31. Practical: An HTML Generation Library, the Compiler

Now you're ready to look at how the FOO compiler works. The main difference between a compiler and an interpreter is that an interpreter processes a program and directly generates some behavior--generating HTML in the case of a FOO interpreter--but a compiler processes the same program and generates code in some other language that will exhibit the same behavior. In FOO, the compiler is a Common Lisp macro that translates FOO into Common Lisp so it can be embedded in a Common Lisp program. Compilers, in general, have the advantage over interpreters that, because compilation happens in advance, they can spend a bit of time optimizing the code they generate to make it more efficient. The FOO compiler does that, merging literal text as much as possible in order to emit the same HTML with a smaller number of writes than the interpreter uses. When the compiler is a Common Lisp macro, you also have the advantage that it's easy for the language understood by the compiler to contain embedded Common Lisp--the compiler just has to recognize it and embed it in the right place in the generated code. The FOO compiler will take advantage of this capability.

The Compiler

The basic architecture of the compiler consists of three layers. First you'll implement a class `html-compiler` that has one slot that holds an adjustable vector that's used to accumulate *ops* representing the calls made to the generic functions in the backend interface during the execution of `process`.

You'll then implement methods on the generic functions in the backend interface that will store the sequence of actions in the vector. Each op is represented by a list consisting of a keyword naming the operation and the arguments passed to the function that generated the op. The function `sexp->ops` implements the first phase of the compiler, compiling a list of FOO forms by calling `process` on each form with an instance of `html-compiler`.

This vector of ops stored by the compiler is then passed to a function that optimizes it, merging consecutive `raw-string` ops into a single op that emits the combined string in one go. The optimization function can also, optionally, strip out ops that are needed only for pretty printing, which is mostly important because it allows you to merge more `raw-string` ops.

Finally, the optimized ops vector is passed to a third function, `generate-code`, that returns a list of Common Lisp expressions that will actually output the HTML. When `*pretty*` is true, `generate-code` generates code that uses the methods specialized on `html-pretty-printer` to output pretty HTML. When `*pretty*` is **`NIL`**, it generates code that writes directly to the stream `*html-output*`.

The macro `html` actually generates a body that contains two expansions, one generated with `*pretty*` bound to **`T`** and one with `*pretty*` bound to **`NIL`**. Which expansion is used is determined by the runtime value of `*pretty*`. Thus, every function that contains a call to `html` will contain code to generate both pretty and compact output.

The other significant difference between the compiler and the interpreter is that the compiler can embed Lisp forms in the code it generates. To take advantage of that, you need to modify the `process` function so it calls the `embed-code` and `embed-value` functions when asked to process an expression that's not a FOO form. Since all self-evaluating objects are valid FOO forms, the only forms that won't be passed to `process-sexp-html` are lists that don't match the syntax for FOO cons forms and non-keyword symbols, the only atoms that aren't self-evaluating. You can assume that any non-FOO cons is code to be run inline and all symbols are variables whose value you should embed.

```
(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))))
```

Now let's look at the compiler code. First you should define two functions that slightly abstract the vector you'll use to save ops in the first two phases of compilation.

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

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

Next you can define the `html-compiler` class and the methods specialized on it to implement the backend interface.

```
(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)))
```

With those methods defined, you can implement the first phase of the compiler, `sexp->ops`.

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

During this phase you don't need to worry about the value of `*pretty*`: just record all the functions called by `process`. Here's what `sexp->ops` makes of a simple FOO form:

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

The next phase, `optimize-static-output`, takes a vector of ops and returns a new vector containing the optimized version. The algorithm is simple--for each `:raw-string` op, it writes the string to a temporary string buffer. Thus, consecutive `:raw-string` ops will build up a single string containing the concatenation of the strings that need to be emitted. Whenever you encounter an op other than a `:raw-string` op, you convert the built-up string into a sequence of alternating `:raw-string` and `:newline` ops with the helper function `compile-buffer` and then add the next op. This function is also where you strip out the pretty printing ops if `*pretty*` is **`NIL`**.

```
(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))
```

The last step is to translate the ops into the corresponding Common Lisp code. This phase also pays attention to the value of `*pretty*`. When `*pretty*` is true, it generates code that invokes the backend generic functions on `*html-pretty-printer*`, which will be bound to an instance of `html-pretty-printer`. When `*pretty*` is **`NIL`**, it generates code that writes directly to `*html-output*`, the stream to which the pretty printer would send its output.

