Tcl and the Tk Toolkit

John K. Ousterhout
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

Note to readers:
This manuscript is a partial draft of a book to be published in early 1994 by Addison-Wesley (ISBN 0-201-63337-X). Addison-Wesley has given me permission to make drafts of the book available to the Tcl community to help meet the need for introductory documentation on Tcl and Tk until the book becomes available. Please observe the restrictions set forth in the copyright notice above: you're welcome to make a copy for yourself or a friend but any sort of large-scale reproduction or reproduction for profit requires advance permission from Addison-Wesley.

I would be happy to receive any comments you might have on this draft; send them to me via electronic mail at `ouster@cs.berkeley.edu`. I'm particularly interested in hearing about things that you found difficult to learn or that weren't adequately explained in this document, but I'm also interested in hearing about inaccuracies, typos, or any other constructive criticism you might have.

Chapter 1
Introduction

1.1 Introduction

This book is about two packages called Tcl and Tk. Together they provide a programming system for developing and using graphical user interface (GUI) applications. Tcl stands for “tool command language” and is pronounced “tickle”; is a simple scripting language for controlling and extending applications. It provides generic programming facilities that are useful for a variety of applications, such as variables and loops and procedures. Furthermore, Tcl is *embeddable*: its interpreter is implemented as a library of C procedures that can easily be incorporated into applications, and each application can extend the core Tcl features with additional commands specific to that application.

One of the most useful extensions to Tcl is Tk. It is a toolkit for the X Window System, and its name is pronounced “tee-kay”. Tk extends the core Tcl facilities with additional commands for building user interfaces, so that you can construct Motif user interfaces by writing Tcl scripts instead of C code. Like Tcl, Tk is implemented as a library of C procedures so it too can be used in many different applications. Individual applications can also extend the base Tk features with new user-interface widgets and geometry managers written in C.

Together, Tcl and Tk provide four benefits to application developers and users. First, Tcl makes it easy for any application to have a powerful scripting language. All that an application needs to do is to implement a few new Tcl commands that provide the basic features of that application. Then the application can be linked with the Tcl interpreter to produce a full-function scripting language that includes both the commands provided by Tcl (called the *Tcl core*) and those implemented by the application (see Figure 1.1).

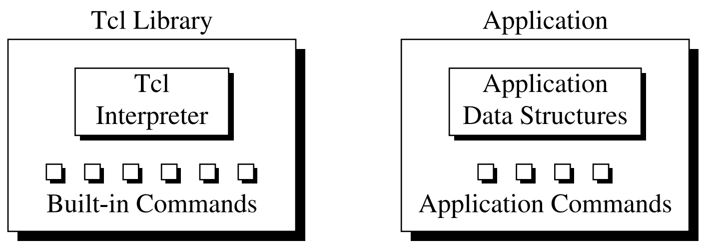


Figure 1.1. To create a new application based on Tcl, an application developer designs new C data structures specific to that application and writes C code to implement a few new Tcl commands. The Tcl library provides everything else that is needed to produce a fully programmable command language. The application can then be modified and extended by writing Tcl scripts.

For example, an application for reading electronic bulletin boards might contain C code that implements one Tcl command to query a bulletin board for new messages and another Tcl command to retrieve a given message. Once these commands exist, Tcl scripts can be written to cycle through the new messages from all the bulletin boards and display them one at a time, or keep a record in disk files of which messages have been read and which haven't, or search one or more bulletin boards for messages on a particular topic. The bulletin board application would not have to implement any of these additional functions in C; they could all be written as Tcl scripts, and users of the application could write additional Tcl scripts to add more functions to the application.

The second benefit of Tcl and Tk is rapid development. For example, many interesting windowing applications can be written entirely as Tcl scripts with no C code at all, using a windowing shell called `wish`. This allows you to program at a much higher level than you would in C or C++, and many of the details that C programmers must address are hidden from you. Compared to toolkits where you program entirely in C, such as Xt/Motif, there is much less to learn in order to use Tcl and Tk and much less code to write. New Tcl/Tk users can often create interesting user interfaces after just a few hours of learning, and many people have reported ten-fold reductions in code size and development time when they switched from other toolkits to Tcl and Tk.

Another reason for rapid development with Tcl and Tk is that Tcl is an interpreted language. When you use a Tcl application such as `wish` you can generate and execute new scripts on-the-fly without recompiling or restarting the application. This allows you to test out new ideas and fix bugs very rapidly. Since Tcl is interpreted it executes more slowly than compiled C code, of course, but modern workstations are surprisingly fast. For example, you can execute scripts with hundreds or even thousands of Tcl commands on each movement of the mouse with no perceptible delay. In the rare cases where performance becomes an issue, you can re-implement the most performance-critical parts of your Tcl scripts in C.

The third benefit of Tcl is that it makes an excellent “glue language”. Because it is embeddable, it can be used for many different purposes in many different programs. Once this happens, it becomes possible to write Tcl scripts that combine the features of all the programs. For example, any windowing application based on Tk can issue a Tcl script to any other Tk application. This feature makes multi-media effects much more accessible: once audio and video applications have been built with Tk (and there exist several already), any Tk application can issue “record” and “play” commands to them. In addition, spreadsheets can update themselves from database applications, user-interface editors can modify the appearance and behavior of live applications as they run, and so on. Tcl provides the *lingua franca* that allows application to work together.

The fourth benefit of Tcl is user convenience. Once a user learns Tcl and Tk, he or she can write scripts for any Tcl and Tk application merely by learning the few application-specific commands for the new application. This should make it possible for more users to personalize and enhance their applications.

1.2 Organization of the book

Chapter 2 uses several simple scripts to provide a quick overview of the most important features of Tcl and Tk. It is intended to give you the flavor of the systems and convince you that they are useful without explaining anything in detail. The remainder of the book goes through everything again in a more comprehensive fashion. It is divided into four parts:

- **Part I** introduces the Tcl scripting language. After reading this section you will be able to write scripts for Tcl applications.
- **Part II** describes the additional Tcl commands provided by Tk, which allow you to create user-interface widgets such as menus and scrollbars and arrange them in windowing applications. After reading this section you’ll be able to create new windowing application as `wish` scripts and write scripts to enhance existing Tk applications.
- **Part III** discusses the C procedures in the Tcl library and how to use them to create new Tcl commands. After reading this section you’ll be able to write new Tcl packages and applications in C.
- **Part IV** describes Tk’s library procedures. After reading this section you’ll be able to create new widgets and geometry managers in C.

Each of these major parts contains about ten short chapters. Each chapter is intended to be a self-contained description of a piece of the system, and you need not necessarily read the chapters in order. I recommend that you start by reading through Chapters 3-9 quickly, then skip to Chapters XXX-YYY, then read other chapters as you need them.

Not every feature of Tcl and Tk is covered here, and the explanations are organized to provide a smooth introduction rather than a terse reference source. A separate set of reference manual entries is available with the Tcl and Tk distributions. These are much more terse but they cover absolutely every feature of both systems.

This book assumes that you are familiar with the C programming language as defined by the ANSI C standard, and that you have some experience with UNIX and X11. In order to understand Part IV you will need to understand many of the features provided by the Xlib interface, such as graphics contexts and window attributes; however, these details are not necessary except in Part IV. You need not know anything about either Tcl or Tk before reading this book; both of them will be introduced from scratch.

1.3 Notation

Throughout the book I use a `Courier` font for anything that might be typed to a computer, such as variable names, procedure and command names, Tcl scripts, and C code. The examples of Tcl scripts use notation like the following:

```
   set a 44
⇒  44
```

Tcl commands such as "`set a 44`" is the example appear in Courier and their results, such as "*`44`*" in the example, appear in Courier oblique. The ⇒ symbol before the result indicates that this is a normal return value. If an error occurs in a Tcl command then the error message appears in Courier oblique, preceded by a ∅ symbol to indicate that this is an error rather than a normal return:

```
   set a 44 55
∅  wrong # args: should be "set varName ?newValue?"
```

When describing the syntax of Tcl commands, Courier oblique is used for formal argument names. If an argument or group of arguments is enclosed in question marks it means that the arguments are optional. For example, the syntax of the `set` command is as follows:

`set` *`varName`* ?*`newValue`*?

This means that the word `set` would be entered verbatim to invoke the command, while *`varName`* and *`newValue`* are the names of `set`'s arguments; when invoking the command you would type a variable name instead of *`varName`* and a new value for the variable instead of *`newValue`*. The *`newValue`* argument is optional.

Chapter 2
An Overview of Tcl and Tk

This chapter introduces Tcl and Tk with a series of scripts that illustrate the main features of the systems. Although you should be able to start writing simple scripts after reading this chapter, the explanations here are not intended to be complete. All of the information in this chapter will be revisited in more detail in later chapters, and several important aspects of the systems, such as their C interfaces, are not discussed at all in this chapter. The purpose of this chapter is to show you the overall structure of Tcl and Tk and the kinds of things they can do, so that when individual features are discussed in detail you'll be able to see why they are useful.

2.1 Getting started

In order to invoke Tcl scripts you must run a Tcl application. If Tcl is installed on your system then there should exist a simple Tcl shell application called `tclsh`, which you can use to try out some of the examples in this chapter (if Tcl has not been installed on your system then refer to Appendix A for information on how to obtain and install it). Type the command

```
tclsh
```

to your shell to invoke `tclsh`; `tclsh` will start up in interactive mode, reading Tcl commands from its standard input and passing them to the Tcl interpreter for evaluation. For starters, type the following command to `tclsh`:

```
expr 2 + 2
```

`Tclsh` will print the result "`4`" and prompt you for another command.

This example illustrates several features of Tcl. First, Tcl commands are similar in form to shell commands. Each command consists of one or more *words* separated by spaces or tabs. In the example there are four words: `expr`, `2`, `+`, and `2`. The first word of each command is its name: the name selects a C procedure in the application that will carry out the function of the command. The other words are *arguments* that are passed to the C procedure. `Expr` is one of the core commands built into the Tcl interpreter, so it exists in every Tcl application. It concatenates its arguments into a single string and evaluates the string as an arithmetic expression.

Each Tcl command returns a result string. For the `expr` command the result is the value of the expression. Results are always returned as strings, so `expr` converts its numerical result back to a string in order to return it. If a command has no meaningful result then it returns an empty string.

From now on I will use notation like the following to describe examples:

```
   expr 2 + 2
⇒  4
```

The first line is the command you type and the second line is the result returned by the command. The ⇒ symbol indicates that the line contains a return value; the ⇒ will not actually be printed out by `tclsh`. I will omit return values in cases where they aren't important, such as sequences of commands where only the last command's result matters.

Commands are normally terminated by newlines, so when you are typing to `tclsh` each line normally becomes a separate command. Semi-colons also act as command separators, in case you wish to enter multiple commands on a single line. It is also possible for a single command to span multiple lines; you'll see how to do this later.

The `expr` command supports an expression syntax similar to that of expressions in ANSI C, including the same precedence rules and most of the C operators. Here are a few examples that you could type to `tclsh`:

```
   expr 3 << 2
⇒  12
   expr 14.1*6
⇒  84.6
   expr (3 > 4) || (6 <= 7)
⇒  1
```

The first example illustrates the bitwise left-shift operator <<. The second example shows that expressions can contain real values as well as integer values. The last example shows the use of relational operators > and <= and the logical or operator `||`. As in C, boolean results are represented numerically with 1 for true and 0 for false.

To leave `tclsh`, invoke the `exit` command:

```
   exit
```

This command will terminate the application and return you to your shell.

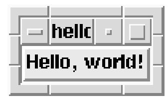


Figure 2.1. The “hello world” application. All of the decorations around the “Hello, world!” button are provided by the `mwm` window manager. If you use a different window manager then your decorations may be different.

2.2 Hello world with Tk

Although Tcl provides a full set of programming features such as variables, loops, and procedures, it is not intended to be a stand-alone programming environment. Tcl is intended to be used as part of applications that provide their own Tcl commands in addition to those in the Tcl core. The application-specific commands provide interesting primitives and Tcl is used to assemble the primitives into useful functions. Tcl by itself isn’t very interesting and it is hard to motivate all of Tcl’s facilities until you have seen some interesting application-specific commands to use them with.

Tk provides a particularly interesting set of commands to use with Tcl’s programming tools. Most of the examples in the book will use an application called `wish`, which is similar to `tclsh` except that it also includes the commands defined by Tk. Tk’s commands allow you to create graphical user interfaces. If Tcl and Tk have been installed on your system then you can invoke `wish` from your shell just like `tclsh`; it will display a small empty window on your screen and then read commands from standard input. Here is a simple `wish` script:

```
button .b -text "Hello, world!" -command exit
pack .b
```

If you type these two Tcl commands to `wish` the window’s appearance will change to what is shown in Figure 2.1. If you then move the pointer over the window and click mouse button 1, the window will disappear and `wish` will exit.

There are several things to explain about this example. First let us deal with the syntactic issues. The example contains two commands, `button` and `pack`, both of which are implemented by Tk. Although these commands look different than the `expr` command in the previous section, they have the same basic structure as all Tcl commands, consisting of one or more words separated by white space. The `button` command contains six words and the pack command contains two words.

The fourth word of the `button` command is enclosed in double quotes. This allows the word to include white space characters: without the quotes “`Hello,`” and “`world!`” would be separate words. The double-quotes are not part of the word itself; they are removed by the Tcl interpreter before the word is passed to the command as an argument.

For the `expr` command the word structure doesn't matter much since `expr` concatenates all its arguments together. However for the `button` and `pack` commands, and for most Tcl commands, the word structure is important. The `button` command expects its first argument to be the name of a window and the following arguments to come in pairs, where the first argument of each pair is the name of a *configuration option* and the second argument is a value for that option. Thus if the double-quotes were omitted the value of the `-text` option would be "`Hello,`" and "`world!`" would be treated as the name of a separate configuration option. Since there is no option defined with the name "`world!`" the command would return an error.

Now let us move on to the behavior of the commands. The basic building block for a graphical user interface in Tk is a *widget*. A widget is a window with a particular appearance and behavior (the terms "widget" and "window" are used synonymously in Tk). Widgets are divided into classes such as buttons, menus, and scrollbars. All the widgets in the same class have the same general appearance and behavior. For example, all button widgets display a text string or bitmap and execute a particular Tcl command when they are invoked with the mouse.

Widgets are organized hierarchically in Tk, with names that reflect their position in the hierarchy. The *main widget*, which appeared on the screen when you started `wish`, has the name ".". The name `.b` refers to a child of the main widget. Widget names in Tk are like file names in UNIX except that they use "`.`" as a separator character instead of "`/`". Thus `.a.b.c` refers to a widget that is a child of widget `.a.b`, which in turn is a child of `.a`, which is a child of the main widget.

Tk provides one command for each class of widgets, which you invoke to create widgets of that class. For example the `button` command creates button widgets. All of the widget creation commands have the same form: the first argument is the name of a new widget to create and additional arguments specify configuration options. Different widget classes support different sets of options. Widgets typically have many options (there are about 20 different options defined for buttons, for example), and default values are provided for the options that you don't specify. When a widget creation command like `button` is invoked it creates a new window by the given name and configures it as specified by the options.

The `button` command in the example specifies two options: `-text`, which is a string to display in the button, and `-command`, which is a Tcl script to execute when the user invokes the button. In this example the `-command` option is `exit`. Here are a few other button options that you can experiment with:

`-background`	The background color for the button.
`-foreground`	The color of the text in the button.
`-font`	The name of the font to use for the button, such as `*-times-medium-r-normal--*-120-*` for a 12-point Times Roman font.

The `pack` command makes the button widget appear on the screen. Creating a widget does not automatically cause it to be displayed. Independent entities called *geometry managers* are responsible for computing the sizes and locations of widgets and making them appear on the screen. The `pack` command in the example asks a geometry manager called the *packer* to manage `.b`. The command asks that `.b` fill the entire area of its parent window; furthermore, if the parent has more space than needed by its child, as in the example, the parent is shrunk so that it is just large enough to hold the child. Thus when you typed the `pack` command the main window shrunk from its original size to the size that appears in Figure 2.1.

2.3 Script files

In the examples so far you have typed Tcl commands interactively to `tclsh` or `wish`. You can also place commands into script files and invoke the script files just like shell scripts. To do this for the hello world example, place the following text in a file named `hello`:

```
#!/usr/local/bin/wish -f
button .b -text "Hello, world!" -command exit
pack .b
```

This script is the same as the one you typed earlier except for the first line. As far as `wish` is concerned this line is a comment but if you make the file executable (type "`chmod 775 hello`" to your shell, for example) you can then invoke the file directly by typing `hello` to your shell. When you do this the system will invoke `wish`, passing it the file as a script to interpret. `Wish` will display the same window shown in Figure 2.1 and wait for you to interact with it. In this case you will not be able to type commands interactively to wish; all you can do is click on the button.

Note: *This script will only work if `wish` is installed in `/usr/local/bin`. If `wish` has been installed somewhere else then you'll need to change the first line to reflect its location on your system.*

In practice users of Tk applications rarely type Tcl commands; they interact with the applications using the mouse and keyboard in the usual ways you would expect for graphical applications. Tcl works behind the scenes where users don't normally see it. The `hello` script behaves just the same as an application that has been coded in C with a toolkit such as Motif and compiled into a binary executable file.

