Generating HTML with Lisp & Templates

Gene Michael Stover

created Tuesday, 27 January 2004
updated Wednesday, 15 July 2004

Contents

1 Introduction

In Generating HTML with Lisp / a tutorial for new programmers[Sto03a], I described a way to produce HTML documents from data structures expressed in Lisp. In this article, I discuss another way of making HTML documents. It uses Lisp, but instead of expressing the content in Lisp data structures which convert to HTML, it reads a file of HTML that also contains Lisp code to execute to fill-in parts of the HTML file. These files are called "templates".

HTML templates are also used in Perl & probably some other languages. A PHP page is inherently a template, too.

2 To Do

This article is still under construction. The biggest problem is that the source code in the figures is out of date. You might check back here after 15 February 2004.

Update source code in the figures.

Edit edit edit.

Insert links to the source code files at strategic loctions.

Considered embedded images for improved mood.

Incorporate the final source code into CyberTiggyr Chill.

Make sure all links to Lisp CGI programs are relative to `CyberTiggyr.COM`, not to the base URL. This is necessary because the server for `lisp-p.org` does not have clisp.

Consider installing a better Lisp on `lisp-p.org`. Might want to update Lisp/CGI Programming Experiments. Might convert that article into a tutorial that shows how to use many Lisps for CGI, though it would not discuss techniques such as generating HTML from Lisp expressions ([Sto03a]) or HTML templates ([Sto04a]).

3 Basic Idea

We'll use a macro to read the template file & produce Lisp code for it. When you execute that Lisp code, it sends the HTML directly to the browser, & it evaluates the embedded Lisp code.

Instead of a macro, we could use a function that produced Lisp code, but to execute the Lisp code, we'd need to use the EVAL function. The EVAL function executes Lisp code at the top level, so local variables, such as those within LET , are not in effect. That's a bit of a bummer, so I chose a macro.

What to call the macro? I think CALL-TEMPLATE is most appropriate. Actually, the name CALL-TEMPLATE is not all that great, but it leaves LOAD-TEMPLATE for a function which produces Lisp code in memory from the template file.

4 Function LOAD-TEMPLATE

The top-level function is LOAD-TEMPLATE. It reads an HTML template from a file & then evaluates it. An optional argument lets you identify the stream to which the template's output is sent.

```
(defun load-template (pn &optional (strm t))
  "Call an HTML template that is in a file.  PN is the
pathname of the file."
  (call-template (slurp-file pn) strm))
```

Figure 1: The LOAD-TEMPLATE function

```
(defun slurp-file-simple (pn)
  "Return a string containing the entire
contents of the file identified by the
pathname PN."
  (with-open-file (strm pn)
  (slurp-stream-simple strm)))
```

Figure 2: A simple but probably slow implementation of function SLURP-FILE

Assuming a SLURP-FILE5 function returns the entire contents of a file as a string, & a CALL-TEMPLATE4 function evaluates an HTML template (as a string), then LOAD-TEMPLATE could be implemented as shown in Figure 1.

5 Function SLURP-FILE

The simplest implementation of this function is probably to read characters from the file & *push* them onto a vector which will later be the string containing the entire contents of the file. This implementation is in Figure 2, but it is probably a slow implemtation of the SLURP-FILE function ([Sto03b]). It's kind of a cheat because it delegates most of the work to a SLURP-STREAM-SIMPLE function, which is in Figure 3.

I wrote separate functions for SLURP-FILE-SIMPLE & SLURP-STREAM-SIMPLE because it seems like I might sometimes have a stream to read, not always a file.

I expect that it would be faster to read the file in big chunks of characters, then concatenate the chunks into a single string. Common Lisp does not offer a "read an entire buffer" function like unix's `read` function. The closest we can get is Common Lisp's READ-LINE function. Let's use that to create a list of lines. Once we have all the lines, we'll concatenate them into a string.

When creating a list that will be concatenated later, it's generally faster to treat the list as a stack while we build it, then reverse it before concatenating it.[1]

A twist is that READ-LINE removes the end-of-line character. We want to preserve those characters. I also think it's easier if our replacement for READ-LINE returns one value: the string in most cases & the stream on end-of-file.

