

There's probably a 12-step program to help those of us who still prefer to use a text-based mail reader in this day of GUI's and multimedia MIME attachments. Some of us still get by quite well with these curses-based dinosaurs. If our mail-reader supports hooks for `metamail` and the default set of viewers, we can read anything that the GUI-based readers can send.

Except HTML. HTML was developed well after `metamail`, and while there are simple hooks in the mailcap file to invoke Netscape, I don't want to start up a full browser just to read one e-mail with an HTML attachment.

So, after grousing about this enough times, I put together a simple little HTML text viewer. The program is pretty simple, a text widget, a scrollbar, a quit button, and a call to the HTML rendering package.

```
#!/usr/local/bin/wish

source "htmllib.tcl"

# build and display the button, text widget, and scrollbar

set b [button .b -text "quit" -command "exit"]
set t [text .t -yscrollcommand ".sy set" ]
set s [scrollbar .sy -orient vertical -command "$t yview"]

grid $t -row 0 -column 0
grid $b -row 1 -column 0
grid $s -row 0 -column 1 -sticky ns

# Read from a file if there's one on the command line, else stdin

if {$argc >= 1} {
    set infl [open [lindex $argv [expr $argc - 1]]]
} else {
    set infl stdin
}

# Initialize the HTML display package.
HMininit_win $t

# Read the input, and display it.
HMparse_html [read $infl ] "HMrender $t"
```

Rather obviously, the main functionality in this program is in `htmllib.tcl`. The command `source "htmllib.tcl"` merges the HTML library into the HTML viewer at run time.

The `source` command loads a script into a Tcl program, and evaluates the commands in that script before evaluating the next line of the original script.

**Syntax:** `source fileName`

This is similar to the c language `#include` or the C-Shell `source` command.

The `source` command is the easiest way to split your package into multiple source code modules. (There are other ways, and I'll get to them in future articles.)

Steve Uhler wrote the `htmlLib.tcl` package while he was with the Tcl group at Sun Labs. Since then, Steve Ball and others have used and modified it. This package shows some tricks you can play with the `proc` and `eval` commands, and demonstrates the power of the Tcl text widget.

Like most programming languages, Tcl allows programmers to split pieces of functionality into subroutines to create modular code. The Tcl subroutine is sometimes called a procedure and sometimes referred to by the name of the Tcl command that creates a procedure, `proc`.

The syntax for the `proc` command is:

**Syntax:** `proc name args body`

The `args` and `body` parameters are generally grouped with braces when you create a procedure in your application:

```
proc mySubroutine {arg1 arg2} {
    puts "arg1 is: $arg1"
    puts "arg2 is: $arg2"
    return "DONE"
}
```

In Tcl every line starts with a command. So, Tcl doesn't declare procedures the way we're used to with C or FORTRAN. The word `proc` is a command, not a declaration. Despite the fact that it *looks* a lot like a declaration that `mySubroutine` is a procedure, what's actually happening is that the `proc` command is creating a new procedure, and adding it to the procedure hash table.

The Tcl interpreter will not only evaluate scripts you've created with an editor, it will also evaluate lines that are created at runtime by the program being evaluated.

The command that will evaluate a line of text is `eval`.

**Syntax:** `eval string`

The `string` is any valid Tcl command.

The `eval` command is the trick that makes the HTML parsing code in `htmlLib.tcl` elegantly simple. The `htmlLib.tcl` package uses a set of regular expressions to convert an HTML page from a list of tags and strings into a set of Tcl commands. The `htmlLib` package uses `eval` to evaluate the Tcl procedures that render the HTML page into displayed text.

Tcl regular expression commands are rich enough to fill more than a single *Tclsh Spot* article. In the `htmlLib.tcl` package, they are used to convert this text:

This is **bold** and *italics*.

to these Tcl commands:

```
HMrender .t {P} {} {} {This is }
HMrender .t {B} {} {} {bold}
HMrender .t {B} {/} {} { and }
HMrender .t {I} {} {} {italics}
HMrender .t {I} {/} {} {.
```

Within the `HMrender` procedure, the HTML tag information is massaged into new procedure names such as `HMtag_b` and `HMtag_\b`. These procedures are then evaluated to insert strings into the text widget with the appropriate formatting.

Which, finally brings us to the `text` widget, and some of the things you can do with it.

The previous *Tclsh Spot* article showed how to create a `text` widget, and insert text. Now, let's look at what we can do to control the appearance of that text.

The contents of a `text` widget are addressed by index. An index is a line and character position in the format `line.character`. To match other Unix utilities, the line numbers start at 1, and the character positions start at 0. The index `1.0` represents the first character in a `text` widget, and the index `2.3` is the fourth character on the second line.

The `text` widget has a feature that allows scripts to tag areas of text and then define how to display text within that area.

A tag consists of a reference name for the tag, and the start and end indices of the area associated with that tag.

A tag is added to a `text` widget with the `textName tag add` command.