During debugging, though, it is common for application developers to type Tcl commands interactively. For example, you could test out the `hello` script by starting `wish` interactively (type `wish` to your shell instead of `hello`). Then type the following Tcl command:

```
source hello
```

`Source` is a Tcl command that takes a file name as argument. It reads the file and evaluates it as a Tcl script. This will generate the same user interface as if you had invoked `hello` directly from your shell, but you can now type Tcl commands interactively too. For example, you could edit the script file to change the `-command` option to

```
-command "puts Good-bye!; exit"
```

then type the following commands interactively to `wish` without restarting the program:

```
destroy .b
source hello
```

The first command will delete the existing button and the second command will recreate the button with the new `-command` option. Now when you click on the button the `puts` command will print a message on standard output before `wish` exits.

2.4 Variables and substitutions

Tcl allows you to store values in variables and use those values in commands. For example, consider the following script, which you could type to either `tclsh` or `wish`:

```
   set a 44
⇒  44
   expr $a*4
⇒  176
```

The first command assigns the value “`44`” to variable `a` and returns the variable’s value. In the secon command t he `$` causes Tcl to perform *variable substitution*: the Tcl interpreter replaces the dollar-sign and the variable name following it with the value of the variable, so that the actual argument received by `expr` is “`44*4`”. Variables need not be declared in Tcl; they are created automatically when assigned to. Variable values are stored as strings and arbitrary string values of any length are allowed. Of course, in this example an error will occur in `expr` if the value of `a` doesn’t make sense as an integer or real number (try other values and see what happens).

Tcl also provides *command substitution*, which allows you to use the result of one command in an argument to another command:

```
   set a 44
   set b [expr $a*4]
⇒  176
```

Square brackets invoke command substitution: everything inside the brackets is evaluated as a separate Tcl script and the result of that script is substituted into the word in place of the bracketed command. In this example the second argument of the second command will be “`176`”.

2.5 Control structures

The next example uses variables and substitutions along with some simple control structures to create a Tcl procedure `power` that raises a base to an integer power:

```
proc power {base p} {
    set result 1
    while {$p > 0} {
        set result [expr $result*$base]
        set p [expr $p-1]
    }
    return $result
}
```

If you type the above lines to `wish` or `tclsh`, or if you enter them into a file and then `source` the file, a new command `power` will become available. The command takes two arguments, a number and an integer power, and its result is the number raised to the power:

```
    power 2 6
⇒   64
    power 1.15 5
⇒   2.01136
```

This example uses one additional piece of Tcl syntax: braces. Braces are like double-quotes in that they can be placed around a word that contains embedded spaces. However, braces are different from double-quotes in two respects. First, braces nest. The last word of the `proc` command starts after the open brace on the first line and contains everything up to the close brace on the last line. The Tcl interpreter removes the outer braces and passes everything between them, including several nested pairs of braces, to `proc` as an argument. The second difference between braces and double-quotes is that no substitutions occur inside braces, whereas they do inside quotes. All of the characters between the braces are passed verbatim to `proc` without any special processing.

The `proc` command takes three arguments: the name of a procedure, a list of argument names separated by white space, and the body of the procedure, which is a Tcl script. `Proc` enters the procedure name into the Tcl interpreter as a new command. Whenever the command is invoked, the body of the procedure will be evaluated. While the procedure body is executing it can access its arguments as variables: `base` will hold the first argument to power and `p` will hold the second argument.

The body of the `power` procedure contains three Tcl commands: `set`, `while`, and `return`. The `while` command does most of the work of the procedure. It takes two arguments, an expression "`$p > 0`" and a body, which is another multi-line Tcl script. The `while` command evaluates its expression argument and if the result is non-zero then it evaluates the body as a Tcl script. It repeats this process over and over until eventually the expression evaluates to zero. In the example, the body of the `while` command multiplies the result value by base and then decrements p. When p reaches zero the result contains the desired power of base.

The return command causes the procedure to exit with the value of variable result as the procedure's result. If it is omitted then the return value of the procedure will be the result of the last command in the procedure's body. In the case of power this would be the result of while, which is always an empty string.

The use of braces in this example is crucial. The single most difficult issue in writing Tcl scripts is managing substitutions: making them happen when you want them and preventing them from happening when you don't want them. Braces prevent substitutions or defer them until later. The body of the procedure must be enclosed in braces because we don't want variable and command substitutions to occur at the time the body is passed to proc as an argument; we want the substitutions to occur later, when the body is evaluated as a Tcl script. The body of the while command is enclosed in braces for the same reason: rather than performing the substitutions once, while parsing the while command, we want the substitutions to be performed over and over, each time the body is evaluated. Braces are also needed in the "{$p > 0}" argument to while. Without them the value of variable p would be substituted when parsing the while command; the expression would have a constant value and while would loop forever (you can try replacing some of the braces in the example with double quotes to see what happens).

In the examples in this book I use a stylized syntax where the open brace for an argument that is a Tcl script appears at the end of one line, the script follows on successive lines indented, and the close brace is on a line by itself after the script. Although I think that this makes for readable scripts, Tcl doesn't require this particular syntax. Script arguments are subject to the same syntax rules as any other arguments; in fact the Tcl interpreter doesn't even know that an argument is a script at the time it parses it. One consequence of this is that the open parenthesis must be on the same line as the preceding portion of the command. If the open brace is moved to a line by itself then the newline before the open brace will terminate the command.

By now you have seen nearly the entire Tcl language syntax. The only remaining syntactic feature is backslash substitution, which allows you to enter special characters such as dollar-signs into a word without enclosing the entire word in braces. Note that while and proc are not special syntactic elements in Tcl. They are just commands that take arguments just like all Tcl commands. The only special thing about while and proc is that they treat some of their arguments as Tcl scripts and cause the scripts to be evaluated. Many other commands also do this. The button command was one example (its -command option is a Tcl script), and you'll read about several other control structures later on, such as for, foreach, case, and eval.

One final note about procedures. The variables in a procedure are normally local to that procedure and will not be visible outside the procedure. In the power example the local variables include the arguments base and p as well as the variable result. A fresh set of local variables is created for each call to a procedure (arguments are passed by copying their values), and when a procedure returns its local variables are deleted. Vari-

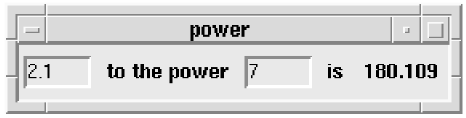


Figure 2.2. A graphical user interface that computes powers of a base.

ables named outside any procedure are called *global variables*; they last forever unless explicitly deleted. You'll find out later how a procedure can access global variables and the local variables of other active procedures.

2.6 Event bindings

The next example provides a graphical front-end for the `power` procedure. In addition to demonstrating two new widget classes it illustrates Tk's *binding* mechanism. A binding causes a particular Tcl script to be evaluated whenever a particular event occurs in a particular window. The `-command` option for buttons is an example of a simple binding implemented by a particular widget class. Tk also includes a more general mechanism that can be used to extend the behavior of arbitrary widgets in nearly arbitrary ways.

To run the example, copy the following script into a file `power` and invoke the file from your shell.

```
#!/usr/local/bin/wish -f
proc power {base p} {
    set result 1
    while {$p > 0} {
        set result [expr $result*$base]
        set p [expr $p-1]
    }
    return $result
}
entry .base -width 6 -relief sunken -textvariable base
label .label1 -text "to the power"
entry .power -width 6 -relief sunken -textvariable power
label .label2 -text "is"
label .result -textvariable result
pack .base .label1 .power .label2 .result \
        -side left -padx 1m -pady 2m
bind .base <Return> {set result [power $base $power]}
bind .power <Return> {set result [power $base $power]}
```

This script will produce a screen display like that in Figure 2.2. There are two entry widgets in which you can click with the mouse and type numbers. If you type return in either of the entries, the result will appear on the right side of the window. You can compute different results by modifying either the base or the power and then typing return again.

This application consists of five widgets: two entries and three labels. Entries are widgets that display one-line text strings that you can edit interactively. The two entries, .base and .power, are used for entering the numbers. Each entry is configured with a -width of 6, which means it will be large enough to display about 6 digits, and a -relief of sunken, which gives the entry a depressed appearance. The -textvariable option for each entry specifies the name of a global variable to hold the entry's text: any changes you make in the entry will be reflected in the variable and vice versa.

Two of the labels, .label1 and .label2, hold decorative text and the third, .result, holds the result of the power computation. The -textvariable option for .result causes it to display whatever string is in global variable result whereas .label1 and .label2 display constant strings.

The pack command arranges the five widgets in a row from left to right. The command occupies two lines in the script; the backslash at the end of the first line is a line-continuation character: it causes the newline to be treated as a space. The -side option means that each widget is placed at the left side of the remaining space in the main widget: first .base is placed at the left edge of the main window, then .label1 is placed at the left side of the space not occupied by .base, and so on. The -padx and -pady options make the display a bit more attractive by arranging for 1 millimeter of extra space on the left and right sides of each widget, plus 2 millimeters of extra space above and below each widget. The "m" suffix specifies millimeters; you could also use "c" for centimeters, "i" for inches, "p" for points, or no suffix for pixels.

The bind commands connect the user interface to the power procedure. Each bind command has three arguments: the name of a window, an event specification, and a Tcl script to invoke when the given event occurs in the given window. <Return> specifies an event consisting of the user typing the return key on the keyboard. Here are a few other event specifiers that you might find useful:

<Button-1>	Mouse button 1 is pressed.
<ButtonRelease-1>	Mouse button 1 is released.
<Double-Button-1>	Double-click on mouse button 1.
<1>	Short-hand for <Button-1>.
<Key-a>	Key "a" is pressed.
<a> or a	Short-hand for <Key-a>.
<Motion>	Pointer motion with no buttons or modifier keys pressed.
<B1-Motion>	Pointer motion with button 1 pressed.
<Any-Motion>	Pointer motion with any (or no) buttons or modifier keys pressed.

The scripts for the bindings invoke `power`, passing it the values in the two entries, and they store the result in `result` so that it will be displayed in the `.result` widget. These bindings extend the generic built-in behavior of the entries (editing text strings) with application-specific behavior (computing a value based on two entries and displaying that value in a third widget).

The script for a binding has access to several pieces of information about the event, such as the location of the pointer when the event occurred. For an example, start up `wish` interactively and type the following command to it:

```
bind . <Any-Motion> {puts "pointer at %x,%y"}
```

Now move the pointer over the window. Each time the pointer moves a message will be printed on standard output giving its new location. When the pointer motion event occurs, Tk scans the script for % sequences and replaces them with information about the event before passing the script to Tcl for evaluation. `%x` is replaced with the pointer's x-coordinate and `%y` is replaced with the pointer's y-coordinate.

2.7 Subprocesses

Normally Tcl executes each command by invoking a C procedure in the application to carry out its function; this is different from a shell program like `sh` where each command is normally executed in a separate subprocess. However, Tcl also allows you to create subprocesses, using the `exec` command. Here is a simple example of `exec`:

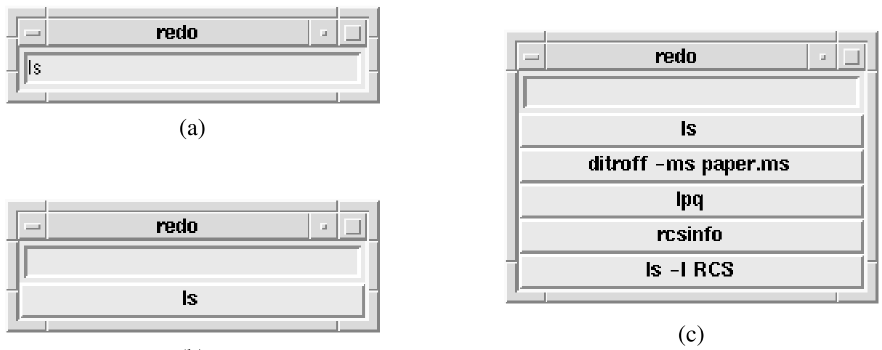
```
exec grep #include tk.h
⇒ #include <tcl.h>
  #include <X11/Xlib.h>
  #include <stddef.h>
```

The `exec` command treats its arguments much like the words of a shell command line. In this example `exec` creates a new process to run the `grep` program and passes it "`#include`" and "`tk.h`" as arguments, just as if you had typed

```
grep #include tk.h
```

to your shell. The `grep` program searches file `tk.h` for lines that contain the string `#include` and prints those lines on its standard output. However, `exec` arranges for standard output from the subprocess to be piped back to Tcl. `Exec` waits for the process to exit and then it returns all of the standard output as its result. With this mechanism you can execute subprocesses and use their output in Tcl scripts. `Exec` also supports input and output redirection using standard shell notation such as <, <<, and >, pipelines with |, and background processes with `&`.

The example below creates a simple user interface for saving and re-invoking commonly used shell commands. Type the following script into a file named `redo` and invoke it:



(a) (b) (c)

Figure 2.3. The `redo` application. The user can type a command in the entry window, as in (a). When the user types return the command is invoked as a subprocess using `exec` and a new button is created that can be used to re-invoke the command later, as in (b). Additional commands can be typed to create additional buttons, up to a limit of five buttons as in (c).

```
#!/usr/local/bin/wish -f
set id 0
entry .entry -width 30 -relief sunken -textvariable cmd
pack .entry -padx 1m -pady 1m
bind .entry <Return> {
    set id [expr $id + 1]
    if {$id > 5} {
        destroy .b[expr $id - 5]
    }
    button .b$id -command "exec <@stdin >@stdout $cmd" \
            -text $cmd
    pack .b$id -fill x
    .b$id invoke
    .entry delete 0 end
}
```

Initially the script creates an interface with a single entry widget. You can type a shell command such as `ls` into the entry, as shown in Figure 2.3(a). When you type return the command gets executed just as if you had typed it to the shell from which you invoked `redo`, and output from the command appears in the shell's window. Furthermore, the script creates a new button widget that displays the command (see Figure 2.3(b)) and you can re-invoke the command later by clicking on the button. As you type more and more commands, more and more buttons appear, up to a limit of five remembered commands as in Figure 2.3(c).

Note: This example suffers from several limitations. For example, you cannot specify wild-cards such as "" in command lines, and the "*`cd`*" command doesn't behave properly. In Part I you'll read about Tcl facilities that you can use to eliminate these limitations.*

The most interesting part of the `redo` script is in the `bind` command. The binding for `<Return>` must execute the command, which is stored in the `cmd` variable, and create a new button widget. First it creates the widget. The button widgets have names like `.b1`, `.b2`, and so on, where the number comes from the variable `id. Id` starts at zero and increments before each new button is created. The notation "`.b$id`" generates a widget name by "`.b`" and the value of `id`. Before creating a new widget the script checks to see if there are already five saved commands; if so then the oldest existing button is deleted. The notation "`.b[expr $id - 5]`" produces the name of the oldest button by subtracting five from the number of the new button and concatenating it with "`.b`". The `-command` option for the new button invokes `exec` and redirects standard input and standard output for the subprocess(es) to `wish`'s standard input and standard output, which are the same as those of the shell from which `wish` was invoked: this causes output from the subprocesses to appear in the shell's window instead of being returned to `wish`.

The command "`pack .b$id -fill x`" makes the new button appear at the bottom of the window. The option "`-fill x`" improves the appearance by stretching the button horizontally so that it fills the width of the window even it it doesn't really need that much space for its text. Try omitting the `-fill` option to see what happens without it.

The last two commands of the binding script are called *widget commands*. Whenever a new widget is created a new Tcl command is also created with the same name as the widget, and you can invoke this command to communicate with the widget. The first argument to a widget command selects one of several operations and additional arguments are used as parameters for that operation. In the `redo` script the first widget command causes the button widget to invoke its `-command` option just as if you had clicked the mouse button on it. The second widget command clears the entry widget in preparation for a new command to be typed.

Each class of widget supports a different set of operations in its widget commands, but many of the operations are similar from widget to widget. For example, every widget class supports a `configure` widget command that can be used to modify any of the configuration options for the widget. If you run the `redo` script interactively you could type the following command to change the background of the entry widget to yellow:

```
.entry configure -background yellow
```

Or, you could type

```
.b1 configure -foreground brown
.b1 flash
```

to change the color of the text in button `.b1` to brown and then cause the button to flash.

One of the most important things about Tcl and Tk is that they make every aspect of an application accessible and modifiable at run-time. For example, the `redo` script modifies its own interface on the fly. In addition, Tk provides commands that you can use to query the structure of the widget hierarchy, and you can use `configure` widget commands to query and modify the configuration options of individual widgets.

2.8 Additional features of Tcl and Tk

The examples in this chapter used every aspect of the Tcl language syntax and they illustrated many of the most important features of Tcl and Tk. However, Tcl and Tk contain many other facilities that are not used in this chapter; all of these will be described later in the book. Here is a sampler of some of the most useful features that haven't been mentioned yet:

Arrays and lists. Tcl provides associative arrays for storing key-value pairs efficiently and lists for managing aggregates of data.

