CHAPTER 11

Regular Expressions

This chapter describes regular expression pattern matching and string processing based on regular expression substitutions. These features provide the most powerful string processing facilities in Tcl. Tcl commands described are: `regexp` and `regsub`.

This chapter is from *Practical Programming in Tcl and Tk*, 3rd Ed.

http://www.beedub.com/book/

Regular expressions are a formal way to describe string patterns. They provide a powerful and compact way to specify patterns in your data. Even better, there is a very efficient implementation of the regular expression mechanism due to Henry Spencer. If your script does much string processing, it is worth the effort to learn about the `regexp` command. Your Tcl scripts will be compact and efficient. This chapter uses many examples to show you the features of regular expressions.

Regular expression substitution is a mechanism that lets you rewrite a string based on regular expression matching. The `regsub` command is another powerful tool, and this chapter includes several examples that do a lot of work in just a few Tcl commands. Stephen Uhler has shown me several ways to transform input data into a Tcl script with `regsub` and then use `subst` or `eval` to process the data. The idea takes a moment to get used to, but it provides a very efficient way to process strings.

Tcl 8.1 added a new regular expression implementation that supports Unicode and *advanced regular expressions* (ARE). This implementation adds more syntax and escapes that makes it easier to write patterns, once you learn the new features! If you know Perl, then you are already familiar with these features. The Tcl advanced regular expressions are almost identical to the Perl 5 regular expressions. The new features include a few very minor incompatibilities with the regular expressions implemented in earlier versions of Tcl 8.0, but these rarely occur in practice. The new regular expression package supports Unicode, of course, so you can write patterns to match Japanese or Hindu documents!

When to Use Regular Expressions

Regular expressions can seem overly complex at first. They introduce their own syntax and their own rules, and you may be tempted to use simpler commands like `string first`, `string range`, or `string match` to process your strings. However, often a single regular expression command can replace a sequence of several `string` commands. Any time you can replace several Tcl commands with one, you get a performance improvement. Furthermore, the regular expression matcher is implemented in optimized C code, so pattern matching is fast.

The regular expression matcher does more than test for a match. It also tells you what part of your input string matches the pattern. This is useful for picking data out of a large input string. In fact, you can capture several pieces of data in just one match by using subexpressions. The `regexp` Tcl command makes this easy by assigning the matching data to Tcl variables. If you find yourself using `string first` and `string range` to pick out data, remember that `regexp` can do it in one step instead.

The regular expression matcher is structured so that patterns are first compiled into an form that is efficient to match. If you use the same pattern frequently, then the expensive compilation phase is done only once, and all your matching uses the efficient form. These details are completely hidden by the Tcl interface. If you use a pattern twice, Tcl will nearly always be able to retrieve the compiled form of the pattern. As you can see, the regular expression matcher is optimized for lots of heavy-duty string processing.

Avoiding a Common Problem

Group your patterns with curly braces.

One of the stumbling blocks with regular expressions is that they use some of the same special characters as Tcl. Any pattern that contains brackets, dollar signs, or spaces must be quoted when used in a Tcl command. In many cases you can group the regular expression with curly braces, so Tcl pays no attention to it. However, when using Tcl 8.0 (or earlier) you may need Tcl to do backslash substitutions on part of the pattern, and then you need to worry about quoting the special characters in the regular expression.

Advanced regular expressions eliminate this problem because backslash substitution is now done by the regular expression engine. Previously, to get `\n` to mean the newline character (or `\t` for tab) you had to let Tcl do the substitution. With Tcl 8.1, `\n` and `\t` inside a regular expression mean newline and tab. In fact, there are now about 20 backslash escapes you can use in patterns. Now more than ever, remember to group your patterns with curly braces to avoid conflicts between Tcl and the regular expression engine.

The patterns in the first sections of this Chapter ignore this problem. The sample expressions in Table 11–7 on page 151 are quoted for use within Tcl scripts. Most are quoted simply by putting the whole pattern in braces, but some are shown without braces for comparison.

Regular Expression Syntax

This section describes the basics of regular expression patterns, which are found in all versions of Tcl. There are occasional references to features added by advanced regular expressions, but they are covered in more detail starting on page 138. There is enough syntax in regular expressions that there are five tables that summarize all the options. These tables appear together starting at page 145.

A regular expression is a sequence of the following items:

- A literal character.
- A matching character, character set, or character class.
- A repetition quantifier.
- An alternation clause.
- A subpattern grouped with parentheses.

Matching Characters

Most characters simply match themselves. The following pattern matches an `a` followed by a `b`:

```
ab
```

The general wild-card character is the period, "`.`". It matches any single character. The following pattern matches an `a` followed by any character:

```
a.
```

Remember that matches can occur anywhere within a string; a pattern does not have to match the whole string. You can change that by using anchors, which are described on page 137.

Character Sets

The matching character can be restricted to a set of characters with the `[`*xyz*`]` syntax. Any of the characters between the two brackets is allowed to match. For example, the following matches either `Hello` or `hello`:

```
[Hh]ello
```

The matching set can be specified as a range over the character set with the `[`*x*`-`*y*`]` syntax. The following matches any digit:

```
[0-9]
```

There is also the ability to specify the complement of a set. That is, the matching character can be anything except what is in the set. This is achieved with the `[^`*xyz*`]` syntax. Ranges and complements can be combined. The following matches anything except the uppercase and lowercase letters:

```
[^a-zA-Z]
```

Using special characters in character sets.

If you want a `]` in your character set, put it immediately after the initial

opening bracket. You do not need to do anything special to include [in your character set. The following matches any square brackets or curly braces:

```
[][{}]
```

Most regular expression syntax characters are no longer special inside character sets. This means you do not need to backslash anything inside a bracketed character set except for backslash itself. The following pattern matches several of the syntax characters used in regular expressions:

```
[][+*?()|\\]
```

Advanced regular expressions add names and backslash escapes as shorthand for common sets of characters like white space, alpha, alphanumeric, and more. These are described on page 139 and listed in Table 11–3 on page 146.

