Writing CGI scripts in Tcl

Don Libes
National Institute of Standards and Technology
libes@nist.gov

Abstract

CGI scripts enable dynamic generation of HTML pages. This paper describes how to write CGI scripts using Tcl. Many people use Tcl for this purpose already but in an ad hoc way and without realizing many of the more non-obvious benefits. This paper reviews these benefits and provides a framework and examples. Canonical solutions to HTML quoting problems are presented. This paper also discusses using Tcl for the generation of different formats from the same document. As an example, FAQ generation in both text and HTML are described.

Keywords: CGI; FAQ; HTML generation; Tcl; World Wide Web

Introduction

CGI scripts enable dynamic generation of HTML pages [BLee]. Specifically, CGI scripts generate HTML in response to requests for Web pages. For example, a static Web page containing the date might look like this:

```
<p>The date is Mon Mar  4 12:50:10 EST
1996.
```

This page was constructed by manually running the date command and pasting its output in the page. The page will show that same date each time it is requested, until the file is manually rewritten with a different date.

Using a CGI script, it is possible to dynamically generate the date. Each time the file is requested, it will show the current date. This script (and all others in this paper) are written in Tcl [Ouster].

```
puts "Content-type: text/html\n"
puts "<p>The date is [exec date]."
```

The first puts command identifies how the browser should treat the remainder of the data – in this case, as text to be interpreted as HTML. For all but esoteric uses, this same first line will be required in every CGI script.

CGI scripts have many advantages over statically written HTML. For example, CGI scripts can automatically adapt to changes in the environment, such as the date in the previous example. CGI scripts can run programs, include and process data, and just about anything that can be done in traditional programs.

CGI scripts are particularly worthwhile in handling Web forms. Web forms allow users to enter data into a page and then send the results to a Web server for processing. The Web form itself does not have to be generated by a CGI script. However, data entered by a user may require a customized response. Therefore, a dynamically generated response via a CGI script is appropriate. Since the response may produce another form, it is common to generate forms dynamically as well as their responses.

CGI Scripts Are Just a Subset of Dynamic HTML Generation

CGI scripts are a special case of generated HTML. Generated HTML means that another program produced the HTML. There can be a payoff in programmatic generation even if it is not demanded by the CGI environment. I will describe this idea further later in the paper.

Simply embedding HTML in Tcl scripts does not in itself provide any payoff. For instance, consider the preparation of a page describing various types of widgets, such as button widgets, dial widgets, etc. Ignoring the body paragraphs, the headers could be generated as follows:

```
puts "<h3>Button Widgets</h3>"
puts "<h3>Dial Widgets</h3>"
```

Much of this is redundant and suggests the use of a procedure such as this one:

```
proc h3 {header} {
   puts "<h3>$header</h3>"
}
```

Now the script can be rewritten:

```
h3 "Button Widget"
```

Reprinted from *The Proceedings of the Fourth Annual Tcl/Tk Workshop '96*, Monterey, CA, July 10-13, 1996.

```
h3 "Dial Widget"
```

Notice that you no longer have to worry about adding closing tags such as /h3 or putting them in the right place. Also, changing the heading level is isolated to one place in each line.

Using a procedure name specifically tied to an HTML tag has drawbacks. For example, consider code that has level 3 headings for both Widgets and Packages. Now suppose you decide to change just the Widgets to level 2. You would have to look at each h3 instance and manually decide whether it is a Widget or a Package.

In order to change groups of headers that are related, it is helpful to use a logical name rather than one specifically tied to an HTML tag. This can be done by defining an application-specific procedure such as one for widget headers:

```
proc widget_header {heading} {
   h2 "$header Widget"
}
proc package_header {heading} {
   h3 "$heading Package"
}
```

The script can then be written:

```
widget_header "Button"
widget_header "Dial"
package_header "Object"
```

Now all the widget header formats are defined in one place – the widget_header procedure. This includes not only the header level, but any additional formatting. Here, the word “Widget” is automatically appended, but you can imagine other formatting such as adding hyperlinks, rules, and images.

This style of scripting makes up for a deficiency of HTML: HTML lacks the ability to define application-specific tags.

Form Generation

The idea of logical tags is equally useful for generation of Web forms. For example, consider generation of an entry box. Naively rendered in Tcl, a 10-character entry box might look this way:

```
puts "<input name=Username size=10>"
```

This is fine if there is only one place in your code which requires a username. If you have several, it is more convenient to place this in a procedure. Dumping this all into a procedure simplifies things a little, but enough additional attributes on the input tag can quickly render the new procedure impenetrable. Applying the same technique shown earlier suggests two procedures: text and username. text is the application-independent HTML interface. username is the application-specific interface. An example definition for username is shown below. Remember that this is specific to a particular application. In this case, a literal prompt is shown (the HTML markup for this would be defined in yet another procedure). Then the 10-character entry box containing some default value.

```
proc username {name defvalue} {
   prompt "Username"
   text $name $defvalue 10
}
```

When the form is filled out, the user’s new value will be provided as the value for the variable named by the first parameter, stored here in “name”. Later in this paper, I’ll go into this in more detail.

A good definition for text is relatively ugly because it must do the hard work of adding quotes around each value at the same time as doing the value substitutions. This is a good demonstration of something you want to write as few times as possible – once, ideally. In contrast, you could have hundreds of application-specific text boxes. Those procedures are trivial to write and make all forms consistent. In the example above, each call to username would always look identical.

```
proc text {name defvalue {size 50}} {
   puts "<input name=\"$name\"\
    value=\"$defvalue\" size=$size>"
}
```

Once all these procedures exist, the actual code to add a username entry to a form is trivial:

```
username new_user $user
```

Many refinements can be made. For example, it is common to use Tcl variables to mirror the form variables. The rewrite in Figure 1 tests whether the named form variable is also a Tcl variable. If so, the value is used as the default for the entry.

If username called this procedure, the second argument could be omitted if the variable name was identical to the first argument. For example:

```
username User
```

An explicit value can be supplied in this way:

```
username User=don
```

And arbitrary tags can be added as follows:

```
username User=don size=10 \
                 maxlength=5
```

Many other procedures are required for a full implementation. Here are two more which will be used in the remainder of the paper. The procedure “p” starts a new paragraph and prints out its argument. The procedure “put” prints its argument with no terminating newline. And puts, of course, can be called directly.

```
proc p {s} {
   puts "<p>$s"
}
proc put {s} {
   puts -nonewline "$s"
}
```

Inline Directives

Some HTML tags affect characters rather than complete elements. For example, a word can be made bold by surrounding it with and . As before, redundancy can be eliminated by using a procedure:

```
proc bold {s} {
   puts "<b>$s</b>"
}
```

Unlike the earlier examples, it is not desirable to have character-based procedures call puts directly. Otherwise, scripts end up looking like this:

```
put "I often use "
bold "Tcl"
put "to program."
```

These character-based procedures can be made more readable by having them return their results like this:

```
proc bold {s} {
   return "<b>$s</b>"
}
```

Using these inline directives, scripts become much more readable:

```
p "I often use [bold Tcl] to
program."
```

Explicit use of a procedure such as bold shares the same drawbacks as explicit use of procedures such as h2 and h3. If you later decide to change a subset of some uses, you must examine all of them. By using logical procedure names, that trap is avoided. For example, suppose that you want hostnames to always appear the same way. But there is no hostname directive in HTML. So you could arbitrarily choose bold and write:

```
proc hostname {s} {
   return [bold $s]
}
```

An example using this is:

```
p "You may ftp the files from [host
$ftphost] or [host $ftpbackuphost]."
```

If you later decide to change the appearance of hostnames to, say, italics, it is now very easy to do so. Simply change the one-line definition of the hostname procedure.

URLs

URLs have a great deal of redundancy in them, so using procedures can provide dramatic benefits in readability and maintainability. Similarly to the previous section, hyperlinks can be treated as inline directives. By prestoring all URLs, generation of a URL then just requires a reference to the appropriate one. While separate variables can be used for each URL, a single array (_cgi_link) provides all URL tags with their own namespace. This namespace is managed with a procedure called link. For example, suppose that you want to produce the following display in the browser:

> I am married to Don Libes who works in the Manufacturing Collaboration Technologies Group at NIST.

