

Generating HTML with Lisp
a tutorial for new programmers

Gene Michael Stover

created Sunday, 10 August 2003
updated Sunday, 2005 December 5

Copyright © 2003, 2005 by Gene Michael Stover. All rights reserved. Permission to copy, store, & view this document unmodified & in its entirety is granted.

Contents

1	Introduction	5
2	Motivation	7
3	Unsatisfactory Alternatives	9
4	Getting Started	11
5	Other Types of Tags	13
6	Attributes	17
7	Some Nice Touches	23
7.1	Newlines Around Tags	23
7.2	Character Arguments	26
7.3	Two Special Tags	28
7.4	Make a Package	28
8	The Library So Far (Draft Release)	31
9	Putting It To Use	33
9.1	Miles Per Gallon	33
9.2	CGI	34
9.3	CGI with a Twist	34
10	Improving Performance	35
10.1	Thoughts about Optimization	35
10.2	The Importance of Profilers	36
10.3	Our Profiler	37
10.4	Let's Profile It Already	38
10.5	Knowing When To Stop	43
10.6	Importance of Profilers, revisited	44
11	The Final Library (Final Release)	45

4

CONTENTS

12 Exercise for the Reader

47

13 Other File Formats

49

Chapter 1

Introduction

This is a tutorial for new Lisp programmers, but not total beginners. I assume you know some basic Lisp, like the syntax, lists, and what `car` & `cdr` mean.

Things I discuss or show in this tutorial include

- A simple, but representative, use of macros,
- Refactoring Lisp code as we develop,
- Moving code into a package,
- Optimization session with the help of a profiler written in Lisp itself & discussion of effective use of a profiler,
- Another programmer's thought processes as he develops some software. It's not that my thought processes are models for a good programmer; it's that it is sometimes useful to see how another practitioner of the same craft thinks.

This is not a CGI tutorial, though I do a tiny bit of CGI near the end. It's just an HTML tutorial.¹

¹But I feel the call to write a Lisp/CGI tutorial, so maybe in a month or two, maybe October 2003.

Chapter 2

Motivation

Let's say you are using Lisp to generate HTML. In fact, I use it to generate my resumé & parts of my web site. It doesn't take long before the print statements get out of hand. Here is a representative chunk of code from the index-generator on my web site:

```
(format strm "~&<DT>")
(format strm "~&<A HREF=\"~A\">" (cdr (assoc 'url article)))
(format strm "~A" (cdr (assoc 'title article)))
(format strm "</A>")
(format strm "</DT>")
(format strm "~&<DD>")
(format strm "~&~A," (cdr (assoc 'author article)))
```

Yuck. It's well-behaved, correct code, but it's an eye-sore. It would be much nicer to write something like this:

```
(html
  (head (title "I am titular"))
  (body
    (h1 "Generating HTML with Lisp")
    (p "This is a tutorial for new Lisp"
      " programmers, but not total"
      " beginners. ...")

    ;; We can use Lisp to generate things
    ;; dynamically, like a numbered list of
    ;; colors.
    (apply #'ol
      (mapcar #'(lambda (color)
                  (li color))
              '(red green yellow blue))))))
```

That'd be much nicer.

Chapter 3

Unsatisfactory Alternatives

Before we start writing a small HTML library, let's talk about some approaches I considered but disqualified.

I have a C programming history, & old habits die hard, so my first thoughts were to write the HTML to a stream, much as I'd probably do with C. I thought I might do something like this:

```
;; This is a BAD example. Don't do this.

;; I wouldn't use a global stream variable
;; in reality. It's just for illustration.
(defvar strm (open "output.html" :direction :output))

;; Here's how I imagined generating HTML.
(html strm
  (head strm (title strm "i'm a title"))
  (body strm
    (h1 strm "I'm a title")
    (p strm "you get the idea..."))))
```

The first thing that's wrong with the idea is that I must type "strm" too many times. All the HTML goes to the same place, so I should type that place just once.

Another thing that's wrong is it's difficult to handle beginning & ending tags when you write to a stream. In fact, I think it's impossible for that example to work if those Lisp forms are function calls. If they are function calls, the inner forms are evaluated before the outer forms, & their output goes to the stream before the output of the enclosing forms. So the HTML tag structure would be completely broken. I mean *it's just wrong!*

It could work if those Lisp forms were macros. I could have an encloser macro that set the destination stream, & then evaluated the inner forms. The inner forms would all be macros, too. Each would print its opening tag, then

evaluate its arguments, then print its ending tag. Your Lisp code could look something like this:

```
(defvar *html-output* nil
  "Output stream for all the HTML-generating
Lisp macros.")

(with-open-file (*html-output* "mine.html"
               :direction :output)
  (html
    (head (title "i'm a title"))
    (body
      (h1 "I'm a title")
      (p "you get the idea...")))))
```

I'm sure it would work, but it relies on at least one global variable & potentially grotesquely large macro expansions, so I chose an approach that uses functions.

For what it's worth, Paul Graham discusses HTML generation in his *On Lisp* book, & he succeeded with a macro approach. I gather that he used his macro-based HTML generator in a production, commercial context.

Chapter 4

Getting Started

Remember that first example of what kind of Lisp code I'd like to type? Here it is again:

```
(html
  (head (title "I am titular")))
  (body
    (h1 "Generating HTML with Lisp")
    (p "This is a tutorial for new Lisp"
      " programmers, but not total"
      " beginners.  ...")

    ;; We can use Lisp to generate things
    ;; dynamically, like a numbered list of
    ;; colors.
    (apply #'ol
      (mapcar #'(lambda (color)
                  (li color))
              '(red green yellow blue))))))
```

If each of those are function calls, then the inner ones will be evaluated first. Their return values will be arguments to the outer function calls. So each function must be prepared to handle multiple arguments. That's easily done with the `&rest` keyword when we define the functions. Each of the functions will need to concatenate its arguments & wrap them in its own beginning & ending tags. Basically, the arguments to each function should be strings, & each function should return a string.

Let's write the `html` function to see what it's like.

```
(defun html (&rest strings)
  "Concatenate the arguments, then wrap them
in <HTML> & </HTML>, & return the new string."
  (apply #'concatenate 'string
```

```
(append (list "<HTML>")
        strings
        (list "</HTML>"))))
```

The `&rest` keyword when we define our function tells Lisp that any unprocessed actual arguments¹ should be grouped into a list & given to our function as a single argument. You can demonstrate or verify this for yourself at the Lisp command line like this:

```
[1]> (defun moo (&rest lst) lst)
MOO
[2]> (moo 1 2 3)
(1 2 3)
[3]>
```

`Moo`² returns its single, “`&rest`” argument, & we can see that it’s a list containing the actual arguments.

So all the strings that are actual arguments to our `html` function become one list, which we link with beginning & ending tags & then convert to a string which we return. Here are examples of `html` in action:

```
[4]> ;; An easy example
(html "abc")
"<HTML>abc</HTML>"
[5]> ;; See what happens if we had HEAD
;; & BODY functions & we called HTML on
;; their results.
(html "<HEAD><TITLE>tee eye tee ell ee</TITLE></HEAD>"
     "<BODY><P>That's the word
for me.</P></BODY>")
"<HTML><HEAD><TITLE>tee eye tee ell
ee</TITLE></HEAD><BODY><P>That's the word
for me.</P></BODY></HTML>"
[6]>
```

(I put a line-break in the string that `html` generated because it was one line & too long for the page. In reality, it was all one line.)