More control structures. Tcl provides several additional commands for controlling the flow of execution, such as `eval`, `for`, `foreach`, and `switch`.

String manipulation. Tcl contains a number of commands for manipulating strings, such as measuring their length and performing regular expression pattern matching and substitution.

File access. You can read and write files from Tcl scripts and retrieve directory information and file attributes such as length and creation time.

More widgets. Tk contains many widget classes besides those shown here, such as menus, scrollbars, a drawing widget called a *canvas*, and a text widget that makes it easy to achieve hypertext effects.

Access to other X facilities. Tk provides commands for accessing all of the major facilities in the X Window System, such as a command for communicating with the window manager (to set the window's title, for example), a command for retrieving the selection, and a command to manage the input focus.

C interfaces. Tcl provides C library procedures that you can use to define your own new Tcl commands in C, and Tk provides a library that you can use to create your own widget classes and geometry managers in C.

2.9 Extensions and applications

Tcl and Tk have an active and rapidly-growing user community that now numbers in the tens of thousands. Many people have built applications based on Tcl and Tk and packages that extend the base functionality of Tcl and Tk. Several of these packages and applications are publically available and widely used in the Tcl/Tk community. There isn't space in this book to discuss all of the exciting Tcl/Tk software in detail but this section gives a quick overview of five of the most popular extensions and applications. See Appendix A for information on how you can obtain them and other contributed Tcl/Tk software.

2.9.1 Expect

`Expect` is one of the oldest Tcl applications and also one of the most popular. It is a program that “talks” to interactive programs. Following a script, `expect` knows what output can be expected from a program and what the correct responses should be. It can be used to automatically control programs like `ftp`, `telnet`, `rlogin`, `crypt`, `fsck`, `tip`, and others that cannot be automated from a shell script because they require interactive input. `Expect` also allows the user to take control and interact directly with the program when desired. For example, the following `expect` script logs into a remote machine using the `rlogin` program, sets the working directory to that of the originating machine, then turns control over to the user:

```
#!/usr/local/bin/expect
spawn rlogin [lindex $argv 1]
expect -re "(%|#) "
send "cd [pwd]\r"
interact
```

The `spawn`, `expect`, `send`, and `interact` commands are implemented by `expect`, and `lindex` and `pwd` are built-in Tcl commands. The `spawn` command starts up `rlogin`, using a command-line argument as the name of the remote machine. The `expect` command waits for `rlogin` to output a prompt (either “`%`” or “`#`”, followed by a space), then `send` outputs a command to change the working directory, just as if a user had typed the command interactively. Finally, `interact` causes `expect` to step out of the way so that the user who invoked the `expect` script can now talk directly to `rlogin`.

`Expect` can be used for many purposes, such as a scriptable front-end to debuggers, mailers, and other programs that don’t have scripting languages of their own. The programs require no changes to be driven by expect. `Expect` is also useful for regression testing of interactive programs. `Expect` can be combined with Tk or other Tcl extensions. For example, using Tk it is possible to make a graphical front end for an existing interactive application without changing the application.

`Expect` was created by Don Libes.

2.9.2 Extended Tcl

Extended Tcl (TclX) is a library package that augments the built-in Tcl commands with many additional commands and procedures oriented towards system programming tasks. It can be used with any Tcl application. Here are a few of the most popular features of TclX:

- Access to many additional POSIX system calls and functions.
- A file scanning facility with functionality much like that of the `awk` program.

- Keyed lists, which provide functionality similar to C structures.
- Commands for manipulating times and dates and converting them to and from ASCII.
- An on-line help facility.
- Facilities for debugging, profiling, and program development.

Many of the best features of TclX are no longer part of TclX: they turned out to be so widely useful that they were incorporated into the Tcl core. Among the Tcl features pioneered by TclX are file input and output, array variables, real arithmetic and transcendental functions, auto-loading, XPG-based internationalization, and the `upvar` command.

Extended Tcl was created by Karl Lehenbauer and Mark Diekhans.

2.9.3 XF

Tk makes it relatively easy to create graphical user interfaces by writing Tcl scripts, but XF makes it even easier. XF is an interactive interface builder: you design a user interface by manipulating objects on the screen, then XF creates a Tcl script that will generate the interface you have designed (see Figure 2.4). XF provides tools for creating and configuring widgets, arranging them with Tk's geometry managers, creating event bindings, and so on. XF manipulates a live application while it is running, so the full effect of each change in the interface can be seen and tested immediately.

XF supports all of Tk's built-in widget classes and allows you to add new widget classes by writing class-specific Tcl scripts for XF to use to handle the classes. You needn't use XF exclusively: you can design part of a user interface with XF and part by writing Tcl scripts. XF supports most of the currently available extensions to Tcl and Tk, and XF itself is written in Tcl.

XF was created by Sven Delmas. It is based on an earlier interface builder for Tk called BYO, which was developed at the Victoria University of Wellington, New Zealand.

2.9.4 Distributed programming

Tcl Distributed Programming (Tcl-DP) is a collection of Tcl commands that simplify the development of distributed programs. Tcl-DP's most important feature is a *remote procedure call* facility, which allows Tcl applications to communicate by exchanging Tcl scripts. For example, the following script uses Tcl-DP to implement a trivial "id server", which returns unique identifiers in response to `GetId` requests:

```
set myId 0
proc GetId {} {
    global myId;
    set myId [expr $myId+1]
    return $myId
}
MakeRPCServer 4545
```

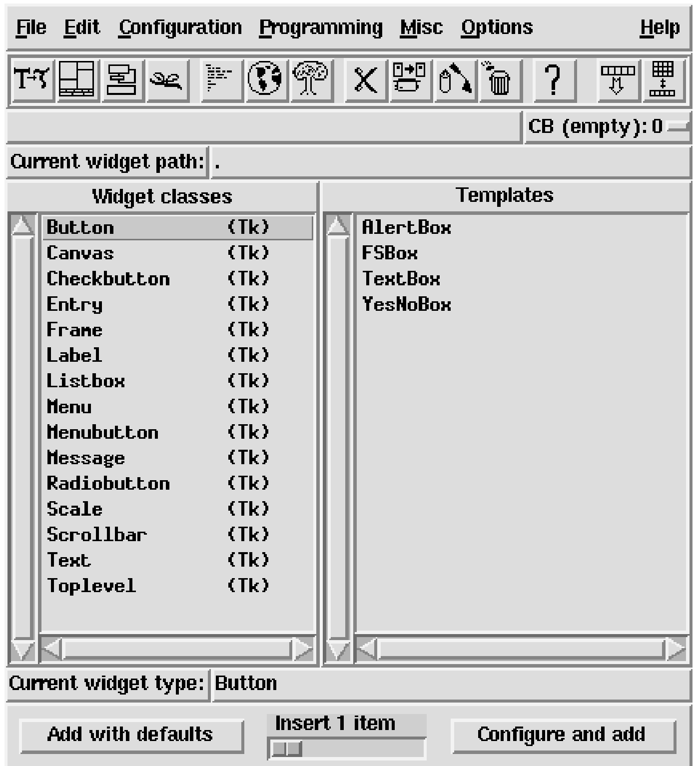


Figure 2.4. A screen dump showing the main window of XF, an interactive application builder for Tcl and Tk.

All of the code in this script except the last line is ordinary Tcl code that defines a global variable `myId` and a procedure `GetId` that increments the variable and returns its new value. The `MakeRPCServer` command is implemented by Tcl-DP; it causes the application to listen for requests on TCP socket 4545.

Other Tcl applications can communicate with this server using scripts that look like the following:

```
set server [MakeRPCClient server.company.com 4545]
RPC $server GetId
```

The first command opens a connection with the server and saves an identifier for that connection. The arguments to `MakeRPCClient` identify the server's host and the socket on which the server is listening. The `RPC` command performs a remote procedure call. Its arguments are a connection identifier and an arbitrary Tcl script. `RPC` forwards the script to the server; the server executes the script and returns its result (a new identifier in this case), which becomes the result of the `RPC` command. Any script whatosever could be substituted in place of the `GetId` command.

Tcl-DP also includes several other features, including asynchronous remote procedure calls, where the client need not wait for the call to complete, a distributed object system in which objects can be replicated in several applications and updates are automatically propagated to all copies, and a simple name service. Tcl-DP has been used for applications such as a video playback system, groupware, and games. Tcl-DP is more flexible than most remote procedure call systems because it is not based on compiled interfaces between clients and servers: it is easy in Tcl-DP to connect an existing client to a new server without recompiling or restarting the client.

Tcl-DP was created by Lawrence A. Rowe, Brian Smith, and Steve Yen.

2.9.5 Ak

Ak is an audio extension for Tcl. It is built on top of AudioFile, a network-transparent, device independent audio system that runs on a variety of platforms. Ak provides Tcl commands for file playback, recording, telephone control, and synchronization. The basic abstractions in Ak are connections to AudioFile servers, device contexts (which encapsulate the state for a particular audio device), and requests such as file playback. For example, here is a script that plays back an audio file on a remote machine:

```
audioserver remote "server.company.com:0"
remote context room -device 1
room create play "announcement-file.au"
```

The first command opens a connection to the audio server on the machine `server.company.com` and gives this connection the name `remote`. It also creates a command named `remote`, which is used to issue commands over the connection. The second command creates a context named `room`, which is associated with audio device 1 on the server, and also creates a command named `room` for communicating with the context. The last command initiates a playback of a particular audio file.

Ak implements a unique model of time that allows clients to specify precisely when audio samples are going to emerge. It also provides a mechanism to execute arbitrary Tcl scripts at specified audio times; this can be used to achieve a variety of hypermedia effects, such as displaying images or video in sync with an audio playback. When combined with Tk, Ak provides a powerful and flexible scripting system for developing multimedia applications such as tutorials and telephone inquiry systems.

Ak was created by Andrew C. Payne.

Part I:

The Tcl Language

Chapter 3
Tcl Language Syntax

In order to write Tcl scripts you must learn two things. First, you must learn the Tcl syntax, which consists of about a half-dozen rules that determine how commands are parsed. The Tcl syntax is the same for every command. Second, you must learn about the individual commands that you use in your scripts. Tcl provides about 60 built-in commands, Tk adds several dozen more, and any application based on Tcl or Tk will add a few more of its own. You'll need to know all of the syntax rules right away, but you can learn about the commands more gradually as you need them.

This chapter describes the Tcl language syntax. The remaining chapters in Part I describe the built-in Tcl commands, and Part II describes Tk's commands.

3.1 Scripts, commands, and words

A Tcl *script* consists of one or more *commands*. Commands are separated by newlines and semi-colons. For example,

```
set a 24
set b 15
```

is a script with two commands separated by a newline character. The same script could be written on a single line using a semi-colon separator:

```
set a 24; set b 15
```

Each command consists of one or more *words*, where the first word is the name of a command and additional words are arguments to that command. Words are separated by spaces and tabs. Each of the commands in the above examples has three words. There may

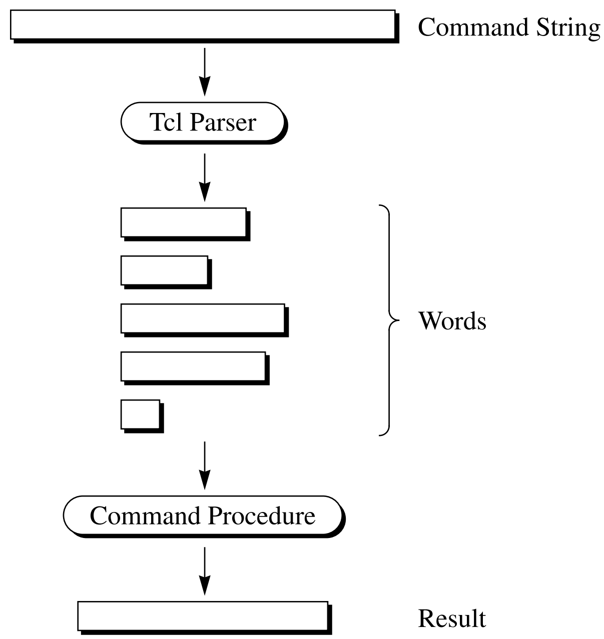


Figure 3.1. Tcl commands are evaluated in two steps. First the Tcl interpreter parses the command string into words, performing substitutions along the way. Then a command procedure processes the words to produce a result string. Each command has a separate command procedure.

be any number of words in a command, and each word may have an arbitrary string value. The white space that separates words is not part of the words, nor are the newlines and semi-colons that terminate commands

3.2 Evaluating a command

Tcl evaluates a command in two steps as shown in Figure 3.1: *parsing* and *execution*. In the parsing step the Tcl interpreter applies the rules described in this chapter to divide the command up into words and perform substitutions. Parsing is done in exactly the same way for every command. During the parsing step the Tcl interpreter does not apply any meaning to the values of the words. Tcl just performs a set of simple string operations such as replacing the characters "`$a`" with the string stored in variable `a`; Tcl does not know or care whether `a` or the resulting word is a number or the name of a widget or anything else.

In the execution step meaning is applied to the words of the command. Tcl treats the first word as a command name, checking to see if the command is defined and locating a *command procedure* to carry out its function. If the command is defined then the Tcl interpreter invokes its command procedure, passing all of the words of the command to the command procedure. The command procedure is free to interpret the words in any way that it pleases, and different commands apply very different meanings to their arguments

Note: *I use the terms "word" and "argument" interchangeably to refer to the values passed to command procedures. The only difference between these two terms is that the first argument is the second word.*

The following commands illustrate some of meanings that are commonly applied to arguments:

`set a 122`

In many cases, such as the `set` command, arguments may take any form whatsoever. The `set` command simply treats the first argument as a variable name and the second argument as a value for the variable. The command "`set 122 a`" is valid too: it creates a variable whose name is "`122`" and whose value is "`a`".

`expr 24/3.2`

The argument to `expr` must be an arithmetic expression that follows the rules described in Chapter 5. Several other commands also take expressions as arguments.

`eval {set a 122}`

The argument to `eval` is a Tcl script. `Eval` passes it to the Tcl interpreter where another round of parsing and execution occurs for the argument. Other control-flow commands such as `if` and `while` also take scripts as arguments.

`lindex {red green blue purple} 2`

The first argument to `lindex` is a *list* consisting of four values separated by spaces. This command will extract element 2 ("`blue`") from the list and return it. Tcl's commands for manipulating lists are described in Chapter 6.

`string length abracadabra`

Some commands, like `string` and the Tk widget commands, are actually several commands rolled into one. The first argument of the command selects one of several operations to perform and determines the meaning of the remaining arguments. For example "`string length`" requires one additional argument and computes its length, whereas "`string compare`" requires two additional arguments.

`button .b -text Hello -fg red`

The arguments starting with `-text` are option-value pairs that allow you to specify the options you care about and use default values for the others.

In writing Tcl scripts one of the most important things to remember is that the Tcl parser doesn't apply any meaning to the words of a command while it parses them. All of the above meanings are applied by individual command procedures, not by the Tcl parser. Another way of saying this is that arguments are quoted by default; if you want evaluation you must request it explicitly. This approach is similar to that of most shell languages but different than most programming languages. For example, consider the following C program:

```
x = 4;
y = x+10;
```

In the first statement C stores the integer value 4 in variable `x`. In the second statement C evaluates the expression "`x+10`", fetching the the value of variable `x` and adding 10, and stores the result in variable `y`. At the end of execution `y` has the integer value 14. If you want to use a literal string in C without evaluation you must enclose it in quotes. Now consider a similar-looking program written in Tcl:

```
set x 4
set y x+10
```

The first command assigns the *string* "`4`" to variable `x`. The value of the variable need not have any particular form. The second command simply takes the string "`x+10`" and stores it as the new value for `y`. At the end of the script y has the string value "`x+10`", not the integer value 14. In Tcl if you want evaluation you must ask for it explicitly:

```
set x 4
set y [expr $x+10]
```

Evaluation is requested twice in this example. First, the second word of the second command is enclosed in brackets, which tells the Tcl parser to evaluate the characters between the brackets as a Tcl script and use the result as the value of the word. Second, a dollar-sign has been placed before `x`. When Tcl parses the `expr` command it substitutes the value of variable `x` for the `$x`. If the dollar-sign were omitted then `expr`'s argument would contain the string "`x`", resulting in a syntax error. At the end of the script `y` has the string value "`14`", which is almost the same as in the C example.

3.3 Variable substitution

Tcl provides three forms of *substitution:* variable substitution, command substitution, and backslash substitution. Each substitution causes some of the original characters of a word to be replaced with some other value. Substitutions may occur in any word of a command, including the command name, and there may be any number of substitutions within a single word.

The first form of substitution is *variable substitution*. It is triggered by a dollar-sign character and it causes the value of a Tcl variable to be inserted into a word. For example, consider the following commands:

```
set kgrams 20
expr $kgrams*2.2046
⇒ 44.092
```

The first command sets the value of variable `kgrams` to `20`. The second command computes the corresponding weight in pounds by multiplying the value of `kgrams` by 2.2046. It does this using variable substitution: the string `$kgrams` is replaced with the value of variable `kgrams`, so that the actual argument received by the `expr` command procedure is "`20*2.2046`".