Using the link procedure, with appropriate link definitions, the scripting to produce this is simple:

```
p "I am married to [link Libes] who
works in the [link MCTG] at [link
NIST]."
```

This expands to a sizeable chunk of HTML:

```
proc text {nameval args} {
   regexp "(\[^=]*)(=?)(.*)" $nameval dummy name q value

   put "<input name=\"$name\""

   if {$q != "="} {
      set value [uplevel set $name]
   }
   puts " value=\"[quote_html $value]\" $args>"
}
```

Figure 1: Procedure to create a generic text entry

```
I am married to <A HREF="http://
elib.cme.nist.gov/msid/staff/libes/
libes.don.html">Don Libes</A> who
works in the <A HREF="http://
elib.cme.nist.gov/msid/groups/
mctg.htm">Manufacturing
Collaboration Technologies Group</A>
at <A HREF="http://
www.nist.gov">NIST</A>.
```

Needless to say, working on such raw text is the bane of HTML page maintainers. Yet HTML has no provisions itself for reducing this complexity.[1]

The link procedure is shown in Figure 2. It returns the formatted link given the tag name as its first argument. The second argument, if given, declare a name to be displayed by the browser. The third argument is the URL.

Links can be defined by handcoding the complete absolute URL. However, it is much simpler to create a few helper variables to further minimize redundancy. Figure 3 shows to refer to several of my colleagues whose home pages all exist in the same staff directory.

If the location of any one staff member's page changes, only one line needs to be changed. More importantly, if the directory for the MSID staff pages changes, only one line needs to be changed. MSID_STAFF is dependent on another variable that defines the hostname. The hostname is stored in a separate variable because 1) it is likely to change and 2) there are other links that depend on it.

Figure 4 shows some examples of hosts.

There are no restrictions on tag names or display names. For example, sometimes it is useful to display "Don". Sometimes, the more formal "Don Libes" is appropriate. This is done by defining two links with different names but pointing to the same URL. This is shown in Figure 5.

1. It is tempting to think that relative URLs can simplify this, but relative URLs only apply to URLs that are, well, relative. In this example, the URLs point to a different host than the one where the referring page lives. Even if this isn't the case, I avoid relative URLs because they prevent other people from copying the raw HTML and pasting it into their own page (again, on another site) without substantial effort in first making the URLs absolute.

```
proc link {args} {
   global _cgi_link

   set tag [lindex $args 0]
   if {[llength $args] == 3} {
      set _cgi_link($tag) \
            "<A HREF=\"[lindex $args 2]\">[lindex $args 1]</A>"
   }
   return $_cgi_link($tag)
```

Figure 2: Procedure to access a database of URL links.

```
set MSID_STAFF $MSID_HOST/msid/staff

link Steve     "Steve Ray"     $MSID_STAFF/ray.steve.html
link Don       "Don Libes"     $MSID_STAFF/libes.don.html
link Josh      "Josh Lubell"   $MSID_STAFF/josh.lubell.html
```

Figure 3: Create links to several colleagues who home pages all exist in the same staff directory.

```
set MSID_HOST  http://elib.cme.nist.gov
set NIST_HOST  http://www.nist.gov
set ORA_HOST   http://www.ora.com
```

Figure 4: Some examples of hosts.

```
link Don       "Don"           $MSID_STAFF/libes.don.html
link Libes     "Don Libes"     $MSID_STAFF/libes.don.html
```

Figure 5: Create links to the same URL but display them to the user differently.

Similarly, there are no restrictions on the tag names themselves. Consider the link definitions in Figure 6. These are used in paragraphs such as this one:

```
p "You can ftp Expect from
ftp.cme.nist.gov as [link Expect.Z]
or [link Expect.gz]"
```

A browser shows this as:

> You can ftp Expect from ftp.cme.nist.gov as pub/expect/expect.tar.Z or ...gz.

Having link dependencies localized to one place greatly aids maintenance and testing. For example, if you have a set of pages that use the definitions (i.e., by sourcing them), editing that one file automatically updates all of the other pages the next time they are regenerated. This is useful for testing groups of pages on a different server, such as a test server before moving them over to a production location. Even smaller moves can benefit. For example, it is common to move directories around or create new directories and just move some of the files around.

Quoting

HTML values must be quoted at different times and in different ways. Unfortunately, the standards are hard to read so most people guess instead. However, intuitively figuring out the quoting rules is tricky because simple cases don't require quoting and many browsers handle various error cases differently. It can be very difficult to deduce what is correct when your own browser accepts erroneous code. This section presents procedures for handling quoting.

CGI Arguments

CGI scripts can receive input from either forms or URLs. For example, in a URL specification such as http://www.nist.gov/expect?help=input+foo, anything to the right of the question mark becomes input to the CGI script (which conversely is to the left of the question mark).

Various peculiar translations must be performed on the raw input to restore it to the original values supplied by the user. For example, the user-supplied string "foo bar" is changed to "foo+bar". This is undone by the first regsub in unquote_input (shown in Figure 7). The remaining conversions are rather interesting but understanding them is outside the point of this paper.

The converse procedure to unquote_input is shown below. This transformation is usually done automatically by Web browsers. However, it can be useful if your CGI script needs to send a URL through some other means such as an advertisement on TV.

```
proc quote_url {in} {
    regsub -all " " $in "+" in
    regsub -all "%" $in "%25" in
    return $in
}
```

In theory, this procedure should perform additional character translations. However, you should avoid generating such characters since receiving URLs outside of a browser requires hand-treatment by users. In these situations, all bizarre character sequences should be avoided. For the purposes of testing (feeding input back), additional translation is also unnecessary since any other unquoted characters will be passed untouched.

Suppressing HTML Interpretation

In most contexts, strings which contain strings that *look* like HTML will be interpreted as HTML. For example, if you want to display the literal string "<A HREF=....>", it must be encoded so that the "<" is not turned into a hyperlink specification. Other special characters must be similarly protected. This can be done using quote_html, shown below:

```
proc quote_html {s} {
    # ampersand must be done first!
    regsub -all {&} $s {\&amp}  s
    regsub -all {"} $s {\&quot} s
    regsub -all {<} $s {\&lt}   s
    regsub -all {>} $s {\&gt}   s
    return $s
}
```

This can be used to simplify other procedures. Adding explicit double quotes before returning the final value allows simplification of many other procedures. Assuming this new procedure is called dquote_html, consider the earlier text entry procedure which had the code fragment

```
value=\"$defvalue\"
```

This could be rewritten:

```
value=[dquote_html $defvalue]
```

```
link Expect.Z   "pub/expect/expect.tar.Z"  $EXPECT_DIR/expect.tar.Z
link Expect.gz  "...gz"                    $EXPECT_DIR/expect.tar.gz
```

Figure 6: Link tags and definitions can be very unusual. There are no restrictions.

Argument Cracking

As described earlier, input strings to a CGI script are encoded by the browser. Besides the transformations described already, the browser also packs all variable values together in the form variable1=value1&variable2 =value2&variable*N*=value*N*.

The input procedure (Figure 8) splits the input back into its specific variable/value pairs leaving them in a global array called _cgi_var. Any variable ending with the string "List" causes its value to be treated as a Tcl list. This allows, for example, multiple elements of a listbox to be extractable as individual elements.

If the procedure is run in the CGI environment (i.e., via an HTTPD server), input is automatically read from the environment. If not run from the CGI environment (i.e., via the command line), the argument is used as input. This is very useful for testing. An explicit argument obviates the need for using a real form page to drive the script and means it is easily run from the command line or a debugger.

If the global variable _cgi(debug) is set to 1, the procedure prints the input string before doing anything else. This is useful because it may then be cut and pasted into the procedure argument for debugging purposes, as was just mentioned.

Import/Export

Variables are not automatically entered into separate global variables or the env array because that would open a security hole. Instead, variables must be explicitly requested. Several procedures simplify this. The procedure most commonly used is "import".

import is called for each variable defined from the invoking form. For example, if a form used an entry with "name=foo", the command "import foo" would define foo as a Tcl variable with the value contained in the entry. The command import_cookie is a variation that obtains the value from a cookie variable – a mechanism that allows client-side caching of variables.

```
proc import {name} {
   upvar $name var
   upvar #0 _cgi_uservar($name) val

   set var $val
}
```

Form variables are automatically exported to the called CGI script. It is sometimes necessary to export other variables. This must be done explicitly. Figure 9 shows the export procedure which exports the named variable. Similar to the text procedure, if the first argument is in the form "var=value", the variable as exported with the given value. Otherwise, the variable is treated as a Tcl variable and its value is used.

```
proc unquote_input {buf} {
   # rewrite "+" back to space
   regsub -all {\+} $buf {\ } buf

   # protect $ so Tcl won't do variable expansion
   regsub -all {\$} $buf {\$} buf

   # protect [ so Tcl doesn't do evaluation
   regsub -all {\[} $buf {\[} buf

   # protect quotes so Tcl doesn't terminate string early
   regsub -all \" $buf \\\" buf

   # replace line delimiters with newlines
   regsub -all "%0D%0A" $buf "\n" buf
   # Mosaic sends just %0A.  This is handled in the next command.

   # prepare to process all %-escapes
   regsub -all {%([A-F0-9][A-F0-9])} $buf {[format %c 0x\1]} buf
   # Mosaic sends just %0A.  This is handled in the next command.

   # prepare to process all %-escapes
   regsub -all {%([A-F0-9][A-F0-9])} $buf {[format %c 0x\1]} buf

   # process %-escapes and undo all protection
   eval return \"$buf\"
}
```

Figure 7: Translate HTML-style input to original data.