In the next chapter, we’ll write functions for many other HTML tags.

¹Terminology refresher: A function has *formal arguments* & *actual arguments*. This is independent of Lisp; it applies to virtually any programming language.

²Why “moo”? I got tired of people saying “foo”, so I’m trying something different. And in an object oriented world (though this tutorial isn’t object oriented), isn’t “moo” more appropriate than “foo”?

Chapter 5

Other Types of Tags

With the `html` function, we've seen how to do it, now we need to create similar functions for other HTML tags such as `head`, `title`, `body`, `p`, `ol`, `strong`, & a bunch of others. That'd be a lot of typing if we did it all ourselves, & then when you wanted to fix a bug, you'd have to fix it in each of those functions. Yuck.

It's better to let Lisp do the work for us, & that's what macros are for. Instead of writing our function, we'll make a macro to do it. We'll tell the macro the name of the function. It'll get the HTML tag name from the function name, & it'll generate code which will define the function for us.

To write the macro, start with the actual code for the `html` function because that's one of the output cases for our macro. In fact, when we invoke the macro with `html`, we want to get the code we already have for our `html` function. So start with that original `html` code. Here it is as a reminder:

```
(defun html (&rest strings)
  "Concatenate the arguments, then wrap them
in <HTML> & </HTML> & return the new string."
  (apply #'concatenate 'string
         (append (list "<HTML>")
                 strings
                 (list "</HTML>")))))
```

Now make it the body of the macro. The macro has one argument, which is the name of the HTML tag. The old code becomes the *output* of the macro, so we'll quote it, but we won't use the normal, forward-tick (a.k.a. apostrophy or ') quote because we need to do some replacements in it. We'll use the backward-tick (a.k.a. `) quote because that allows us to do some substitutions within the quoted form. So put the `html` function within the macro, like this:

```
(defmacro defhtml-region (name)
  `(defun html (&rest strings)
     "Concatenate the arguments, then wrap them
```

```
in <HTML> & </HTML> & return the new string."
  (apply #'concatenate 'string
         (append (list "<HTML>")
                 strings
                 (list "</HTML>")))))
```

See that “html” right after the `defun`? We want that to be the name we give the macro. So we replace “html” with “,name”. Notice the comma in front of `name`. Within the back-tick quote, the comma evaluates the expression that follows it. We want it to evaluate the `name` we give the macro. After you do the replacement, you’ll have a macro like this:

```
(defmacro defhtml-region (name)
  `(defun ,name (&rest strings)
     "Concatenate the arguments, then wrap them
in <HTML> & </HTML> & return the new string."
     (apply #'concatenate 'string
            (append (list "<HTML>")
                    strings
                    (list "</HTML>")))))
```

You can paste that into your Lisp, or save it in a file that you load into your Lisp. Then try it out, like this:

```
[12]> (defhtml-region html)
HTML
[13]> (html "abc")
"<HTML>abc</HTML>" ; Cool!
[14]> (defhtml-region p)
P
[15]> (p "abc" " 123")
"<HTML>abc 123</HTML>" ; Woops.
```

It defines the functions we’ve requested, but we need to use the actual tag name instead of a hard-coded “<HTML>”. We can generate the actual tag as a string with `format`, but because I’m lazy (& thinking ahead), let’s put it in another function. We’ll have one function to create a beginning tag as a string & another function to create an ending tag as a string. Those functions might look like this:

```
(defun tag-begin (tag)
  (format nil "<~A>" tag))
(defun tag-end (tag)
  (format nil "</~A>" tag))
```

Now modify our `defhtml-region` macro to use those. (And while we’re at it, we’ll remove the documentation string from the function that the macro produces because it’s taking up space without offering much value for now.) Now our `defhtml-region` macro looks like this:

```
(defmacro defhtml-region (name)
  '(defun ,name (&rest strings)
    (apply #'concatenate 'string
      (append (list (tag-begin ',name))
              strings
              (list (tag-end ',name))))))
```

Notice our two new uses of the comma (,). We have them in front of `name` again, but they are behind quotes. Weird, at first thought. Those commas evaluate `name` like the first comma does, but we want to quote the resulting symbol. To see for yourself, type “(macroexpand-1 '(defhtml-region p))” into your Lisp & look at the results. Then remove those two quotes from the macro, use it to redefine one of your HTML functions, maybe `p`, & do the `macroexpand-1` thing again to see the difference in code. Then try to evaluate “(p "abc")” to see the difference in functionality. (This difference is that you’ll now see an error.)

Load that macro into your Lisp & use it on the command line, like this:

```
[35]> (defhtml-region head)
HEAD
[36]> (defhtml-region body)
BODY
[37]> (defhtml-region h1)
H1
[38]> (defhtml-region h2)
H2
[39]> (defhtml-region p)
P
[40]> (defhtml-region ol)
OL
[41]> (defhtml-region li)
LI
[42]> (defhtml-region em)
EM
[43]> (defhtml-region strong)
STRONG
[44]> (defhtml-region title)
TITLE
[45]> (html (head (title "title"))) (body (h1 "An important
Paper")
  (p "Blah blah "
    (em "bluie")
    "!"))
"<HTML><HEAD><TITLE>title</TITLE></HEAD><BODY><H1>An important
Paper</H1><P>Blah blah <EM>bluie</EM>!</P></BODY></HTML>"
```

Now create a file called “html.lisp”. Put our two functions, `tag-begin` &

tag-end in it. Put our macro in it. Then append these lines to it:

```
(defhtml-region body)
(defhtml-region em)
(defhtml-region h1)
(defhtml-region h2)
(defhtml-region h3)
(defhtml-region h4)
(defhtml-region head)
(defhtml-region html)
(defhtml-region li)
(defhtml-region ol)
(defhtml-region p)
(defhtml-region strong)
(defhtml-region table)
(defhtml-region title)
(defhtml-region td)
(defhtml-region th)
(defhtml-region tr)
(defhtml-region ul)
```

Notice how much typing the `defhtml-region` macro saved us. We already have a workable HTML generating library for Lisp, though it's missing features.

Chapter 6

Attributes

Possibly the most important missing feature is attributes for tags. We need them for colors & fonts. The “A” tag is useless without attributes, & HTML is virtually castrated without the A tag.

I’d like to specify attributes as an association list. For example, to center something, I’d like to type “’((align . "CENTER"))”. For a table that has a border & is as wide as the screen, I’d like to type “’((border . "1") (width . "100%"))”.

Both of those example lists in the previous paragraph are *association lists*, a very old Lisp data structure. Programmers of yore used association lists much as we use hash tables today.¹ Association lists are searched linearly, so the cost of searching them is $O(N)$, whereas a hash table is $O(K)$, but for short lists, that won’t matter. Association lists are easy for the programmer to type & easy for Lisp to process at run-time. Attribute lists for HTML tags will almost always be very short, even empty, so association lists will be a fine way to represent the attributes.

