30. Practical: An HTML Generation Library, the Interpreter

In this chapter and the next you'll take a look under the hood of the FOO HTML generator that you've been using in the past few chapters. FOO is an example of a kind of programming that's quite common in Common Lisp and relatively uncommon in non-Lisp languages, namely, *language-oriented* programming. Rather than provide an API built primarily out of functions, classes, and macros, FOO provides language processors for a domain-specific language that you can embed in your Common Lisp programs.

FOO provides two language processors for the same s-expression language. One is an interpreter that takes a FOO "program" as data and interprets it to generate HTML. The other is a compiler that compiles FOO expressions, possibly with embedded Common Lisp code, into Common Lisp that generates HTML and runs the embedded code. The interpreter is exposed as the function `emit-html` and the compiler as the macro `html`, which you used in previous chapters.

In this chapter you'll look at some of the infrastructure shared between the interpreter and the compiler and then at the implementation of the interpreter. In the next chapter, I'll show you how the compiler works.

Designing a Domain-Specific Language

Designing an embedded language requires two steps: first, design the language that'll allow you to express the things you want to express, and second, implement a processor, or processors, that accepts a "program" in that language and either performs the actions indicated by the program or translates the program into Common Lisp code that'll perform equivalent behaviors.

So, step one is to design the HTML-generating language. The key to designing a good domain-specific language is to strike the right balance between expressiveness and concision. For instance, a highly expressive but not very concise "language" for generating HTML is the language of literal HTML strings. The legal "forms" of this language are strings containing literal HTML. Language processors for this "language" could process such forms by simply emitting them as-is.

```
(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*))
```

This "language" is highly expressive since it can express *any* HTML you could possibly want to generate.[1] On the other hand, this language doesn't win a lot of points for its concision because it gives you zero compression--its input *is* its output.

To design a language that gives you some useful compression without sacrificing too much expressiveness, you need to identify the details of the output that are either redundant or uninteresting. You can then make those aspects of the output implicit in the semantics of the language.

For instance, because of the structure of HTML, every opening tag is paired with a matching closing tag.[2] When you write HTML by hand, you have to write those closing tags, but you can improve the concision of your HTML-generating language by making the closing tags implicit.

Another way you can gain concision at a slight cost in expressiveness is to make the language processors responsible for adding appropriate whitespace between elements--blank lines and indentation. When you're generating HTML programmatically, you typically don't care much about which elements have line breaks before or after them or about whether different elements are indented relative to their parent elements. Letting the language processor insert whitespace according to some rule means you don't have to worry about it. As it turns out, FOO actually supports two modes--one that uses the minimum amount of whitespace, which allows it to generate extremely efficient code and compact HTML, and another that generates nicely formatted HTML with different elements indented and separated from other elements according to their role.

Another detail that's best moved into the language processor is the escaping of certain characters that have a special meaning in HTML such as <, >, and `&`. Obviously, if you generate HTML by just printing strings to a stream, then it's up to you to replace any occurrences of those characters in the string with the appropriate escape sequences, `<`, `>` and `&`. But if the language processor can know which strings are to be emitted as element data, then it can take care of automatically escaping those characters for you.

The FOO Language

So, enough theory. I'll give you a quick overview of the language implemented by FOO, and then you'll look at the implementation of the two FOO language processors--the interpreter, in this chapter, and the compiler, in the next.

Like Lisp itself, the basic syntax of the FOO language is defined in terms of forms made up of Lisp objects. The language defines how each legal FOO form is translated into HTML.

The simplest FOO forms are self-evaluating Lisp objects such as strings, numbers, and keyword symbols.[3] You'll need a function `self-evaluating-p` that tests whether a given object is

self-evaluating for FOO's purposes.

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

Objects that satisfy this predicate will be emitted by converting them to strings with **`PRINC-TO-STRING`** and then escaping any reserved characters, such as <, >, or `&`. When the value is being emitted as an attribute, the characters `"`, and `'` are also escaped. Thus, you can invoke the `html` macro on a self-evaluating object to emit it to `*html-output*` (which is initially bound to **`*STANDARD-OUTPUT*`**). Table 30-1 shows how a few different self-evaluating values will be output.