Error Handling

The CGI environment makes no special provisions for errors. Thus, error processing requires explicit handling by the application programmer. If none is made, any error messages produced (e.g., by the Tcl interpreter) are sent on to the client browser. These are rarely meaningful to the user. Even worse, they can be misinterpreted as HTML in which case the result is incomprehensible even to the script creator.

The procedure in Figure 10 provides a framework to evaluate the body of the CGI script, to automatically catch errors, and attempt to do something useful. The two arguments, head and body, are blocks of Tcl commands which create the head and body of an HTML form. An example is shown later.

If the global value _cgi(debug) is 1, the script error is formatted and printed to the screen so that it is readable. If debug is 0, a simple message is printed saying that an error occurred and that the "diagnostics are being emailed to the service system administrator". At the same time, mail is sent to the service administrator. The mail includes everything about the environment that is necessary to reproduce the problem including the error,

```
proc input {{fakeinput {}}} {
   global env _cgi _cgi_uservar

   if {![info exists env(REQUEST_METHOD)]} {
      set input $fakeinput;# running by hand, so fake it
   } elseif { $env(REQUEST_METHOD) == "GET" } {
      set input $env(QUERY_STRING)
   } else {
      set input [read stdin $env(CONTENT_LENGTH)]
   }
   # if script blows up later, enable access to the original input.
   set _cgi(input) $input

   # good for debugging!
   if {$_cgi(debug)} {
      puts "<pre>$input</pre>"
   }

   set pairs [split $input &]
   foreach pair $pairs {
      regexp (.*)=(.*) $pair dummy varname val

      set val [unquote_input $val]

      # handle lists of values correctly
      if [regexp List$ $varname] {
         lappend _cgi_uservar($varname) $val
      } else {
         set _cgi_uservar($varname) $val
      }
   }

   # repeat loop above but for cookies
}
```

Figure 8: Retrieve CGI input

```
proc export {nameval} {
   regexp "(\[^=]*)(=?)(.*)" $nameval dummy name q value

   if {$q != "="} {
      set value [uplevel set $name]
   }

   put "<input type=hidden name=$name \
                  value=[dquote_html $value]>"
}
```

Figure 9: Export a variable to the CGI script.

the script name, and the input. The implementation shown here is skeletal. In the actual definition, a variety of other interesting problems are handled. For instance, cookie definitions must appear in the output before any HTML. However, cookies are more easily generated as one of the final results in a script. This and other problems are solved by the full implementation, however the details are beyond the scope of this paper.

Using the procedures defined, CGI scripts become very simple. They all start out by sourcing the CGI support routines. Then cgi_eval is called with arguments to create the head and body. The head generates titles, link colors, etc., while the body is responsible for importing, exporting, and generation of text and graphical elements as has already been described. A skeletal example is shown in Figure 11

The title procedure (not shown) produces all of the usual HTML boilerplate including titles, backgrounds, etc. A form procedure simplifies the calling conventions for establishing any forms. This is not difficult. However, of critical importance is noting that a form is in progress. Because some browsers won't show anything if a form hasn't been ended (i.e., "/form"), the error handler must prematurely close the form if an unexpected

```
proc cgi_eval {head body} {
   global env _cgi

   set _cgi(body) "$head;cgi_body_start;app_body_start;$body;app_body_end"
   uplevel #0 {
      cgi_body_start
      if 1==[catch $_cgi(body)] {      # errors occurred, handle them
         set _cgi(errorInfo) $errorInfo

         # close possible open form because some
         # browsers won't show errors otherwise
         if [info exists _cgi(form_in_progress)] {
            puts "</form>"
         }

         h3 "An internal error was detected in the service software. \
               The diagnostics are being emailed to the service\
               system administrator."

         if {$_cgi(debug)} {
            puts "Heck, since you're debugging, I'll show you the\
                  errors right here:"
            # suppress formatting
            puts "<xmp>$_cgi(errorInfo)</xmp>"
         } else {
            mail_start $_cgi(email_admin)
            mail_add "Subject: $_cgi(name) problem"
            mail_add
            if {$env(REQUEST_METHOD) != "by hand"} {
               mail_add "CGI environment:"
               mail_add "REQUEST_METHOD: $env(REQUEST_METHOD)"
               mail_add "SCRIPT_NAME: $env(SCRIPT_NAME)"
               catch {mail_add "HTTP_USER_AGENT: $env(HTTP_USER_AGENT)"}
               catch {mail_add "REMOTE_ADDR: $env(REMOTE_ADDR)"}
               catch {mail_add "REMOTE_HOST: $env(REMOTE_HOST)"}
            }
            mail_add "input:"
            mail_add "$_cgi(input)"
            mail_add "errorInfo:"
            mail_add "$_cgi(errorInfo)"
            mail_end
         }
      }
      cgi_body_end
   }
}
```

Figure 10: Framework to catch errors and report them intelligently.

error occurs. Saving this information is done with a simple global variable. The form procedure is shown in Figure 12.

Many other utilities are necessary such as procedures for each type of form element. Space prevents inclusion of them. Several other miscellaneous utilities complete the basic implementation of the procedures that appear in this paper. A few are mentioned here to give a flavor for what is necessary:

cgi
: Converts a form name to a complete URL.

mail_start
: Generates headers and writes them to a new file representing a mail message to be sent.

mail_add
: Writes a new line to the temporary mail file.

mail_end
: Appends a signature to the temporary mail file, sends it, and deletes the file.

cgi_body_start
: Generates the <body> tag and handles user requests such as backgrounds and various color options. cgi_body_end is analogous.

All of the procedures described so far can be invoked with "cgi_" prepended (if they do not already begin that way). In practice, CGI scripts are generally quite short so this isn't often useful – and writing things like "cgi_h2" is particularly irritating. However conflicts with other namespaces can occasionally make such prefixes a necessary evil.

Several procedures are expected to be redefined by the user. Here are two examples that appear in the body procedure earlier.

app_body_start
: Application-supplied procedure, typically for writing initial images or headers common to all pages.

app_body_end
: Application-supplied procedure, typically for writing signature lines, last-update-by, etc.

```
source cgi.tcl

cgi_eval {
   title "Password Change Acknowledgment"
   input "name=libes&old=swordfish&new1=tgif23&new2=tgif23
} {
   import name
   import old

   ... other stuff
   form password {

      spawn /bin/passwd
      expect "Password:"
      ...
   }
}
```

Figure 11: Skeletal example of the CGI procedures in use.

```
proc form {name cmd} {
   global _cgi

   set _cgi(form_in_progress) 1
   puts "<form method=POST action=[cgi $x]>"
   uplevel $cmd
   puts "</form>"
   unset _cgi(form_in_progress)
}
```

Figure 12: The form procedure creates an HTML-style form.

FAQ generation

Earlier I mentioned that CGI scripts are just a subset of HTML generation. As an example, consider the task of building an FAQ in HTML. There is no benefit to dynamically generating an FAQ – it rarely changes. However, an FAQ has some of the same problems as I described earlier. For example, it can include many links which must be kept current.

Another reason that it makes sense to think about generating HTML for an FAQ is that an FAQ is highly stylized. For example, an FAQ always has a set of questions. These questions are then repeated but with answers. Written manually, you would have to literally repeat the questions and create the links. If a new question was added or an old one deleted, you would have to carefully make sure that both entries were handled identically.

Intuitively, this could be automated using two loops. First, the questions and answers would be defined. Then the first loop would print the questions. The second loop would print the questions (again) interspersed with the answers. In pseudocode:

```
define QAs              ;# pseudocode!

foreach qa $QAs {
   print_question $qa
}

foreach qa $QAs {
   print_question $qa
   print_answer $qa
}
```

It suffices to store the questions and answers in an array. The following code numbers each pair and stores question N in qa(N,q) and the corresponding answer in qa(N,a). At the same time, the question is printed out. Thus, there is no need for the first loop in the earlier pseudocode.

```
proc question {q a} {
      global index qa

      incr index

      set qa($index,q) $q
      set qa($index,a) $a

      puts "<A HREF=\"#q$index\">"
      puts "<li>$q"
      puts "</A>"
   }
```

Each question automatically links to its corresponding answer, linked as #qN. When the question/answer pairs are later printed, they will have A HREF tags defining the #qN targets.