The elements of an association list are pairs. Remember that `cons` cells link together the elements of a list in Lisp. Right? In fact, a list is a `cons` whose tail is `nil` or is another `cons`. And another name for a list’s tail in Lisp is `cdr` because a `cons` has two parts: a `car` (head) & a `cdr` (tail).²

When you type a list of dotted pairs, you tell `read` to create an association list. Actually, it creates an association list as a side-effect of creating a regular list whose elements are the special-case dotted pairs.

We could make the attributes (association list) for an HTML tag be the second argument of our HTML functions. So we’d use them like this:

```
(html '())
```

¹Hash table reliance might not be in the vogue among Lisp programmers, but Perl programmers use them at the drop of a hat.

²I pronounce `cdr` like “could-er”. People also pronounce it like “coo-drè”. I presume I don’t need to tell anyone how I pronounce `car`. Anyone can see it’s pronounced like “boogiepop jiggy bottom”.

```
(head '() (title "I am titular"))
(body '()
  (h1 '((align . "CENTER")) "Generating HTML with Lisp")
  (p '() "This is a tutorial for new Lisp"
      " programmers, but not total"
      " beginners. ...")

  (table '((border . 1))
    (tr '()
      (td '((align . "RIGHT")) "eenie")
      (td '((align . "LEFT")) "meanie"))
    (tr '()
      (td '((align . "RIGHT")) "minie")
      (td '((align . "LEFT")) "moe")))))
```

While that isn't bad, those empty lists of attributes don't contribute to readability. Because we're using `&rest` in the argument lists of our HTML functions, we can't use a keyword argument before it, & an optional argument wouldn't help much. Either the attributes must be a required argument, or the functions could look at the first argument in their `&rest` lists, & if that first argument is a list, the function could use it as an attribute list. Otherwise, it stays with the rest of the arguments to become the body of the HTML region.³

Here's what I'm talking about. We could make it a required argument to our HTML functions. In that case, our functions (as generated by `defhtml-region`) would look like this: “(defun html (attribs &rest strings) ...)”.

Or we could make it an optional argument by having our HTML functions examine their first argument. The HTML functions (as generated by `defhtml-region`) would look like this:

```
(defun html (&rest args)
  (let ((attribs (and (consp (first args))
                     (first args)))
        (strings (if (consp (first args))
                     (rest args)
                     args))))
    ;; Process the attributes
    ...

    ;; Process the strings, as we already
    ;; do.
    ..))
```

I like this solution because we'll need to type the attributes for a tag only when there are attributes for a tag. For most tags, where we don't need at-

³A third possibility is to accomplish this optional attribute lists feature by use of CLOS, but I don't wanna.

tributes, we only type the strings that will go in the tags. A possible disadvantage is that, if we ever mistakenly give a list of strings to one of our HTML functions, it will attempt to use the list as tag attributes, which will be an error that is possibly difficult to debug.

First, let's re-write `defhtml-region`. It must extract the attributes, if any, from the arguments. If we put the burden of processing the attributes in the `tag-begin` function, then `defhtml-region` can remain unchanged otherwise. We'll need to add an attributes argument to `tag-begin`. The modified function & the modified macro will be:

```
(defun tag-begin (tag attribs)
  ;; This is incorrect, but just roll with it
  ;; for the moment. Will fix it soon.
  (format nil "<~A ~A>" tag attribs))

(defmacro defhtml-region (name)
  '(defun ,name (&rest args)
    (let ((attribs (and (consp (first args)) (first args)))
          (strings (if (consp (first args))
                      (rest args)
                      args)))
      (apply #'concatenate 'string
             (append (list (tag-begin ',name attribs))
                     strings
                     (list (tag-end ',name)))))))
```

For now, `tag-begin` puts the list of attributes in the beginning tag. That won't be correct HTML, but it'll let us debug the macro; we'll fix the attributes later.

The main change is to `defhtml-region`. The first part of the `let` figures out whether the arguments start with a list of attributes. If they do, then `attribs` gets the attributes & `strings` gets the rest, which should be a list of strings. If the arguments do not begin with a list, then `attribs` gets the empty list (`nil`), & `strings` gets the entire list of arguments.

Let's test the changes. Copy them to your `html.lisp` file & load them into your Lisp. Then try tests like these:

```
[1]> (load "html.lisp")
T
;; Test TAG-BEGIN with no attributes.
[2]> (tag-begin 'HTML '())
"<HTML NIL>" ; The NIL is incorrect HTML, but
              ; we'll fix that soon.
;; Try it with one attribute.
[5]> (tag-begin 'p '((align . "CENTER")))
"<P ((ALIGN . CENTER))>"
```

So we know that `tag-begin` prints the attributes, even though it prints them as incorrect HTML. We'll fix that soon; at least we know that `tag-begin` recognizes its new argument.

Now test the new `defhtml-region` itself.

```
[13]> (html (head (title "no attribs"))
(body
  (h1 '((align . "CENTER")) "attributes")
  (p "Again, no attributes.")))
"<HTML NIL><HEAD NIL>
 <TITLE NIL>no attribs</TITLE>
</HEAD><BODY NIL>
 <H1 ((ALIGN . CENTER))>attributes</H1>
 <P NIL>Again, no attributes.</P>
</BODY></HTML>"
```

(I've inserted line breaks & some whitespace in the result for readability & so that the line isn't too wide for the page.)

When we don't give attributes to an HTML function, the resulting begin tag has "NIL" in it, so it correctly interpreted the arguments as not having attributes (though they were incorrectly inserted into the HTML – but we'll fix that). When we did use attributes, for the `h1` tag, those attributes went into the begin tag. That's all as it should be.

Now we need to fix the `tag-begin` function. It always has an attributes argument, but when that argument is the empty list, there are no attributes. Otherwise (the attributes list is not empty), it's an association list. Each element in the association list is a pair whose `car` is the attribute name & whose `cdr` is its value. In that case, we need to convert each of those "(attr . val)" pairs into "ATTR=VAL". Edit `tag-begin` to look like this:

```
(defun tag-begin (tag attribs)
  (apply #'concatenate 'string
    (append (list (format nil "<~A" tag))
      (mapcar #'(lambda (pair)
        (format nil
          "~A=~A\"
          (car pair)
          (cdr pair)))
        attribs)
      (list ">"))))
```

`Tag-begin` is more complex now. A begin tag always starts with "<" & the tag's name. You can see that in the first `format`. It always ends with ">", & you can see that in the last line of the function.

Between the first `format` & the last line of the function (the second `list`), we have to convert the pairs in the list of attributes to "ATTR=VAL" strings. We do that with `mapcar`. Notice that when the list of attributes is empty, `mapcar`

will return an empty list. We convert those three lists to one with `append` & then concatenate them into a single string by applying `concatenate` to them. Whew!

Edit the `tag-begin` function in your `html.lisp` file, load it into your Lisp, & try it out:

```
[15]> (load "html.lisp")
;; Loading file html.lisp ...
;; Loading of file html.lisp is finished.
T
[16]> (h1 '((align . "center")) "i'm the title")
"<H1 ALIGN=\"center\">i'm the title</H1>"
[21]> (table '((border . 1))
  (tr (td "no attrs")
      (td '((align . right)) "with attrs"))))
"<TABLE BORDER=\"1\"><TR>
<TD>no attrs</TD>
<TD ALIGN=\"RIGHT\">with attrs</TD>
</TR>
</TABLE>"
```

(Once again, I've added line breaks for readability.)