**Syntax:** `textName tag add tagName startIndex ?end1? ?start2? ... ?endN?`

<code>textName</code>	The <code>text</code> widget that will contain the tag.
<code>tag add</code>	Add a tag at the defined index points.
<code>startIndex ?end?</code>	The tag will be attached to the character at <code>startIndex</code> , and will contain all characters up to, but not including, the character at <code>end</code> . If the <code>end</code> is less than the <code>startIndex</code> , or if the <code>startIndex</code> does not refer to any character in the <code>text</code> widget, then no characters are tagged.

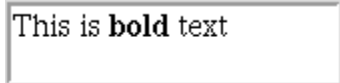
Text within a tagged area is manipulated with the `textName tag configure` command.

**Syntax:** `textName tag configure tagName -parameter value`

<code>textName</code>	The name of the <code>text</code> widget.
<code>tag configure</code>	Configure the text within a tagged area.
<code>tagName</code>	The name of the tag that defines the range of characters that will be configured.
<code>-parameter</code>	The parameter to be defined for this area. The parameters include: <ul style="list-style-type: none"> <li><code>-font</code> Set the font for this text.</li> <li><code>-foreground</code> Set the foreground color for this text.</li> <li><code>-background</code> Set the background color for this text.</li> </ul>
<code>value</code>	The value to use for this parameter.

As a simple example, this short script generates this display:

```
text .t -height 2 -width 20 -font {times 16 normal}
pack .t
.t insert end "This is bold text"
.t tag configure loud -font {times 16 bold}
.t tag add loud 1.8 1.12
```



That explains how the `text` widget can render different styles of text, but how does it handle hypertext references?

The `text` and `canvas` widget support a `bind` command that lets you bind an action to an event. Whenever a button press or `RESIZE` event happens, the defined action will occur. The action is a Tcl script to be evaluated. The action may be multiple lines of commands, or a single call to a Tcl procedure.

The `text` widget supports binding actions to events that happen to tagged areas. The syntax for this is:

**Syntax:** `textName tag bind tagName ?eventType? ?script?`

<i>textName</i>	The name of the <code>text</code> widget.
<i>tag bind</i>	Bind an action to an event occurring on the tagged section of text, or return the script to be evaluated when an event occurs.
<i>tagName</i>	The name of the tag that defines the range of characters that will accept an event.
<i>?eventType?</i>	If the <i>eventType</i> field is set, this defines the event that will trigger this action. The event types are the same as those defined for canvas events in section 9.5.
<i>script</i>	The script to evaluate when this event occurs.

Adding this line to the previous example would turn the word `bold` red when a user clicks on it.

```
.t tag bind loud {.t tag configure loud -foreground red}
```

In the `TclTutor` package, I use this technique to bring up `TkMan` (or the Windows help viewer) when a user clicks on a word.

In the `htmlLib.tcl` package, the `bind` command is used to invoke a procedure to handle hypertext references.

Finally, the HTML viewer uses a `scrollbar` to shift the displayed section of the `text` widget.

The Tcl `scrollbar` widget sends commands to a target widget requesting that widget modify its state to match the `scrollbar`. The `scrollbar` receives commands from the target widget to modify its appearance when the target's state changes.

Using the Tcl `scrollbar` to control a `text` or `canvas` widget is fairly simple:

1. Your script creates a `scrollbar` and a widget to be controlled by the `scrollbar`
2. The two widgets are linked with the `-command` and `-yscrollcommand` (or `-xscrollcommand`) options to exchange information when their state changes.

After those steps are complete, everything else is automatic.

**Syntax:** `scrollbar scrollbarName ?options?`

<code>scrollbar</code>	Create a scrollbar widget.
<code>scrollbarName</code>	The name for this <code>scrollbar</code>
<code>options</code>	This widget supports several options. The <code>-command</code> option is required. <code>-command "procName ?args?"</code> This defines the command to invoke when the state of the <code>scrollbar</code> changes. Arguments that define the changed state will be appended to the arguments defined in this option. <code>-orient direction</code> Defines the orientation for the scrollbar. The <code>direction</code> may <code>horizontal</code> or <code>vertical</code> . Defaults to <code>vertical</code> .

In the HTML viewer, the code `-command "$t yview"` tells the `scrollbar` to invoke the `text` widget's `yview` subcommand with new parameters when a user modifies the `scrollbar`. The `text` widget option: `-yscrollcommand ".sy set"` causes the `text` widget to send information to the `scrollbar` whenever its state changes.

That describes some of the details hiding inside the trivial little 20 line HTML viewing script. All you need to do is copy that file into your `/usr/local/bin` directory, and add the following line to your `mailcap` file, and you can view HTML messages from `elm`, `pine`, or your favorite curses-based mail reader. (Assuming you run your curses-based mail reader from an X-Term window.)

```
text/html; /usr/local/bin/htmlview.tcl
```

A version of the `htmlview.tcl` program with `htmllib.tcl` included is available from <http://www.cflynt.com>.