The source for an example question/answer definition is shown in Figure 13.

The question is now only stated once and it is always paired with the answer. This simplifies maintenance.

Notice that the answer is not simply a string. The answer is Tcl code. This makes it possible to use all of the techniques mentioned earlier. For example, the example above uses p to generate new paragraphs and link to generate hyperlinks.

The code is evaluated by passing the answer to eval whenever it is needed. An answer procedure does this and generates the hyperlink target at the same time.

```
proc answer {i} {
```

```
question {I keep hearing about Expect.  So what is it?} {

p "Expect is a tool primarily for automating interactive applications
such as telnet, ftp, passwd, fsck, rlogin, tip, etc.  Expect really
makes this stuff trivial.  Expect is also useful for testing these
same applications.  Expect is described in many books, articles,
papers, and FAQs.  There is an entire [link book] on it available
from [link ORA]."

p "You can ftp Expect from ftp.cme.nist.gov as [link Expect.Z] or
[link Expect.gz]"

p "Expect requires Tcl.  If you don't already have Tcl, you can get
it in the same directory (above) as [link Tcl.Z] or [link Tcl.gz]."

p "Expect is free and in the public domain."

};# end question
```

Figure 13: Source to an example question/answer definition.

```
    global qa

    puts "<p>"
    puts "<A NAME=\"q$i\">"
    puts "<li><b>$qa($i,q)</b>"
    puts "</a>"
    puts "<p>"
    eval $qa($i,a)
}
```

For example, “answer 0” would produce the beginning of the output from the earlier question. The full HTML would begin like this:

```
<p>
<A NAME="q0">
<li><b>I keep hearing about Expect.
So what is it?</b>
</a>
<p>Expect is a tool primarily for
automating interactive . . .
```

The answer procedure itself is called from a loop in another procedure called answers (Figure 14). An answer_header procedure prints out a header if one has been associated with the current question. This provides a way of breaking the FAQ into sections. A matching procedure (question_header) defines and prints the headers as they are encountered.

```
proc answer_header {i} {
    global qa

    h3 "$qa($i,h)"
}

proc question_header {h} {
    global index qa

    set qa($index,h) $h
    puts "<A HREF=\"#h$index\">"
    h3 $h
    puts "</A>"
}
```

Translation to Other Formats

Another benefit of using logical tags is that different output formats can be generated by changing the application-specific procedures. For instance, suppose a horizontal rule is produced using the hr command. Obviously this can be defined as “puts <hr>”. It is easily changed to produce text using the following procedure:

```
proc hr {} {
    puts ============================
}
```

Here are analogous definitions for h1 and h2. Others are similar.

```
proc h1 {s} {
    puts ""
    puts "*"
    puts "* $s"
    puts "*"
    puts ""
}

proc h2 {s} {
    puts "*** $s ***"
}
```

For example, with this new definition, “h1 Questions” reasonably simulates a level 1 header using only text as:

```
*
* Questions
*
```

The ability to generate the FAQ in different forms is convenient. For example, it means that people can read the FAQ without having an HTML browser.

The generation of different formats is simplified by avoiding use of explicit HTML tags and instead using logical procedure names. A particular output format can be produced merely by providing an appropriate set of procedure definitions. Although I have not done so, it should be possible to adapt the framework and ideas shown here to produce output in such formats as TEX, MIF, and others. Even without translation, avoiding explicit HTML is a good idea for the reasons mentioned earlier – maintenance and readability.

```
proc answers {} {
    uplevel #0 {
        start_answers
        for {set index 0} {$index < $maxindex} {incr index} {
            catch {answer_header $index}
            answer $index
            hr
        }
    }
}
```

Figure 14: Generate all the answers in the FAQ.

A Translation Framework

Translation is further simplified by separating the application-specific definitions from the content of the particular document. For example, multiple FAQs could reuse the same set of FAQ support definitions. Each FAQ would start by loading the FAQ definitions by means of a source command appropriate to the desired output:

```
source FAQdriver.$argv
```

A driver for each output format defines the procedures to produce the FAQ in that particular format. For example, FAQdriver.html would begin:

```
# driver.html - Tcl to HTML procs
proc hr {} {puts "<hr>"}
```

FAQdriver.text would start similarly:

```
# driver.text - Tcl to text procs
proc hr {} {puts ===================}
```

If short enough, all of the different definitions can be maintained as a single file which simply uses a switch to define the appropriate definitions.

```
switch $argv {
   html {
      proc emphasis {s} {
         puts "<em>$s</em>"
      }
      . . .
   }
   text {
      proc emphasis {s} {puts "*$s*"}
      . . .
   }
}
```

In either case, output generation is then accomplished by executing the document with the argument describing the desired format. For example, assuming the FAQ source is stored in ExpectFAQ, HTML is generated from the command line as:

```
% ExpectFAQ html
```

Text output is generated as:

```
% ExpectFAQ text
```

Experiences

The techniques described in this paper have been used successfully in building several projects consisting of large numbers of pages including the NIST Application Protocol Information Base [Lubell] and the NIST Identifier Collaboration Service [Libes95]. In addition, they have been used to construct and maintain several FAQs including the Expect FAQ [Libes96].

Readers interested in comparative strategies to CGI generation should consult the Yahoo database [Yahoo] which lists CGI libraries for dozens of languages, often with multiple entries for each. Readers should also explore alternative strategies to CGI, such as the Tcl-based server-side programming demonstrated by Audience1 [Sah] and NeoScript [Lehen] which elegantly solve problems that CGI alone cannot address adequately.

The other aspect of this paper, dynamic document generation, is also an area rich in development. Various attempts are being made to solve this in other ways including SGML and its extensions and alternatives. Good discussion of these can be found in [Harman].

Concluding Notes

This paper has shown the benefits of generating HTML from Tcl scripts. CGI scripts are an obvious use of this. However, even static documents benefit by increasing readability and improving maintainability.

Traditionally, Perl has been the language of choice for CGI scripting. However, use of Tcl for CGI scripting has increased significantly. Part of this is simply due to the number of people who already know Tcl. But Tcl brings with it many beneficial attributes: Tcl is a simple language to learn. Its portability is excellent, it is robust, and it has no significant startup overhead. And of course it is easily embeddable in other applications making it that much easier to leverage ongoing development in languages such as C and C++.

These are all characteristics that make Tcl very attractive for CGI scripting. However, Tcl does not have a history of use for CGI scripting and there is little documentation to help beginners get started. Hopefully, this paper will make it easier for more people to get starting writing CGI scripts in Tcl.

Availability

The CGI library described is available at http://www.cme.nist.gov/pub/expect/cgi.tcl.tar.Z. The FAQ library described can be retrieved from the Expect FAQ itself [Libes96]. This software is in the public domain. NIST and I would appreciate credit if you use this software.

Acknowledgments

Thanks to Josh Lubell, John Buckman, Mark Williamson, Steve Ray, and the Tcl '96 program committee for valuable suggestions on this paper.

References

[BLee] T. Berners-Lee, D. Connolly, "Hypertext Markup Language – 2.0, RFC 1866, HTML Working Group, IETF, Corporation for National Research Initiatives, URL: http://www.w3.org/pub/WWW/MarkUp/html-spec/html-spec_toc.html, September 22, 1995.

[Harman] Harman, D., "Overview of the Third Text REtrieval Conference (TREC-3), NIST Special Publication 500-225, NIST, Gaithersburg, MD, April 1995.

[Lehen] Lehenbauer, K., "NeoScript", URL: http://www.NeoSoft.com/neoscript/, 1996.

[Libes95] Libes, D., "NIST Identification Collaboration Service", URL: http://www-i.cme.nist.gov/cgi-bin/ns/src/welcome.cgi, National Institute of Standards and Technology, 1995.

[Libes96] Libes, D., "Expect FAQ", URL: http://www.cme.nist.gov/pub/expect/FAQ.html", National Institute of Standards and Technology, 1996.

[Lubell] Lubell, J., "NIST Identification Collaboration Service", URL: http://www-i.cme.nist.gov/proj/apde/www/apib.htm, National Institute of Standards and Technology, 1996.

[Ouster] Ousterhout, J., "Tcl and the Tk Toolkit", Addison-Wesley Publishing Co., 1994.

[Sah] Sah, A., Brown, K., and Brewer, E., "Programming the Internet from the Server-Side with Tcl and Audience1", Tcl/Tk Workshop 96, Monterey, CA, July 10-13, 1996.

[Yahoo] "Yahoo!", URL: http://www.yahoo.com/Computers_and_Internet/Internet/World_Wide_Web/CGI___Common_Gateway_Interface/, April, 1996.