Now let's create that all-important "A" tag. To your `html.lisp` file, add this one line: `(defhtml-region a)`. That's it! Load the file into Lisp & try it out:

```
[23]> (load "html.lisp")
;; Loading file html.lisp ...
;; Loading of file html.lisp is finished.
T
[26]> (a '((href . "http://cybertiggyr.com/gene/"))
  "http://cybertiggyr.com/gene/")
"<A HREF=\"http://cybertiggyr.com/gene/\">http://cybertiggyr.com/gene/</A>"
```

Add a `defhtml-region` form for the `IMG` tag while you're at it. (That's an exercise for the reader.)

Chapter 7

Some Nice Touches

We have a usable HTML library, but there are still some features I'd like to add.

7.1 Newlines Around Tags

It'd be nice if the output wasn't in one long line. I'd feel safer with a newline after the closing HTML tag. I'd also like a newline before some other opening tags to improve readability.

We could do it with keyword arguments to the `defhtml-region` macro. To insert a newline after the closing tag, we could use a keyword argument `:append-newline`. It would default to `nil`; any other value would make the HTML function append a newline to the string it generates. We could have another keyword argument, `:prepend-newline`, which would enable a newline before the beginning tag.

There are a couple of ways to implement this. One is to make the “to newline or not to newline” decision in `defhtml-region`. For run-time efficiency, that might be the best choice because it can decide whether or not to insert the newline before it defines the function for the HTML tag.

Look at the definition of `defhtml-region` we have been using (before the modifications I've discussed in the past few paragraphs). Here it is:

```
(defmacro defhtml-region (name)
  '(defun ,name (&rest args)
    (let ((attribs (and (consp (first args)) (first args)))
          (strings (if (consp (first args))
                       (rest args)
                       args)))
      (apply #'concatenate 'string
              (append (list (tag-begin ',name attribs))
                      strings
                      (list (tag-end ',name)))))))
```

We want the macro to adjust the arguments to `append`. If the macro's `prepend-newline` keyword argument is true, we want the macro to insert a newline (newline as a string in a list, actually) as the first argument to `append`. Similarly, if the macro's `append-newline` argument is true, we want it to insert a newline (as a string in a list) as the last argument to `append`.

Here's how I edited `defhtml-region`. I'd explain how it works & how I got there, but I'm not sure. I'm not all that great with macros, & we've moved into the area where macros make my head hurt. I wish I could explain it, but I can't. Similarly, I expect that macro I've written is not the greatest.

```
(defmacro defhtml-region (name &key
                          prepend-newline
                          append-newline)
  ;; This FORMAT is for debugging.
  ; (format t "~&defhtml-region ~A" name)
  '(defun ,name (&rest args)
    (let ((attribs (and (consp (first args))
                       (first args)))
          (strings (if (consp (first args))
                       (rest args)
                       args)))
      (apply #'concatenate 'string
             (append ,(if prepend-newline
                          '(,(format nil "~%"))
                          ())
                    (list (tag-begin ',name attribs))
                    strings
                    (list (tag-end ',name))
                    ,(if append-newline
                        '(,(format nil "~%"))
                        ())))))
```

We've added two keyword arguments to `defhtml-region`. They are `prepend-newline` & `append-newline`. If `prepend-newline` is true, the function for the HTML tag will always insert a newline character before the beginning tag. If `append-newline` is true, the function will append a newline character after the ending tag. Both `prepend-newline` & `append-newline` default to false because I have not provided a specific default value for them, & that's the way keyword arguments work.

You might notice the two “`(format nil " %")`” forms. I did that because it's one safe & easy way to create a string with an embedded newline. It's not the most efficient way to do it because it creates a new string each time we define a function for some HTML tag, but it works, it's safe, & it's easy.

Update your `html.lisp` file with this new definition of `defhtml-region`. While you're at it, update `defhtml-region` calls so that the right tags will append newlines or prepend newlines. I put `append-newline` on the HTML tag

because I feel better when the file ends with a newline, though I know browsers don't require it. I put `prepend-newline` on some other tags to aid readability for humans (well, for me, anyway). Here is the entire list of `defhtml-region` forms I have now:

```
(defhtml-region a)
(defhtml-region body :prepend-newline t)
(defhtml-region em)
(defhtml-region h1 :prepend-newline t)
(defhtml-region h2 :prepend-newline t)
(defhtml-region h3 :prepend-newline t)
(defhtml-region h4 :prepend-newline t)
(defhtml-region head :prepend-newline t)
(defhtml-region html :append-newline t)
(defhtml-region img)
(defhtml-region li :prepend-newline t)
(defhtml-region ol :prepend-newline t)
(defhtml-region p :prepend-newline t)
(defhtml-region strong)
(defhtml-region table :prepend-newline t)
(defhtml-region title :prepend-newline t)
(defhtml-region td)
(defhtml-region th)
(defhtml-region tr :prepend-newline t)
(defhtml-region ul)
```

Now load `html.lisp` into your Lisp & test it. Here's what I did:

```
[61]> (load "html.lisp")
;; Loading file html.lisp ...
;; Loading of file html.lisp is finished.
T
[62]> (html
      (head
        (title "title")))
      (body
        (h1 "title")
        (p "I'm a paragraph.")
        (p "I'm another.")
        (h2 '((align . center))
          "Attributes still work.))))
"<HTML>
<HEAD>
<TITLE>title</TITLE></HEAD>
<BODY>
<H1>title</H1>
<P>I'm a paragraph.</P>
```

```
<P>I'm another.</P>
<H2 ALIGN="CENTER">Attributes still work.</H2></BODY></HTML>
"
```

This time I did not insert line breaks for readability. What you see is exactly what came out of the HTML functions.

Notice that we made these changes without actually editing all the HTML tag functions we have. Instead, we edited `defhtml-region` & let it do the rest of the work. Well, yes, we had to edit the invocations of `defhtml-region`, too.

7.2 Character Arguments

Sometimes I'd like to use characters & symbols as arguments to the HTML tag functions. Maybe I want to embed a newline in the HTML code for human readability. Or maybe I want to embed a quote character. I could create strings in Lisp & use those as arguments to the HTML functions, but I'm lazy; I'd rather let our library do the work.

You might notice that I have used symbols, integers, & strings for the attribute arguments. I can get away with that because `tag-begin` uses `format` to convert attributes & their values to strings, & the `format` field I used, `"~A"`, silently converts almost any type of argument. We can use the same trick to convert characters & numbers to strings. Then I'll be able to use strings, characters, numbers, symbols, & who knows what else as arguments to the HTML functions.