The actual function, `generate-code`, is trivial.

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

All the work is done by methods on the generic function `op->code` specializing the `op` argument with an **`EQL`** specializer on the name of the op.

```
(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)))
```

The two most interesting `op->code` methods are the ones that generate code for the `:embed-value` and `:embed-code` ops. In the `:embed-value` method, you can generate slightly different code depending on the value of the `escapes` operand since if `escapes` is **`NIL`**, you don't need to generate a call to `escape`. And when both `*pretty*` and `escapes` are **`NIL`**, you can generate code that uses **`PRINC`** to emit the value directly to the stream.

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

Thus, something like this:

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

works because `html` translates `(:p x)` into something like this:

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

When that code replaces the call to `html` in the context of the **`LET`**, you get the following:

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

and the reference to `x` in the generated code turns into a reference to the lexical variable from the **`LET`** surrounding the `html` form.

The `:embed-code` method, on the other hand, is interesting because it's so trivial. Because `process` passed the form to `embed-code`, which stashed it in the `:embed-code` op, all you have to do is pull it out and return it.

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

This allows code like this to work:

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

The outer call to `html` expands into code that does something like this:

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

Then if you expand the call to `html` in the body of the **`DOLIST`**, you'll get something like this:

```
(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*))
```

This code will, in fact, generate the output you saw.

FOO Special Operators

You could stop there; certainly the FOO language is expressive enough to generate nearly any HTML you'd care to. However, you can add two features to the language, with just a bit more code, that will make it quite a bit more powerful: special operators and macros.

Special operators in FOO are analogous to special operators in Common Lisp. Special operators provide ways to express things in the language that can't be expressed in the language supported by the basic evaluation rule. Or, another way to look at it is that special operators provide access to the primitive mechanisms used by the language evaluator.[1]

To take a simple example, in the FOO compiler, the language evaluator uses the embed-value function to generate code that will embed the value of a variable in the output HTML. However, because only symbols are passed to embed-value, there's no way, in the language I've described so far, to embed the value of an arbitrary Common Lisp expression; the process function passes cons cells to embed-code rather than embed-value, so the values returned are ignored. Typically this is what you'd want, since the main reason to embed Lisp code in a FOO program is to use Lisp control constructs. However, sometimes you'd like to embed computed values in the generated HTML. For example, you might like this FOO program to generate a paragraph tag containing a random number:

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

But that doesn't work because the code is run and its value discarded.

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

In the language, as you've implemented it so far, you could work around this limitation by computing the value outside the call to html and then embedding it via a variable.

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

But that's sort of annoying, particularly when you consider that if you could arrange for the form (random 10) to be passed to embed-value instead of embed-code, it'd do exactly what you want. So, you can define a special operator, :print, that's processed by the FOO language processor according to a different rule than a normal FOO expression. Namely, instead of generating a <print> element, it passes the form in its body to embed-value. Thus, you can generate a paragraph containing a random number like this:

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

Obviously, this special operator is useful only in compiled FOO code since embed-value doesn't work in the interpreter. Another special operator that can be used in both interpreted and compiled FOO code is :format, which lets you generate output using the **FORMAT** function. The arguments to the :format special operator are a string used as a format control string and then any arguments to be interpolated. When all the arguments to :format are self-evaluating objects, a string is generated by passing them to **FORMAT**, and that string is then emitted like any other string. This allows such :format forms to be used in FOO passed to emit-html. In compiled FOO, the arguments to :format can be any Lisp expressions.

Other special operators provide control over what characters are automatically escaped and to explicitly emit newline characters: the :noescape special operator causes all the forms in its body to be evaluated as regular FOO forms but with *escapes* bound to **NIL**, while :attribute evaluates the forms in its body with *escapes* bound to *attribute-escapes*. And :newline is translated into code to emit an explicit newline.

So, how do you define special operators? There are two aspects to processing special operators: how does the language processor recognize forms that use special operators, and how does it know what code to run to process each special operator?