Writing CGI scripts in Tcl

Don Libes
National Institute of Standards and Technology
libes@nist.gov

Abstract

CGI scripts enable dynamic generation of HTML pages. This paper describes how to write CGI scripts using Tcl. Many people use Tcl for this purpose already but in an ad hoc way and without realizing many of the more non-obvious benefits. This paper reviews these benefits and provides a framework and examples. Canonical solutions to HTML quoting problems are presented. This paper also discusses using Tcl for the generation of different formats from the same document. As an example, FAQ generation in both text and HTML are described.

Keywords: CGI; FAQ; HTML generation; Tcl; World Wide Web

Introduction

CGI scripts enable dynamic generation of HTML pages [BLee]. Specifically, CGI scripts generate HTML in response to requests for Web pages. For example, a static Web page containing the date might look like this:

```
<p>The date is Mon Mar  4 12:50:10 EST
1996.
```

This page was constructed by manually running the date command and pasting its output in the page. The page will show that same date each time it is requested, until the file is manually rewritten with a different date.

Using a CGI script, it is possible to dynamically generate the date. Each time the file is requested, it will show the current date. This script (and all others in this paper) are written in Tcl [Ouster].

```
puts "Content-type: text/html\n"
puts "<p>The date is [exec date]."
```

The first puts command identifies how the browser should treat the remainder of the data – in this case, as text to be interpreted as HTML. For all but esoteric uses, this same first line will be required in every CGI script.

CGI scripts have many advantages over statically written HTML. For example, CGI scripts can automatically adapt to changes in the environment, such as the date in the previous example. CGI scripts can run programs, include and process data, and just about anything that can be done in traditional programs.

CGI scripts are particularly worthwhile in handling Web forms. Web forms allow users to enter data into a page and then send the results to a Web server for processing. The Web form itself does not have to be generated by a CGI script. However, data entered by a user may require a customized response. Therefore, a dynamically generated response via a CGI script is appropriate. Since the response may produce another form, it is common to generate forms dynamically as well as their responses.

CGI Scripts Are Just a Subset of Dynamic HTML Generation

CGI scripts are a special case of generated HTML. Generated HTML means that another program produced the HTML. There can be a payoff in programmatic generation even if it is not demanded by the CGI environment. I will describe this idea further later in the paper.

Simply embedding HTML in Tcl scripts does not in itself provide any payoff. For instance, consider the preparation of a page describing various types of widgets, such as button widgets, dial widgets, etc. Ignoring the body paragraphs, the headers could be generated as follows:

```
puts "<h3>Button Widgets</h3>"
puts "<h3>Dial Widgets</h3>"
```

Much of this is redundant and suggests the use of a procedure such as this one:

```
proc h3 {header} {
   puts "<h3>$header</h3>"
}
```

Now the script can be rewritten:

```
h3 "Button Widget"
```

Reprinted from *The Proceedings of the Fourth Annual Tcl/Tk Workshop '96*, Monterey, CA, July 10-13, 1996.

```
h3 "Dial Widget"
```

Notice that you no longer have to worry about adding closing tags such as /h3 or putting them in the right place. Also, changing the heading level is isolated to one place in each line.

Using a procedure name specifically tied to an HTML tag has drawbacks. For example, consider code that has level 3 headings for both Widgets and Packages. Now suppose you decide to change just the Widgets to level 2. You would have to look at each h3 instance and manually decide whether it is a Widget or a Package.

In order to change groups of headers that are related, it is helpful to use a logical name rather than one specifically tied to an HTML tag. This can be done by defining an application-specific procedure such as one for widget headers:

```
proc widget_header {heading} {
   h2 "$header Widget"
}
proc package_header {heading} {
   h3 "$heading Package"
}
```

The script can then be written:

```
widget_header "Button"
widget_header "Dial"
package_header "Object"
```

Now all the widget header formats are defined in one place – the widget_header procedure. This includes not only the header level, but any additional formatting. Here, the word "Widget" is automatically appended, but you can imagine other formatting such as adding hyperlinks, rules, and images.

This style of scripting makes up for a deficiency of HTML: HTML lacks the ability to define application-specific tags.

Form Generation

The idea of logical tags is equally useful for generation of Web forms. For example, consider generation of an entry box. Naively rendered in Tcl, a 10-character entry box might look this way:

```
puts "<input name=Username size=10>"
```

This is fine if there is only one place in your code which requires a username. If you have several, it is more convenient to place this in a procedure. Dumping this all into a procedure simplifies things a little, but enough additional attributes on the input tag can quickly render the new procedure impenetrable. Applying the same technique shown earlier suggests two procedures: text and username. text is the application-independent HTML interface. username is the application-specific interface. An example definition for username is shown below. Remember that this is specific to a particular application. In this case, a literal prompt is shown (the HTML markup for this would be defined in yet another procedure). Then the 10-character entry box containing some default value.

```
proc username {name defvalue} {
   prompt "Username"
   text $name $defvalue 10
}
```

When the form is filled out, the user's new value will be provided as the value for the variable named by the first parameter, stored here in "name". Later in this paper, I'll go into this in more detail.

A good definition for text is relatively ugly because it must do the hard work of adding quotes around each value at the same time as doing the value substitutions. This is a good demonstration of something you want to write as few times as possible – once, ideally. In contrast, you could have hundreds of application-specific text boxes. Those procedures are trivial to write and make all forms consistent. In the example above, each call to username would always look identical.

```
proc text {name defvalue {size 50}} {
   puts "<input name=\"$name\"\
    value=\"$defvalue\" size=$size>"
}
```

Once all these procedures exist, the actual code to add a username entry to a form is trivial:

```
username new_user $user
```

Many refinements can be made. For example, it is common to use Tcl variables to mirror the form variables. The rewrite in Figure 1 tests whether the named form variable is also a Tcl variable. If so, the value is used as the default for the entry.

If username called this procedure, the second argument could be omitted if the variable name was identical to the first argument. For example:

```
username User
```

An explicit value can be supplied in this way:

```
username User=don
```

And arbitrary tags can be added as follows:

```
username User=don size=10 \
                 maxlength=5
```

Many other procedures are required for a full implementation. Here are two more which will be used in the remainder of the paper. The procedure “p” starts a new paragraph and prints out its argument. The procedure “put” prints its argument with no terminating newline. And puts, of course, can be called directly.

```
proc p {s} {
   puts "<p>$s"
}
proc put {s} {
   puts -nonewline "$s"
}
```

Inline Directives

Some HTML tags affect characters rather than complete elements. For example, a word can be made bold by surrounding it with and . As before, redundancy can be eliminated by using a procedure:

```
proc bold {s} {
   puts "<b>$s</b>"
}
```

Unlike the earlier examples, it is not desirable to have character-based procedures call puts directly. Otherwise, scripts end up looking like this:

```
put "I often use "
bold "Tcl"
put "to program."
```

These character-based procedures can be made more readable by having them return their results like this:

```
proc bold {s} {
   return "<b>$s</b>"
}
```

Using these inline directives, scripts become much more readable:

```
p "I often use [bold Tcl] to
program."
```

Explicit use of a procedure such as bold shares the same drawbacks as explicit use of procedures such as h2 and h3. If you later decide to change a subset of some uses, you must examine all of them. By using logical procedure names, that trap is avoided. For example, suppose that you want hostnames to always appear the same way. But there is no hostname directive in HTML. So you could arbitrarily choose bold and write:

```
proc hostname {s} {
   return [bold $s]
}
```

An example using this is:

```
p "You may ftp the files from [host
$ftphost] or [host $ftpbackuphost]."
```

If you later decide to change the appearance of hostnames to, say, italics, it is now very easy to do so. Simply change the one-line definition of the hostname procedure.

URLs

URLs have a great deal of redundancy in them, so using procedures can provide dramatic benefits in readability and maintainability. Similarly to the previous section, hyperlinks can be treated as inline directives. By prestoring all URLs, generation of a URL then just requires a reference to the appropriate one. While separate variables can be used for each URL, a single array (_cgi_link) provides all URL tags with their own namespace. This namespace is managed with a procedure called link. For example, suppose that you want to produce the following display in the browser:

> I am married to Don Libes who works in the Manufacturing Collaboration Technologies Group at NIST.