To refresh your memory, here is the `defhtml-region` macro as it exists now, before making the changes I'm discussing.

```
(defmacro defhtml-region (name &key
                          prepend-newline
                          append-newline)
  '(defun ,name (&rest args)
    (let ((attribs (and (consp (first args))
                       (first args)))
          (strings (if (consp (first args))
                      (rest args)
                      args)))
      (apply #'concatenate 'string
              (append ,(if prepend-newline
                          '(,(format nil "~%"))
                        ())
                    (list (tag-begin ',name attribs))
                    strings
                    (list (tag-end ',name))
                    ,(if append-newline
                        '(,(format nil "~%"))
                        ())))))
```

The *strings* variable currently must be a list of strings. We can apply `format` & its “~A” field to each element of *strings* to convert whatever they are to strings. We could do it by using a `mapcar` where that *strings* is now, but `defhtml-region` is getting pretty big. Let’s do this extra work in a function that takes the *strings* list as its only argument, converts each item to a string, & returns a new list of things that are definitely strings. Let’s call the new function `ensure-strings`. Here it is:

```
(defun ensure-strings (lst)
  "Convert each element of LST to a string &
  return a new list of the new strings."
  (mapcar #'(lambda (x)
            (format nil "~A" x))
          lst))
```

It’s about as simple as a non-trivial function can get. Add that to your `html.lisp` file. Now let’s update `defhtml-region` to use `ensure-strings`. Also, because *strings* isn’t necessarily a list of strings any more, I’ve changed its name to *lst*. Here’s the new version:

```
(defmacro defhtml-region (name &key
                          prepend-newline
                          append-newline)
  `(defun ,name (&rest args)
    (let ((attribs (and (consp (first args))
                       (first args)))
          (lst (if (consp (first args))
                  (rest args)
                  args)))
      (apply #'concatenate 'string
             (append ,(if prepend-newline
                          '(,(format nil "~%"))
                          ())
                   (list (tag-begin ',name attribs))
                       (ensure-strings lst)
                       (list (tag-end ',name))
                       ,(if append-newline
                          '(,(format nil "~%"))
                          ())))))
```

Now load `html.lisp` into Lisp & give it a whirl, like this:

```
[64]> (load "html.lisp")
;; Loading file html.lisp ...
;; Loading of file html.lisp is finished.
T
[65]> (p 1 #\Space 2 #\Space 3 #\Newline 4)
```

```
"
<P>1 2 3
4</P>"
```

In Lisp, you specify a character with “#\” followed by a character. There are some distinguished symbols which, when following #\, refer to certain characters. So #\Space indicates a space character, & #\Newline indicates a newline character. Also in this little test you can see that integers work, too.

7.3 Two Special Tags

In HTML, the `br` & `hr` tags are special because they do not have closing tags. (You can use closing tags with them, but you get weird results on some browsers, at least with `br`.) So our `defhtml-region` macro won’t work for them.

Rather than expand `defhtml-region` (yet again), let’s just write these two functions directly. I don’t think I’ve ever seen attributes to `br`, & I’ve rarely seen attributes to `hr`, so let’s take the easy way out & make them no-argument functions. Here they are:

```
(defun br () "<BR>")
(defun hr () "<HR>")
```

Add those to your `html.lisp` file.

7.4 Make a Package

Now that we have an HTML library, let’s put it in a package. It’ll help avoid name collisions, it’ll group all the functions of our library into one place (the package), & hey, every self-respecting library has to be in its own package.

Use `defpackage` to make a package. It’s a complex form, what with nested lists that begin with keywords & such. We’ll want to export the symbols that we expect other packages to use. For safety, we’ll want to import, or “use”, all the symbols in the Common Lisp package. At the beginning of your `html.lisp` file, put this `defpackage` form. And put the `in-package` form after it, too. They should both come before the other forms in the `html.lisp` file. Here’s what I have:

```
(defpackage html
  (:use "COMMON-LISP")
  (:export "A"
           "BODY"
           "BR"
           "EM"
           "ENCODE"
           "H1"
```

```
"H2"  
"H3"  
"H4"  
"HEAD"  
"HR"  
"HTML"  
"IMG"  
"LI"  
"OL"  
"P"  
"SMALL"  
"STRONG"  
"TABLE"  
"TITLE"  
"TD"  
"TH"  
"TR"  
"UL"))  
(in-package html)
```

The `in-package` tells Lisp that what follows, until another `in-package` or until the file ends, is *in* the package called HTML.

From now on, code that accesses our HTML library will either need to import the HTML symbols, such as with “`use-package`”, or it will need to prefix the symbols with “`html:`” so the Lisp reader can find them. I’ll show how to do this in the examples in Chapter 9.

Chapter 8

The Library So Far (Draft Release)

If you've followed along & edited your own `html.lisp` file as you read, yours is probably the same as mine. Nevertheless, if you want to see my completed source it is `./html-draft.lisp`. (It's a draft because we'll be optimizing it in Chapter 10.

Chapter 9

Putting It To Use

Let's put our new library through its paces, see what it can do.

9.1 Miles Per Gallon

For starters, let's generate a report into an HTML file. It's hardly as glamorous as CGI, but it's a real use of HTML.¹

Let's create a report of miles per gallon & dollars per mile.

Because our HTML library is now in its own package, we'll need to import its symbols. If we used only some of those symbols, we might `import` the individual symbols we intended to use, or we might qualify them explicitly with `"html:"`. We'll use most symbols in our library, so we'll import all of them with one `use-package` form. So the Lisp file for our report will start like this:

```
;;; -*- mode: Lisp -*-  
  
(load "html.lisp")  
(use-package 'html)
```

The code for the program is in `./mpg.lisp`. I've hard-coded some data into it; that's the global `*gene*` symbol. And yes, the data is real, from my own car, a 1999 Suzuki Swift. (The gasoline prices are in United States dollars, mostly in or around Seattle, Washington.)

The `one-row` function generates a string for one row of in the report. The `report` function loops over all the data. The `rf` function is for convenience; it runs `report` & writes the output to a file which you may open in your web browser.

Sample output from the program is in `./mpg.html`.

¹This is how I generate many of the for my web site.

9.2 CGI

This isn't a tutorial on CGI, but let's try a little CGI anyway, but just some basic stuff. We'll generate a report (again, but a different one) without any input from the viewer. No *post* or *get* arguments.

To keep it simple but allow for long output (for the next example), let's generate random lists of some Lisp symbols. I'll just hard-code the list of symbols that can do into the lists.² Also in preparation for the next example, we'll put the code for this in a file called `excgi.lisp`.

Now we make a CGI program that uses the functions in `excgi.lisp`. The source for that program is `excgi0.cgi.sh`. The program itself is at <http://CyberTiggyr.COM/gene/lh/excgi0.cgi>. You can run it by pointing your browser at that URL.

The page takes about twenty seconds from the time you visit it until you've downloaded everything, so be patient. For what it's worth, my web server is Apache running on Open BSD on a 200 MHz Pentium.

9.3 CGI with a Twist

For the final example, let's generate the same report as in the previous example, but let's make use of the "Content-Length" HTTP reply header in the hopes that browsers will show the percentage of the document that has loaded. If all goes well, your browser will show a progress bar as it downloads the page. That's an advantage of generating the HTML in a single string before sending it.

The source for this second CGI example is `excgi1.cgi.sh`. The program itself is <http://CyberTiggyr.COM/gene/lh/excgi1.cgi>. You can run it by pointing your browser at that URL.