Variable substitution can occur anywhere within a word and any number of times as in the following command:

```
expr $result*$base
```

The variable name consists of all of the numbers, letters, and underscores following the dollar-sign. Thus the first variable name (`result`) extends up to the `*` and the second variable name (`base`) extends to the end of the word.

The examples above show only the simplest form of variable substitution. There are two other forms of variable substitution, which are used for associative array references and to provide more explicit control over the extent of a variable name (e.g. so that there can be a letter immediately following the variable name). These other forms are discussed in Chapter 4.

3.4 Command substitution

The second form of substitution provided by Tcl is *command substitution*. Command substitution causes part or all of a command word to be replaced with the result of another Tcl command. Command substitution is invoked by enclosing a nested command in brackets:

```
set kgrams 20
set lbs [expr $kgrams*2.2046]
⇒ 44.092
```

The characters between the brackets must constitute a valid Tcl script. The script may contain any number of commands separated by newlines or semi-colons in the usual fashion. The brackets and all of the characters in between are replaced with the result of the script. Thus in the example above the `expr` command is executed while parsing the words for `set`; its result, the string "`44.092`", becomes the second argument to `set`. As with variable substitution, command substitution can occur anywhere in a word and there may be more than one command substitution within a single word.

3.5 Backslash substitution

The final form of substitution in Tcl is *backslash substitution*. It is used to insert special characters such as newlines into words and also to insert characters like `[` and `$` without them being treated specially by the Tcl parser. For example, consider the following command:

```
set msg Eggs:\ \$2.18/dozen\nGasoline:\ \$1.49/gallon
⇒ Eggs: $2.18/dozen
  Gasoline: $1.49/gallon
```

There are two sequences of backslash followed by space; each of these sequences is replaced in the word by a single space and the space characters are not treated as word separators. There are also two sequences of backslash followed by dollar-sign; each of these is replaced in the word with a single dollar-sign, and the dollar signs are treated like ordinary characters (they do not trigger variable substitution). The backslash followed by `n` is replaced with a newline character

Table 3.1 lists all of the backslash sequences supported by Tcl. These include all of the sequences defined for ANSI C, such as `\t` to insert a tab character and `\xd4` to insert the character whose hexadecimal value is 0xd4. If a backslash is followed by any character not listed in the table, as in `\$` or `\[`, then the backslash is dropped from the word and the following character is included in the word as an ordinary character. This allows you to include any of the Tcl special characters in a word without the character being treated specially by the Tcl parser. The sequence `\\` will insert a single backslash into a word.

The sequence backslash-newline can be used to spread a long command across multiple lines, as in the following example:

```
pack .base .label1 .power .label2 .result \
        -side left -padx 1m -pady 2m
```

The backslash and newline, plus any leading space on the next line, are replaced by a single space character in the word. Thus the two lines together form a single command.

Note: *Backslash-newline sequences are unusual in that they are replaced in a separate preprocessing step before the Tcl interpreter parses the command. This means, for example, that the space character that replaces backslash-newline will be treated as a word separator unless it is between double-quotes or braces.*

3.6 Quoting with double-quotes

Tcl provides several ways for you to prevent the parser from giving special interpretation to characters such as `$` and semi-colon. These techniques are called *quoting*. You have already seen one form of quoting in backslash subsitution; for example, `\$` causes a dollar-sign to be inserted into a word without triggering variable substitution. In addition to backslash substitution Tcl provides two other forms of quoting: double-quotes and braces.

Backslash Sequence	Replaced By
`\a`	Audible alert (0x7)
`\b`	Backspace (0x8)
`\f`	Form feed (0xc)
`\n`	Newline (0xa)
`\r`	Carriage return (0xd)
`\t`	Tab (0x9)
`\v`	Vertical tab (0xb)
`\`*`ddd`*	Octal value given by *ddd* (one, two, or three *d*'s)
`\x`*`hh`*	Hex value given by *hh* (any number of *h*'s)
`\`*`newline space`*	A single space character.

Table 3.1. Backslash substitutions supported by Tcl. Each of the sequences in the first column is replaced by the corresponding character from the second column. If a backslash is followed by a character other than those in the first column, then the two characters are replaced by the second character.

Double-quotes disable word and command separators, while braces disable almost all special characters.

If a word is enclosed in double-quotes then spaces, tabs, newlines, and semi-colons are treated as ordinary characters within the word. The example from page 30 can be rewritten more cleanly with double-quotes as follows:

```
set msg "Eggs: \$2.18/dozen\nGasoline: \$1.49/gallon"
```

⇒ *`Eggs: $2.18/dozen`*
`Gasoline: $1.49/gallon`

Note that the quotes themselves are not part of the word. The `\n` in the example could also be replaced with an actual newline character, as in

```
set msg "Eggs: \$2.18/dozen
Gasoline: \$1.49/gallon"
```

but I think the script is more readable with `\n`.

Variable substitutions, command substitutions, and backslash substitutions all occur as usual inside double-quotes. For example, the following script sets `msg` to a string containing the name of a variable, its value, and the square of its value:

```
set a 2.1
set msg "a is $a; the square of a is [expr $a*$a]"
```

⇒ *a is 2.1; the square of a is 4.41*

If you would like to include a double-quote in a word enclosed in double-quotes, then use backlash substitution:

```
set name a.out
set msg "Couldn't open file \"$name\""
```

⇒ *Couldn't open file "a.out"*

3.7 Quoting with braces

Braces provide a more radical form of quoting where all the special charaters lose their meaning. If a word is enclosed in braces then the characters between the braces are the value of the word, verbatim. No substitutions are performed on the word and spaces, tabs, newlines, and semi-colons are treated as ordinary characters. The example on page 30 can be rewritten with braces as follows:

```
set msg {Eggs: $2.18/dozen
Gasoline: $1.49/gallon}
```

The dollar-signs in the word do not trigger variable substitution and the newline does not act as a command separator. In this case `\n` cannot be used to insert a newline into the wod as on page 31, because the `\n` will be included in the argument as-is without triggering backslash substitution:

```
set msg {Eggs: $2.18/dozen\nGasoline: $1.49/gallon}
```

⇒ *Eggs: $2.18/dozen\nGasoline: $1.49/gallon*

One of the most important uses for braces is to *defer evaluation*. Deferred evaluation means that special characters aren't processed immediately by the Tcl parser. Instead they will be passed to the command procedure as part of its argument and the command procedure will process the special characters itself. Braces are almost always used when passing scripts to Tcl commands, as in the following example that computes the factorial of five:

```
set result 1
set i 5
while {$i > 0} {
    set result [expr $result*$i]
    set i [expr $i-1]
}
```

The body of the `while` loop is enclosed in braces to defer substitutions. `While` passes the script back into Tcl for evaluation during each iteration of the loop and the subsitutions will be performed at that time. In this case it is important to defer the substitutions so that they are done afresh each time that `while` evaluates the loop body, rather than once-and-for-all while parsing the `while` command.

Braces nest, as in the following example:

```
proc power {base p} {
    set result 1
    while {$p > 0} {
        set result [expr $result*base]
        set p [expr $p-1]
    }
    return $result
}
```

In this case the third argument to `proc` contains two pairs of nested braces (the outermost braces are removed by the Tcl parser). The command substitution requested with "`[expr $p-1]`" will not be performed when the `proc` command is parsed, or even when the `while` command is parsed as part of executing the procedure's body, but only when `while` evaluates its second argument to execute the loop.

Note: *If a brace is backslashed then it does not count in finding the matching close brace for a word enclosed in braces. The backslash will not be removed when the word is parsed.*

Note: *The only form of substitution that occurs between braces is for backslash-newline. As discussed in Section 3.5, backslash-newline sequences are actually removed in a pre-processing step before the command is parsed.*

3.8 Comments

If the first non-blank character of a command is # then the # and all the characters following it up through the next newline are treated as a comment and discarded. Note that the hash-mark must occur in a position where Tcl is expecting the first character of a command. If a hash-mark occurs anywhere else then it is treated as an ordinary character that forms part of a command word:

```
# This is a comment
set a 100            # Not a comment
∅ wrong # args: should be "set varName ?newValue?"
set b 101;           # This is a comment
⇒ 101
```

The # on the second line is not treated as a comment character because it occurs in the middle of a command. As a result the first `set` command receives 6 arguments and generates an error. The last # is treated as a comment character, since it occurs just after the command was terminated with a semi-colon.

3.9 Normal and exceptional returns

A Tcl command can terminate in several different ways. A *normal return* is the most common case; it means that the command completed successfully and the return includes a string result. Tcl also supports *exceptional returns* from commands. The most frequent form of exceptional return is an error. When an error return occurs, it means that the command could not complete its intended function. The command is aborted and any commands that follow it in the script are skipped. An error return includes a string identifying what went wrong; the string is normally displayed by the application. For example, the following `set` command generates an error because it has too many arguments:

```
set state West Virginia
∅ wrong # args: should be "set varName ?newValue?"
```

Different commands generate errors under different conditions. For example, `expr` accepts any number of arguments but requires the arguments to have a particular syntax; it generates an error if, for example, parentheses aren't matched:

```
expr 3 * (20+4
∅ unmatched parentheses in expression "3 * (20+4"
```

The complete exceptional return mechanism for Tcl is discussed in Chapter 9. It supports a number of exceptional returns other than errors, provides additional information about errors besides the error message mentioned above, and allows errors to be "caught" so that effects of the error can be contained within a piece of Tcl code. For now, though, all you need to know is that commands normally return string results but they sometimes return errors that cause Tcl command interpretation to be aborted.

Note: You may also find the `errorInfo` variable useful. After an error Tcl sets `errorInfo` to hold a stack trace indicating exactly where the error occurred. You can print out this variable with the command "`set errorInfo`".

3.10 More on substitutions

The most common difficulty for new Tcl users is understanding when substitutions do and do not occur. A typical scenario is for a user to be surprised at the behavior of a script because a substitution didn't occur when the user expected it to happen, or a substitution occurred when it wasn't expected. However, I think that you'll find Tcl's substitution mechanism to be simple and predictable if you just remember two related rules:

1. Tcl parses a command and makes substitutions in a single pass from left to right. Each character is scanned exactly once.
2. At most a single layer of substitution occurs for each character; the result of one substitution is not scanned for further substitutions.

Tcl's substitutions are simpler and more regular than you may be used to if you've programmed with UNIX shells (particularly `csh`). When new users run into problems with Tcl substitutions it is often because they have assumed a more complex model than actually exists.

For example, consider the following command:

```
    set x [format {Earnings for July: $%.2f} $earnings]
⇒   Earnings for July: $1400.26
```

The characters between the brackets are scanned exactly once, during command substitution, and the value of the `earnings` variable is substituted at that time. It is *not* the case that Tcl first scans the whole `set` command to substitute variables, then makes another pass to perform command substitution; everything happens in a single scan. The result of the `format` command is passed verbatim to `set` as its second argument without any additional scanning (for example, the dollar-sign in `format`'s result does not trigger variable substitution).

One consequence of the substitution rules is that all the word boundaries within a command are immediately evident and are not affected by substitutions. For example, consider the following script:

```
set city "Los Angeles"
set bigCity $city
```

The second `set` command is guaranteed to have exactly three words regardless of the value of variable `city`. In this case `city` contains a space character but the space is *not* treated as a word separator.

In some situations the single-layer-of-substitutions rule can be a hindrance rather than a help. For example, the following script is an erroneous attempt to delete all files with names ending in "`.o`":

```
    exec rm [glob *.o]
∅   rm: a.o b.o c.o nonexistent
```

The `glob` command returns a list of all file names that match the pattern "`*.o`", such as "`a.o b.o c.o`". The `exec` command then attempts to invoke the `rm` program to delete all of these files. However, the entire list of files is passed to `rm` as a single argument; `rm` reports an error because it cannot find a file named "`a.o b.o c.o`". For `rm` to work correctly the result of `glob` must be split up into multiple words.

Fortunately, it is easy to add additional layers of parsing if you want them. Remember that Tcl commands are evaluated in two phases: parsing and execution. The substitution rules apply only to the parsing phase. Once Tcl passes the words of a command to a command procedure for execution, the command procedure can do anything it likes with them. Some commands will reparse their words, for example by passing them back to the Tcl interpreter again. `Eval` is an example of such a command, and it can be used to solve the problems with `rm` above:

```
eval exec rm [glob *.o]
```

`Eval` concatenates all of its arguments with spaces in-between and then evaluates the result as a Tcl script, at which point another round of parsing and evaluation occurs. In this example `eval` receives three arguments: "`exec`", "`rm`", and "`a.o b.o c.o`". It concatenates them to form the string "`exec rm a.o b.o c.o`". When this string is parsed as a Tcl script it yields five words; each of the file names is passed to `exec` and then to the `rm` program as a separate argument, so the files are all removed successfully. See Section 7.5 for more details on this.

One final note. It is possible to use substitutions in very complex ways but I urge you not to do so. Substitutions work best when used in very simple ways such as "`set a $b`". If you use a great many substitutions in a single command, and particularly if you use lots of backslashes, your code is unlikely to be unreadable and it's also unlikely to work reliably. In situations like these I suggest breaking up the offending command into several commands that build up the arguments in simple stages. Tcl provides several commands, such as `format` and `list`, that should make this easy to do.

Chapter 4
Variables

Tcl supports two kinds of variables: simple variables and associative arrays. This chapter describes the basic Tcl commands for manipulating variables and arrays, and it also provides a more complete description of variable substitution. See Table 4.1 for a summary of the commands discussed in this chapter.

4.1 Simple variables and the set command

A simple Tcl variable consists of two things: a name and a value. Both the name and the value may be arbitrary strings of characters. For example, it is possible to have a variable named "`xyz !# 22`" or "`March earnings: $100,472`". In practice variable names usually start with a letter and consist of a combination of letters, digits, and underscores, since that makes it easier to use variable substitution.

Variables may be created, read, and modified with the `set` command, which takes either one or two arguments. The first argument is the name of a variable and the second, if present, is a new value for the variable:

```
   set a {Eggs: $2.18/dozen}
⇒  Eggs: $2.18/dozen
   set a
⇒  Eggs: $2.18/dozen
   set a 44
⇒  44
```

`append` *`varName value`* ?*`value`* ...?	Appends each of the *`value`* arguments to variable *`varName`*, in order. If *`varName`* doesn't exist then it is created with an empty value before appending. The return value is the new value of *`varName`*.
`incr` *`varName`* ?*`increment`*?	Adds *`increment`* to the value of variable *`varName`*. *`Increment`* and the old value of *`varName`* must both be integer strings (decimal, hexadecimal, or octal). If *`increment`* is omitted then it defaults to `1`. The new value is stored in *`varName`* as a decimal string and returned as the result of the command.
`set varName` ?*`value`*?	If *`value`* is specified, sets the value of variable *`varName`* to *`value`*. In any case the command returns the (new) value of the variable.
`unset` *`varName`* ?*`varName varName`* ...?	Deletes the variables given by the *`varName`* arguments. Returns an empty string.

Table 4.1. A summary of the basic commands for manipulating variables. Optional arguments are indicated by enclosing them in question-marks.

The first command above creates a new variable `a` if it doesn't already exist and sets its value to the character sequence "`Eggs: $2.18/dozen`". The result of the command is the new value of the variable. The second `set` command has only one argument: `a`. In this form it simply returns the current value of the variable. The third `set` command changes the value of `a` to `44` and returns that new value.

Although the final value of `a` looks like a decimal integer, it is stored as a character string. Tcl variables can be used to represent many things, such as integers, floating-point numbers, names, lists, and Tcl scripts, but they are always stored as strings. This use of a single representation for all values allows different values to be manipulated in the same way and communicated easily.

Tcl variables are created automatically when they are assigned values. Variables don't have types so there is no need for declarations.

4.2 Arrays

In addition to simple variables Tcl also provides *arrays*. An array is a collection of *elements*, each of which is a variable with its own name and value. The name of an array element has two parts: the name of the array and the name of the element within that array. Both array names and element names may be arbitrary strings. For this reason Tcl arrays are sometimes called *associative arrays* to distinguish them from arrays in other languages where the element names must be integers.

Array elements are referenced using notation like `earnings(January)` where the array name (`earnings` in this case) is followed by the element name in parentheses (`January` in this case). Arrays may be used anywhere that simple variables may be used, such as in the `set` command:

```
   set earnings(January) 87966
⇒  87966
   set earnings(February) 95400
⇒  95400
   set earnings(January)
⇒  87966
```

The first command creates an array named `earnings`, if it doesn't already exist. Then it creates an element `January` within the array, if it doesn't already exist, and assigns it the value `87966`. The second command assigns a value to the `February` element of the array, and the third command returns the value of the `January` element.

4.3 Variable substitution

Chapter 3 introduced the use of `$`-notation for substituting variable values into Tcl commands. This section describes the mechanism in more detail.

Variable substitution is triggered by the presence of an unquoted `$` character in a Tcl command. The characters following the `$` are treated as a variable name, and the `$` and name are replaced in the word by the value of the variable. Tcl provides three forms of variable substitution. So far you have seen only the simplest form, which is used like this:

```
expr $a+2
```

In this form the `$` is followed by a variable name consisting of letters, digits, and underscores. The first character that is not a letter or digit or underscore ("+" in the example) terminates the name.