[1] I know this from experience. It probably belongs in [Sto04b].

```
(defun slurp-stream-simple (strm)
  "Return a string containing the entire
contents of the stream."
  (let ((str (make-array 1024
                         :element-type 'character
                         :adjustable t
                         :fill-pointer 0)))
    (do ((ch (read-char strm nil strm)
     (read-char strm nil strm)))
((eq ch strm) str)
(vector-push-extend ch str))))
```

Figure 3: A simple but probably slow implementation of function SLURP-STREAM-SIMPLE

```
(defun slurp-file-faster (pn)
  "This turned out to be slower than the 'simple' version."
  (with-open-file (strm pn)
    (slurp-stream-faster strm)))
```

Figure 4: The SLURP-FILE-FASTER function

The new SLURP-FILE-FASTER function is in Figure 4. The SLURP-STREAM-FASTER function is in Figure 5. Some functions & a global object that they use are in Figure 6.

The SLURP-STREAM-FASTER function is more complex but hopefully faster than SLURP-STREAM-SIMPLE. I ran some performance comparisons & learned that SLURP-FILE-FASTER is pretentiously named. Where SLURP-FILE-SIMPLE required 3.42 seconds, SLURP-FILE-FASTER required 6.98 seconds. Some of the functions I used to run the performance comparison are in Figure 7.

So much for performance tests. The final SLURP-FILE function is in Figure 8. It is effectively the same as the SLURP-FILE-SIMPLE function.

I guess another implementation could push characters onto a list, then reverse the list & COERCE the list to a string. The vector-appending SLURP-FILE function I have is good enough, though. So much for SLURP-FILE.

6 Function CONVERT-TEMPLATE

Fix this. The code in the figures here isn't up-to-date with the code in lht.lisp.

The CONVERT-TEMPLATE function has one argument, which is the HTML template as a string. It must return an expression in Lisp which produces the HTML for the template.

For example, if we have the HTML template from Figure 9, we want CONVERT-TEMPLATE to produce something like the Lisp expression in Figure 10.

```
(defun slurp-stream-faster (strm)
  "Slurp the contents of the stream.  Return them in a
string.  This turned out to be slower than the 'simple'
version."
  (do ((lst () (cons line lst))
       (line (xread-line strm) (xread-line strm)))
      ((eq line strm)
       ;; I tried apply #'concatenate 'string blah blah, but got a
       ;; stack overflow in clisp.  So I wrote a function to do the
       ;; concatenation.
       (strcatlst (nreverse lst)))))
```

Figure 5: The SLURP-STREAM-FASTER function

```
(defconstant *newline-string* (format nil "~%"))

(defun xread-line (strm)
  "Returns a string containing the next line in the file
with the end-of-line character(s).  On end-of-file,
returns STRM."
  (let ((x (read-line strm nil strm)))
    (cond ((eq x strm) x)
          (t (concatenate 'string x *newline-string*)))))

(defun strcatlst (lst)
  "Concatenate the strings in a list of strings.  Return a
new string.  Neither the list nor the strings in it are
modified.  Another implementation I considered is
(format nil '~{~A~}' lst).  It would be shorter."
  (let* ((len (loop for x in lst sum (length x)))
 (str (make-string len)))
    (do ((x lst (rest x))
 (i 0 (+ i (length (first x)))))
((endp x) str)
(setq str (replace str (first x) :start1 i)))))
```

Figure 6: Some objects used by SLURP-STREAM-FASTER

```
(defun make-tester (fn)
  #'(lambda ()
      (funcall fn "big.tmp")))

(defun moo ()
  (time
   (timetable
    (list (cons (make-tester #'slurp-file-simple)
                'slurp-file-simple)
  (cons (make-tester #'slurp-file)
                'slurp-file)))))
```

Figure 7: Some of the functions I used to compare the two slurp-file functions

```
(defun slurp-stream (strm)
  (let ((str (make-array 1024
                         :element-type 'character
                         :adjustable t
                         :fill-pointer 0)))
    (do ((ch (read-char strm nil strm)
     (read-char strm nil strm)))
((eq ch strm) str)
(vector-push-extend ch str))))

(defun slurp-file (pn)
  "Return a string containing the entire contents
of the file  identified by the pathname PN."
  (with-open-file (strm pn)
  (slurp-file strm)))
```

Figure 8: The final SLURP-FILE function

```
<html>
<head>
<title>Titular Title</title>
</head>
<body><p>Here's a numbered list of some
things:</p><ol>
<lisp>
  (dolist (x (list 'red 'potatoe 'pie))
</lisp>
<li> <lisp>(format t "~A" x)</lisp> <li>
<lisp>
  )
</lisp>
</ol><p>Whew!</p></body></html>
```

Figure 9: An example HTML template

The Lisp expressions are surrounded by *lisp* tags, as if they were HTML. They aren't HTML; I just thought that was a clever way to identify the Lisp parts of the template. Notice that you can mix Lisp with HTML. See how I've done that with the DOLIST ; part of it's body is Lisp, & part is HTML. Personally, I think that's ugly, but I wanted to show that it can be done.