You could hack `process-sexp-html` to recognize each special operator and handle it in the appropriate manner--special operators are, logically, part of the implementation of the language, and there aren't going to be that many of them. However, it'd be nice to have a slightly more modular way to add new special operators--not because users of FOO will be able to but just for your own sanity.

Define a *special form* as any list whose **`CAR`** is a symbol that's the name of a special operator. You can mark the names of special operators by adding a non-**`NIL`** value to the symbol's property list under the key `html-special-operator`. So, you can define a function that tests whether a given form is a special form like this:

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

The code that implements each special operator is responsible for taking apart the rest of the list however it sees fit and doing whatever the semantics of the special operator require. Assuming you'll also define a function `process-special-form`, which will take the language processor and a special form and run the appropriate code to generate a sequence of calls on the processor object, you can augment the top-level `process` function to handle special forms like this:

```
(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))))
```

You must add the `special-form-p` clause first because special forms can look, syntactically, like regular FOO expressions just the way Common Lisp's special forms can look like regular function calls.

Now you just need to implement `process-special-form`. Rather than define a single monolithic function that implements all the special operators, you should define a macro that allows you to define special operators much like regular functions and that also takes care of adding the `html-special-operator` entry to the property list of the special operator's name. In fact, the value you store in the property list can be a function that implements the special operator. Here's the macro:

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

This is a fairly advanced type of macro, but if you take it one line at a time, there's nothing all that tricky about it. To see how it works, take a simple use of the macro, the definition of the special operator **`:noescape`**, and look at the macro expansion. If you write this:

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

it's as if you had written this:

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

The **EVAL-WHEN** special operator, as I discussed in Chapter 20, ensures that the effects of code in its body will be made visible during compilation when you compile with **COMPILE-FILE**. This matters if you want to use `define-html-special-operator` in a file and then use the just-defined special operator in that same file.

Then the **SETF** expression sets the property `html-special-operator` on the symbol `:noescape` to an anonymous function with the same parameter list as was specified in `define-html-special-operator`. By defining `define-html-special-operator` to split the parameter list in two parts, `processor` and everything else, you ensure that all special operators accept at least one argument.

The body of the anonymous function is then the body provided to `define-html-special-operator`. The job of the anonymous function is to implement the special operator by making the appropriate calls on the backend interface to generate the correct HTML or the code that will generate it. It can also use `process` to evaluate an expression as a FOO form.

The `:noescape` special operator is particularly simple--all it does is pass the forms in its body to `process` with `*escapes*` bound to **NIL**. In other words, this special operator disables the normal character escaping preformed by `process-sexp-html`.

With special operators defined this way, all `process-special-form` has to do is look up the anonymous function in the property list of the special operator's name and **APPLY** it to the processor and rest of the form.

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

Now you're ready to define the five remaining FOO special operators. Similar to `:noescape` is `:attribute`, which evaluates the forms in its body with `*escapes*` bound to `*attribute-escapes*`. This special operator is useful if you want to write helper functions that output attribute values. If you write a function like this:

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

the `html` macro is going to generate code that escapes the characters in `*element-escapes*`. But if you're planning to use `foo-value` like this:

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

then you want it to generate code that uses `*attribute-escapes*`. So, instead, you can write it like this:[2]

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

The definition of `:attribute` looks like this:

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

The next two special operators, `:print` and `:format`, are used to output values. The `:print` special operator, as I discussed earlier, is used in compiled FOO programs to embed the value of an arbitrary Lisp expression. The `:format` special operator is more or less equivalent to generating a string with `(format nil ...)` and then embedding it. The primary reason to define `:format` as a special operator is for convenience. This:

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

is nicer than this:

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

It also has the slight advantage that if you use `:format` with arguments that are all self-evaluating, FOO can evaluate the `:format` at compile time rather than waiting until runtime. The definitions of `:print` and `:format` are as follows:

```
(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))))
```

The `:newline` special operator forces an output of a literal newline, which is occasionally handy.

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

Finally, the `:progn` special operator is analogous to the **`PROGN`** special operator in Common Lisp. It simply processes the forms in its body in sequence.

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

In other words, the following:

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

will generate the same code as this:

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

This might seem like a strange thing to need since normal FOO expressions can have any number of forms in their body. However, this special operator will come in quite handy in one situation--when writing FOO macros, which brings you to the last language feature you need to implement.