Using the link procedure, with appropriate link definitions, the scripting to produce this is simple:

```
p "I am married to [link Libes] who
works in the [link MCTG] at [link
NIST]."
```

This expands to a sizeable chunk of HTML:

```
proc text {nameval args} {
   regexp "(\[^=]*)(=?)(.*)" $nameval dummy name q value

   put "<input name=\"$name\""

   if {$q != "="} {
      set value [uplevel set $name]
   }
   puts " value=\"[quote_html $value]\" $args>"
}
```

Figure 1: Procedure to create a generic text entry

```
I am married to <A HREF="http://
elib.cme.nist.gov/msid/staff/libes/
libes.don.html">Don Libes</A> who
works in the <A HREF="http://
elib.cme.nist.gov/msid/groups/
mctg.htm">Manufacturing
Collaboration Technologies Group</A>
at <A HREF="http://
www.nist.gov">NIST</A>.
```

Needless to say, working on such raw text is the bane of HTML page maintainers. Yet HTML has no provisions itself for reducing this complexity.[1]

The link procedure is shown in Figure 2. It returns the formatted link given the tag name as its first argument.

1. It is tempting to think that relative URLs can simplify this, but relative URLs only apply to URLs that are, well, relative. In this example, the URLs point to a different host than the one where the referring page lives. Even if this isn't the case, I avoid relative URLs because they prevent other people from copying the raw HTML and pasting it into their own page (again, on another site) without substantial effort in first making the URLs absolute.

The second argument, if given, declare a name to be displayed by the browser. The third argument is the URL.

Links can be defined by handcoding the complete absolute URL. However, it is much simpler to create a few helper variables to further minimize redundancy. Figure 3 shows to refer to several of my colleagues whose home pages all exist in the same staff directory.

If the location of any one staff member's page changes, only one line needs to be changed. More importantly, if the directory for the MSID staff pages changes, only one line needs to be changed. MSID_STAFF is dependent on another variable that defines the hostname. The hostname is stored in a separate variable because 1) it is likely to change and 2) there are other links that depend on it.

Figure 4 shows some examples of hosts.

There are no restrictions on tag names or display names. For example, sometimes it is useful to display "Don". Sometimes, the more formal "Don Libes" is appropriate. This is done by defining two links with different names but pointing to the same URL. This is shown in Figure 5.

```
proc link {args} {
   global _cgi_link

   set tag [lindex $args 0]
   if {[llength $args] == 3} {
      set _cgi_link($tag) \
            "<A HREF=\"[lindex $args 2]\">[lindex $args 1]</A>"
   }
   return $_cgi_link($tag)
```

Figure 2: Procedure to access a database of URL links.

```
set MSID_STAFF $MSID_HOST/msid/staff

link Steve      "Steve Ray"    $MSID_STAFF/ray.steve.html
link Don        "Don Libes"    $MSID_STAFF/libes.don.html
link Josh       "Josh Lubell"  $MSID_STAFF/josh.lubell.html
```

Figure 3: Create links to several colleagues who home pages all exist in the same staff directory.

```
set MSID_HOST  http://elib.cme.nist.gov
set NIST_HOST  http://www.nist.gov
set ORA_HOST   http://www.ora.com
```

Figure 4: Some examples of hosts.

```
link Don        "Don"          $MSID_STAFF/libes.don.html
link Libes      "Don Libes"    $MSID_STAFF/libes.don.html
```

Figure 5: Create links to the same URL but display them to the user differently.

Similarly, there are no restrictions on the tag names themselves. Consider the link definitions in Figure 6. These are used in paragraphs such as this one:

```
p "You can ftp Expect from
ftp.cme.nist.gov as [link Expect.Z]
or [link Expect.gz]"
```

A browser shows this as:

> You can ftp Expect from ftp.cme.nist.gov as pub/expect/expect.tar.Z or ...gz.

Having link dependencies localized to one place greatly aids maintenance and testing. For example, if you have a set of pages that use the definitions (i.e., by sourcing them), editing that one file automatically updates all of the other pages the next time they are regenerated. This is useful for testing groups of pages on a different server, such as a test server before moving them over to a production location. Even smaller moves can benefit. For example, it is common to move directories around or create new directories and just move some of the files around.

Quoting

HTML values must be quoted at different times and in different ways. Unfortunately, the standards are hard to read so most people guess instead. However, intuitively figuring out the quoting rules is tricky because simple cases don't require quoting and many browsers handle various error cases differently. It can be very difficult to deduce what is correct when your own browser accepts erroneous code. This section presents procedures for handling quoting.

CGI Arguments

CGI scripts can receive input from either forms or URLs. For example, in a URL specification such as http://www.nist.gov/expect?help=input+foo, anything to the right of the question mark becomes input to the CGI script (which conversely is to the left of the question mark).

Various peculiar translations must be performed on the raw input to restore it to the original values supplied by the user. For example, the user-supplied string “foo bar” is changed to “foo+bar”. This is undone by the first regsub in unquote_input (shown in Figure 7). The remaining conversions are rather interesting but understanding them is outside the point of this paper.

The converse procedure to unquote_input is shown below. This transformation is usually done automatically by Web browsers. However, it can be useful if your CGI script needs to send a URL through some other means such as an advertisement on TV.

```
proc quote_url {in} {
    regsub -all " " $in "+" in
    regsub -all "%" $in "%25" in
    return $in
}
```

In theory, this procedure should perform additional character translations. However, you should avoid generating such characters since receiving URLs outside of a browser requires hand-treatment by users. In these situations, all bizarre character sequences should be avoided. For the purposes of testing (feeding input back), additional translation is also unnecessary since any other unquoted characters will be passed untouched.

Suppressing HTML Interpretation

In most contexts, strings which contain strings that *look* like HTML will be interpreted as HTML. For example, if you want to display the literal string “<A HREF=....>”, it must be encoded so that the “<” is not turned into a hyperlink specification. Other special characters must be similarly protected. This can be done using quote_html, shown below:

```
proc quote_html {s} {
    # ampersand must be done first!
    regsub -all {&} $s {\&amp}  s
    regsub -all {"} $s {\&quot} s
    regsub -all {<} $s {\&lt}   s
    regsub -all {>} $s {\&gt}   s
    return $s
}
```

This can be used to simplify other procedures. Adding explicit double quotes before returning the final value allows simplification of many other procedures. Assuming this new procedure is called dquote_html, consider the earlier text entry procedure which had the code fragment

```
value=\"$defvalue\"
```

This could be rewritten:

```
value=[dquote_html $defvalue]
```

```
link Expect.Z   "pub/expect/expect.tar.Z"  $EXPECT_DIR/expect.tar.Z
link Expect.gz  "...gz"                    $EXPECT_DIR/expect.tar.gz
```

Figure 6: Link tags and definitions can be very unusual. There are no restrictions.

Argument Cracking

As described earlier, input strings to a CGI script are encoded by the browser. Besides the transformations described already, the browser also packs all variable values together in the form variable1=value1&variable2 =value2&variable*N*=value*N*.

The input procedure (Figure 8) splits the input back into its specific variable/value pairs leaving them in a global array called _cgi_var. Any variable ending with the string "List" causes its value to be treated as a Tcl list. This allows, for example, multiple elements of a listbox to be extractable as individual elements.

If the procedure is run in the CGI environment (i.e., via an HTTPD server), input is automatically read from the environment. If not run from the CGI environment (i.e., via the command line), the argument is used as input. This is very useful for testing. An explicit argument obviates the need for using a real form page to drive the script and means it is easily run from the command line or a debugger.

If the global variable _cgi(debug) is set to 1, the procedure prints the input string before doing anything else. This is useful because it may then be cut and pasted into the procedure argument for debugging purposes, as was just mentioned.

Import/Export

Variables are not automatically entered into separate global variables or the env array because that would open a security hole. Instead, variables must be explicitly requested. Several procedures simplify this. The procedure most commonly used is "import".

import is called for each variable defined from the invoking form. For example, if a form used an entry with "name=foo", the command "import foo" would define foo as a Tcl variable with the value contained in the entry. The command import_cookie is a variation that obtains the value from a cookie variable – a mechanism that allows client-side caching of variables.

```
proc import {name} {
    upvar $name var
    upvar #0 _cgi_uservar($name) val

    set var $val
}
```

Form variables are automatically exported to the called CGI script. It is sometimes necessary to export other variables. This must be done explicitly. Figure 9 shows the export procedure which exports the named variable. Similar to the text procedure, if the first argument is in the form "var=value", the variable as exported with the given value. Otherwise, the variable is treated as a Tcl variable and its value is used.

```
proc unquote_input {buf} {
    # rewrite "+" back to space
    regsub -all {\+} $buf {\ } buf

    # protect $ so Tcl won't do variable expansion
    regsub -all {\$} $buf {\$} buf

    # protect [ so Tcl doesn't do evaluation
    regsub -all {\[} $buf {\[} buf

    # protect quotes so Tcl doesn't terminate string early
    regsub -all \" $buf \\\" buf

    # replace line delimiters with newlines
    regsub -all "%0D%0A" $buf "\n" buf
    # Mosaic sends just %0A.  This is handled in the next command.

    # prepare to process all %-escapes
    regsub -all {%([A-F0-9][A-F0-9])} $buf {[format %c 0x\1]} buf
    # Mosaic sends just %0A.  This is handled in the next command.

    # prepare to process all %-escapes
    regsub -all {%([A-F0-9][A-F0-9])} $buf {[format %c 0x\1]} buf

    # process %-escapes and undo all protection
    eval return \"$buf\"
}
```

Figure 7: Translate HTML-style input to original data.