To create the Lisp expression in Figure 10, I placed "`(progn (princ "`" at the beginning of the template & "`"))`" after the template. Then I replaced every "`<lisp>`" with "`")`" & every "`</lisp>`" with "`(princ "`". I didn't reformat the code; that's in the hopes of making my changes more aparent. The resuling code is formatted terribly if you want to see what it does, so Figure 11 has the same code formatted more readably. Because it contains so many string constants with embedded newlines, it's still fairly difficult to read.

The description of the changes I made by hand to convert the example HTML template to an example Lisp expression tell what the CONVERT-TEMPLATE function must do. Assuming we have a function STRING-REPLACE-ALL which replaces all occurences of an *old* substring with a *new* substring in a big string, then CONVERT-TEMPLATE may be implemented as in Figure 12.

It's ugly as sin because of those string constants which themselves contain fragments of Lisp expressions, but it's a really simple function. Now we need a STRING-REPLACE-ALL function.

Lisp doesn't have many string functions. I tried to find a string library which did what I need here. I found a couple but had troubles installing them. So let's write STRING-REPLACE-ALL here & now. It could be useful in other libraries.

If I'm not worried about efficiency, the STRING-REPLACE-ALL function could search for the *old* string & replace it with the *new* string. It could continue the process until it cannot find any occurrences of the *old* string. Such an implementation is in Figure 13.

```
(progn
  (princ "<html>
<head>
<title>Titular Title</title>
</head>
<body><p>Here's a numbered list of some
things:</p><ol>
")
  (dolist (x (list 'red 'potatoe 'pie))
  (princ "
<li> ") (format t "~A" x) (princ " <li>
")
  )
  (princ "
</ol><p>Whew!</p></body></html>
"))
```

Figure 10: One possible Lisp expression for the HTML template in Figure 9

```
(progn
  (princ "<html>
<head>
<title>Titular Title</title>
</head>
<body><p>Here's a numbered list of some
things:</p><ol>
")
  (dolist (x (list 'red 'potatoe 'pie))
  (princ "
<li> ")
  (format t "~A" x)
  (princ " <li>
")
  )
  (princ "
</ol><p>Whew!</p></body></html>
"))
```

Figure 11: Indented Lisp code that was produced for the example HTML template

```
(defun convert-template (template)
  "Evaluates to a closure that evaluates the
HTML template.  The template is a string containing
HTML & embedded Lisp.
  This function uses EVAL, which usually indicates
it should be a macro, but even as a macro, I
couldn't figure out how to make it work except when
the template is a string constant.  I hesitatingly
suggest that convert-template must be a function
thta uses EVAL."
  (eval
   (first
    (multiple-value-list
     (read-from-string
      (concatenate
       'string
       "#'(lambda (strm) (princ \""
       (string-replace-all
"<lisp>"
"\" strm) "
(string-replace-all
 "</lisp>"
 " (princ \""
 template))
       "\" strm) strm)"))))))
```

Figure 12: The CONVERT-TEMPLATE function

```
(defun string-replace-all (old new big)
  "Replace all occurences of OLD string with NEW string in BIG string."
  (do ((i (search old big) (search old big)))
      ((null i) big)
      (setq big
            (concatenate 'string
              (subseq big 0 i)
              new
              (subseq big (+ i (length old)))))))
```

Figure 13: The STRING-REPLACE-ALL function

```
(defun string-replace-all-simple (old new big)
  "Replace all occurences of OLD string with NEW
string in BIG string."
  (do ((i (search old big) (search old big)))
      ((null i) big)
      (setq big
            (concatenate 'string
              (subseq big 0 i)
              new
              (subseq big (+ i (length old)))))))
```

Figure 14: The STRING-REPLACE-ALL-SIMPLE function

```
(defun string-replace-all-faster (old new big)
  "Replace all occurences of OLD string with NEW
string in BIG string."
  (do ((newlen (length new))
       (i (search old big)
          (search old big :start2 (+ i newlen))))
      ((null i) big)
      (setq big
            (concatenate 'string
              (subseq big 0 i)
              new
              (subseq big (+ i (length old)))))))
```

Figure 15: The STRING-REPLACE-ALL-FASTER function

(It turned out to be much easier to write than I had expected. Cool!)