Quantifiers

Repetition is specified with *, for zero or more, +, for one or more, and ?, for zero or one. These *quantifiers* apply to the previous item, which is either a matching character, a character set, or a subpattern grouped with parentheses. The following matches a string that contains b followed by zero or more a's:

```
ba*
```

You can group part of the pattern with parentheses and then apply a quantifier to that part of the pattern. The following matches a string that has one or more sequences of ab:

```
(ab)+
```

The pattern that matches anything, even the empty string, is:

```
.*
```

These quantifiers have a *greedy* matching behavior: They match as many characters as possible. Advanced regular expressions add nongreedy matching, which is described on page 140. For example, a pattern to match a single line might look like this:

```
.*\n
```

However, as a greedy match, this will match all the lines in the input, ending with the last newline in the input string. The following pattern matches up through the first newline.

```
[^\n]*\n
```

We will shorten this pattern even further on page 140 by using nongreedy quantifiers. There are also special newline sensitive modes you can turn on with some options described on page 143.

Alternation

Alternation lets you test more than one pattern at the same time. The matching engine is designed to be able to test multiple patterns in parallel, so alternation is efficient. Alternation is specified with |, the pipe symbol. Another way to match either Hello or hello is:

```
hello|Hello
```

You can also write this pattern as:

```
(h|H)ello
```

or as:

```
[hH]ello
```


Anchoring a Match

By default a pattern does not have to match the whole string. There can be unmatched characters before and after the match. You can anchor the match to the beginning of the string by starting the pattern with ^, or to the end of the string by ending the pattern with $. You can force the pattern to match the whole string by using both. All strings that begin with spaces or tabs are matched with:

```
^[ \t]+
```

If you have many text lines in your input, you may be tempted to think of ^ as meaning "beginning of line" instead of "beginning of string." By default, the ^ and $ anchors are relative to the whole input, and embedded newlines are ignored. Advanced regular expressions support options that make the ^ and $ anchors line-oriented. They also add the \A and \Z anchors that always match the beginning and end of the string, respectively.

Backslash Quoting

Use the backslash character to turn off these special characters :

```
. * ? + [ ] ( ) ^ $ | \
```

For example, to match the plus character, you will need:

```
\+
```

Remember that this quoting is not necessary inside a bracketed expression (i.e., a character set definition.) For example, to match either plus or question mark, either of these patterns will work:

```
(\+|\?)
[+?]
```

To match a single backslash, you need two. You must do this everywhere, even inside a bracketed expression. Or you can use \B, which was added as part of advanced regular expressions. Both of these match a single backslash:

```
\\
\B
```

Unknown backslash sequences are an error.

Versions of Tcl before 8.1 ignored unknown backslash sequences in regular expressions. For example, \= was just =, and \w was just w. Even \n was just n, which was probably frustrating to many beginners trying to get a newline into their pattern. Advanced regular expressions add backslash sequences for tab, newline, character classes, and more. This is a convenient improvement, but in rare cases it may change the semantics of a pattern. Usually these cases are

where an unneeded backslash suddenly takes on meaning, or causes an error because it is unknown.

Matching Precedence

If a pattern can match several parts of a string, the matcher takes the match that occurs earliest in the input string. Then, if there is more than one match from that same point because of alternation in the pattern, the matcher takes the longest possible match. The rule of thumb is: *first, then longest*. This rule gets changed by nongreedy quantifiers that prefer a shorter match.

Watch out for *, which means zero or more, because zero of anything is pretty easy to match. Suppose your pattern is:

```
[a-z]*
```

This pattern will match against `123abc`, but not how you expect. Instead of matching on the letters in the string, the pattern will match on the zero-length substring at the very beginning of the input string! This behavior can be seen by using the `-indices` option of the `regexp` command described on page 148. This option tells you the location of the matching string instead of the value of the matching string.

Capturing Subpatterns

Use parentheses to capture a subpattern. The string that matches the pattern within parentheses is remembered in a matching variable, which is a Tcl variable that gets assigned the string that matches the pattern. Using parentheses to capture subpatterns is very useful. Suppose we want to get everything between the `<td>` and `</td>` tags in some HTML. You can use this pattern:

```
<td>([^<]*)</td>
```

The matching variable gets assigned the part of the input string that matches the pattern inside the parentheses. You can capture many subpatterns in one match, which makes it a very efficient way to pick apart your data. Matching variables are explained in more detail on page 148 in the context of the `regexp` command.

Sometimes you need to introduce parentheses but you do not care about the match that occurs inside them. The pattern is slightly more efficient if the matcher does not need to remember the match. Advanced regular expressions add noncapturing parentheses with this syntax:

```
(?:pattern)
```

Advanced Regular Expressions

The syntax added by advanced regular expressions is mostly just shorthand notation for constructs you can make with the basic syntax already described. There are also some new features that add additional power: nongreedy quantifi-

ers, back references, look-ahead patterns, and named character classes. If you are just starting out with regular expressions, you can ignore most of this section, except for the one about backslash sequences. Once you master the basics, of if you are already familiar with regular expressions in Tcl (or the UNIX *vi* editor or *grep* utility), then you may be interested in the new features of advanced regular expressions.

Compatibility with Patterns in Tcl 8.0

Advanced regular expressions add syntax in an upward compatible way. Old patterns continue to work with the new matcher, but advanced regular expressions will raise errors if given to old versions of Tcl. For example, the question mark is used in many of the new constructs, and it is artfully placed in locations that would not be legal in older versions of regular expressions. The added syntax is summarized in Table 11–2 on page 145.

If you have unbraced patterns from older code, they are very likely to be correct in Tcl 8.1 and later versions. For example, the following pattern picks out everything up to the next newline. The pattern is unbraced, so Tcl substitutes the newline character for each occurrence of `\n`. The square brackets are quoted so that Tcl does not think they delimit a nested command:

```
regexp "(\[^\n\]+)\n" $input
```

The above command behaves identically when using advanced regular expressions, although you can now also write it like this:

```
regexp {([^\n]+)\n} $input
```

The curly braces hide the brackets from the Tcl parser, so they do not need to be escaped with backslash. This saves us two characters and looks a bit cleaner.

Backslash Escape Sequences

The most significant change in advanced regular expression syntax is backslash substitutions. In Tcl 8.0 and earlier, a backslash is only used to turn off special characters such as: `. + * ? [ ]`. Otherwise it was ignored. For example, `\n` was simply `n` to the Tcl 8.0 regular expression engine. This was a source of confusion, and it meant you could not always quote patterns in braces to hide their special characters from Tcl's parser. In advanced regular expressions, `\n` now means the newline character to the regular expression engine, so you should never need to let Tcl do backslash processing.

Again, *always group your pattern with curly braces* to avoid confusion.