Table 30-1. FOO Output for Self-Evaluating Objects

FOO Form	Generated HTML
`"foo"`	`foo`
`10`	`10`
`:foo`	`FOO`
`"foo & bar"`	`foo &amp; bar`

Of course, most HTML consists of tagged elements. The three pieces of information that describe each element are the tag, a set of attributes, and a body containing text and/or more HTML elements. Thus, you need a way to represent these three pieces of information as Lisp objects, preferably ones that the Lisp reader already knows how to read.[4] If you forget about attributes for a moment, there's an obvious mapping between Lisp lists and HTML elements: any HTML element can be represented by a list whose **`FIRST`** is a symbol where the name is the name of the element's tag and whose **`REST`** is a list of self-evaluating objects or lists representing other HTML elements. Thus:

```
<p>Foo</p> <==> (:p "Foo")

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

Now the only problem is where to squeeze in the attributes. Since most elements have no attributes, it'd be nice if you could use the preceding syntax for elements without attributes. FOO provides two ways to notate elements with attributes. The first is to simply include the attributes in the list immediately following the symbol, alternating keyword symbols naming the attributes and objects representing the attribute value forms. The body of the element starts with the first item in the list that's in a position to be an attribute name and isn't a keyword symbol. Thus:

```
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
```

For folks who prefer a bit more obvious delineation between the element's attributes and its body, FOO supports an alternative syntax: if the first element of a list is itself a list with a keyword as *its* first element, then the outer list represents an HTML element with that keyword indicating the tag, with the **`REST`** of the nested list as the attributes, and with the **`REST`** of the outer list as the body. Thus, you could write the previous two expressions like this:

```
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
```

The following function tests whether a given object matches either of these syntaxes:

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

You should parameterize the `test` function because later you'll need to test the same two syntaxes with a slightly different predicate on the name.

To completely abstract the differences between the two syntax variants, you can define a function, `parse-cons-form`, that takes a form and parses it into three elements, the tag, the attributes plist, and the body list, returning them as multiple values. The code that actually evaluates cons forms will use this function and not have to worry about which syntax was used.

```
(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))))
```

Now that you have the basic language specified, you can think about how you're actually going to implement the language processors. How do you get from a series of FOO forms to the desired HTML? As I mentioned previously, you'll be implementing two language processors for FOO: an interpreter that walks a tree of FOO forms and emits the corresponding HTML directly and a compiler that walks a tree and translates it into Common Lisp code that'll emit the same HTML. Both the interpreter and compiler will be built on top of a common foundation of code, which provides support for things such as escaping reserved characters and generating nicely indented output, so it makes sense to start there.

Character Escaping

The first bit of the foundation you'll need to lay is the code that knows how to escape characters with a special meaning in HTML. There are three such characters, and they must not appear in the text of an element or in an attribute value; they are <, >, and &. In element text or attribute values, these characters must be replaced with the *character reference entities* `<`, `>`, and `&`. Similarly, in attribute values, the quotation marks used to delimit the value must be escaped, ' with `&apos;` and " with `&quot;`. Additionally, any character can be represented by a numeric character reference entity consisting of an ampersand, followed by a sharp sign, followed by the numeric code as a base 10 integer, and followed by a semicolon. These numeric escapes are sometimes used to embed non-ASCII characters in HTML.

The Package

Since FOO is a low-level library, the package you develop it in doesn't rely on much external code--just the usual dependency on names from the `COMMON-LISP` package and, almost as usual, on the names of the macro-writing macros from `COM.GIGAMONKEYS.MACRO-UTILITIES`. On the other hand, the package needs to export all the names needed by code that uses FOO. Here's the **`DEFPACKAGE`** from the source that you can download from the book's Web site:

```
(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))
```

The following function accepts a single character and returns a string containing a character reference entity for that character:

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

You can use this function as the basis for a function, `escape`, that takes a string and a sequence of characters and returns a copy of the first argument with all occurrences of the characters in the second argument replaced with the corresponding character entity returned by `escape-char`.

```
(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))))
```

You can also define two parameters: `*element-escapes*`, which contains the characters you need to escape in normal element data, and `*attribute-escapes*`, which contains the set of characters to be escaped in attribute values.

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

Here are some examples:

```
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;"
```

Finally, you'll need a variable, `*escapes*`, that will be bound to the set of characters that need to be escaped. It's initially set to the value of `*element-escapes*`, but when generating attributes, it will, as you'll see, be rebound to the value of `*attribute-escapes*`.

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

Indenting Printer

To handle generating nicely indented output, you can define a class `indenting-printer`, which wraps around an output stream, and functions that use an instance of that class to emit strings to the stream while keeping track of when it's at the beginning of the line. The class looks like this:

```
(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)))
```

The main function that operates on `indenting-printers` is `emit`, which takes the printer and a string and emits the string to the printer's output stream, keeping track of when it emits a newline so it can reset the `beginning-of-line-p` slot.