If it turned out that I needed a faster version, I guess we could give SEARCH the index after the last occurence we found because it wouldn't need to search earlier occurrences, but I don't think it'll be necessary.

Oh hell. I can't resist an opportunity to compare performances. Let's do it.

First, rename the function to STRING-REPLACE-ALL-SIMPLE. Though that change is trivial, Figure 14 shows the new function.

My idea for an optimization is to alter the second SEARCH callso that it won't examine the part of the *big* string that it has already examined. The index where it should start searching is the location it last found, plus the length of the *new* string. While we're at it, let's take the length of the *new* string once at the beginning of the loop & store it into a temporary variable.

The new, hypothetically optimized function is STRING-REPLACE-ALL-FASTER, & it's in Figure 15.

Those functions & others are in the file lht.lisp. The expressions I used for the performance comparison & their results are in Figure 16.

```
> (setq big (make-string 10000 :initial-element #\a))
; big long string of a's
> (defun make-tester (fn)
    #'(lambda ()
        (funcall fn "aaa" "zaz" big)))
MAKE-TESTER
> (timetable
    (list (cons (make-tester #'string-replace-all-simple)
                'string-replace-all-simple)
          (cons (make-tester #'string-replace-all-faster)
                'string-replace-all-faster)))
 1  STRING-REPLACE-ALL-SIMPLE  1.82E+1
 2  STRING-REPLACE-ALL-FASTER  1.69E+1
NIL
```

Figure 16: Comparison of the two implementations of STRING-REPLACE-ALL

The "simple" function required 18.2 seconds to complete the test, while the "faster" function required 16.9 seconds. So the faster function is about 7 percent faster. So I'll use it for STRING-REPLACE-ALL.

7 Bringing the Library Together

All the source code for this tiny library is in the file lht.lisp. There are test programs in the file test.lisp.

8 Example CGI Programs

Let's make some CGI programs to demonstrate this little HTML template library.

8.1 No-Lisp Example

For a first example, let's use an HTML template that does not embed any Lisp at all. The source code for the program is in `ex1.lisp`. The template file it uses is `ex1.tmpl`. You can see the program run at ex1.cgi.

8.2 A Little Lisp Example

This second example generates some of its output with Lisp code. It doesn't use any arguments from *get* or *post*. The source code for the program is in ex2.lisp. The template file it uses is ex2.tmpl. You can see the program run at ex2.cgi.

A Change Log

2004-Jul-01 Fixed a bug in the STRING-REPLACE-ALL function. Thanks to Chris Chapel & Coy Stewart for independently calling my attention to the problem.

2004-Jul-14 Fixed the links to the source code. The Web server at http://lisp-p.org/ doesn't like `*.lisp` files.

B Other File Formats

This document is available in multi-file HTML format at http://lisp-p.org/lht/.

This document is available in DVI format at http://lisp-p.org/lht/lht.dvi.

This document is available in PostScript format at http://lisp-p.org/lht/lht.ps.

This document is available in Pointless Document Format at http://lisp-p.org/lht/lht.pdf.[2]

References

[Sto03a] Gene Michael Stover. *Generating HTML with Lisp / a tutorial for new programmers.* personal web site, August 2003. http://CyberTiggyr.COM/gene/lh/.

[Sto03b] Gene Michael Stover. Implementation of tcpmux using lisp. *personal web site*, December 2003. http://cybertiggyr.com/gene/tcpmux/.

[Sto04a] Gene Michael Stover. Generating html with lisp & templates. *personal web site*, January 2004. http://cybertiggyr.com/gene/lht/.

[Sto04b] Gene Michael Stover. Lisp idioms. *personal web site*, January 2004. http://cybertiggyr.com/gene/lid/.

[2]If Microsloth Winders allowed PostScript printers to print PostScript files directly, the way nature intended, there would be no need for PDF. Also, for viewing on the screen as an interactive document, PDF is inferior to HTML.