Advanced regular expressions add a lot of new backslash sequences. They are listed in Table 11–4 on page 146. Some of the more useful ones include `\s`, which matches space-like characters, `\w`, which matches letters, digit, and the underscore, `\y`, which matches the beginning or end of a word, and `\B`, which matches a backslash.

Character Classes

Character classes are names for sets of characters. The named character class syntax is valid only inside a bracketed character set. The syntax is

```
[:identifier:]
```

For example, `alpha` is the name for the set of uppercase and lowercase letters. The following two patterns are *almost* the same:

```
[A-Za-z]
[[:alpha:]]
```

The difference is that the alpha character class also includes accented characters like è. If you match data that contains nonASCII characters, the named character classes are more general than trying to name the characters explicitly.

There are also backslash sequences that are shorthand for some of the named character classes. The following patterns to match digits are equivalent:

```
[0-9]
[[:digit:]]
\d
```

The following patterns match space-like characters including backspace, form feed, newline, carriage return, tag, and vertical tab:

```
[ \b\f\n\r\t\v]
[:space:]
\s
```

The named character classes and the associated backslash sequence are listed in Table 11–3 on page 146.

You can use character classes in combination with other characters or character classes inside a character set definition. The following patterns match letters, digits, and underscore:

```
[[:digit:][:alpha:]_]
[\d[:alpha:]_]
[[:alnum:]_]
\w
```

Note that `\d`, `\s` and `\w` can be used either inside or outside character sets. When used outside a bracketed expression, they form their own character set. There are also `\D`, `\S`, and `\W`, which are the complement of `\d`, `\s`, and `\w`. These escapes (i.e., `\D` for not-a-digit) cannot be used inside a bracketed character set.

There are two special character classes, `[[:<:]` and `[[:>:]]`, that match the beginning and end of a word, respectively. A word is defined as one or more characters that match `\w`.

Nongreedy Quantifiers

The *, +, and ? characters are *quantifiers* that specify repetition. By default these match as many characters as possible, which is called *greedy* matching. A *nongreedy* match will match as few characters as possible. You can specify non-

greedy matching by putting a question mark after these quantifiers. Consider the pattern to match "one or more of not-a-newline followed by a newline." The not-a-newline must be explicit with the greedy quantifier, as in:

```
[^\n]+\n
```

Otherwise, if the pattern were just

```
.+\n
```

then the "." could well match newlines, so the pattern would greedily consume everything until the very last newline in the input. A nongreedy match would be satisfied with the very first newline instead:

```
.+?\n
```

By using the nongreedy quantifier we've cut the pattern from eight characters to five. Another example that is shorter with a nongreedy quantifier is the HTML example from page 138. The following pattern also matches everything between <td> and </td>:

```
<td>(.*?)</td>
```

Even ? can be made nongreedy, ??, which means it prefers to match zero instead of one. This only makes sense inside the context of a larger pattern. Send me e-mail if you have a compelling example for it!

Bound Quantifiers

The {m,n} syntax is a quantifier that means match at least m and at most n of the previous matching item. There are two variations on this syntax. A simple {m} means match exactly m of the previous matching item. A {m,} means match m or more of the previous matching item. All of these can be made nongreedy by adding a ? after them.

Back References

A back reference is a feature you cannot easily get with basic regular expressions. A back reference matches the value of a subpattern captured with parentheses. If you have several sets of parentheses you can refer back to different captured expressions with \1, \2, and so on. You count by left parentheses to determine the reference.

For example, suppose you want to match a quoted string, where you can use either single or double quotes. You need to use an alternation of two patterns to match strings that are enclosed in double quotes or in single quotes:

```
("[^"]*"|'[^']*')
```

With a back reference, \1, the pattern becomes simpler:

```
('|").*?\1
```

The first set of parenthesis matches the leading quote, and then the \1 refers back to that particular quote character. The nongreedy quantifier ensures that the pattern matches up to the first occurrence of the matching quote.

Look-ahead

Look-ahead patterns are subexpressions that are matched but do not consume any of the input. They act like constraints on the rest of the pattern, and they typically occur at the end of your pattern. A positive look-ahead causes the pattern to match if it also matches. A negative look-ahead causes the pattern to match if it would not match. These constraints make more sense in the context of matching variables and in regular expression substitutions done with the `regsub` command. For example, the following pattern matches a filename that begins with `A` and ends with `.txt`

```
^A.*\.txt$
```

The next version of the pattern adds parentheses to group the file name suffix.

```
^A.*(\.txt)$
```

The parentheses are not strictly necessary, but they are introduced so that we can compare the pattern to one that uses look-ahead. A version of the pattern that uses look-ahead looks like this:

```
^A.*(?=\.txt)$
```

The pattern with the look-ahead constraint matches only the part of the filename before the `.txt`, but only if the `.txt` is present. In other words, the `.txt` is not consumed by the match. This is visible in the value of the matching variables used with the `regexp` command. It would also affect the substitutions done in the `regsub` command.

There is negative look-ahead too. The following pattern matches a filename that begins with `A` and does not end with `.txt`.

```
^A.*(?!\.txt)$
```

Writing this pattern without negative look-ahead is awkward.

Character Codes

The `\`*nn* and `\`*mmm* syntax, where *n* and *m* are digits, can also mean an 8-bit character code corresponding to the octal value *nn* or *mmm*. This has priority over a back reference. However, I just wouldn't use this notation for character codes. Instead, use the Unicode escape sequence, `\u`*nnnn*, which specifies a 16-bit value. The `\x`*nn* sequence also specifies an 8-bit character code. Unfortunately, the `\x` escape consumes all hex digits after it (not just two!) and then truncates the hexadecimal value down to 8 bits. This misfeature of `\x` is not considered a bug and will probably not change even in future versions of Tcl.

The `\U`*yyyyyyyy* syntax is reserved for 32-bit Unicode, but I don't expect to see that implemented anytime soon.

Collating Elements

Collating elements are characters or long names for characters that you can use inside character sets. Currently, Tcl only has some long names for various

ASCII punctuation characters. Potentially, it could support names for every Unicode character, but it doesn't because the mapping tables would be huge. This section will briefly mention the syntax so that you can understand it if you see it. But its usefulness is still limited.