FOO Macros

FOO macros are similar in spirit to Common Lisp's macros. A FOO macro is a bit of code that accepts a FOO expression as an argument and returns a new FOO expression as the result, which is then evaluated according to the normal FOO evaluation rules. The actual implementation is quite similar to the implementation of special operators.

As with special operators, you can define a predicate function to test whether a given form is a macro form.

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

You use the previously defined function `cons-form-p` because you want to allow macros to be used in either of the syntaxes of nonmacro FOO cons forms. However, you need to pass a different predicate function, one that tests whether the form name is a symbol with a non-**`NIL`** `html-macro` property. Also, as in the implementation of special operators, you'll define a macro for defining FOO macros, which is responsible for storing a function in the property list of the macro's name, under the key `html-macro`. However, defining a macro is a bit more complicated because FOO supports two flavors of macro. Some macros you'll define will behave much like normal HTML elements and may want to have easy access to a list of attributes. Other macros will simply want raw access to the elements of their body.

You can make the distinction between the two flavors of macros implicit: when you define a FOO macro, the parameter list can include an `&attributes` parameter. If it does, the macro form will be parsed like a regular cons form, and the macro function will be passed two values, a plist of attributes and a list of expressions that make up the body of the form. A macro form without an `&attributes` parameter won't be parsed for attributes, and the macro function will be invoked with a single argument, a list containing the body expressions. The former is useful for what are essentially HTML templates. For example:

```
(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
```

The latter kind of macro is more useful for writing macros that manipulate the forms in their body. This type of macro can function as a kind of HTML control construct. As a trivial example, consider the following macro that implements an `:if` construct:

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

This macro allows you to write this:

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

instead of this slightly more verbose version:

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

To determine which kind of macro you should generate, you need a function that can parse the parameter list given to `define-html-macro`. This function returns two values, the name of the `&attributes` parameter, or **`NIL`** if there was none, and a list containing all the elements of `args` after removing the `&attributes` marker and the subsequent list element.[3]

```
(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)
```

The element following `&attributes` in the parameter list can also be a destructuring parameter list.

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

Now you're ready to write `define-html-macro`. Depending on whether there was an `&attributes` parameter specified, you need to generate one form or the other of HTML macro so the main macro simply determines which kind of HTML macro it's defining and then calls out to a helper function to generate the right kind of code.

```
(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))))
```

The functions that actually generate the expansion look like this:

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

The macro functions you'll define accept either one or two arguments and then use **DESTRUCTURING-BIND** to take them apart and bind them to the parameters defined in the call to `define-html-macro`. In both expansions you need to save the macro function in the name's property list under `html-macro` and a boolean indicating whether the macro takes an `&attributes` parameter under the property `html-macro-wants-attributes`. You use that property in the following function, `expand-macro-form`, to determine how the macro function should be invoked:

```
(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))))
```

The last step is to integrate macros by adding a clause to the dispatching **COND** in the top-level `process` function.

```
(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))))
```

This is the final version of `process`.

The Public API

Now, at long last, you're ready to implement the `html` macro, the main entry point to the FOO compiler. The other parts of FOO's public API are `emit-html` and `with-html-output`, which I discussed in the previous chapter, and `define-html-macro`, which I discussed in the previous section. The `define-html-macro` macro needs to be part of the public API because FOO's users will want to write their own HTML macros. On the other hand, `define-html-special-operator` isn't part of the public API because it requires too much knowledge of FOO's internals to define a new special operator. And there should be very little that can't be done using the existing language and special operators.[4]

One last element of the public API, before I get to `html`, is another macro, `in-html-style`. This macro controls whether FOO generates XHTML or regular HTML by setting the `*xhtml*` variable. The reason this needs to be a macro is because you'll want to wrap the code that sets `*xhtml*` in an **`EVAL-WHEN`** so you can set it in a file and have it affect uses of the `html` macro later in that same file.