The second form of variable substitution allows array elements to be substituted. This form is like the first one except that the variable name is followed immediately by an element name enclosed in parentheses. Variable, command, and backslash substitutions are performed on the element name in the same way as a command word in double-quotes, and spaces in the element name are treated as part of the name rather than as word separators. For example, consider the following script:

```
set yearTotal 0
foreach month {Jan Feb Mar Apr May Jun Jul Aug Sep \
        Oct Nov Dec} {
    set yearTotal [expr $yearTotal+$earnings($month)]
}
```

In the `expr` command "`$earnings($month)`" is replaced with the value of an element of the array `earnings`. The element's name is given by the value of the `month` variable, which varies from iteration to iteration.

The last form of substitution is used for simple variables in places where the variable name is followed by a letter or number or underscore. For example, suppose that you wish to pass a value like "`1.5m`" to a command as an argument but the number is in a variable `size` (in Tk you might do this to specify a size in millimeters). If you try to substitute the variable value with a form like "`$sizem`" then Tcl will treat the `m` as part of the variable name. To get around this problem you can enclose the variable name in braces as in the following command:

```
.canvas configure -width ${size}m
```

You can also use braces to specify variable names containing characters other than letters or numbers or underscores.

Note: Braces can only be used to delimit simple variables. However, they shouldn't be needed for arrays since the parentheses already indicate where the variable name ends.

Tcl's variable substitution mechanism is only intended to handle the most common situations; there exist scenarios where none of the above forms of substitution achieves the desired effect. More complicated situations can be handled with a sequence of commands. For example, the `format` command can be used to generate a variable name of almost any imaginable form, `set` can be used to read or write the variable with that name, and command substitution can be used to substitute the value of the variable into other commands.

4.4 Removing variables: unset

The `unset` command destroys variables. It takes any number of arguments, each of which is a variable name, and removes all of the variables. Future attempts to read the variables will result in errors just as if the variables had never been set in the first place. The arguments to `unset` may be either simple variables, elements of arrays, or whole arrays, as in the following example:

```
unset a earnings(January) b
```

In this case the variables `a` and `b` are removed entirely and the `January` element of the `earnings` array is removed. The `earnings` array continues to exist after the `unset` command. If `a` or `b` is an array then all of the elements of that array are removed along with the array itself.

4.5 Multi-dimensional arrays

Tcl only implements one-dimensional arrays, but multi-dimensional arrays can be simulated by concatenating multiple indices into a single element name. The program below simulates a two-dimensional array indexed with integers:

```
set matrix(1,1) 140
set matrix(1,2) 218
set matrix(1,3) 84
set i 1
set j 2
set cell $matrix($i,$j)
⇒ 218
```

`Matrix` is an array with three elements whose names are "`1,1`" and "`1,2`" and "`1,3`". However, the array behaves just as if it were a two-dimensional array; in particular, variable substitution occurs while scanning the element name in the `expr` command, so that the values of `i` and `j` get combined into an appropriate element name.

4.6 The incr and append commands

`Incr` and `append` provide simple ways to change the value of a variable. `Incr` takes two arguments, which are the name of a variable and an integer; it adds the integer to the variable's value, stores the result back into the variable as a decimal string, and returns the variable's new value as result:

```
set x 43
incr x 12
⇒ 55
```

The number can have either a positive or negative value. It can also be omitted, in which case it defaults to `1`:

```
set x 43
incr x
⇒ 44
```

Both the variable's original value and the increment must be integer strings, either in decimal, octal (indicated by a leading `0`), or hexadecimal (indicated by a leading `0x`).

The `append` command adds text to the end of a variable. It takes two arguments, which are the name of the variable and the new text to add. It appends the new text to the variable and returns the variable's new value. The following example uses `append` to compute a table of squares:

```
set msg ""
foreach i {1 2 3 4 5} {
    append msg "$i squared is [expr $i*$i]\n"
}
set msg
⇒ 1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
```

Neither `incr` nor `append` adds any new functionality to Tcl, since the effects of both of these commands can be achieved in other ways. However, they provide simple ways to do common operations. In addition, `append` is implemented in a fashion that avoids character copying. If you need to construct a very large string incrementally from pieces it will be much more efficient to use a command like

```
append x $piece
```

instead of a command like

```
set x "$x$piece"
```

4.7 Preview of other variable facilities

Tcl provides a number of other commands for manipulating variables. These commands will be introduced in full after you've learned more about the Tcl language, but this section contains a short preview of some of the facilities.

The `trace` command can be used to monitor a variable so that a Tcl script gets invoked whenever the variable is set or read or unset. Variable tracing is sometimes useful during debugging, and it allows you to create read-only variables. You can also use traces for *propagation* so that, for example, a database or screen display gets updated whenever a variable changes value. Variable tracing is discussed in Section 13.4.

The `array` command can be used to find out the names of all the elements in an array and to step through them one at a time (see Section 13.1). It's possible to find out what variables exist using the `info` command (see Section 13.2).

The `global` and `upvar` commands can be used by a procedure to access variables other than its own local variables. These commands are discussed in Chapter 8.

Chapter 5
Expressions

Expressions combine values (or *operands*) with *operators* to produce new values. For example, the expression "4+2" contains two operands, "4" and "2", and one operator, "+"; it evaluates to 6. Many Tcl commands expect one or more of their arguments to be expressions. The simplest such command is `expr`, which just evaluates its arguments as an expression and returns the result as a string:

```
expr (8+4) * 6.2
⇒ 74.4
```

Another example is `if`, which evaluates its first argument as an expression and uses the result to determine whether or not to evaluate its second argument as a Tcl script:

```
if $x<2 then {set x 2}
```

This chapter uses the `expr` command for all of its examples, but the same syntax, substitution, and evaluation rules apply to all other uses of expressions too. See Table 5.1 for a summary of the `expr` command.

5.1 Numeric operands

Expression operands are normally integers or real numbers. Integers are usually specified in decimal, but if the first character is 0 (zero) then the number is read in octal (base 8) and if the first two characters are `0x` then the number is read in hexadecimal (base 16). For example, `335` is a decimal number, `0517` is an octal number with the same value, and `0x14f` is a hexadecimal number with the same value. `092` is not a valid integer: the leading `0` causes the number to be read in octal but `9` is not a valid octal digit. Real operands

`expr arg ?arg arg ...?`	Concatenates all the *arg* values together (with spaces in between), evaluates the result as an expression, and returns a string corresponding to the expression's value.

Table 5.1. A summary of the `expr` command.

may be specified using most of the forms defined for ANSI C, including the following examples:

```
2.1
7.91e+16
6E4
3.
```

Note: These same forms are allowable not just in expressions but anywhere in Tcl that an integer or real value is required.

Expression operands can also be non-numeric strings. String operands are discussed in Section 5.5.

5.2 Operators and precedence

Table 5.2 lists all of the operators supported in Tcl expressions; they are similar to the operators for expressions in ANSI C. Horizontal lines separate groups of operators with the same precedence, and operators with higher precedence appear in the table above operators with lower precedence. For example, `4*2<7` evaluates to `0` because the `*` operator has higher precedence than <. Except in the simplest and most obvious cases you should use parentheses to indicate the way operators should be grouped; this will prevent errors by you or by others who modify your programs.

Operators with the same precedence group from left to right. For example, `10-4-3` is the same as `(10-4)-3`; it evaluates to `3`.

5.2.1 Arithmetic operators

Tcl expressions support the arithmetic operators `+`, `-`, `*`, `/`, and `%`. The `-` operator may be used either as a binary operator for subtraction, as in `4-2`, or as a unary operator for negation, as in `-(6*$i)`. The `/` operator truncates its result to an integer value if both operands are integers. `%` is the modulus operator: its result is the remainder when its first operand is divided by the second. Both of the operands for `%` must be integers.

Note: The / and % operators have a more consistent behavior in Tcl than in ANSI C. In Tcl the remainder is always positive and has an absolute value less than the absolute value of the

Syntax	Result	Operand Types
`-a`	Negative of *a*	int, float
`!a`	Logical NOT: 1 if *a* is zero, 0 otherwise	int, float
`~a`	Bit-wise complement of *a*	int
`a*b`	Multiply *a* and *b*	int, float
`a/b`	Divide *a* by *b*	int, float
`a%b`	Remainder after dividing *a* by *b*	int
`a+b`	Add *a* and *b*	int, float
`a-b`	Subtract *b* from *a*	int, float
`a<<b`	Left-shift *a* by *b* bits	int
`a>>b`	Arithmetic right-shift *a* by *b* bits	int
`a<b`	1 if *a* is less than *b*, 0 otherwise	int, float, string
`a>b`	1 if *a* is greater than *b*, 0 otherwise	int, float, string
`a<=b`	1 if *a* is less than or equal to *b*, 0 otherwise	int, float, string
`a>=b`	1 if *a* is greater than or equal to *b*, 0 otherwise	int, float, string
`a==b`	1 if *a* is equal to *b*, 0 otherwise	int, float, string
`a!=b`	1 if *a* is not equal to *b*, 0 otherwise	int, float, string
`a&b`	Bit-wise AND of *a* and *b*	int
`a^b`	Bit-wise exclusive OR of *a* and *b*	int
`a|b`	Bit-wise OR of *a* and *b*	int
`a&&b`	Logical AND: 1 if both *a* and *b* are non-zero, 0 otherwise	int, float
`a||b`	Logical OR: 1 if either *a* is non-zero or *b* is non-zero, 0 otherwise	int, float
`a?b:c`	Choice: if *a* is non-zero then *b*, else *c*	*a*: int, float

Table 5.2. Summary of the operators allowed in Tcl expressions. These operators have the same behavior as in ANSI C except that some of the operators allow string operands. Groups of operands between horizontal lines have the same precedence; higher groups have higher precedence.

divisor. ANSI C guarantees only the second property: In both ANSI C and Tcl the quotient will always have the property that `(x/y)*y + x%y` *is* `x`*., for all* `x` *and* `y`*.*

5.2.2 Relational operators

The operators < (less than), <= (less than or equal), >=(greater than or equal), > (greater than), == (equal), and `!=` (not equal) are used for comparing two values. Each operator produces a result of `1` (true) if its operands meet the condition and `0` (false) if they don't.

5.2.3 Logical operators

The logical operators `&&`, `||`, and `!` are typically used for combining the results of relational operators, as in the expression

```
($x > 4) && ($x < 10)
```

Each operator produces a `0` or `1` result. `&&` (logical "and") produces a `1` result if both its operands are non-zero, `||` (logical "or") produces a `1` result if either of its operands is non-zero, and `!` ("not") produces a `1` result if its single operand is zero.

In Tcl, as in ANSI C, a zero value is treated as false and anything other than zero is treated as true. Whenever Tcl generates a true/false value it uses `1` for true and `0` for false.

5.2.4 Bitwise operators

Tcl provides six operators that manipulate the individual bits of integers: `&`, `|`, `^`, <<, >>, and `~`. These operators require their operands to be integers. The `&`, `|`, and `^` operators perform bitwise and, or, and exclusive or: each bit of the result is generated by applying the given operation to the corresponding bits of the left and right operands. Note that `&` and `|` do not always produce the same result as `&&` and `||`:

```
expr 8&&2
⇒ 1
expr 8&2
⇒ 0
```

The operators << and >> use the right operand as a shift count and produce a result consisting of the left operand shifted left or right by that number of bits. During left shifts zeros are shifted into the low-order bits. Right shifting is always "arithmetic right shift", meaning that it shifts in zeroes for positive numbers and ones for negative numbers. This behavior is different from right-shifting in ANSI C, which is machine-dependent.

The ~ operand ("ones complement") takes only a single operand and produces a result whose bits are the opposite of those in the operand: zeroes replace ones and vice versa.

5.2.5 Choice operator

The ternary operator `?:` may be used to select one of two results:

```
expr {($a < $b) ? $a : $b}
```

This expression returns the smaller of `$a` and `$b`. The choice operator checks the value of its first operand for truth or falsehood. If it is true (non-zero) then the argument following the `?` is evaluated and becomes the result; if the first operand is false (zero) then the third operand is evaluated and becomes the result. Only one of the second and third arguments is evaluated.

5.3 Math functions

Tcl expressions support a number of mathematical functions such as `sin` and `exp`. Math functions are invoked using standard functional notation:

```
expr 2*sin($x)
expr hypot($x, $y) + $z
```

The arguments to math functions may be arbitrary expressions, and multiple arguments are separated by commas. See Table 5.3 for a list of all the built-in functions.

5.4 Substitutions

Substitutions can occur in two ways for expression operands. The first way is through the normal Tcl parser mechanisms, as in the following command:

```
expr 2*sin($x)
```

In this case the Tcl parser substitutes the value of variable `x` before executing the command, so the first argument to `expr` will have a value such as "`2*sin(0.8)`". The second way is through the expression evaluator, which performs an additional round of variable and command substitution on the expression while evaluating it. For example, consider the command:

```
expr {2*sin($x)}
```

In this case the braces prevent the Tcl parser from substituting the value of `x`, so the argument to `expr` is "`2*sin($x)`". When the expression evaluator encounters the dollar-sign it performs variable substitution itself, using the value of variable `x` as the argument to `sin`.

Having two layers of substitution doesn't usually make any difference for the `expr` command, but it is vitally important for other commands like `while` that evaluate an expression repeately and expect to get different results each time. For example, consider the following script that raises a base to a power:

```
set result 1
while {$power>0} {
    set result [expr $result*$base]
    incr power -1
}
```

Function	Result
`abs(`*x*`)`	Absolute value of *x*.
`acos(`*x*`)`	Arc cosine of *x*, in the range 0 to π.
`asin(`*x*`)`	Arc sine of *x*, in the range $-\pi/2$ to $\pi/2$.
`atan(`*x*`)`	Arc tangent of *x*, in the range $-\pi/2$ to $\pi/2$.
`atan2(`*x*`,`*y*`)`	Arc tangent of *x*/*y*, in the range $-\pi/2$ to $\pi/2$.
`ceil(`*x*`)`	Smallest integer not less than *x*.
`cos(`*x*`)`	Cosine of *x* (*x* in radians).
`cosh(`*x*`)`	Hyperbolic cosine of *x*.
`double(`*i*`)`	Real value equal to integer *i*.
`exp(`*x*`)`	*e* raised to the power *x*.
`floor(`*x*`)`	Largest integer not greater than *x*.
`fmod(`*x*`,`*y*`)`	Floating-point remainder of *x* divided by *y*.
`hypot(`*x*`,`*y*`)`	Square root of $(x^2 + y^2)$.
`int(`*x*`)`	Integer value produced by truncating *x*.
`log(`*x*`)`	Natural logarithm of *x*.
`log10(`*x*`)`	Base 10 logarithm of *x*.
`pow(`*x*`,`*y*`)`	*x* raised to the power *y*.
`round(`*x*`)`	Integer value produced by rounding *x*.
`sin(`*x*`)`	Sine of *x* (*x* in radians).
`sinh(`*x*`)`	Hyperbolic sine of *x*.
`sqrt(`*x*`)`	Square root of *x*.
`tan(`*x*`)`	Tangent of *x* (*x* in radians).
`tanh(`*x*`)`	Hyperbolic tangent of *x*.

Table 5.3. The mathematical functions supported in Tcl expressions. In most cases the functions have the same behavior as the ANSI C library procedures with the same names.

The expression "`$power>0`" gets evaluated by `while` at the beginning of each iteration to decide whether or not to terminate the loop. It is essential that the expression evaluator use a new value of `power` each time. If the variable substitution were performed while parsing the `while` command, for example "`while $power>0 ...`", then `while`'s argument would be a constant expression such as "`5>0`"; either the loop would never execute or it would execute forever.

Note: When the expression evaluator performs variable or command substitution the value substituted must be an integer or real number (or a string, as described below). It cannot be an arbitrary expression.

5.5 String manipulation

Unlike expressions in ANSI C, Tcl expressions allow som simple string operations, as in the following command:

```
if {$x == "New York"} {
...
}
```

In this example the expression evaluator compares the value of variable `x` to the string "`New York`" using string comparison; the body of the `if` will be executed if they are identical. In order to specify a string operand you must either enclose it in quotes or braces or use variable or command substitution. It is important that the expression in the above example is enclosed in braces so that the expression evaluator substitutes the value of `x`; if the braces are left out then the argument to `if` will be a string like

```
Los Angeles == "New York"
```

The expression parser will not be able to parse "`Los`" (it isn't a number, it doesn't make sense as a function name, and it can't be interpreted as a string because it isn't delimited) so a syntax error will occur.

If a string is enclosed in quotes then the expression evaluator performs command, variable, and backslash substitution on the characters between the quotes. If a string is enclosed in braces then no substitutions are performed. Braces nest for strings in expressions in the same way that they nest for words of a command.

The only operators that allow string operands are <, >, <=, >=, ==, and !=. For all other operators the operands must be numeric. For operators like < the strings are compared lexicographically using the system's `strcmp` library function; the sorting order may vary from system to system.