Within a bracketed expression, the following syntax is used to specify a collating element:

```
[.identifier.]
```

The identifier can be a character or a long name. The supported long names can be found in the `generic/regc_locale.c` file in the Tcl source code distribution. A few examples are shown below:

```
[.c.]
[.#.]
[.number-sign.]
```


Equivalence Classes

An equivalence class is all characters that sort to the same position. This is another feature that has limited usefulness in the current version of Tcl. In Tcl, characters sort by their Unicode character value, so there are no equivalence classes that contain more than one character! However, you could imagine a character class for 'o', 'ò', and other accented versions of the letter o. The syntax for equivalence classes within bracketed expressions is:

```
[=char=]
```

where *char* is any one of the characters in the character class. This syntax is valid only inside a character class definition.

Newline Sensitive Matching

By default, the newline character is just an ordinary character to the matching engine. You can make the newline character special with two options: `lineanchor` and `linestop`. You can set these options with flags to the `regexp` and `regsub` Tcl commands, or you can use the embedded options described later in Table 11–5 on page 147.

The `lineanchor` option makes the `^` and `$` anchors work relative to newlines. The `^` matches immediately after a newline, and `$` matches immediately before a newline. These anchors continue to match the very beginning and end of the input, too. With or without the `lineanchor` option, you can use `\A` and `\Z` to match the beginning and end of the string.

The `linestop` option prevents `.` (i.e., period) and character sets that begin with `^` from matching a newline character. In other words, unless you explicitly include `\n` in your pattern, it will not match across newlines.

Embedded Options

You can start a pattern with embedded options to turn on or off case sensitivity, newline sensitivity, and expanded syntax, which is explained in the next section. You can also switch from advanced regular expressions to a literal string, or to older forms of regular expressions. The syntax is a leading:

```
(?chars)
```

where *chars* is any number of option characters. The option characters are listed in Table 11–5 on page 147.

Expanded Syntax

Expanded syntax lets you include comments and extra white space in your patterns. This can greatly improve the readability of complex patterns. Expanded syntax is turned on with a `regexp` command option or an embedded option.

Comments start with a # and run until the end of line. Extra white space and comments can occur anywhere except inside bracketed expressions (i.e., character sets) or within multicharacter syntax elements like `(?=`. When you are in expanded mode, you can turn off the comment character or include an explicit space by preceding them with a backslash. Example 11–1 shows a pattern to match URLs. The leading `(?x)` turns on expanded syntax. The whole pattern is grouped in curly braces to hide it from Tcl. This example is considered again in more detail in Example 11–3 on page 150:

Example 11–1 Expanded regular expressions allow comments.

```
regexp {(?x)        # A pattern to match URLS
       ([^:]+):     # The protocol before the initial colon
       //([^:/]+)   # The server name
       (:([0-9]+))? # The optional port number
       (/.*)        # The trailing pathname
} $input
```

Syntax Summary

Table 11–1 summarizes the syntax of regular expressions available in all versions of Tcl:

Table 11–1 Basic regular expression syntax.

`.`	Matches any character.
`*`	Matches zero or more instances of the previous pattern item.
`+`	Matches one or more instances of the previous pattern item.
`?`	Matches zero or one instances of the previous pattern item.
`( )`	Groups a subpattern. The repetition and alternation operators apply to the preceding subpattern.
`|`	Alternation.
`[ ]`	Delimit a set of characters. Ranges are specified as [*x-y*]. If the first character in the set is ^, then there is a match if the remaining characters in the set are *not* present.
`^`	Anchor the pattern to the beginning of the string. Only when first.
`$`	Anchor the pattern to the end of the string. Only when last.

Advanced regular expressions, which were introduced in Tcl 8.1, add more syntax that is summarized in Table 11–2:

Table 11–2 Additional advanced regular expression syntax.

`{`*m*`}`	Matches *m* instances of the previous pattern item.
`{`*m*`}?`	Matches *m* instances of the previous pattern item. Nongreedy.
`{m,}`	Matches *m* or more instances of the previous pattern item.
`{m,}?`	Matches *m* or more instances of the previous pattern item. Nongreedy.
`{m,n}`	Matches *m* through *n* instances of the previous pattern item.
`{m,n}?`	Matches *m* through *n* instances of the previous pattern item. Nongreedy.
`*?`	Matches zero or more instances of the previous pattern item. Nongreedy.
`+?`	Matches one or more instances of the previous pattern item. Nongreedy.
`??`	Matches zero or one instances of the previous pattern item. Nongreedy.
`(?:`*re*`)`	Groups a subpattern, *re*, but does not capture the result.
`(?=`*re*`)`	Positive look-ahead. Matches the point where *re* begins.
`(?!`*re*`)`	Negative look-ahead. Matches the point where *re* does not begin.
`(?`*abc*`)`	Embedded options, where *abc* is any number of option letters listed in Table 11–5.

Table 11–2 Additional advanced regular expression syntax. (Continued)

`\`*c*	One of many backslash escapes listed in Table 11–4.
`[: :]`	Delimits a character class within a bracketed expression. See Table 11–3.
`[. .]`	Delimits a collating element within a bracketed expression.
`[= =]`	Delimits an equivalence class within a bracketed expression.

Table 11–3 lists the named character classes defined in advanced regular expressions and their associated backslash sequences, if any. Character class names are valid inside bracketed character sets with the `[:`*class*`:]` syntax.

Table 11–3 Character classes.

`alnum`	Upper and lower case letters and digits.
`alpha`	Upper and lower case letters.
`blank`	Space and tab.
`cntrl`	Control characters: `\u0001` through `\u001F`.
`digit`	The digits zero through nine. Also `\d`.
`graph`	Printing characters that are not in `cntrl` or `space`.
`lower`	Lowercase letters.
`print`	The same as `alnum`.
`punct`	Punctuation characters.
`space`	Space, newline, carriage return, tab, vertical tab, form feed. Also `\s`.
`upper`	Uppercase letters.
`xdigit`	Hexadecimal digits: zero through nine, a-f, A-F.

Table 11–4 lists backslash sequences supported in Tcl 8.1.

Table 11–4 Backslash escapes in regular expressions.