Error Handling

The CGI environment makes no special provisions for errors. Thus, error processing requires explicit handling by the application programmer. If none is made, any error messages produced (e.g., by the Tcl interpreter) are sent on to the client browser. These are rarely meaningful to the user. Even worse, they can be misinterpreted as HTML in which case the result is incomprehensible even to the script creator.

The procedure in Figure 10 provides a framework to evaluate the body of the CGI script, to automatically catch errors, and attempt to do something useful. The two arguments, head and body, are blocks of Tcl commands which create the head and body of an HTML form. An example is shown later.

If the global value _cgi(debug) is 1, the script error is formatted and printed to the screen so that it is readable. If debug is 0, a simple message is printed saying that an error occurred and that the "diagnostics are being emailed to the service system administrator". At the same time, mail is sent to the service administrator. The mail includes everything about the environment that is necessary to reproduce the problem including the error,

```
proc input {{fakeinput {}}} {
   global env _cgi _cgi_uservar

   if {![info exists env(REQUEST_METHOD)]} {
      set input $fakeinput;# running by hand, so fake it
   } elseif { $env(REQUEST_METHOD) == "GET" } {
      set input $env(QUERY_STRING)
   } else {
      set input [read stdin $env(CONTENT_LENGTH)]
   }
   # if script blows up later, enable access to the original input.
   set _cgi(input) $input

   # good for debugging!
   if {$_cgi(debug)} {
      puts "<pre>$input</pre>"
   }

   set pairs [split $input &]
   foreach pair $pairs {
      regexp (.*)=(.*) $pair dummy varname val

      set val [unquote_input $val]

      # handle lists of values correctly
      if [regexp List$ $varname] {
         lappend _cgi_uservar($varname) $val
      } else {
         set _cgi_uservar($varname) $val
      }
   }

   # repeat loop above but for cookies
}
```

Figure 8: Retrieve CGI input

```
proc export {nameval} {
   regexp "(\[^=]*)(=?)(.*)" $nameval dummy name q value

   if {$q != "="} {
      set value [uplevel set $name]
   }

   put "<input type=hidden name=$name \
                   value=[dquote_html $value]>"
}
```

Figure 9: Export a variable to the CGI script.

the script name, and the input. The implementation shown here is skeletal. In the actual definition, a variety of other interesting problems are handled. For instance, cookie definitions must appear in the output before any HTML. However, cookies are more easily generated as one of the final results in a script. This and other problems are solved by the full implementation, however the details are beyond the scope of this paper.

Using the procedures defined, CGI scripts become very simple. They all start out by sourcing the CGI support routines. Then cgi_eval is called with arguments to create the head and body. The head generates titles, link colors, etc., while the body is responsible for importing, exporting, and generation of text and graphical elements as has already been described. A skeletal example is shown in Figure 11

The title procedure (not shown) produces all of the usual HTML boilerplate including titles, backgrounds, etc. A form procedure simplifies the calling conventions for establishing any forms. This is not difficult. However, of critical importance is noting that a form is in progress. Because some browsers won't show anything if a form hasn't been ended (i.e., "/form"), the error handler must prematurely close the form if an unexpected

```
proc cgi_eval {head body} {
   global env _cgi

   set _cgi(body) "$head;cgi_body_start;app_body_start;$body;app_body_end"
   uplevel #0 {
      cgi_body_start
      if 1==[catch $_cgi(body)] {      # errors occurred, handle them
         set _cgi(errorInfo) $errorInfo

         # close possible open form because some
         # browsers won't show errors otherwise
         if [info exists _cgi(form_in_progress)] {
            puts "</form>"
         }

         h3 "An internal error was detected in the service software. \
               The diagnostics are being emailed to the service\
               system administrator."

         if {$_cgi(debug)} {
            puts "Heck, since you're debugging, I'll show you the\
                  errors right here:"
            # suppress formatting
            puts "<xmp>$_cgi(errorInfo)</xmp>"
         } else {
            mail_start $_cgi(email_admin)
            mail_add "Subject: $_cgi(name) problem"
            mail_add
            if {$env(REQUEST_METHOD) != "by hand"} {
               mail_add "CGI environment:"
               mail_add "REQUEST_METHOD: $env(REQUEST_METHOD)"
               mail_add "SCRIPT_NAME: $env(SCRIPT_NAME)"
               catch {mail_add "HTTP_USER_AGENT: $env(HTTP_USER_AGENT)"}
               catch {mail_add "REMOTE_ADDR: $env(REMOTE_ADDR)"}
               catch {mail_add "REMOTE_HOST: $env(REMOTE_HOST)"}
            }
            mail_add "input:"
            mail_add "$_cgi(input)"
            mail_add "errorInfo:"
            mail_add "$_cgi(errorInfo)"
            mail_end
         }
      }
      cgi_body_end
   }
}
```

Figure 10: Framework to catch errors and report them intelligently.

error occurs. Saving this information is done with a simple global variable. The form procedure is shown in Figure 12.

Many other utilities are necessary such as procedures for each type of form element. Space prevents inclusion of them. Several other miscellaneous utilities complete the basic implementation of the procedures that appear in this paper. A few are mentioned here to give a flavor for what is necessary:

- cgi — Converts a form name to a complete URL.
- mail_start — Generates headers and writes them to a new file representing a mail message to be sent.
- mail_add — Writes a new line to the temporary mail file.
- mail_end — Appends a signature to the temporary mail file, sends it, and deletes the file.
- cgi_body_start — Generates the <body> tag and handles user requests such as backgrounds and various color options. cgi_body_end is analogous.

All of the procedures described so far can be invoked with "cgi_" prepended (if they do not already begin that way). In practice, CGI scripts are generally quite short so this isn't often useful – and writing things like "cgi_h2" is particularly irritating. However conflicts with other namespaces can occasionally make such prefixes a necessary evil.

Several procedures are expected to be redefined by the user. Here are two examples that appear in the body procedure earlier.

- app_body_start — Application-supplied procedure, typically for writing initial images or headers common to all pages.
- app_body_end — Application-supplied procedure, typically for writing signature lines, last-update-by, etc.

```
source cgi.tcl

cgi_eval {
   title "Password Change Acknowledgment"
   input "name=libes&old=swordfish&new1=tgif23&new2=tgif23
} {
   import name
   import old

   ... other stuff
   form password {

      spawn /bin/passwd
      expect "Password:"
      ...
   }
}
```

Figure 11: Skeletal example of the CGI procedures in use.

```
proc form {name cmd} {
   global _cgi

   set _cgi(form_in_progress) 1
   puts "<form method=POST action=[cgi $x]>"
   uplevel $cmd
   puts "</form>"
   unset _cgi(form_in_progress)
}
```

Figure 12: The form procedure creates an HTML-style form.

FAQ generation

Earlier I mentioned that CGI scripts are just a subset of HTML generation. As an example, consider the task of building an FAQ in HTML. There is no benefit to dynamically generating an FAQ – it rarely changes. However, an FAQ has some of the same problems as I described earlier. For example, it can include many links which must be kept current.

Another reason that it makes sense to think about generating HTML for an FAQ is that an FAQ is highly stylized. For example, an FAQ always has a set of questions. These questions are then repeated but with answers. Written manually, you would have to literally repeat the questions and create the links. If a new question was added or an old one deleted, you would have to carefully make sure that both entries were handled identically.

Intuitively, this could be automated using two loops. First, the questions and answers would be defined. Then the first loop would print the questions. The second loop would print the questions (again) interspersed with the answers. In pseudocode:

```
define QAs              ;# pseudocode!

foreach qa $QAs {
   print_question $qa
}

foreach qa $QAs {
   print_question $qa
   print_answer $qa
}
```

It suffices to store the questions and answers in an array. The following code numbers each pair and stores question N in qa(N,q) and the corresponding answer in qa(N,a). At the same time, the question is printed out. Thus, there is no need for the first loop in the earlier pseudocode.

```
proc question {q a} {
       global index qa

       incr index

       set qa($index,q) $q
       set qa($index,a) $a

       puts "<A HREF=\"#q$index\">"
       puts "<li>$q"
       puts "</A>"
    }
```

Each question automatically links to its corresponding answer, linked as #qN. When the question/answer pairs are later printed, they will have A HREF tags defining the #qN targets.