5.6 Types and conversions

Tcl evaluates expressions numerically whenever possible. String operations are only performed for the relational operators and only if one or both of the operands doesn't make sense as a number. Most operators permit either integer or real operands but a few, such as << and `&`, allow only integers.

If the operands for an operator have different types then Tcl automatically converts one of them to the type of the other. If one operand is an integer and the other is a real then the integer operand is converted to real. If one operand is a non-numeric string and the other is an integer or real then the integer or real operand is converted to a string. The result of an operation always has the same type as the operands except for relational operators like <, which always produce 0/1 integer results. You can use the math function `double` to explicitly promote an integer to a real, and `int` and `round` to convert a real value back to integer by truncation or rounding.

5.7 Precision

During expression evaluation Tcl represents integers internally with the C type `int`, which provides at least 32 bits of precision on most machines. Real numbers are represented with with the C type `double`, which is usually represented with 64-bit values (about 15 decimal digits of precision) using the IEEE Floating Point Standard.

Numbers are kept in internal form throughout the evaluation of an expression and are only converted back to strings when necessary, such as when `expr` returns its result. Integers are converted to signed decimal strings without any loss of precision. When a real value is converted to a string only six significant digits are retained by default:

```
expr 1.11111111 + 1.11111111
⇒ 2.22222
```

If you would like more significant digits to be retained when real values are converted to strings you can set the `tcl_precision` global variable with the desired number of significant digits:

```
set tcl_precision 12
expr 1.11111111 + 1.11111111
⇒ 2.22222222
```

The `tcl_precision` variable is used not just for the `expr` command but anywhere that a Tcl application converts a real number to a sting.

Note: If you set `tcl_precision` to 17 on a machine that uses IEEE floating point, you will guarantee that string conversions do not lose information: if an expression result is converted to a string and then later used in a different expression, the internal form after conversion back from the string will be identical to the internal form before converting to the string.

Chapter 6
Lists

Lists are used in Tcl to deal with collections of things, such as all the users in a group or all the files in a directory or all the options for a widget. Lists allow you to collect together any number of values in one place, pass around the collection as a single entity, and later get the component values back again. A list is an ordered collection of *elements* where each element can have any string value, such as a number, a person's name, the name of a window, or a word of a Tcl command. Lists are represented as strings with a particular structure; this means that you can store lists in variables, type them to commands, and nest them as elements of other lists.

This chapter describes the structure of lists and presents a dozen basic commands for manipulating lists. The commands perform operations like creating lists, inserting and extracting elements, and searching for particular elements (see Table 6.1 for a summary). There are other Tcl commands besides those described in this chapter that take lists as arguments or return them as results; these other commands will be described in later chapters.

6.1 Basic list structure and the lindex command

In its simplest form a list is a string containing any number of elements separated by spaces or tabs. For example, the string

```
John Anne Mary Jim
```

`concat` *`list`* ?*`list`* ...?	Joins multiple lists into a single list (each element of each *`list`* becomes an element of the result list) and returns the new list.
`join` *`list`* ?*`joinString`*?	Concatenates list elements together with *`joinString`* as separator and returns the result.
`lappend` *`varName value`* ?*`value`* ...?	Appends each *`value`* to variable *`varName`* as a list element and returns the new value of the variable. Creates the variable if it doesn't already exist.
`lindex` *`list index`*	Returns the *`index`*'th element from *`list`*.
`linsert` *`list index value`* ?*`value`* ...?	Returns a new list formed by inserting all of the *`value`* arguments as list elements before *`index`*'th element of *`list`*.
`list` *`value`* ?*`value`* ...?	Returns a list whose elements are the *`value`* arguments.
`llength` *`list`*	Returns the number of elements in *`list`*.
`lrange` *`list first last`*	Returns a list consisting of elements *`first`* through *`last`* of *`list`*. If *`last`* is `end` then it selects all elements up to the end of the list.
`lreplace` *`list first last`* ?*`value value`* ...?	Returns a new list formed by replacing elements *`first`* through *`last`* of *`list`* with zero or more new elements, each formed from one *`value`* argument.
`lsearch ?-exact? ?-glob? ?-regexp?` *`list pattern`*	Returns the index of the first element in *`list`* that matches *`pattern`* or `-1` if none. The optional switch selects a pattern-matching technique (default: `-glob`).
`lsort ?-ascii? ?-integer? ?-real? ?-command` *`command`*`? \` `?-increasing? ?-decreasing?` *`list`*	Returns a new list formed by sorting the elements of *`list`*. The switches determine the comparison function and sorted order (default: `-ascii -increasing`).
`split` *`string`* ?*`splitChars`*?	Returns a list formed by splitting *`string`* at instances of *`splitChars`* and turning the characters between these instances into list elements.

Table 6.1. A summary of the list-related commands in Tcl.

is a list with four elements. There can be any number of elements in a list, and each element can be an arbitrary string. In the simple form above, elements cannot contain spaces, but there is additional list syntax that allows spaces within elements (see below).

The `lindex` command extracts an element from a list:

```
   lindex {John Anne Mary Jim} 1
⇒ Anne
```

`Lindex` takes two arguments, a list and an index, and returns the selected element of the list. An index of `0` corresponds to the first element of the list, `1` corresponds to the second element, and so on. If the index is outside the range of the list then an empty string is returned.

When a list is entered in a Tcl command the list is usually enclosed in braces, as in the above example. The braces are not part of the list; they are needed on the command line to pass the entire list to the command as a single word. When lists are stored in variables or printed out, there are no braces around them:

```
   set x {John Anne Mary Jim}
⇒ John Anne Mary Jim
```

Curly braces and backslashes within list elements are handled by the list commands in the same way that the Tcl command parser treats them in words. This means that you can enclose a list element in braces if it contains spaces, and you can use backslash substitution to get special characters such as braces into list elements. Braces are often used to nest lists within lists, as in the following example:

```
   lindex {a b {c d e} f} 2
⇒ c d e
```

In this case element 2 of the list is itself a list with three elements. There is no limit on how deeply lists may be nested.

6.2 Creating lists: concat, list, and llength

Tcl provides two commands that combine strings together to produce lists: `concat` and `list`. Each of these commands accepts an arbitrary number of arguments, and each produces a list as a result. However, they differ in the way they combine their arguments. The `concat` command takes one or more lists as arguments and joins all of the elements of the argument lists together into a single large list:

```
   concat {a b c} {d e} f {g h i}
⇒ a b c d e f g h i
```

`Concat` expects its arguments to have proper list structure; if the arguments are not well-formed lists then the result may not be a well-formed list either. In fact, all that `concat` does is to concatenate its argument strings into one large string with space characters between the arguments. The same effect as `concat` can be achieved using double-quotes:

```
set x {a b c}
set y {d e}
set z [concat $x $y]
⇒ a b c d e
set z "$x $y"
⇒ a b c d e
```

The `list` command joins its arguments together so that each argument becomes a distinct element of the resulting list:

```
list {a b c} {d e} f {g h i}
⇒ {a b c} {d e} f {g h i}
```

In this case, the result list contains only four elements. The `list` command will always produce a list with proper structure, regardless of the structure of its arguments (it adds braces or backslashes as needed), and the `lindex` command can always be used to extract the original elements of a list created with `list`. The arguments to `list` need not themselves be well-formed lists.

The `llength` command returns the number of elements in a list:

```
llength {{a b c} {d e} f {g h i}}
⇒ 4
llength a
⇒ 1
llength {}
⇒ 0
```

As you can see from the examples, a simple string like “a” is a proper list with one element and an empty string is a proper list with zero elements.

6.3 Modifying lists: linsert, lreplace, lrange, and lappend

The `linsert` command forms a new list by adding one or more elements to an existing list:

```
set x {a b {c d} e}
⇒ a b {c d} e
linsert $x 2 X Y Z
⇒ a b X Y Z {c d} e
linsert $x 0 {X Y} Z
⇒ {X Y} Z a b {c d} e
```

`Linsert` takes three or more arguments. The first is a list, the second is the index of an element within that list, and the third and additional arguments are new elements to insert into the list. The return value from `linsert` is a list formed by inserting the new elements just before the element indicated by the index. If the index is zero then the new elements go at the beginning of the list; if it is one then the new elements go after the first element in the old list; and so on. If the index is greater than or equal to the number of elements in the original list then the new elements are inserted at the end of the list.

The `lreplace` command deletes elements from a list and optionally adds new elements in their place. It takes three or more arguments. The first argument is a list and the second and third arguments give the indices of the first and last elements to be deleted. If only three arguments are specified then the result is a new list produced by deleting the given range of elements from the original list:

```
   lreplace {a b {c d} e} 3 3
⇒  a b {c d}
```

If additional arguments are specified to `lreplace` as in the example below, then they are inserted into the list in place of the elements that were deleted.

```
   lreplace {a b {c d} e} 1 2 {W X} Y Z
⇒  a {W X} Y Z e
```

The `lrange` command extracts a range of elements from a list. It takes as arguments a list and two indices and it returns a new list consisting of the range of elements that lie between the two indices (inclusive):

```
   set x {a b {c d} e}
⇒  a b {c d} e
   lrange $x 1 3
⇒  b {c d} e
   lrange $x 0 1
⇒  a b
```

The `lappend` command provides an efficient way to append new elements to a list stored in a variable. It takes as arguments the name of a variable and any number of additional arguments. Each of the additional arguments is appended to the variable's value as a new list element and `lappend` returns the variable's new value:

```
   set x {a b {c d} e}
⇒  a b {c d} e
   lappend x XX {YY ZZ}
⇒  a b {c d} e XX {YY ZZ}
   set x
⇒  a b {c d} e XX {YY ZZ}
```

`Lappend` is similar to `append` except that it enforces proper list structure. As with append, it isn't strictly necessary. For example, the command

```
lappend x $a $b $c
```

could be written instead as

```
set x "$x [list $a $b $c]"
```

However, as with `append`, `lappend` is implemented in a way that avoids string copies. For large lists this can make a big difference in performance.

6.4 Searching lists: lsearch

The `lsearch` command searches a list for an element with a particular value. It takes two arguments, the first of which is a list and second of which is a pattern:

```
set x {John Anne Mary Jim}
lsearch $x Mary
⇒ 2
lsearch $x Phil
⇒ -1
```

`Lsearch` returns the index of the first element in the list that matches the pattern, or `-1` if there was no matching element.

One of three different pattern matching techniques can be selected by specifying one of the switches `-exact`, `-glob`, and `-regexp` before the list argument:

```
lsearch -glob $x A*
⇒ 1
```

The `-glob` switch causes matching to occur with the rules of the `string match` command described in Section 10.1. A `-regexp` switch causes matching to occur with regular expression rules as described in Section 10.2, and `-exact` insists on an exact match only. If no switch is specified then `-glob` is assumed by default.

6.5 Sorting lists: lsort

The `lsort` command takes a list as argument and returns a new list with the same elements, but sorted in increasing lexicographic order:

```
lsort {John Anne Mary Jim}
⇒ Anne Jim John Mary
```

You can precede the list with any of several switches to control the sort. For example, `-decreasing` specifies that the result should have the “largest” element first and `-integer` specifies that the elements should be treated as integers and sorted according to integer value:

```
lsort -decreasing {John Anne Mary Jim}
⇒ Mary John Jim Anne
lsort {10 1 2}
⇒ 1 10 2
```

```
lsort -integer {10 1 2}
⇒ 1 2 10
```

You can use the `-command` option to specify your own sorting function (see the reference documentation for details).

6.6 Converting between strings and lists: split and join

The `split` command breaks up a string into component pieces so that you can process the pieces independently. It creates a list whose elements are the pieces, so that you can use any of the list commands to process the pieces. For example, suppose a variable contains a UNIX file name with components separated by slashes, and you want to convert it to a list with one element for each component:

```
set x a/b/c
set y /usr/include/sys/types.h
split $x /
⇒ a b c
split $y /
⇒ {} usr include sys types.h
```

The first argument to `split` is the string to be split up and the second argument contains one or more *split characters*. `Split` locates all instances of any of the split characters in the string. It then creates a list whose elements consist of the substrings between the split characters. The ends of the string are also treated as split characters. If there are consecutive split characters or if the string starts or ends with a split character as in the second example, then empty elements are generated in the result list. The split characters themselves are discarded. Several split characters can be specified, as in the following example:

```
split xbaybz ab
⇒ x {} y z
```

If an empty string is specified for the split characters then each character of the string is made into a separate list element:

```
split {a b c} {}
⇒ a { } b { } c
```

The `join` command is approximately the inverse of `split`. It concatenates list elements together with a given separator string between them:

```
join {{} usr include sys types.h} /
⇒ /usr/include/sys/types.h
set x {24 112 5}
expr [join $x +]
⇒ 141
```

`Join` takes two arguments: a list and a separator string. It extracts all of the elements from the list and concatenates them together with the separator string between each pair of elements. The separator string can contain any number of characters, including zero. In the first example above a file name is generated by joining the list elements with "`/`". In the second example a Tcl expression is generated by joining the list elements with "+".

One of the most common uses for `split` and `join` is for dealing with file names as shown above. Another common use is for splitting up text into lines by using newline as the split character.

6.7 Lists and commands

There is a very important relationship between lists and commands in Tcl. Any proper list is also a well-formed Tcl command. If a list is evaluated as a Tcl script then it will consist of a single command whose words are the list elements. In other words, the Tcl parser will perform no substitutions whatsoever: it will simply extract the list elements with each element becoming one word of the command. This property is very important because it allows you to generate Tcl commands that are guaranteed to parse in a particular fashion even if some of the command's words contain special characters like spaces or `$`.

For example, suppose you are creating a button widget in Tk, and when the user clicks on the widget you would like to reset a variable to a particular value. You might create such a widget with a command like this:

```
button .b -text "Reset" -command {set x 0}
```

The Tcl script "`set x 0`" will be evaluated whenever the user clicks on the button. Now suppose that the value to be stored in the variable is not constant, but instead is computed just before the `button` command and must be taken from a variable `initValue`. Furthermore, suppose that `initValue` could contain any string whatsoever. You might rewrite the command as

```
button .b -text "Reset" -command {set x $initValue}
```

The script "`set x $initValue`" will be evaluated when the user clicks on the button. However, this will use the value of `initValue` at the time the user clicks on the button, which may not be the same as the value when the button was created. For example, the same variable might be used to create several buttons, each with a different intended reset value.

To solve this problem you must generate a Tcl command that contains the *value* of the `initValue` variable, not its name, and use this as part of the `-command` option for the `button` command. Unfortunately, a simple approach like

```
button .b -text "Reset" -command "set x $initValue"
```

will not work in general. If the value of `initValue` is something simple like `47` then this will work fine: the resulting command will be "`set x 47`", which will produce the desired result. However, what if `initValue` contains "`New York`"? In this case the resulting command will be "`set x New York`", which has four words; `set` will generate an error because there are too many arguments. Even worse, what if `initValue` contains special characters like "`$`" or "`[`"? These characters could cause unwanted substitutions to occur when the command is evaluated.

The only solution that is guaranteed to work for any value of `initValue` is to use list commands to generate the command, as in the following example:

```
button .b -text "Reset" -command [list set x $initValue]
```

The result of the `list` command is a Tcl command whose first word will be `set`, whose second word will be `x`, and whose third word will be the value of `initValue`. The command will always produce the desired result: whatever value is stored in `initValue` at the time `button` is invoked will be stored in `x` when the widget is invoked. For example, suppose that the value of `initValue` is "`New York`". The command generated by `list` will be "`set x {New York}`", which will parse and execute correctly. Any of the Tcl special characters will also be handled correctly by `list`:

```
   set initValue {Earnings: $1410.13}
   list set x $initValue
⇒ set x {Earnings: $1410.13}
   set initValue "{ \\"
   list set x $initValue
⇒ set x \{\ \\
```

Chapter 7
Control Flow

This chapter describes the Tcl commands for controlling the flow of execution in a script. Tcl's control flow commands are similar to the control flow statements in the C programming language and `csh`, including `if`, `while`, `for`, `foreach`, `switch`, and `eval`. Table 7.1 summarizes these commands.

7.1 The if command

The `if` command evaluates an expression, tests its result, and conditionally executes a script based on the result. For example, consider the following command, which sets variable `x` to zero if it was previously negative:

```
if {$x < 0} {
    set x 0
}
```

In this case `if` receives two arguments. The first is an expression and the second is a Tcl script. The expression can have any of the forms for expressions described in Chapter 5. The `if` command evaluates the expression and tests the result; if it is non-zero then `if` evaluates the Tcl script. If the value is zero then `if` returns without taking any further action.