`\a`	Alert, or "bell", character.
`\A`	Matches only at the beginning of the string.
`\b`	Backspace character, `\u0008`.
`\B`	Synonym for backslash.
`\c`*X*	Control-*X*.
`\d`	Digits. Same as `[[:digit:]]`
`\D`	Not a digit. Same as `[^[:digit:]]`

Table 11–4 Backslash escapes in regular expressions. (Continued)

`\e`	Escape character, `\u001B`.
`\f`	Form feed, `\u000C`.
`\m`	Matches the beginning of a word.
`\M`	Matches the end of a word.
`\n`	Newline, `\u000A`.
`\r`	Carriage return, `\u000D`.
`\s`	Space. Same as `[[:space:]]`
`\S`	Not a space. Same as `[^[:space:]]`
`\t`	Horizontal tab, `\u0009`.
`\u`*XXXX*	A 16-bit Unicode character code.
`\v`	Vertical tab, `\u000B`.
`\w`	Letters, digit, and underscore. Same as `[[:alnum:]_]`
`\W`	Not a letter, digit, or underscore. Same as `[^[:alnum:]_]`
`\x`*hh*	An 8-bit hexadecimal character code. Consumes all hex digits after `\x`.
`\y`	Matches the beginning or end of a word.
`\Y`	Matches a point that is not the beginning or end of a word.
`\Z`	Matches the end of the string.
`\0`	NULL, `\u0000`
`\`*x*	Where *x* is a digit, this is a back-reference.
`\`*xy*	Where *x* and *y* are digits, either a decimal back-reference, or an 8-bit octal character code.
`\`*xyz*	Where *x*, *y* and *z* are digits, either a decimal back-reference or an 8-bit octal character code.

Table 11–5 lists the embedded option characters used with the `(?`*abc*`)` syntax.

Table 11–5 Embedded option characters used with the (?*x*) syntax.

`b`	The rest of the pattern is a basic regular expression (a la *vi* or *grep*).
`c`	Case sensitive matching. This is the default.
`e`	The rest of the pattern is an extended regular expression (*a la* Tcl 8.0).
`i`	Case insensitive matching.
`m`	Synonym for the `n` option.

Table 11–5 Embedded option characters used with the (?*x*) syntax. (Continued)

n	Newline sensitive matching . Both `lineanchor` and `linestop` mode.
p	Partial newline sensitive matching. Only `linestop` mode.
q	The rest of the pattern is a literal string.
s	No newline sensitivity. This is the default.
t	Tight syntax; no embedded comments. This is the default.
w	Inverse partial newline-sensitive matching. Only `lineanchor` mode.
x	Expanded syntax with embedded white space and comments.

The `regexp` Command

The `regexp` command provides direct access to the regular expression matcher. Not only does it tell you whether a string matches a pattern, it can also extract one or more matching substrings. The return value is 1 if some part of the string matches the pattern; it is 0 otherwise. Its syntax is:

```
regexp ?flags? pattern string ?match sub1 sub2...?
```

The *`flags`* are described in Table 11–6:

Table 11–6 Options to the regexp command.

`-nocase`	Lowercase characters in *`pattern`* can match either lowercase or uppercase letters in *`string`*.
`-indices`	The match variables each contain a pair of numbers that are in indices delimiting the match within *`string`*. Otherwise, the matching string itself is copied into the match variables.
`-expanded`	The pattern uses the expanded syntax discussed on page 144.
`-line`	The same as specifying both `-lineanchor` and `-linestop`.
`-lineanchor`	Change the behavior of ^ and $ so they are line-oriented as discussed on page 143.
`-linestop`	Change matching so that `.` and character classes do not match newlines as discussed on page 143.
`-about`	Useful for debugging. It returns information about the pattern instead of trying to match it against the input.
`--`	Signals the end of the options. You must use this if your pattern begins with `-`.

The *`pattern`* argument is a regular expression as described earlier. If *`string`* matches *`pattern`*, then the results of the match are stored in the variables named in the command. These match variable arguments are optional. If present, *`match`* is set to be the part of the string that matched the pattern. The

remaining variables are set to be the substrings of *string* that matched the corresponding subpatterns in *pattern*. The correspondence is based on the order of left parentheses in the pattern to avoid ambiguities that can arise from nested subpatterns.

Example 11–2 uses regexp to pick the hostname out of the DISPLAY environment variable, which has the form:

```
hostname:display.screen
```

Example 11–2 Using regular expressions to parse a string.

```
set env(DISPLAY) sage:0.1
regexp {([^:]*):} $env(DISPLAY) match host
=> 1
set match
=> sage:
set host
=> sage
```

The pattern involves a complementary set, [^:], to match anything except a colon. It uses repetition, *, to repeat that zero or more times. It groups that part into a subexpression with parentheses. The literal colon ensures that the DISPLAY value matches the format we expect. The part of the string that matches the complete pattern is stored into the match variable. The part that matches the subpattern is stored into host. The whole pattern has been grouped with braces to quote the square brackets. Without braces it would be:

```
regexp (\[^:\]*): $env(DISPLAY) match host
```

With advanced regular expressions the nongreedy quantifier *? can replace the complementary set:

```
regexp (.*?): $env(DISPLAY) match host
```

This is quite a powerful statement, and it is efficient. If we had only had the string command to work with, we would have needed to resort to the following, which takes roughly twice as long to interpret:

```
set i [string first : $env(DISPLAY)]
if {$i >= 0} {
     set host [string range $env(DISPLAY) 0 [expr $i-1]]
}
```

A Pattern to Match URLs

Example 11–3 demonstrates a pattern with several subpatterns that extract the different parts of a URL. There are lots of subpatterns, and you can determine which match variable is associated with which subpattern by counting the left parenthesis. The pattern will be discussed in more detail after the example:

Example 11–3 A pattern to match URLs.

```
set url http://www.beedub.com:80/index.html
regexp {([^:]+)://([^:/]+)(:([0-9]+))?(/.*)} $url \
   match protocol x serverport path
=> 1
set match
=> http://www.beedub.com:80/index.html
set protocol
=> http
set server
=> www.beedub.com
set x
=> :80
set port
=> 80
set path
=> /index.html
```

Let's look at the pattern one piece at a time. The first part looks for the protocol, which is separated by a colon from the rest of the URL. The first part of the pattern is one or more characters that are not a colon, followed by a colon. This matches the `http:` part of the URL:

```
[^:]+:
```

Using nongreedy `+?` quantifier, you could also write that as:

```
.+?:
```

The next part of the pattern looks for the server name, which comes after two slashes. The server name is followed either by a colon and a port number, or by a slash. The pattern uses a complementary set that specifies one or more characters that are *not* a colon or a slash. This matches the `//www.beedub.com` part of the URL:

```
//[^:/]+
```

The port number is optional, so a subpattern is delimited with parentheses and followed by a question mark. An additional set of parentheses are added to capture the port number without the leading colon. This matches the `:80` part of the URL:

```
(:([0-9]+))?
```

The last part of the pattern is everything else, starting with a slash. This matches the `/index.html` part of the URL:

```
/.*
```

Use subpatterns to parse strings.