(I don't see a progress bar in the three browsers I've tried (Mozilla, Lynx, & Internet Explorer). I'm surprised; I thought that, if the browser had the length of the coming document, it would naturally display a progress bar. Live & learn.)

²It's a contrived, none too useful, example. I know. Sorry. But I've written this whole thing in one day, & I'm really tired. Maybe I'll have a better example in a later revision.

Chapter 10

Improving Performance

I'd like to make the two previous CGI programs run faster. Twenty seconds is a long time to wait for a web page. Maybe we can get it down to five seconds.

10.1 Thoughts about Optimization

I enjoy optimization. You start with a (hopefully) working program, & make it faster, which impresses most people except for the very stupid or the similarly technically experienced. If you do it right, you won't break the program along the way, so you'll always have a working program. Optimization is like debugging except that the program is already working, so you don't get angry because it doesn't work or because someone (maybe you) wrote crappy code. It's easier than programming from scratch. It's a good way to spend a few hours, & it mixes well with a set of headphones & your favorite tunes.

Optimization is one of the few activities in computer science that could legitimately claim to be a science. A discipline isn't a science unless it uses the scientific method. The essential features of the scientific method are in Figure 10.1.

observe Observe some phenomenon.

hypothesize Create a hypothesis to describe the causes & effects involved in the phenomenon.

experiment Perform an experiment featuring a *control group* & an *experimental group*.

repeat Return to the observation step by comparing the results of your experiment with your hypothesis.

Figure 10.1: The essential features of the scientific method

To optimize a program, you observe its performance then hypothesize about what consumes the time & how you could reduce that. Using your hypothesis, you change the program in ways that, you hope, will improve performance. You run it again on the same inputs & measure the new program's performance. The new program is the experimental group, & the old program is the control group. Of course, you compare their performances & repeat until the program is fast enough.

10.2 The Importance of Profilers

You can't do a proper job of optimizing without a profiler. If you think of optimization as a science as I described in Section 10.1, a profiler makes your observations vastly – *vastly* – more precise. Without a profiler, your guesses about where to optimize are just guesses. You still need to make some educated guesses to use a profiler (because you need to guess at what to profile), but your optimization efforts will be directed instead of just SWAGs.¹

Let's test my claim that your guesses without a profiler (well, my guesses without a profiler) are just guesses. Before I've run a profiler on the CGI programs I'd like to optimize, here are my guesses about where the run-time goes:

1. We construct long strings when generating HTML. We make bigger & bigger strings from smaller ones by concatenating them. I suspect that the largest single chunk of time goes into concatenating the strings because `concatenate` must allocate large chunks of contiguous memory & then copy lots of characters.
2. I have heard that `format` is a notorious time pig. I've never tested that claim, but it could be right, so I'll bet it's the second largest eater of time.
3. I use `apply` several times, & sometimes it'll have large strings as arguments. I suspect this is the third-largest consumer of run-time.

I swear that those were my expectations before I ran the profiler. Let's see how right I am.²

Let's also take a moment to consider the implications if my predictions are correct. If `concatenate` really does consume most of the time, one fix might be to remote concatenate. Each of the HTML functions would keep the same argument list (the optional attributes list & the `&rest` argument), but each would return a list of strings instead of one string. At the very top, we could concatenate all those strings just once, or we could even write each string to the output stream without concatenating them at all.

¹A SWAG is a Scientific Wild-Assed Guess. A perfect example of a SWAG is what you reply when your boss asks how long it'll take to write a program.

²If my predictions are right, then I'm right. If my predictions are wrong, then I'm write because I claim that your performance predictions without a profiler are guesses (i.e., they're wrong), so I'm right.

A change like that wouldn't be painful (I hope), but it could run into the limit on the maximum number of function arguments. (It's in a constant defined by Common Lisp. The constant is named `call-arguments-limit` or something similar, but I don't remember the exact name.) Common Lisp only specifies that the limit should be at least 50. I can easily imagine generating more than fifty strings in an HTML page. Maybe each of the HTML functions could examine the list of strings it's about to return, & if it's close to the limit on number of call arguments, it could use concatenate. It'd work, & it wouldn't require too much typing on our part because we are using a macro to write the HTML functions for it, but eew! It's just gross.

It's all speculation for now, but see how much work we might have to do if we ran ahead with my speculations before we did a proper profiling run.

10.3 Our Profiler

I use `clisp`³, which doesn't have a profiler⁴, but that won't stop us. In *Paradigms of Artificial Intelligence: Case Studies in Common Lisp* ([Nor92]), Peter Norvig implements profiling tools in Lisp itself.

I copied that code into a file, `profile.lisp`. I had to write one function, `profile-count`, because I did not see it in the book. Maybe it was omitted from the text, or maybe I just didn't see it. Also, I made a small modification to `profile-report` so that it would order the report with the biggest time pigs at the beginning. Other than those two changes, that file is a copy of the code from [Nor92]. Download that file & save it on your system. I assume the file is called `profile.lisp`.

I won't explain how Norvig's profiler works; Doctor Norvig does that in [Nor92]. Here's a brief description of how to use it.

The easiest way to use it is with the `with-profiling` macro. It's analogous to the `with-open-file` & other `with-...` macros in Common Lisp. The first argument to `with-profiling` is a list of the names of functions you want to profile. The other arguments are forms to evaluate. If I wanted to profile functions `f`, `g`, & `format`, I might do this:

```
> (load "profile.lisp")
T
> (with-profiling (f g format)
  (f) ; presumably, F & G call
  (g)) ; format.
```

Before it exits, `with-profiling` will print a report. Here's an actual profiling run so you can see:

³<http://www.cliki.net/CLISP>

⁴It seems that whenever I claim `clisp` doesn't do something, someone enlightens me & I learn that it does. So maybe it does & I'll be updating this article in the next few weeks. As of 16 August 2003, I believe `clisp` does not have a profiler.

```

[1]> (load "profile.lisp")
T
[2]> (defun f () t)
F
[3]> (defun g () (sleep 2))
G
[6]> (with-profiling (f g) (f) (g))
Total elapsed time: 2.01 seconds.
  Count  Secs Time% Name
    1    2.01  100% G
    1    0.00   0% F
NIL

```

To see a list of the functions being profiled, evaluate “(profile)”.

If you start getting weird behaviours, maybe because some error prevented the profiler from un-profiling some functions, evaluate “(unprofile)”. That should remove the profiler from all functions that are being profiled. If that doesn’t fix things, exit your Lisp & re-start it.

10.4 Let’s Profile It Already

Download `excgi.lisp` because that’s what we’ll be optimizing. You might want to look at the source for `excgi0.cgi.sh` to see how it uses the functions from `excgi.lisp`, but you won’t actually need the code from it.

Also download `profile.lisp`.

You might want to re-start your Lisp, just in case. Prime it with the functions we’ll need by evaluating these forms:

```

> (load "html.lisp")
T
> (load "excgi.lisp")
T
> (load "profile.lisp")
T

```

If you didn’t restart your Lisp, & if you have been editing your `html.lisp` file as you go, you’ll just need to evaluate “(load "profile.lisp)”.