`If` commands can also include one or more `elseif` clauses with additional tests and scripts, plus a final `else` clause with a script to evaluate if no test succeeds:

`break`	Terminates the innermost nested looping command.
`continue`	Terminates the current iteration of the innermost looping command and goes on to the next iteration of that command.
`eval` *`arg`* ?*`arg arg`* ...?	Concatenates all of the *`arg`*'s with separator spaces, then evaluates the result as a Tcl script and returns its result.
`for` *`init test reinit body`*	Executes *`init`* as a Tcl script. Then evaluates *`test`* as an expression. If it evaluates to non-zero then executes *`body`* as a Tcl script, executes *`reinit`* as a Tcl script, and re-evaluates *`test`* as an expression. Repeats until *`test`* evaluates to zero. Returns an empty string.
`foreach` *`varName list body`*	For each element of *`list`*, in order, set variable *`varName`* to that value and execute *`body`* as a Tcl script. Returns an empty string. *`List`* must be a valid Tcl list.
`if` *`test1`* ?`then`? *`body1`* ?`elseif` *`test2`* ?`then`? *`body2`* `elseif` ...? \ ?`else`? ?*`bodyn`*?	Evaluates *`test`* as an expression. If its value is non-zero then executes *`body1`* as a Tcl script and returns its value . Otherwise evaluates *`test2`* as an expression; if its value is non-zero then executes *`body2`* as a script and returns its value. If no test succeeds then executes *`bodyn`* as a Tcl script and returns its result.
`source` *`fileName`*	Reads the file whose name is *`fileName`* and evaluates its contents as a Tcl script. Returns the result of the script.
`switch` ?*`options`*? *`string pattern body`* ?*`pattern body`* ...? `switch` ?*`options`*? *`string`* {*`pattern body`* ?*`pattern body`* ...?}	Matches *`string`* against each *`pattern`* in order until a match is found, then executes the *`body`* corresponding to the matching *`pattern`*. If the last `pattern` is `default` then it matches anything. Returns the result of the *`body`* executed, or an empty string if no pattern matches. *`Options`* may be any of `-exact`, `-glob`, `-regexp`, or `--`.
`while` *`test body`*	Evaluates *`test`* as an expression. If its value is non-zero then executes *`body`* as a Tcl script and re-evaluates *`test`*. Repeats until *`test`* evaluates to zero. Returns an empty string.

Table 7.1. A summary of the Tcl commands for controlling the flow of execution.

```
if {$x < 0} {
    ...
} elseif {$x == 0} {
    ...
} elseif {$x == 1} {
    ...
} else {
    ...
}
```

This command will execute one of the four scripts indicated by "..." depending on the value of `x`. The result of the command will be the result of whichever script is executed. If an `if` command has no `else` clause and none of its tests succeeds then it returns an empty string.

The argument `else` is an optional "noise word". It is also legal to have `then` noise words after any of the expressions to test. The `elseif` words are not optional: they are needed to distinguish `elseif` clauses from `else` clauses.

Remember that the expressions and scripts for `if` and other control flow commands are parsed using the same approach as all arguments to all Tcl commands. It is almost always a good idea to enclose the expressions and scripts in braces so that substitutions are deferred until the the command is executed. Furthermore, each open brace must be on the same line as the preceding word or else the newline will be treated as a command separator. The following script is parsed as two commands, which probably isn't the desired result:

```
if {$x < 0}
{
    set x 0
}
```

7.2 Looping commands: while, for, and foreach

Tcl provides three commands for looping: `while`, `for`, and `foreach`. `While` and `for` are similar to the corresponding C statements and `foreach` is similar to the corresponding feature of the `csh` shell. Each of these commands executes a nested script over and over again; they differ in the kinds of setup they do before each iteration and in the ways they decide to terminate the loop.

The `while` command takes two arguments: an expression and a Tcl script. It evaluates the expression and if the result is non-zero then it executes the Tcl script. This process repeats over and over until the expression evaluates to zero, at which point the `while` command terminates and returns an empty string. For example, the script below copies a list from variable `b` to variable `a`, reversing the order of the elements along the way:

```
set b ""
set i [expr [llength $a] -1]
while {$i >= 0} {
    lappend b [lindex $a $i]
    incr i -1
}
```

The `for` command is similar to `while` except that it provides more explicit loop control. The program to reverse the elements of a list can be rewritten using `for` as follows:

```
set b ""
for {set i [expr [llength $a]-1]} {$i >= 0} {incr i -1} {
    lappend b [lindex $a $i]
}
```

The first argument to `for` is an initialization script, the second is an expression that determines when to terminate the loop, the third is a reinitialization script, which is evaluated after each execution of the loop body before evaluating the test again, and the fourth argument is a script that forms the body of the loop. `For` executes its first argument (the initialization script) as a Tcl command, then evaluates the expression. If the expression evaluates to non-zero, then `for` executes the body followed by the reinitialization script and re-evaluates the expression. It repeats this sequence over and over again until the expression evaluates to zero. If the expression evaluates to zero on the first test then neither the body script nor the reinitialization script is ever executed. Like `while`, `for` returns an empty string as result.

`For` and `while` are equivalent in that anything you can write using one command you can also write using the other command. However, `for` has the advantage of placing all of the loop control information in one place where it is easy to see. Typically the initialization, test, and re-initialization arguments are used to select a set of elements to operate on (integer indices in the above example) and the body of the loop carries out the operations on the chosen elements. This clean separation between element selection and action makes `for` loops easier to understand and debug. Of course, there are some situations where a clean separation between selection and action is not possible, and in these cases a `while` loop may make more sense.

The `foreach` command iterates over all of the elements of a list. For example, the following script provides yet another implementation of list reversal:

```
set b "";
foreach i $a {
    set b [linsert $b 0 $i]
}
```

`Foreach` takes three arguments. The first is the name of a variable, the second is a list, and the third is a Tcl script that forms the body of the loop. `Foreach` will execute the body script once for each element of the list, in order. Before executing the body in each iteration, `foreach` sets the variable to hold the next element of the list. Thus if variable `a` has the value "`first second third`" in the above example, the body will be executed three times. In the first iteration `i` will have the value `first`, in the second iteration it will have the value `second`, and in the third iteration it will have the value `third`. At the end of the loop, b will have the value "`third second first`" and `i` will have the value "`third`". As with the other looping commands, `foreach` always returns an empty string.

7.3 Loop control: break and continue

Tcl provides two commands that can be used to abort part or all of a looping command: `break` and `continue`. These commands have the same behavior as the corresponding statements in C. Neither takes any arguments. The `break` command causes the innermost enclosing looping command to terminate immediately. For example, suppose that in the list reversal example above it is desired to stop as soon as an element equal to `ZZZ` is found in the source list. In other words, the result list should consist of a reversal of only those source elements up to (but not including) a `ZZZ` element. This can be accomplished with `break` as follows:

```
set b "";
foreach i $a {
    if {$i == "ZZZ"} break
    set b [linsert $b 0 $i]
}
```

The `continue` command causes only the current iteration of the innermost loop to be terminated; the loop continues with its next iteration. In the case of `while`, this means skipping out of the body and re-evaluating the expression that determines when the loop terminates; in `for` loops, the re-initialization script is executed before re-evaluating the termination condition. For example, the following program is another variant of the list reversal example, where `ZZZ` elements are simply skipped without copying them to the result list:

```
set b "";
foreach i $a {
    if {$i == "ZZZ"} continue
    set b [linsert $b 0 $i]
}
```

7.4 The switch command

The `switch` command tests a value against a number of patterns and executes one of several Tcl scripts depending on which pattern matched. The same effect as `switch` can be achieved with an `if` command that has lots of `elseif` clauses, but `switch` provides a more compact encoding. Tcl's `switch` command has two forms; here is an example of the first form:

```
switch $x {a {incr t1} b {incr t2} c {incr t3}}
```

The first argument to `switch` is the value to be tested (the contents of variable `x` in the example). The second argument is a list containing one or more pairs of elements. The first argument in each pair is a pattern to compare against the value, and the second is a script to execute if the pattern matches. The `switch` command steps through these pairs in order, comparing the pattern against the value. As soon as it finds a match it executes the corresponding script and returns the value of that script as its value. If no pattern matches then no script is executed and `switch` returns an empty string. This particular command increments variable `t1` if `x` has the value a, `t2` if `x` has the value `b`, `t3` if `x` has the value `c`, and does nothing otherwise.

The second form spreads the patterns and scripts out into separate arguments rather than combining them all into one list:

```
switch $x a {incr t1} b {incr t2} c {incr t3}
```

This form has the advantage that you can invoke substitutions on the pattern arguments more easily, but most people prefer the first form because you can easily spread the patterns and scripts across multiple lines like this:

```
switch $x {
    a {incr t1}
    b {incr t2}
    c {incr t3}
}
```

The outer braces keep the newlines from being treated as command separators. With the second form you would have to use backslash-newlines like this:

```
switch $x \
    a {incr t1} \
    b {incr t2} \
    c {incr t3} \
}
```

The `switch` command supports three forms of pattern matching. You can precede the value to test with a switch that selects the form you want: `-exact` selects exact comparison, `-glob` selects pattern matching as in the `string match` command (see Section 10.1 for details) and `-regexp` selects regular-expression matching as described in Section 10.2. The default is `-glob`.

If the last pattern in a `switch` command is `default` then it matches any value. Its script will thus be executed if no other patterns match. For example, the script below will examine a list and produce three counters. The first, `t1`, counts the number of elements in the list that contain an `a`. The second, `t2`, counts the number of elements that are unsigned decimal integers. The third, `t3`, counts all of the other elements:

```
set t1 0
set t2 0
set t3 0
foreach i $x {
    switch -regexp $i in {
        a           {incr t1}
        ^[0-9]*$    {incr t2}
        default     {incr t3}
    }
}
```

If a script in a `switch` command is “-” then `switch` uses the script for the next pattern instead. This makes it easy to have several patterns that all execute the same script, as in the following example:

```
switch $x {
    a -
    b -
    c {incr t1}
    d {incr t2}
}
```

This script increments variable `t1` if `x` is `a`, `b`, or `c` and it increments `t2` if `x` is `d`.

7.5 Eval

`Eval` is a general-purpose building block for creating and executing Tcl scripts. It accepts any number of arguments, concatenates them together with separator spaces, and then executes the result as a Tcl script. One use of `eval` is for generating commands, saving them in variables, and then later evaluating the variables as Tcl scripts. For example, the script

```
set cmd "set a 0"
...
eval $cmd
```

clears variable `a` to `0` when the `eval` command is invoked.

Perhaps the most important use for `eval` is to force another level of parsing. The Tcl parser performs only level of parsing and substitution when parsing a command; the results of one substitution are not reparsed for other substitutions. However, there are occasionally times when another level of parsing is desirable, and `eval` provides the mechanism to achieve this. For example, suppose that a variable `vars` contains a list of variables and that you wish to unset each of these variables. One solution is to use the following script:

```
foreach i $vars {
    unset $i
}
```

This script will work just fine, but the `unset` command takes any number of arguments so it should be possible to unset all of the variables with a single command. Unfortunately the following script will not work:

```
unset $vars
```

The problem with this script is that all of the variable names are passed to `unset` as a single argument, rather than using a separate argument for each name. The solution is to use `eval`, as with the following command:

```
eval unset $vars
```

`Eval` generates a string consisting of "`unset` " followed by the list of variable names and then passes the string to Tcl for evaluation. The string gets re-parsed so each variable name ends up in a different argument to `unset`.

Note: This approach works even if some of the variable names contain spaces or special characters such as `$`. As described in Section 6.7, the only safe way to generate Tcl commands is using list operations such as `list` and `concat`. The command "`eval unset $vars`" is identical to the command "`eval [concat unset $vars]`"; in either case the script evaluated by `eval` is a proper list whose first element is "`unset`" and whose other elements are the elements of `vars`.

7.6 Executing from files: source

The `source` command is similar to the command by the same name in the `csh` shell: it reads a file and executes the contents of the file as a Tcl script. It takes a single argument that contains the name of the file. For example, the command

```
source init.tcl
```

will execute the contents of the file `init.tcl`. The return value from `source` will be the value returned when the file contents are executed, which is the return value from the last command in the file. In addition, `source` allows the `return` command to be used in the file's script to terminate the processing of the file. See Section 8.1 for more information on `return`.

Chapter 8
Procedures

A Tcl procedure is a command that is implemented with a Tcl script rather than C code. You can define new procedures at any time with the `proc` command described in this chapter. Procedures make it easy for you to package up solutions to problems so that they can be re-used easily. Procedures also provide a simple way for you to prototype new features in an application: once you've tested the procedures, you can reimplement them in C for higher performance; the C implementations will appear just like the original procedures so none of the scripts that invoke them will have to change.

Tcl provides special commands for dealing with variable scopes. Among other things, these commands allow you to pass arguments by reference instead of by value and to implement new Tcl control structures as procedures. Table 8.1 summarizes the Tcl commands related to procedures.

8.1 Procedure basics: proc and return

Procedures are created with the `proc` command, as in the following example:

```
proc plus {a b} {expr $a+$b}
```

The first argument to `proc` is the name of the procedure to be created, `plus` in this case. The second argument is a list of names of arguments to the procedure (`a` and `b` in the example). The third argument to `proc` is a Tcl script that forms the body of the new procedure. `Proc` creates a new command and arranges that whenever the command is invoked the procedure's body will be evaluated. In this case the new command will have the name `plus`; whenever `plus` is invoked it must receive two arguments. While the

`global` *`name1`* ?*`name2`* ...?	Binds variable names *`name1`*, *`name2`*, etc. to global variables. References to these names will refer to global variables instead of local variables for the duration of the current procedure. Returns an empty string.
`proc` *`name argList body`*	Defines a procedure whose name is *`name`*, replacing any existing command by that name. *`ArgList`* is a list with one element for each of the procedure's arguments, and *`body`* contains a Tcl script that is the procedure's body. Returns an empty string.
`return` ?*`options`*? ?*`value`*?	Returns from the innermost nested procedure or `source` command with *`value`* as the result of the procedure. *`Value`* defaults to an empty string. Additional options may be used to trigger an exceptional return (see Section 9.4).
`uplevel` ?*`level`*? *`arg`* ?*`arg arg`* ...?	Concatenates all of the *`arg`*'s with spaces as separators, then executes the resulting Tcl script in the variable context of stack level *`level`* . *`Level`* consists of a number or a number preceded by `#`, and defaults to `-1`. Returns the result of the script.
`upvar` ?*`level`*? *`otherVar1 myVar1`* ?*`otherVar2 myVar2`* ...?	Binds the local variable name *`myVar1`* to the variable at stack level *`level`* whose name is *`otherVar1`*. For the duration of the current procedure, variable references to *`myVar1`* will be directed to *`otherVar1`* instead. Additional bindings may be specified with *`otherVar2`* and *`myVar2`*, etc. *`Level`* has the same syntax and meaning as for `uplevel` and defaults to `-1`. Returns an empty string.

Table 8.1. A summary of the Tcl commands related to procedures and variable scoping.

body of `plus` is executing the variables `a` and `b` will contain the values of the arguments. The return value from the `plus` command is the value returned by the last command in `plus`'s body. Here are some correct and incorrect invocations of `plus`:

```
   plus 3 4
⇒  7
   plus 3 -1
⇒  2
   plus 1
∅  no value given for parameter "b" to "plus"
```

If you wish for a procedure to return early without executing its entire script, you can invoke the `return` command: it causes the enclosing procedure to return immediately and the argument to `return` will be the result of the procedure. Here is an implementation of factorial that uses `return`:

```
proc fac x {
    if {$x <= 1} {
        return 1
    }
    expr $x * [fac [expr $x-1]]
}
fac 4
⇒ 24
fac 0
⇒ 1
```

If the argument to `fac` is less than or equal to one then `fac` invokes `return` to return immediately. Otherwise it executes the `expr` command. The `expr` command is the last one in the procedure's body, so its result is returned as the result of the procedure.

8.2 Local and global variables

When the body of a Tcl procedure is evaluated it uses a different set of variables from its caller. These variables are called *local variables*, since they are only accessible within the procedure and are deleted when the procedure returns. Variables referenced outside any procedure are called *global variables*. It is possible to have a local variable with the same name as a global variable or a local variable in another active procedure, but these will be different variables: changes to one will not affect any of the others. If a procedure is invoked recursively then each recursive invocation will have a distinct set of local variables.

The arguments to a procedure are just local variables whose values are set from the words of the command that invoked the procedure. When execution begins in a procedure, the only local variables with values are those corresponding to arguments. Other local variables are created automatically when they are set.

A procedure can reference global variables with the `global` command. For example, the following command makes the global variables `x` and `y` accessible inside a procedure:

```
global x y
```

The `global` command treats each of its arguments as the name of a global variable and sets up bindings so that references to those names within the procedure will be directed to global variables instead of local ones. `Global` can be invoked at any time during a procedure; once it has been invoked, the bindings will remain in effect until the procedure returns.