To make this pattern really useful, we delimit several subpatterns with parentheses:

```
([^:]+)://([^:/]+)(:([0-9]+))?(/.*)
```

These parentheses do not change the way the pattern matches. Only the optional port number really needs the parentheses in this example. However, the `regexp` command gives us access to the strings that match these subpatterns. In

one step regexp can test for a valid URL and divide it into the protocol part, the server, the port, and the trailing path.

The parentheses around the port number include the : before the digits. We've used a dummy variable that gets the : and the port number, and another match variable that just gets the port number. By using noncapturing parentheses in advanced regular expressions, we can eliminate the unused match variable. We can also replace both complementary character sets with a nongreedy .+? match. Example 11–4 shows this variation:

Example 11–4 An advanced regular expression to match URLs.

```
set url http://www.beedub.com:80/book/
regexp {(.+?)://(.+?)(?::([0-9]+))?(/.*)} $url \
   match protocol server port path
=> 1
set match
=> http://www.beedub.com:80/book/
set protocol
=> http
set server
=> www.beedub.com
set port
=> 80
set path
=> /book/
```

Sample Regular Expressions

The table in this section lists regular expressions as you would use them in Tcl commands. Most are quoted with curly braces to turn off the special meaning of square brackets and dollar signs. Other patterns are grouped with double quotes and use backslash quoting because the patterns include backslash sequences like \n and \t. In Tcl 8.0 and earlier, these must be substituted by Tcl before the regexp command is called. In these cases, the equivalent advanced regular expression is also shown.

Table 11–7 Sample regular expressions.

{^[yY]}	Begins with y or Y, as in a Yes answer.
{^(yes|YES|Yes)$}	Exactly "yes", "Yes", or "YES".
"^\[^ \t:\]+:"	Begins with colon-delimited field that has no spaces or tabs.
{^\S+:}	Same as above, using \S for "not space".
"^\[\t]*$"	A string of all spaces or tabs.
{(?n)^\s*$}	A blank line using newline sensitive mode.

Table 11–7 Sample regular expressions. (Continued)

`"(\n|^)\[^\n\]*(\n|$)"`	A blank line, the hard way.
`{^[A-Za-z]+$}`	Only letters.
`{^[[:alpha:]]+$}`	Only letters, the Unicode way.
`{[A-Za-z0-9_]+}`	Letters, digits, and the underscore.
`{\w+}`	Letters, digits, and the underscore using `\w`.
`{[][${}\\]}`	The set of Tcl special characters:] [$ { } \
`"\[^\n\]*\n"`	Everything up to a newline.
`{.*?\n}`	Everything up to a newline using nongreedy `*?`
`{\.}`	A period.
`{[][$^?+*()|\\]}`	The set of regular expression special characters:] [$ ^ ? + * ( ) | \
`<H1>(.*?)</H1>`	An `H1` HTML tag. The subpattern matches the string between the tags.
`<!--.*?-->`	HTML comments.
`{[0-9a-hA-H][0-9a-hA-H]}`	2 hex digits.
`{[[:xdigit:]]{2}}`	2 hex digits, using advanced regular expressions.
`{\d{1,3}}`	1 to 3 digits, using advanced regular expressions.

The `regsub` Command

The `regsub` command does string substitution based on pattern matching. It is very useful for processing your data. It can perform simple tasks like replacing sequences of spaces and tabs with a single space. It can perform complex data transforms, too, as described in the next section. Its syntax is:

```
regsub ?switches? pattern string subspec varname
```

The `regsub` command returns the number of matches and replacements, or 0 if there was no match. `regsub` copies *`string`* to *`varname`*, replacing occurrences of *`pattern`* with the substitution specified by *`subspec`*. If the pattern does not match, then *`string`* is copied to *`varname`* without modification. The optional switches include:

- `-all`, which means to replace all occurrences of the pattern. Otherwise, only the first occurrence is replaced.
- The `-nocase`, `-expanded`, `-line`, `-linestop`, and `-lineanchor` switches are the same as in the `regexp` command. They are described on page 148.
- The `--` switch separates the pattern from the switches, which is necessary if your pattern begins with a `-`.

The replacement pattern, *subspec*, can contain literal characters as well as the following special sequences:

- & is replaced with the string that matched the pattern.
- *x* , where *x* is a number, is replaced with the string that matched the corresponding subpattern in *pattern*. The correspondence is based on the order of left parentheses in the pattern specification.

The following replaces a user's home directory with a ~:

```
regsub ^$env(HOME)/ $pathname ~/ newpath
```

The following constructs a C compile command line given a filename:

```
set file tclIO.c
regsub {([^\.]*)\.c$} $file {cc -c & -o \1.o} ccCmd
```

The matching pattern captures everything before the trailing .c in the file name. The & is replaced with the complete match, tclIO.c, and \1 is replaced with tclIO, which matches the pattern between the parentheses. The value assigned to ccCmd is:

```
cc -c tclIO.c -o tclIO.o
```

We could execute that with:

```
eval exec $ccCmd
```

The following replaces sequences of multiple space characters with a single space:

```
regsub -all {\s+} $string " " string
```

It is perfectly safe to specify the same variable as the input value and the result. Even if there is no match on the pattern, the input string is copied into the output variable.

The regsub command can count things for us. The following command counts the newlines in some text. In this case the substitution is not important:

```
set numLines [regsub -all \n $text {} ignore]
```

Transforming Data to Program with regsub

One of the most powerful combinations of Tcl commands is regsub and subst. This section describes a few examples that use regsub to transform data into Tcl commands, and then use subst to replace those commands with a new version of the data. This technique is very efficient because it relies on two subsystems that are written in highly optimized C code: the regular expression engine and the Tcl parser. These examples are primarily written by Stephen Uhler.

URL Decoding

When a URL is transmitted over the network, it is encoded by replacing special characters with a %*xx* sequence, where *xx* is the hexadecimal code for the character. In addition, spaces are replaced with a plus (+). It would be tedious

and very inefficient to scan a URL one character at a time with Tcl statements to undo this encoding. It would be more efficient to do this with a custom C program, but still very tedious. Instead, a combination of `regsub` and `subst` can efficiently decode the URL in just a few Tcl commands.