The source for an example question/answer definition is shown in Figure 13.

The question is now only stated once and it is always paired with the answer. This simplifies maintenance.

Notice that the answer is not simply a string. The answer is Tcl code. This makes it possible to use all of the techniques mentioned earlier. For example, the example above uses p to generate new paragraphs and link to generate hyperlinks.

The code is evaluated by passing the answer to eval whenever it is needed. An answer procedure does this and generates the hyperlink target at the same time.

```
proc answer {i} {
```

```
question {I keep hearing about Expect.  So what is it?} {

p "Expect is a tool primarily for automating interactive applications
such as telnet, ftp, passwd, fsck, rlogin, tip, etc.  Expect really
makes this stuff trivial.  Expect is also useful for testing these
same applications.  Expect is described in many books, articles,
papers, and FAQs.  There is an entire [link book] on it available
from [link ORA]."

p "You can ftp Expect from ftp.cme.nist.gov as [link Expect.Z] or
[link Expect.gz]"

p "Expect requires Tcl.  If you don't already have Tcl, you can get
it in the same directory (above) as [link Tcl.Z] or [link Tcl.gz]."

p "Expect is free and in the public domain."

};# end question
```

Figure 13: Source to an example question/answer definition.

```
    global qa

    puts "<p>"
    puts "<A NAME=\"q$i\">"
    puts "<li><b>$qa($i,q)</b>"
    puts "</a>"
    puts "<p>"
    eval $qa($i,a)
}
```

For example, “answer 0” would produce the beginning of the output from the earlier question. The full HTML would begin like this:

```
<p>
<A NAME="q0">
<li><b>I keep hearing about Expect.
So what is it?</b>
</a>
<p>Expect is a tool primarily for
automating interactive . . .
```

The answer procedure itself is called from a loop in another procedure called answers (Figure 14). An answer_header procedure prints out a header if one has been associated with the current question. This provides a way of breaking the FAQ into sections. A matching procedure (question_header) defines and prints the headers as they are encountered.

```
proc answer_header {i} {
    global qa

    h3 "$qa($i,h)"
}

proc question_header {h} {
    global index qa

    set qa($index,h) $h
    puts "<A HREF=\"#h$index\">"
    h3 $h
    puts "</A>"
}
```

Translation to Other Formats

Another benefit of using logical tags is that different output formats can be generated by changing the application-specific procedures. For instance, suppose a horizontal rule is produced using the hr command. Obviously this can be defined as “puts <hr>”. It is easily changed to produce text using the following procedure:

```
proc hr {} {
    puts ============================
}
```

Here are analogous definitions for h1 and h2. Others are similar.

```
proc h1 {s} {
    puts ""
    puts "*"
    puts "* $s"
    puts "*"
    puts ""
}

proc h2 {s} {
    puts "*** $s ***"
}
```

For example, with this new definition, “h1 Questions” reasonably simulates a level 1 header using only text as:

```
*
* Questions
*
```

The ability to generate the FAQ in different forms is convenient. For example, it means that people can read the FAQ without having an HTML browser.

The generation of different formats is simplified by avoiding use of explicit HTML tags and instead using logical procedure names. A particular output format can be produced merely by providing an appropriate set of procedure definitions. Although I have not done so, it should be possible to adapt the framework and ideas shown here to produce output in such formats as TEX, MIF, and others. Even without translation, avoiding explicit HTML is a good idea for the reasons mentioned earlier – maintenance and readability.

```
proc answers {} {
    uplevel #0 {
        start_answers
        for {set index 0} {$index < $maxindex} {incr index} {
            catch {answer_header $index}
            answer $index
            hr
        }
    }
}
```

Figure 14: Generate all the answers in the FAQ.

A Translation Framework

Translation is further simplified by separating the application-specific definitions from the content of the particular document. For example, multiple FAQs could reuse the same set of FAQ support definitions. Each FAQ would start by loading the FAQ definitions by means of a source command appropriate to the desired output:

```
source FAQdriver.$argv
```

A driver for each output format defines the procedures to produce the FAQ in that particular format. For example, FAQdriver.html would begin:

```
# driver.html - Tcl to HTML procs
proc hr {} {puts "<hr>"}
```

FAQdriver.text would start similarly:

```
# driver.text - Tcl to text procs
proc hr {} {puts ==================}
```

If short enough, all of the different definitions can be maintained as a single file which simply uses a switch to define the appropriate definitions.

```
switch $argv {
   html {
      proc emphasis {s} {
         puts "<em>$s</em>"
      }
      . . .
   }
   text {
      proc emphasis {s} {puts "*$s*"}
      . . .
   }
}
```

In either case, output generation is then accomplished by executing the document with the argument describing the desired format. For example, assuming the FAQ source is stored in ExpectFAQ, HTML is generated from the command line as:

```
% ExpectFAQ html
```

Text output is generated as:

```
% ExpectFAQ text
```

Experiences

The techniques described in this paper have been used successfully in building several projects consisting of large numbers of pages including the NIST Application Protocol Information Base [Lubell] and the NIST Identifier Collaboration Service [Libes95]. In addition, they have been used to construct and maintain several FAQs including the Expect FAQ [Libes96].

Readers interested in comparative strategies to CGI generation should consult the Yahoo database [Yahoo] which lists CGI libraries for dozens of languages, often with multiple entries for each. Readers should also explore alternative strategies to CGI, such as the Tcl-based server-side programming demonstrated by Audience1 [Sah] and NeoScript [Lehen] which elegantly solve problems that CGI alone cannot address adequately.

The other aspect of this paper, dynamic document generation, is also an area rich in development. Various attempts are being made to solve this in other ways including SGML and its extensions and alternatives. Good discussion of these can be found in [Harman].

Concluding Notes

This paper has shown the benefits of generating HTML from Tcl scripts. CGI scripts are an obvious use of this. However, even static documents benefit by increasing readability and improving maintainability.

Traditionally, Perl has been the language of choice for CGI scripting. However, use of Tcl for CGI scripting has increased significantly. Part of this is simply due to the number of people who already know Tcl. But Tcl brings with it many beneficial attributes: Tcl is a simple language to learn. Its portability is excellent, it is robust, and it has no significant startup overhead. And of course it is easily embeddable in other applications making it that much easier to leverage ongoing development in languages such as C and C++.

These are all characteristics that make Tcl very attractive for CGI scripting. However, Tcl does not have a history of use for CGI scripting and there is little documentation to help beginners get started. Hopefully, this paper will make it easier for more people to get starting writing CGI scripts in Tcl.

Availability

The CGI library described is available at http://www.cme.nist.gov/pub/expect/cgi.tcl.tar.Z. The FAQ library described can be retrieved from the Expect FAQ itself [Libes96]. This software is in the public domain. NIST and I would appreciate credit if you use this software.

Acknowledgments

Thanks to Josh Lubell, John Buckman, Mark Williamson, Steve Ray, and the Tcl '96 program committee for valuable suggestions on this paper.

References

[BLee] T. Berners-Lee, D. Connolly, "Hypertext Markup Language – 2.0, RFC 1866, HTML Working Group, IETF, Corporation for National Research Initiatives, URL: http://www.w3.org/pub/WWW/MarkUp/html-spec/html-spec_toc.html, September 22, 1995.

[Harman] Harman, D., "Overview of the Third Text REtrieval Conference (TREC-3), NIST Special Publication 500-225, NIST, Gaithersburg, MD, April 1995.

[Lehen] Lehenbauer, K., "NeoScript", URL: http://www.NeoSoft.com/neoscript/, 1996.

[Libes95] Libes, D., "NIST Identification Collaboration Service", URL: http://www-i.cme.nist.gov/cgi-bin/ns/src/welcome.cgi, National Institute of Standards and Technology, 1995.

[Libes96] Libes, D., "Expect FAQ", URL: http://www.cme.nist.gov/pub/expect/FAQ.html", National Institute of Standards and Technology, 1996.

[Lubell] Lubell, J., "NIST Identification Collaboration Service", URL: http://www-i.cme.nist.gov/proj/apde/www/apib.htm, National Institute of Standards and Technology, 1996.

[Ouster] Ousterhout, J., "Tcl and the Tk Toolkit", Addison-Wesley Publishing Co., 1994.

[Sah] Sah, A., Brown, K., and Brewer, E., "Programming the Internet from the Server-Side with Tcl and Audience1", Tcl/Tk Workshop 96, Monterey, CA, July 10-13, 1996.

[Yahoo] "Yahoo!", URL: http://www.yahoo.com/Computers_and_Internet/Internet/World_Wide_Web/CGI___Common_Gateway_Interface/, April, 1996.