。其余的代码用来输出换行并根据元素的类型来调整缩进。

```
(defun emit-element-body (processor tag body)
  (when (block-element-p tag)
    (freshline processor)
    (indent processor))
  (when (preserve-whitespace-p tag) (toggle-indenting processor))
  (dolist (item body)
    (process processor item))
  (when (preserve-whitespace-p tag) (toggle-indenting processor))
  (when (block-element-p tag)
    (unindent processor)
    (freshline processor)))
```

最后，正如你可能想到的那样，emit-close-tag输出闭合的标签（除非不需要闭合标签，例如当主体为空并且你要么在输出XHTML，要么该元素是特殊的空元素之一）。无论是否实际输出闭合标签，你都需要为块和段落元素输出一个结束的换行。

```
(defun emit-close-tag (processor tag body-p)
  (unless (and (or *xhtml* (empty-element-p tag)) (not body-p))
    (raw-string processor (format nil "</~(~a~)>" tag)))
  (when (or (paragraph-element-p tag) (block-element-p tag))
    (freshline processor)))
```

函数process是基本的FOO解释器。为了让其更易于使用，你可以定义函数emit-html，它调用process并向其传递html-pretty-printer和一个需要求值的Lisp形式。你可以定义并使用助手函数get-pretty-printer来得到美化打印器对象，该函数在*html-pretty-printer*被绑定的情况下返回该变量的值；否则，它生成html-pretty-printer的一个新实例，它使用*html-output*作为其输出流。

```
(defun emit-html (sexp) (process (get-pretty-printer) sexp))

(defun get-pretty-printer ()
  (or *html-pretty-printer*
      (make-instance
       'html-pretty-printer
       :printer (make-instance 'indenting-printer :out *html-output*))))
```

有了这个函数，就可以将HTML输出到*html-output*了。与其将变量*html-output*作为FOO公共API的一部分暴露出来，你不如定义一个宏with-html-output，让它来为你处理流的绑定。它还可以让你指定是否想要美化HTML输出，默认值是变量*pretty*的值。

```
(defmacro with-html-output ((stream &key (pretty *pretty*)) &body body)
  `(let* ((*html-output* ,stream)
          (*pretty* ,pretty))
     ,@body))
```

因此，如果你想要使用emit-html来生成HTML到一个文件里，那么就可以这样来写：

```
(with-open-file (out "foo.html" :direction output)
  (with-html-output (out :pretty t)
    (emit-html *some-foo-expression*)))
```

30.8 下一步是什么

在第31章里，你将看到如何实现一个宏来将FOO表达式编译成Common Lisp，这样你就可以将HTML生成代码直接嵌入到Lisp程序中了。你还能扩展FOO语言，通过添加它自己的特殊操作符和宏来使其具有更强的表达能力。