Replacing the + with spaces requires quoting the + because it is the one-or-more special character in regular expressions:

```
regsub -all {\+} $url { } url
```

The *%xx* are replaced with a `format` command that will generate the right character:

```
regsub -all {%([0-9a-hA-H][0-9a-hA-H])} $url \
    {[format %c 0x\1]} url
```

The `%c` directive to `format` tells it to generate the character from a character code number. We force a hexadecimal interpretation with a leading `0x`. Advanced regular expressions let us write the "2 hex digits" pattern a bit more cleanly:

```
regsub -all {%([[:xdigit:]]{2})} $url \
    {[format %c 0x\1]} url
```

The resulting string is passed to `subst` to get the `format` commands substituted:

```
set url [subst $url]
```

For example, if the input is `%7ewelch`, the result of the `regsub` is:

```
[format %c 0x7e]welch
```

And then `subst` generates:

```
~welch
```

Example 11–5 encapsulates this trick in the `Url_Decode` procedure.

Example 11–5 The `Url_Decode` procedure.

```
proc Url_Decode {url} {
   regsub -all {\+} $url { } url
   regsub -all {%([:xdigit:]]{2})} $url \
      {[format %c 0x\1]} url
   return [subst $url]
}
```

CGI Argument Parsing

Example 11–6 builds upon `Url_Decode` to decode the inputs to a CGI program that processes data from an HTML form. Each form element is identified by a name, and the value is URL encoded. All the names and encoded values are passed to the CGI program in the following format:

```
name1=value1&name2=value2&name3=value3
```

Example 11–6 shows `Cgi_List` and `Cgi_Query`. `Cgi_Query` receives the form data from the standard input or the `QUERY_STRING` environment variable,

depending on whether the form data is transmitted with a POST or GET request. These HTTP operations are described in detail in Chapter 17. Cgi_List uses split to get back a list of names and values, and then it decodes them with Url_Decode. It returns a Tcl-friendly name, value list that you can either iterate through with a foreach command, or assign to an array with array set:

Example 11–6 The Cgi_Parse and Cgi_Value procedures.

```
proc Cgi_List {} {
    set query [Cgi_Query]
    regsub -all {\+} $query { } query
    set result {}
    foreach {x} [split $query &=] {
        lappend result [Url_Decode $x]
    }
    return $result
}
proc Cgi_Query {} {
    global env
    if {![info exists env(QUERY_STRING)] ||
            [string length $env(QUERY_STRING)] == 0} {
        if {[info exists env(CONTENT_LENGTH)] &&
                [string length $env(CONTENT_LENGTH)] != 0} {
            set query [read stdin $env(CONTENT_LENGTH)]
        } else {
            gets stdin query
        }
        set env(QUERY_STRING) $query
        set env(CONTENT_LENGTH) 0
    }
    return $env(QUERY_STRING)
}
```

An HTML form can have several form elements with the same name, and this can result in more than one value for each name. If you blindly use array set to map the results of Cgi_List into an array, you will lose the repeated values. Example 11–7 shows Cgi_Parse and Cgi_Value that store the query data in a global cgi array. Cgi_Parse adds list structure whenever it finds a repeated form value. The global cgilist array keeps a record of how many times a form value is repeated. The Cgi_Value procedure returns elements of the global cgi array, or the empty string if the requested value is not present.

Example 11–7 Cgi_Parse and Cgi_Value store query data in the cgi array.

```
proc Cgi_Parse {} {
    global cgi cgilist
    catch {unset cgi cgilist}
    set query [Cgi_Query]
    regsub -all {\+} $query { } query
    foreach {name value} [split $query &=] {
        set name [CgiDecode $name]
```

```
        if {[info exists cgilist($name)] &&
               ($cgilist($name) == 1)} {
            # Add second value and create list structure
            set cgi($name) [list $cgi($name) \
               [Url_Decode $value]]
        } elseif {[info exists cgi($name)]} {
            # Add additional list elements
            lappend cgi($name) [CgiDecode $value]
        } else {
            # Add first value without list structure
            set cgi($name) [CgiDecode $value]
            set cgilist($name) 0    ;# May need to listify
        }
        incr cgilist($name)
    }
    return [array names cgi]
}
proc Cgi_Value {key} {
    global cgi
    if {[info exists cgi($key)]} {
        return $cgi($key)
    } else {
        return {}
    }
}
proc Cgi_Length {key} {
    global cgilist
    if {[info exist cgilist($key)]} {
        return $cgilist($key)
    } else {
        return 0
    }
}
```

Decoding HTML Entities

The next example is a decoder for HTML *entities*. In HTML, special characters are encoded as entities. If you want a literal < or > in your document, you encode them as the entities `<` and `>`, respectively, to avoid conflict with the *<tag>* syntax used in HTML. HTML syntax is briefly described in Chapter 3 on page 32. Characters with codes above 127 such as copyright © and egrave è are also encoded. There are named entities, such as `<` for < and `&egrave;` for è. You can also use decimal-valued entities such as `&#169;` for ©. Finally, the trailing semicolon is optional, so `&lt` or `<` can both be used to encode <.

The entity decoder is similar to `Url_Decode`. In this case, however, we need to be more careful with `subst`. The text passed to the decoder could contain special characters like a square bracket or dollar sign. With `Url_Decode` we can rely on those special characters being encoded as, for example, `%24`. Entity encoding is different (do not ask me why URLs and HTML have different encoding standards), and dollar signs and square brackets are not necessarily encoded. This

requires an additional pass to quote these characters. This regsub puts a backslash in front of all the brackets, dollar signs, and backslashes.

```
regsub -all {[][$\\]} $text {\\&} new
```

The decimal encoding (e.g., ©) is also more awkward than the hexadecimal encoding used in URLs. We cannot force a decimal interpretation of a number in Tcl. In particular, if the entity has a leading zero (e.g.,
) then Tcl interprets the value (e.g., 010) as octal. The scan command is used to do a decimal interpretation. It scans into a temporary variable, and set is used to get that value:

```
regsub -all {&#([0-9][0-9]?[0-9]?);?}  $new \
    {[format %c [scan \1 %d tmp; set tmp]]} new
```

With advanced regular expressions, this could be written as follows using bound quantifiers to specify one to three digits:

```
regsub -all {&#(\d{1,3});?}  $new \
    {[format %c [scan \1 %d tmp;set tmp]]} new
```