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

Finally let's look at `html` itself. The only tricky bit about implementing `html` comes from the need to generate code that can be used to generate both pretty and compact output, depending on the runtime value of the variable `*pretty*`. Thus, `html` needs to generate an expansion that contains an **`IF`** expression and two versions of the code, one compiled with `*pretty*` bound to true and one compiled with it bound to **`NIL`**. To further complicate matters, it's common for one `html` call to contain embedded calls to `html`, like this:

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

If the outer `html` expands into an **`IF`** expression with two versions of the code, one for when `*pretty*` is true and one for when it's false, it's silly for nested `html` forms to expand into two versions too. In fact, it'll lead to an exponential explosion of code since the nested `html` is already going to be expanded twice--once in the `*pretty*`-is-true branch and once in the `*pretty*`-is-false branch. If each expansion generates two versions, then you'll have four total versions. And if the nested `html` form contained another nested `html` form, you'd end up with eight versions of that code. If the compiler is smart, it'll eventually realize that most of that generated code is dead and will eliminate it, but even figuring that out can take quite a bit of time, slowing down compilation of any function that uses nested calls to `html`.

Luckily, you can easily avoid this explosion of dead code by generating an expansion that locally redefines the `html` macro, using **`MACROLET`**, to generate only the right kind of code. First you define a helper function that takes the vector of ops returned by `sexp->ops` and runs it through `optimize-static-output` and `generate-code`--the two phases that are affected by the value of `*pretty*`--with `*pretty*` bound to a specified value and that interpolates the resulting code into a **`PROGN`**. (The **`PROGN`** returns **`NIL`** just to keep things tidy.).

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

With that function, you can then define `html` like this:

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

The **`&whole`** parameter represents the original `html` form, and because it's interpolated into the expansion in the bodies of the two **`MACROLET`**s, it will be reprocessed with each of the new definitions of `html`, the one that generates pretty-printing code and the other that generates non-pretty-printing code. Note that the variable `*pretty*` is used both during macro expansion *and* when the resulting code is run. It's used at macro expansion time by `codegen-html` to cause `generate-code` to generate one kind of code or the other. And it's used at runtime, in the **`IF`** generated by the top-level `html` macro, to determine whether the pretty-printing or non-pretty-printing code should actually run.

The End of the Line

As usual, you could keep working with this code to enhance it in various ways. One interesting avenue to pursue is to use the underlying output generation framework to emit other kinds of output. In the version of FOO you can download from the book's Web site, you'll find some code that implements CSS output that can be integrated into HTML output in both the interpreter and compiler. That's an interesting case because CSS's syntax can't be mapped to s-expressions in such a trivial way as HTML's can. However, if you look at that code, you'll see it's still possible to define an s-expression syntax for representing the various constructs available in CSS.

A more ambitious undertaking would be to add support for generating embedded JavaScript. Done right, adding JavaScript support to FOO could yield two big wins. One is that after you define an s-expression syntax that you can map to JavaScript syntax, then you can start writing macros, in Common Lisp, to add new constructs to the language you use to write client-side code, which will then be compiled to JavaScript. The other is that, as part of the FOO s-expression JavaScript to regular JavaScript translation, you could deal with the subtle but annoying differences between JavaScript implementations in different browsers. That is, the JavaScript code that FOO generates could either contain the appropriate conditional code to do one thing in one browser and another in a different browser or could generate different code depending on which browser you wanted to support. Then if you use FOO in dynamically generated pages, it could use information about the User-Agent making the request to generate the right flavor of JavaScript for that browser.

But if that interests you, you'll have to implement it yourself since this is the end of the last practical chapter of this book. In the next chapter I'll wrap things up, discussing briefly some topics that I haven't touched on elsewhere in the book such as how to find libraries, how to optimize Common Lisp code, and how to deliver Lisp applications.

---

[1]The analogy between FOO's special operators, and macros, which I'll discuss in the next section, and Lisp's own is fairly sound. In fact, understanding how FOO's special operators and macros work may give you some insight into why Common Lisp is put together the way it is.

[2]The `:noescape` and `:attribute` special operators must be defined as special operators because FOO determines what escapes to use at compile time, not at runtime. This allows FOO to escape literal values at compile time, which is much more efficient than having to scan all output at runtime.

[3]Note that `&attributes` is just another symbol; there's nothing intrinsically special about names that start with `&`.

[4]The one element of the underlying language-processing infrastructure that's not currently exposed through special operators is the indentation. If you wanted to make FOO more flexible, albeit at the cost of making its API that much more complex, you could add special operators for manipulating the underlying indenting printer. But it seems like the cost of having to explain the extra special operators would outweigh the rather small gain in expressiveness.