We’ll want to use the profiler on the same Lisp expression(s) in the same way multiple times, so let’s do that in a function. I’m calling it `moo`, & here it is (type this into your Lisp):

```

(defun moo ()
  (with-profiling (excgi html::tag-begin
                        html::tag-end html::ensure-strings
                        append
                        concatenate

```

```

        format list)
(length (excgi 1000)))

```

The “(length (excgi 1000))” expression is the meat from `excgi0.cgi.sh`, but instead of printing it, we’ll just generate it. Also, instead of returning it from `moo` where that multi-kilobyte string would scroll my screen for pages, we’ll return its length.

The first argument to `with-profiling` is a list of the names of the functions to profile. So how did I choose them? That’s where your pre-profiling guesses go. Since I officially suspect `concatenate`, & `format`, I included those. A lesser suspicion is `list` because it allocates memory, so I included it. (If our HTML library made much use of `cons`, `make-array`, & other functions which allocate memory, I would have included them.)

So why are `html::tag-begin` & `html::tag-end` in the list of functions? We could profile each of the HTML generating functions, but I didn’t want to type all their names. (A macro could have done it, but I didn’t even feel like writing that. I’m lazy today.) Our HTML functions all use `concatenate`, `append`, `list`, `format`, `tag-begin`, `tag-end`, & `ensure-strings`. Some of those functions are already being profiled for reasons I’ve already explained. If we also profile the last three of those functions, we’ll be checking most of the work done by our HTML functions without actually needing to profile each of the HTML functions.

I included `excgi` as a sort of sanity check. Since it’s the root of the tree of functions we’re profiling, we probably won’t learn anything from it. I mean, *all* of the execution time while you call `excgi` happens in `excgi`. That’s not news, & it’s not useful. Still, it seemed like a good idea to see its position in the profiling report. Like I said, it’s just a sanity check.

Originally, I also tried to include `apply` in the profiler’s report, but that gets a nasty run-time error. Specifically, it gets a stack overflow. That’s because the profiler itself uses `apply`. I can live without profiling `apply` because I don’t strongly suspect it. If we really, really needed to include `apply` in the profiling run, we could create our own wrapper for it & have our HTML functions use that wrapper, or maybe we could twiddle the profiler itself to be smart about how it uses `apply`.

We’re finally ready to run the profiler:

```

[2]> (moo)
Total elapsed time: 8.76 seconds.
  Count  Secs  Time%  Name
100198  5.87   67%    FORMAT
   1005  1.75   20%    ENSURE-STRINGS
     1   0.75    9%    EXCGI
   2011  0.14    2%    CONCATENATE
   1005  0.12    1%    TAG-BEGIN
   2010  0.06    1%    APPEND
   4020  0.05    1%    LIST

```

```

1005  0.03  0% TAG-END
283249

```

From the third & fourth columns of the table, we can see that 67 percent of the run-time goes into `format`, which was my suspected second-highest time pig. My first pick as the time-consumption culprit, `concatenate`, consumes two percent of the run-time, which pretty much doesn't matter. My third pick, `append`, consumes even less time.

It might seem like my suspicions were totally wrong, but your guesses usually don't even register, so they were remarkably good. (Since my prediction was that my predictions would be wrong, am I wrong because I was right, or right because I was wrong?)

The performance report we have, with `format` accounting for 67 percent of the time, is typical of an un-optimized program. If you add the times spent by the first two items on the report, `format` & `ensure-strings`, you account for over 80 percent of the run-time. This, too, is typical. In a program that has not previously been optimized with a profiler, you will almost always see that 80 percent of the run-time is eaten by one or two culprits. (This happens so frequently that you might consider betting money on it.) These are the low-hanging fruits of optimization opportunities. They are where a few small changes to your program will achieve a vast improvement in performance. In my experience, you usually have two to five of these opportunities in a program that hasn't previously been optimized. (This rule is not as certain as the 80 percent rule.)

So let's see what we can do about these easy opportunities.

Optional: Before changing the HTML library, you might want to copy the current version to another file, such as `html-draft.lisp`. That's so you can compare performances as we go. Or you could edit your only copy & take notes of old performances as we go. Or you could just rely on the numbers I show from my own computer.

Take a look at our HTML library in a text editor. Search for "format" to see the places where we use it. I see four key places: `tag-begin`, `tag-end`, `ensure-strings`, & `defhtml-region`. It's safe (*probably*) to ignore calls that happen just once, such as initializations of variables. Any one call of a function doesn't matter much. It's when we call a function many times; that's what matters. Either the function needs to be faster, or its caller needs to use it fewer times.

Let's think about each of those places where we use `format`.

`Tag-begin` calls `format` in a loop (via `mapcar`). That could be the culprit, but it's not glaringly inefficient. In fact, it looks pretty reasonable. We might replace the inner `format` with a `concatenate`, but then we'd lose the easy conversion of `(car pair)` and `(cdr pair)` to strings. We're looking for some glaring inefficiency, & `tag-begin` probably isn't it. Let's remember `tag-begin` as a suspect place if we don't find any others, but let's look elsewhere for now.

`Tag-end` is even less suspicious than `tag-begin`. Move on.

`Ensure-strings` calls `format` in a loop, & the profile report says that

`ensure-strings` is the second-biggest offender. Now *that's* the kind of thing we're looking for. And remember how we call `ensure-strings`. Each of our HTML functions calls it to convert each of its arguments to strings. What's more, the output of each function becomes the arguments of the enclosing HTML function, which is itself processed by `ensure-strings`. I'd bet that `ensure-strings` is our culprit. It goes to the top of the suspects list. In fact, I'd start twiddling it right now, but I want to checkout `defhtml-region` first, since it's the only item remaining on our list of places where we use `format`.

`Defhtml-region` is a macro, & we need to remember that when we analyze it. Macros are generally evaluated at load time or compile time, whereas functions are evaluated at run-time (which is also called eval time). Look at the first `format` call inside `defhtml-region`. Here is (out of context): “,(`format nil "~%"`)”. That comma in front of it causes it to be evaluated when the macro is expanded. Same goes for the other use of `format` in the macro. So those particular uses of `format` occur only when we define our HTML functions. We define only 20 HTML functions with `defhtml-region`, which would call `format` forty times. Those forty calls don't nearly account for all those uses of `format` that the profiler told us about. So `defhtml-region` is not the culprit. Let's tackle `ensure-strings`.

The job of `ensure-strings` is to ensure that each item in a list is a string. It should convert symbols, numbers, characters, & anything else to a string. It uses `format` to do that. But what about elements that are already strings? There's no need to convert them. What's more, we know that most arguments will already be strings. So we can probably reduce `ensure-string's` use of `format` just by recognizing when a list element is a string & returning that same element instead of using `format` to convert it to a string. So let's modify `ensure-strings` to look like this:

```
(defun ensure-strings (lst)
  "Convert each element of LST to a string &
  return a new list of the new strings."
  (mapcar #'(lambda (x)
             (if (stringp x)
                 x
                 (format nil "~A" x)))
          lst))
```

The inner, anonymous function now checks its argument, *x*. If *x* is already a string, there's no need to convert it, so it returns it. If *x* isn't a string, we convert it in the old way.