The named entities are converted with an array that maps from the entity names to the special character. The only detail is that unknown entity names (e.g., &foobar;) are not converted. This mapping is done inside HtmlMapEntity, which guards against invalid entities.

```
regsub -all {&([a-zA-Z]+)(;?)} $new \
    {[HtmlMapEntity \1 \\\2 ]} new
```

If the input text contained:

```
[x &lt; y]
```

then the regsub would transform this into:

```
\[x [HtmlMapEntity lt \; ] y\]
```

Finally, subst will result in:

```
[x < y]
```

Example 11–8 Html_DecodeEntity.

```
proc Html_DecodeEntity {text} {
   if {![regexp & $text]} {return $text}
   regsub -all {[][$\\]} $text {\\&} new
   regsub -all {&#([0-9][0-9]?[0-9]?);?}  $new {\
      [format %c [scan \1 %d tmp;set tmp]]} new
   regsub -all {&([a-zA-Z]+)(;?)} $new \
      {[HtmlMapEntity \1 \\\2 ]} new
   return [subst $new]
}
proc HtmlMapEntity {text {semi {}}} {
   global htmlEntityMap
   if {[info exist htmlEntityMap($text)]} {
      return $htmlEntityMap($text)
   } else {
      return $text$semi
   }
}
```

```
# Some of the htmlEntityMap
array set htmlEntityMap {
    lt <  gt >  amp &
    aring \xe5  atilde \xe3
    copy  \xa9  ecirc  \xea  egrave \xe8
}
```

A Simple HTML Parser

The following example is the brainchild of Stephen Uhler. It uses `regsub` to transform HTML into a Tcl script. When it is evaluated the script calls a procedure to handle each tag in an HTML document. This provides a general framework for processing HTML. Different callback procedures can be applied to the tags to achieve different effects. For example, the `html_library-0.3` package on the CD-ROM uses `Html_Parse` to display HTML in a Tk text widget.

Example 11–9 `Html_Parse`.

```
proc Html_Parse {html cmd {start {}}} {

    # Map braces and backslashes into HTML entities
    regsub -all \{ $html {\&ob;} html
    regsub -all \} $html {\&cb;} html
    regsub -all {\\} $html {\&bsl;} html

    # This pattern matches the parts of an HTML tag
    set s" \t\r\n"   ;# white space
    set exp <(/?)(\[^$s>]+)\[$s]*(\[^>]*)>

    # This generates a call to cmd with HTML tag parts
    # \1 is the leading /, if any
    # \2 is the HTML tag name
    # \3 is the parameters to the tag, if any
    # The curly braces at either end group of all the text
    # after the HTML tag, which becomes the last arg to $cmd.
    set sub "\}\n$cmd {\\2} {\\1} {\\3} \{"
    regsub -all $exp $html $sub html

    # This balances the curly braces,
    # and calls $cmd with $start as a pseudo-tag
    # at the beginning and end of the script.
    eval "$cmd {$start} {} {} {$html}"
    eval "$cmd {$start} / {} {}"
}
```

The main `regsub` pattern can be written more simply with advanced regular expressions:

```
set exp {<(/?)(\S+?)\s*(.*?)>}
```

An example will help visualize the transformation. Given this HTML:

```
<Title>My Home Page</Title>
<Body bgcolor=white text=black>
<H1>My Home</H1>
This is my <b>home</b> page.
```

and a call to Html_Parse that looks like this:

```
Html_Parse $html {Render .text} hmstart
```

then the generated program is this:

```
Render .text {hmstart} {} {} {}
Render .text {Title} {} {} {My Home Page}
Render .text {Title} {/} {} {
}
Render .text {Body} {} {bgcolor=white text=black} {
}
Render .text {H1} {} {} {My Home}
Render .text {H1} {/} {} {
This is my }
Render .text {b} {} {} {home}
Render .text {b} {/} {} { page.
}
Render .text {hmstart} / {} {}
```

One overall point to make about this example is the difference between using eval and subst with the generated script. The decoders shown in Examples 11–5 and 11–8 use subst to selectively replace encoded characters while ignoring the rest of the text. In Html_Parse we must process all the text. The main trick is to replace the matching text (e.g., the HTML tag) with some Tcl code that ends in an open curly brace and starts with a close curly brace. This effectively groups all the unmatched text.

When eval is used this way you must do something with any braces and backslashes in the unmatched text. Otherwise, the resulting script does not parse correctly. In this case, these special characters are encoded as HTML entities. We can afford to do this because the cmd that is called must deal with encoded entities already. It is not possible to quote these special characters with backslashes because all this text is inside curly braces, so no backslash substitution is performed. If you try that the backslashes will be seen by the cmd callback.

Finally, I must admit that I am always surprised that this works:

```
eval "$cmd {$start} {} {} {$html}"
```

I always forget that $start and $html are substituted in spite of the braces. This is because double quotes are being used to group the argument, so the quoting effect of braces is turned off. Try this:

```
set x hmstart
set y "foo {$x} bar"
=> foo {hmstart} bar
```

Stripping HTML Comments

The `Html_Parse` procedure does not correctly handle HTML comments. The problem is that the syntax for HTML commands allows tags inside comments, so there can be > characters inside the comment. HTML comments are also used to hide Javascript inside pages, which can also contain >. We can fix this with a pass that eliminates the comments.

The comment syntax is this:

```
<!-- HTML comment, could contain <markup> -->
```

Using nongreedy quantifiers, we can strip comments with a single `regsub`:

```
regsub -all <!--.*?--> $html {} html
```

Using only greedy quantifiers, it is awkward to match the closing `-->` without getting stuck on embedded > characters, or without matching too much and going all the way to the end of the last comment. Time for another trick:

```
regsub -all --> $html \x81 html
```

This replaces all the end comment sequences with a single character that is not allowed in HTML. Now you can delete the comments like this:

```
regsub -all "<!--\[^\x81\]*\x81" $html {} html
```

Other Commands That Use Regular Expressions

Several Tcl commands use regular expressions.

- `lsearch` takes a `-regexp` flag so that you can search for list items that match a regular expression. The `lsearch` command is described on page 64.
- `switch` takes a `-regexp` flag, so you can branch based on a regular expression match instead of an exact match or a `string match` style match. The `switch` command is described on page 71.
- The Tk text widget can search its contents based on a regular expression match. Searching in the text widget is described on page 461.
- The *Expect* Tcl extension can match the output of a program with regular expressions. *Expect* is the subject of its own book, *Exploring Expect* (O'Reilly, 1995) by Don Libes.