```
(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))
```

To actually emit the string, it uses the function `emit/no-newlines`, which emits any needed indentation, via the helper `indent-if-necessary`, and then writes the string to the stream. This function can also be called directly by other code to emit a string that's known not to contain any newlines.

```
(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)))
```

The helper `indent-if-necessary` checks `beginning-of-line-p` and `indenting-p` to determine whether it needs to emit indentation and, if they're both true, emits as many spaces as indicated by the value of `indentation`. Code that uses the `indenting-printer` can control the indentation by manipulating the `indentation` and `indenting-p` slots. Incrementing and decrementing `indentation` changes the number of leading spaces, while setting `indenting-p` to **`NIL`** can temporarily turn off indentation.

```
(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)))
```

The last two functions in the `indenting-printer` API are `emit-newline` and `emit-freshline`, which are both used to emit a newline character, similar to the `~%` and `~&` **`FORMAT`** directives. That is, the only difference is that `emit-newline` always emits a newline, while `emit-freshline` does so only if `beginning-of-line-p` is false. Thus, multiple calls to `emit-freshline` without any intervening `emit`s won't result in a blank line. This is handy when one piece of code wants to generate some output that should end with a newline while another piece of code wants to generate some output that should start on a newline but you don't want a blank line between the two bits of output.

```
(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)))
```

With those preliminaries out of the way, you're ready to get to the guts of the FOO processor.

HTML Processor Interface

Now you're ready to define the interface that'll be used by the FOO language processor to emit HTML. You can define this interface as a set of generic functions because you'll need two implementations--one that actually emits HTML and another that the `html` macro can use to collect a list of actions that need to be performed, which can then be optimized and compiled into code that emits the same output in a more efficient way. I'll call this set of generic functions the *backend interface*. It consists of the following eight generic functions:

```
(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))
```

While several of these functions have obvious correspondence to `indenting-printer` functions, it's important to understand that these generic functions define the abstract operations that are used by the FOO language processors and won't always be implemented in terms of calls to the `indenting-printer` functions.

That said, perhaps the easiest way to understand the semantics of these abstract operations is to look at the concrete implementations of the methods specialized on `html-pretty-printer`, the class used to generate human-readable HTML.

The Pretty Printer Backend

You can start by defining a class with two slots--one to hold an instance of `indenting-printer` and one to hold the tab width--the number of spaces you want to increase the indentation for each level of nesting of HTML elements.

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

Now you can implement methods specialized on `html-pretty-printer` on the eight generic functions that make up the backend interface.

The FOO processors use the `raw-string` function to emit strings that don't need character escaping, either because you actually want to emit normally reserved characters or because all reserved characters have already been escaped. Usually `raw-string` is invoked with strings that don't contain newlines, so the default behavior is to use `emit/no-newlines` unless the caller specifies a non-**`NIL`** `newlines-p` argument.

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

The functions `newline`, `freshline`, `indent`, `unindent`, and `toggle-indenting` implement fairly straightforward manipulations of the underlying `indenting-printer`. The only wrinkle is that the HTML pretty printer generates pretty output only when the dynamic variable `*pretty*` is true. When it's **`NIL`**, you should generate compact HTML with no unnecessary whitespace. So, these methods, with the exception of `newline`, all check `*pretty*` before doing anything:[5]

```
(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))))

(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)))))
```

Finally, the functions `embed-value` and `embed-code` are used only by the FOO compiler--`embed-value` is used to generate code that'll emit the value of a Common Lisp expression, while `embed-code` is used to embed a bit of code to be run and its result discarded. In the interpreter, you can't meaningfully evaluate embedded Lisp code, so the methods on these functions always signal an error.