Make that change to your `html.lisp` file & let's test it:

```
[3]> (load "html.lisp")
T
[4]> (moo)
Total elapsed time: 5.22 seconds.
```

Count	Secs	Time%	Name
51610	2.93	56%	FORMAT
1005	1.07	21%	ENSURE-STRINGS
1	0.77	15%	EXCGI
2011	0.22	4%	CONCATENATE
1005	0.11	2%	TAG-BEGIN
4020	0.05	1%	LIST
2010	0.03	1%	APPEND
1005	0.03	1%	TAG-END

287322

We have some success. We reduced the total run-time by about 3 seconds. We reduced `format`'s relative time from 67 percent to 56 percent. (I'll admit that I'm surprised at the small change.) Notice that we reduced the number of times `format` is used by almost half, from 100,198 to 51,610. That's significant.

What if we could reduce it more? Are there other data types that could convert to a string more efficiently than via `format`? Symbols have names, which are strings. So when an item is a symbol, we could use `symbol-name` to get its name. That should be faster than `format`.

Before we modify `ensure-strings` again, let's think about this. If we realize that even more types can be converted to strings efficiently, we'll need to use lots of nested `ifs`. That'd be bad, so we could use `cond`, but even then, we'd be typing all sorts of `eq` expressions to compare the item's type. Instead of that, let's use `typecase`, which is a `case` form except that it looks at the type of its first argument. So here's the new `ensure-strings`:

```
(defun ensure-strings (lst)
  "Convert each element of LST to a string &
  return a new list of the new strings."
  (mapcar #'(lambda (x)
             (typecase x
              (string x)
              (symbol (symbol-name x))
              (t (format nil "~A" x))))
          lst))
```

Now, if `x` is already a string, we use it. If it's a symbol, we use its `symbol-name`. Otherwise, we convert it to a string with `format` (which is what we've been doing all along). Save that change in `html.lisp`, & let's observe it again:

```
[9]> (load "html.lisp")
;; Loading file html.lisp ...
;; Loading of file html.lisp is finished.
T
[10]> (moo)
Total elapsed time: 1.96 seconds.
  Count  Secs Time% Name
```

```

      1   0.73   37% EXCGI
    1005  0.56   28% ENSURE-STRINGS
    2015  0.30   15% FORMAT
    2011  0.20   10% CONCATENATE
    1005  0.12    6% TAG-BEGIN
    4020  0.06    3% LIST
    2010  0.03    2% APPEND
    1005  0.03    1% TAG-END
275068

```

Oh yeah! `Format` now consumes a mere 15 percent of the run-time. It has moved from top of the list to third. What's more, run-time is now 1.96 seconds, which is a mere 22 percent of the original 8.76-second run-time.

Let's see what else is possible. `Excgi` itself is now the biggest time-consumer. Take a look in it. So you don't have to look elsewhere, here it is:

```

(defun excgi (count)
  (html:html
    (html:head (html:title "excgi.lisp"))
    (html:body
      (html:h1 '((align . center)) "excgi.lisp")
      (apply #'concatenate 'string
        (loop for i from 1 to count
              collect (apply #'html:p (randlist)))))))

```

The loop is the only suspicious thing I see. Maybe we could add `randlist` & the functions it calls to our profile run. I don't see anything that looks all that suspicious, & it runs fast enough for me, so I'll stop here.

10.5 Knowing When To Stop

My requirements here were to optimize until I felt I had made my point & I didn't feel like doing more. That's why I quit so abruptly.

At work, it can be less obvious when to stop. The first time you run a profiler on a program, you'll often see a glaring time eater which consumes nearly 80 percent of the run-time. Like I said, that's the low-hanging fruit; a few small changes can easily more than double the program's performance. (Notice that we improved performance by nearly a factor of 4.5.)

Each time you improve something & then run the profiler, the new biggest offender will be less obvious. Notice that on our first profiling run, `format` consumed a whopping 67 percent of the time. After the first improvement, it dropped to 54 percent, which is still obvious but not quite as glaring. After the second fix, `format` dropped to 15 percent, third place on the list of offenders. As we left it, the top offender, `excgi`, consumes 37 percent of the run-time, & the next offender consumes 28 percent. That's far less obvious than `format`'s original 67 percent. After another two good improvements, I wouldn't be surprised if resultant top four offenders were at 20 percent, each.

As you improve performance of the obvious offenders, you flatten the relative times consumed by each function. It also gets more difficult to make each improvement. Eventually, the amount of work required for the next improvement eclipses the potential improvement. For example, maybe you've gone through many improvements, & the top nine offenders each use 10 percent of the time. The last improvement required a full day to pinpoint the location & to write the improvement. The next improvement will probably take at least that much time, & the potential benefit is only 10 percent. Remember that killing the first few big time-consumers could have improved performance by a factor of five, so current program requires 20 percent of the time the original program did. If the next improvement gets you a ten percent improvement, the program will run in 18 percent of the time required by the original program. Is it worth it to spend over a day for that kind of improvement? You'll have to decide. Specific, objective, measurable performance requirements can make the decision for you: Until you achieve that performance, you have to keep optimizing, but as soon as you achieve that performance, you can stop. Sadly, you rarely have good performance requirements like those. You usually have "It needs to be fast, really fast." Those kind of requirements are useless, at best.

10.6 Importance of Profilers, revisited

I wasn't tracking the time, but I think I did our changes to `ensure-strings` and write about them in 30 minutes. (Maybe it was 45, but no way was it more.)

So thirty minutes of work (which could have been less if I hadn't been writing an article) reduced the run-time by 88 percent.

Remember my musings in Section 10.2 about the changes that might be involved if `concatenate` was revealed as the culprit? I think I could implement them in about 30 minutes, which is pretty quick. If you look at the original profile report, you can see that `concatenate` consumed 2 percent of the run-time. If I had spent 30 minutes on my original hunch & reduced the cost of `concatenate` to zero (& I wouldn't have made that much of an improvement), I would have reduced run-time by 2 percent.

This is why a profiler is important. Without it, you can't make an informed decision about where to spend your optimization efforts. A profiler lets you take advantage of those low-hanging fruits of optimization opportunities.

Chapter 11

The Final Library (Final Release)

If you've followed along & edited your own `html.lisp` file as you read, yours is probably the same as mine. Nevertheless, if you want to see my final, optimized, completed source it is `./html-opti.lisp`.

Chapter 12

Exercise for the Reader

You may have noticed that I prefer to use upper-case HTML tags & attributes. If you prefer lower-case HTML tags & attributes, there's a simple modification you can make to `tag-begin` & `tag-end` to ensure that all your HTML tags & their attributes will have lower-case.

You might even create a global flag to select between upper-case tags & lower-case tags. Be sure to export the new symbol so functions outside of the HTML package can access it after they evaluate “`(use-package 'html)`”.

Chapter 13

Other File Formats

- This document is available in multi-file HTML format at <http://cybertiggyr.com/gene/lh/>.
- This document is available in Pointless Document Format (PDF) at <http://cybertiggyr.com/gene/lh/lh.pdf>.

Bibliography

- [Nor92] Peter Norvig. *Paradigms of Artificial Intelligence: Case Studies in Common Lisp*. Morgan Kaufmann Publishers, 1992. ISBN 1-55860-191-0.