```
(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))
```

Using Conditions to Have Your Cake and Eat It Too

An alternate approach would be to use **EVAL** to evaluate Lisp expressions in the interpreter. The problem with this approach is that **EVAL** has no access to the lexical environment. Thus, there's no way to make something like this work:

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

when `x` is a lexical variable. The symbol `x` that's passed to `emit-html` at runtime has no particular connection to the lexical variable named with the same symbol. The Lisp compiler arranges for references to `x` in the code to refer to the variable, but after the code is compiled, there's no longer necessarily any association between the name `x` and that variable. This is the main reason that when you think **EVAL** is the solution to your problem, you're probably wrong.

However, if `x` was a dynamic variable, declared with **DEFVAR** or **DEFPARAMETER** (and likely named `*x*` instead of `x`), **EVAL** could get at its value. Thus, it might be useful to allow the FOO interpreter to use **EVAL** in some situations. But it's a bad idea to always use **EVAL**. You can get the best of both worlds by combining the idea of using **EVAL** with the condition system.

First define some error classes that you can signal when `embed-value` and `embed-code` are called in the interpreter.

```
(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)))))
```

Now you can implement `embed-value` and `embed-code` to signal those errors *and* provide a restart that'll evaluate the form with **EVAL**.

```
(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))))
```

Now you can do something like this:

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

and you'll get dropped into the debugger with this message:

```
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.
```

If you invoke the `evaluate` restart, `embed-value` will **`EVAL`** `*x*`, get the value `10`, and generate this HTML:

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

Then, as a convenience, you can provide restart functions--functions that invoke the `evaluate` restart--in certain situations. The `evaluate` restart function unconditionally invokes the restart, while `eval-dynamic-variables` and `eval-code` invoke it only if the form in the condition is a dynamic variable or potential code.

```
(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)))
```

Now you can use **`HANDLER-BIND`** to set up a handler to automatically invoke the `evaluate` restart for you.

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

Finally, you can define a macro to provide a nicer syntax for binding handlers for the two kinds of errors.

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

With this macro defined, you can write this:

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

The Basic Evaluation Rule

Now to connect the FOO language to the processor interface, all you need is a function that takes an object and processes it, invoking the appropriate processor functions to generate HTML. For instance, when given a simple form like this:

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

this function might execute this sequence of calls on the processor:

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

For now you can define a simple function that just checks whether a form is, in fact, a legal FOO form and, if it is, hands it off to the function `process-sexp-html` for processing. In the next chapter, you'll add some bells and whistles to this function to allow it to handle macros and special operators. But for now it looks like this:

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

The function `sexp-html-p` determines whether the given object is a legal FOO expression, either a self-evaluating form or a properly formatted cons.

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

Self-evaluating forms are easily handled: just convert to a string with **`PRINC-TO-STRING`** and escape the characters in the variable `*escapes*`, which, as you'll recall, is initially bound to the value of `*element-escapes*`. Cons forms you pass off to `process-cons-sexp-html`.

```
(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)))
```

The function `process-cons-sexp-html` is then responsible for emitting the opening tag, any attributes, the body, and the closing tag. The main complication here is that to generate pretty HTML, you need to emit fresh lines and adjust the indentation according to the type of the element being emitted. You can categorize all the elements defined in HTML into one of three categories: block, paragraph, and inline. Block elements--such as `body` and `ul`--are emitted with fresh lines before and after both their opening and closing tags and with their contents indented one level. Paragraph elements--such as `p`, `li`, and `blockquote`--are emitted with a fresh line before the opening tag and after the closing tag. Inline elements are simply emitted in line. The following three parameters list the elements of each type:

```
(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))
```

The functions `block-element-p` and `paragraph-element-p` test whether a given tag is a member of the corresponding list.[6]

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

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

Two other categorizations with their own predicates are the elements that are always empty, such as `br` and `hr`, and the three elements, `pre`, `style`, and `script`, in which whitespace is supposed to be preserved. The former are handled specially when generating regular HTML (in other words, not XHTML) since they're not supposed to have a closing tag. And when emitting the three tags in which whitespace is preserved, you can temporarily turn off indentation so the pretty printer doesn't add any spaces that aren't part of the element's actual contents.

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

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

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

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

The last piece of information you need when generating HTML is whether you're generating XHTML since that affects how you emit empty elements.

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

With all that information, you're ready to process a cons FOO form. You use `parse-cons-form` to parse the list into three parts, the tag symbol, a possibly empty plist of attribute key/value pairs, and a possibly empty list of body forms. You then emit the opening tag, the body, and the closing tag with the helper functions `emit-open-tag`, `emit-element-body`, and `emit-close-tag`.

```
(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)))
```

In `emit-open-tag` you have to call `freshline` when appropriate and then emit the attributes with `emit-attributes`. You need to pass the element's body to `emit-open-tag` so when it's emitting XHTML, it knows whether to finish the tag with `/>` or `>`.

```
(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)) "/>" ">")))
```

In `emit-attributes` the attribute names aren't evaluated since they must be keyword symbols, but you should invoke the top-level `process` function to evaluate the attribute values, binding `*escapes*` to `*attribute-escapes*`. As a convenience for specifying boolean attributes, whose value should be the name of the attribute, if the value is **`T`**--not just any true value but actually **`T`**--then you replace the value with the name of the attribute.[7]

```
(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 "'")))
```

Emitting the element's body is similar to emitting the attribute values: you can loop through the body calling `process` to evaluate each form. The rest of the code is dedicated to emitting fresh lines and adjusting the indentation as appropriate for the type of element.

```
(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)))
```

Finally, `emit-close-tag`, as you'd probably expect, emits the closing tag (unless no closing tag is necessary, such as when the body is empty and you're either emitting XHTML or the element is one of the special empty elements). Regardless of whether you actually emit a close tag, you need to emit a final fresh line for block and paragraph elements.

```
(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)))
```

The function `process` is the basic FOO interpreter. To make it a bit easier to use, you can define a function, `emit-html`, that invokes `process`, passing it an `html-pretty-printer` and a form to evaluate. You can define and use a helper function, `get-pretty-printer`, to get the pretty printer, which returns the current value of `*html-pretty-printer*` if it's bound; otherwise, it makes a new instance of `html-pretty-printer` with `*html-output*` as its output stream.

```
(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*))))
```

With this function, you can emit HTML to `*html-output*`. Rather than expose the variable `*html-output*` as part of FOO's public API, you should define a macro, `with-html-output`, that takes care of binding the stream for you. It also lets you specify whether you want pretty HTML output, defaulting to the value of the variable `*pretty*`.

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

So, if you wanted to use `emit-html` to generate HTML to a file, you could write the following:

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

What's Next?

In the next chapter, you'll look at how to implement a macro that compiles FOO expressions into Common Lisp so you can embed HTML generation code directly into your Lisp programs. You'll also extend the FOO language to make it a bit more expressive by adding its own flavor of special operators and macros.

---

[1]In fact, it's probably *too* expressive since it can also generate all sorts of output that's not even vaguely legal HTML. Of course, that might be a feature if you need to generate HTML that's not strictly correct to compensate for buggy Web browsers. Also, it's common for language processors to accept programs that are syntactically correct and otherwise well formed that'll nonetheless provoke undefined behavior when run.

[2]Well, almost every tag. Certain tags such as `IMG` and `BR` don't. You'll deal with those in the section "The Basic Evaluation Rule."

[3]In the strict language of the Common Lisp standard, keyword symbols aren't *self-evaluating*, though they do, in fact, evaluate to themselves. See section 3.1.2.1.3 of the language standard or HyperSpec for a brief discussion.

[4]The requirement to use objects that the Lisp reader knows how to read isn't a hard-and-fast one. Since the Lisp reader is itself customizable, you could also define a new reader-level syntax for a new kind of object. But that tends to be more trouble than it's worth.

[5]Another, more purely object-oriented, approach would be to define two classes, perhaps `html-pretty-printer` and `html-raw-printer`, and then define no-op methods specialized on `html-raw-printer` for the methods that should do stuff only when `*pretty*` is true. However, in this case, after defining all the no-op methods, you'd end up with more code, and then you'd have the hassle of making sure you created an instance of the right class at the right time. But in general, using polymorphism to replace conditionals is a good strategy.

[6]You don't need a predicate for `*inline-elements*` since you only ever test for block and paragraph elements. I include the parameter here for completeness.

[7]While XHTML requires boolean attributes to be notated with their name as the value to indicate a true value, in HTML it's also legal to simply include the name of the attribute with no value, for example, **`<option selected>`** rather than **`<option selected='selected'>`**. All HTML 4.0-compatible browsers should understand both forms, but some buggy browsers understand only the no-value form for certain attributes. If you need to generate HTML for such browsers, you'll need to hack `emit-attributes` to emit those attributes a bit differently.