Note: Tcl does not provide a form of variable equivalent to "static" variables in C, which are limited in scope to a given procedure but have values that persist across calls to the procedure. In Tcl you must use global variables for purposes like this. To avoid name conflicts with other such variables you should include the name of the procedure or the name of its enclosing package in the variable name, for example "`Hypertext_numLinks`*".*

8.3 Defaults and variable numbers of arguments

In the examples so far, the second argument to `proc` (which describes the arguments to the procedure) has taken a simple form consisting of the names of the arguments. Three additional features are available for specifying arguments. First, the argument list may be specified as an empty string. In this case the procedure takes no arguments. For example, the following command defines a procedure that prints out two global variables:

```
proc printVars {} {
    global a b
    puts "a is $a, b is $b"
}
```

The second additional feature is that defaults may be specified for some or all of the arguments. The argument list is actually a list of lists, with each sublist corresponding to a single argument. If a sublist has only a single element (which has been the case up until now) that element is the name of the argument. If a sublist has two arguments, the first is the argument's name and the second is a default value for it. For example, here is a procedure that increments a given value by a given amount, with the amount defaulting to 1:

```
proc inc {value {increment 1}} {
    expr $value+$increment
}
```

The first element in the argument list, `value`, specifies a name with no default value. The second element specifies an argument with name `increment` and a default value of `1`. This means that `inc` can be invoked with either one or two arguments:

```
inc 42 3
⇒ 45
inc 42
⇒ 43
```

If a default isn't specified for an argument in the `proc` command then that argument must be supplied whenever the procedure is invoked. The defaulted arguments, if any, must be the last arguments for the procedure: if a particular argument is defaulted then all the arguments after it must also be defaulted.

The third special feature in argument lists is support for variable numbers of arguments. If the last argument in the argument list is the special value `args`, then the procedure may be called with varying numbers of arguments. Arguments before `args` in the argument list are handled as before, but any number of additional arguments may be specified. The procedure's local variable `args` will be set to a list whose elements are all of the extra arguments. If there are no extra arguments then `args` will be set to an empty string. For example, the following procedure takes any number of arguments and returns their sum:

```
    proc sum args {
        set s 0
        foreach i $args {
            incr s $i
        }
        return $s
    }
    sum 1 2 3 4 5
⇒   15
    sum
⇒   0
```

If a procedure's argument list contains additional arguments before `args` then they may be defaulted as described above. Of course, if this happens there will be no extra arguments so `args` will be set to an empty string. No default value may be specified for `args`: the empty string is its default.

8.4 Call by reference: upvar

The `upvar` command provides a general mechanism for accessing variables outside the context of a procedure. It can be used to access either global variables or local variables in some other active procedure. Most often it is used to implement call-by-reference argument passing. Here is a simple example of `upvar` in a procedure that prints out the contents of an array:

```
    proc parray name {
        upvar $name a
        foreach el [lsort [array names a]] {
            puts "$el = $a($el)"
        }
    }
    set info(age) 37
    set info(position) "Vice President"
    parray info
⇒   age = 37
    position = "Vice President"
```

When `parray` is invoked it is given the name of an array as argument. The `upvar` command then makes this array accessible through a local variable in the procedure. The first argument to `upvar` is the name of a variable accessible to the procedure's caller. This may be either a global variable, as in the example, or a local variable in a calling procedure. The second argument is the name of a local variable. `Upvar` arranges things so that accesses to local variable `a` will actually refer to the variable in the caller whose name is given by variable `name`. In the example this means that when `parray` reads elements of `a` it is actually reading elements of the `info` global variable. If `parray` were to write `a` it would modify `info`. `Parray` uses the "`array names`" command to retrieve a list of all the elements in the array, sorts them with `lsort`, then prints out each the elements in order.

Note: In the example it appears as if the output is returned as the procedure's result; in fact it is printed directly to standard output and the result of the procedure is an empty string.

The first variable name in an `upvar` command normally refers to the context of the current procedure's caller. However, it is also possible to access variables from any level on the call stack, including global level. For example,

```
upvar #0 other x
```

makes global variable `other` accessible via local variable `x` (the `#0` argument specifies that `other` should be interpreted as a global variable, regardless of how many nested procedure calls are active), and

```
upvar -2 other x
```

makes variable `other` in the caller of the caller of the current procedure accessible as local variable `x` (`-2` specifies that the context of `other` is 2 levels up the call stack). See the reference documentation for more information on specifying a level in `upvar`.

8.5 Creating new control structures: uplevel

The `uplevel` command is a cross between `eval` and `upvar`. It evaluates its argument(s) as a script, just like `eval`, but the script is evaluated in the variable context of a different stack level, like `upvar`. With `uplevel` you can define new control structures as Tcl procedures. For example, here is a new control flow command called `do`:

```
proc do {varName first last body} {
    upvar $varName v
    for {set v $first} {$v <= $last} {incr v} {
        uplevel $body
    }
}
```

The first argument to `do` is the name of a variable. `Do` sets that variable to consecutive integer values in the range between its second and third arguments, and executes the fourth argument as a Tcl command once for each setting. Given this definition of `do`, the following script creates a list of squares of the first five integers:

```
set a {}
do i 1 5 {
    lappend a [expr $i*$i]
}
set a
⇒ 1 4 9 16 25
```

The do procedure uses upvar to access the loop variable (i in the example) as local variable v. Then it uses the for command to increment the loop variable through the desired range. For each value it invokes uplevel to execute the loop body in the variable context of the caller; this causes references to variables a and i in the body of the loop to refer to variables in do's caller. If eval were used instead of uplevel then a and i would be treated as local variables in do, which would not produce the desired effect.

Note: *This implementation of* do *does not handle exceptional conditions properly. For example, if the body of the loop contains a* return *command it will only cause the* do *procedure to return, which is more like the behavior of* break*. If a* return *occurs in the body of a built-in control-flow command like* for *or* while *then it causes the procedure that invoked the command to return. In Chapter 9 you will see how to implement this behavior for* do*.*

As with upvar, uplevel takes an optional initial argument that specifies an explicit stack level. See the reference documentation for details.

Chapter 9
Errors and Exceptions

As you have seen in previous chapters, there are many things that can result in errors in Tcl commands. Errors can occur because a command doesn't exist, or because it doesn't receive the right number of arguments, or because the arguments have the wrong form, or because some other problem occurs in executing the command, such as an error in a system call for file I/O. In most cases errors represent severe problems that make it impossible for the application to complete the script it is processing. Tcl's error facilities are intended to make it easy for the application to unwind the work in progress and display an error message to the user that indicates what went wrong. Presumably the user will fix the problem and retry the operation.

Errors are just one example of a more general phenomenon called *exceptions*. Exceptions are events that cause scripts to be aborted; they include the `break`, `continue`, and `return` commands as well as errors. Tcl allows exceptions to be "caught" by scripts so that only part of the work in progress is unwound. After catching an exception the script can ignore it or take steps to recover from it. If the script can't recover then it can reissue the exception. Table 9.1 summarizes the Tcl commands related to exceptions.

9.1 What happens after an error?

When a Tcl error occurs the current command is aborted. If that command is part of a larger script then the script is also aborted. If the error occurs while executing a Tcl procedure, then the procedure is aborted, along with the procedure that called it, and so on until all the active procedures have aborted. After all Tcl activity has been unwound in this way, control eventually returns to C code in the application, along with an indication that an

`catch` *`command`* ?*`varName`*?	Evaluates *`command`* as a Tcl script and returns an integer code that identifies the completion status of the command. If *`varName`* is specified then it gives the name of a variable, which will be modified to hold the return value or error message generated by *`command`*.
`error` *`message`* ?*`info`*? ?*`code`*?	Generates an error with *`message`* as the error message. If *`info`* is specified and is not an empty string then it is used to initialize the `errorInfo` variable. If *`code`* is specified then it is stored in the `errorCode` variable.
`return -code` *`code`* ?`-errorinfo` *`info`*? ?`-errorcode` *`code`*? ?*`string`*?	Causes the current procedure to return an exceptional condition. *`Code`* specifies the condition and must be `ok`, `error`, `return`, `break`, `continue`, or an integer. The `-errorinfo` option may be used to specify a starting value for the `errorInfo` variable, and `-errorcode` may be used to specify a value for the `errorCode` variable. *`String`* gives the return value or error message associated with the return; it defaults to an empty string.

Table 9.1. A summary of the Tcl commands related to exceptions.

error occurred and a message describing the error. It is up to the application to decide how to handle this situation, but most interactive applications will display the error message for the user and continue processing user input. In a batch-oriented application where the user can't see the error message and adjust future actions accordingly, the application might print the error message into a log and abort.

For example, consider the following script, which is intended to sum the elements of a list:

```
set list {44 16 123 98 57}
set sum 0
foreach el $list {
    set sum [expr $sum+$element]
}
```

∅ *`can't read "element": no such variable`*

This script is incorrect because there is no variable `element`: the variable name `element` in the `expr` command should have been `el` to match the loop variable for the `foreach` command. When the script is executed an error will occur as Tcl parses the `expr` command: Tcl will attempt to substitute the value of variable `element` but will not be able to find a variable by that name, so it will signal an error. This error indication will be returned to the `foreach` command, which had invoked the Tcl interpreter to evaluate the loop body. When `foreach` sees that an error has occurred, it will abort its loop and return the same error indication as its own result. This in turn will cause the overall script to be aborted. The error message "`can't read "element": no such variable`" will be returned along with the error, and will probably be displayed for the user.

In many cases the error message will provide enough information for you to pinpoint where and why the error occurred so you can avoid the problem in the future. However, if the error occurred in a deeply nested set of procedure calls the message alone may not provide enough information to figure out where the error occurred. To help pinpoint the location of the error, Tcl creates a stack trace as it unwinds the commands that were in progress, and it stores the stack trace in the global variable `errorInfo`. The stack trace describes each of the nested calls to the Tcl interpreter. For example, after the above error `errorInfo` will have the following value:

```
can't read "element": no such variable
    while executing
"expr $sum+$element"
    invoked from within
"set sum [expr $sum+$element]..."
    ("foreach" body line 2)
    invoked from within
"foreach el $list {
    set sum [expr $sum+$element]
}"
```

Tcl provides one other piece of information after errors, in the global variable `errorCode`. `ErrorCode` has a format that is easy to process with Tcl scripts; it is most commonly used in Tcl scripts that attempt to recover from errors using the `catch` command described below. The `errorCode` variable consists of a list with one or more elements. The first element identifies a general class of errors and the remaining elements provide more information in a class-dependent fashion. For example, if the first element of `errorCode` is `POSIX` then it means that an error occurred in a POSIX system call. `ErrorCode` will contain two additional elements giving the POSIX name for the error, such as `ENOENT`, and a human-readable message describing the error. See the reference documentation for a complete description of all the forms `errorCode` can take, or refer to the descriptions of individual commands that set `errorCode`, such as those in Chapter 11 and Chapter 12.

The `errorCode` variable is a late-comer to Tcl and is only filled in by a few commands, mostly dealing with file access and child processes. If a command generates an error without setting `errorCode` then Tcl fills it in with the value `NONE`.

9.2 Generating errors from Tcl scripts

Most Tcl errors are generated by the C code that implements the Tcl interpreter and the built-in commands. However, it is also possible to generate an error by executing the `error` Tcl command as in the following example:

```
if {($x < 0) || ($x > 100)} {
    error "x is out of range ($x)"
}
```

The `error` command generates an error and uses its argument as the error message.

As a matter of programming style, you should only use the `error` command in situations where the correct action is to abort the script being executed. If you think that an error is likely to be recovered from without aborting the entire script, then it is probably better to use the normal return value mechanism to indicate success or failure (e.g. return one value from a command if it succeeded and another if it failed, or set variables to indicate success or failure). Although it is possible to recover from errors (you'll see how in Section 9.3 below) the recovery mechanism is more complicated than the normal return value mechanism. Thus it's best to generate errors only in situations where you won't usually want to recover.

9.3 Trapping errors with catch

Errors generally cause all active Tcl commands to be aborted, but there are some situations where it is useful to continue executing a script after an error has occurred. For example, suppose that you want to unset variable `x` if it exists, but it may not exist at the time of the `unset` command. If you invoke `unset` on a variable that doesn't exist then it generates an error:

```
  unset x
∅ can't unset "x": no such variable
```

You can use the `catch` command to ignore the error in this situation:

```
  catch {unset x}
⇒ 1
```

The argument to `catch` is a Tcl script, which `catch` evaluates. If the script completes normally then `catch` returns 0. If an error occurs in the script then `catch` traps the error (so that the `catch` command itself is not aborted by the error) and returns 1 to indicate that an error occurred. The example above ignores any errors in `unset` so `x` is unset if it existed and the script has no effect if `x` didn't previously exist.

The `catch` command can also take a second argument. If the argument is provided then it is the name of a variable and `catch` modifies the variable to hold either the script's return value (if it returns normally) or the error message (if the script generates an error):

```
  catch {unset x} msg
⇒ 1
  set msg
⇒ can't unset "x": no such variable
```

In this case the `unset` command generates an error so `msg` is set to contain the error message. If variable `x` had existed then `unset` would have returned successfully, so the return value from `catch` would have been `0` and `msg` would have contained the return value from the `unset` command, which is an empty string. This longer form of `catch` is useful if you need access to the return value when the script completes successfully. It's also useful if you need to do something with the error message after an error, such as logging it to a file.

9.4 Exceptions in general

Errors are not the only things in Tcl that cause work in progress to be aborted. Errors are just one example of a set of events called *exceptions*. In addition to errors there are three other kinds of exceptions in Tcl, which are generated by the `break`, `continue`, and `return` commands. All exceptions cause active scripts to be aborted in the same way, except for two differences. First, the `errorInfo` and `errorCode` variables are only set during error exceptions. Second, the exceptions other than errors are almost always caught by an enclosing command, whereas errors usually unwind all the work in progress. For example, `break` and `continue` commands are normally invoked inside a looping command such as `foreach`; `foreach` will catch break and continue exceptions and terminate the loop or skip to the next iteration. Similarly, `return` is normally only invoked inside a procedure or a file being `source`'d. Both the procedure implementation and the `source` command catch return exceptions.

Note: If a `break` or `continue` command is invoked outside any loop then active scripts unwind until the outermost script for a procedure is reached or all scripts in progress have been unwound. At this point Tcl turns the break or continue exception into an error with an appropriate message.

All exceptions are accompanied by a string value. In the case of an error, the string is the error message. In the case of `return`, the string is the return value for the procedure or script. In the case of `break` and `continue` the string is always empty.

The `catch` command actually catches all exceptions, not just errors. The return value from `catch` indicates what kind of exception occurred and the variable specified in `catch`'s second argument is set to hold the string associated with the exception (see Table 9.2). For example:

```
   catch {return "all done"} string
⇒ 2
   set string
⇒ all done
```

Whereas `catch` provides a general mechanism for catching exception of all types, `return` provides a general mechanism for generating exceptions of all types. If its first argument consists of the keyword `-code`, as in

Return value from `catch`	Description	Caught by
0	Normal return. String gives return value.	Not applicable
1	Error. String gives message describing the problem.	`Catch`
2	The `return` command was invoked. String gives return value for procedure or `source` command.	`Catch`, `source`, procedures
3	The `break` command was invoked. String is empty.	`Catch`, `for`, `foreach`, `while`, procedures
4	The `continue` command was invoked. String is empty.	`Catch`, `for`, `foreach`, `while`, procedures
anything else	Defined by user or application.	`Catch`

Table 9.2. A summary of Tcl exceptions. The first column indicates the value returned by `catch` in each instance. The second column describes when the exception occurs and the meaning of the string associated with the exception. The last column lists the commands that catch exceptions of that type (“procedures” means that the exception is caught by a Tcl procedure when its entire body has been aborted). The top row refers to normal returns where there is no exception.

```
return -code return 42
```

then its second argument is the name of an exception (`return` in this case) and the third argument is the string associated with the exception. The enclosing procedure will return immediately, but instead of a normal return it will return with the exception described by the `return` command’s arguments. In the example above the procedure will generate a return exception, which will then cause the calling procedure to return as well.

In Section 8.5 you saw how a new looping command `do` could be implemented as a Tcl procedure using `upvar` and `uplevel`. However, the example in Section 8.5 did not properly handle exceptions within the loop body. Here is a new implementation of `do` that uses `catch` and `return` to deal with exceptions properly:

```
proc do {varName first last body} {
    global errorInfo errorCode
    upvar $varName v
    for {set v $first} {$v <= $last} {incr v} {
        switch [catch {uplevel $body} string] {
            1 {return -code error -errorinfo $errorInfo \
                    -errorCode $errorcode $string}
            2 {return -code return $string}
            3 return
        }
    }
}
```

This new implemenation evaluates the loop body inside a `catch` command and then checks to see how the body terminates. If no exception occurs (return value 0 from `catch`) or if the exception is a continue (return value 4) then `do` just goes on to the next iteration. If an error or return occurs (return value 1 or 2 from `catch`) then `do` uses the `return` command to reflect the exception upward to the caller. If a break exception occurs (return value 3 from `catch`) then `do` returns to its caller normally, ending the loop.

When `do` reflects an error upwards it uses the `-errorinfo` option to `return` to make sure that a proper stack trace is available after the error. If that option were omitted then a fresh stack trace would be generated starting with `do`'s error return; the stack trace would not indicate where in `body` the error occurred. The context within `body` is available in the `errorInfo` variable at the time `catch` returns, and the `-errorinfo` option causes this value to be used as the initial contents of the stack trace when `do` returns an error. As additional unwinding occurs more information gets added to the initial value, so that the final stack trace includes both the context within `body` and the context of the call to `do`. The `-errorcode` option serves a similar purpose for the `errorCode` variable, retaining the `errorCode` value from the original error as the `errorCode` value when `do` propagates the error. Without the `-errorcode` option the `errorCode` variable will always end up with the value `NONE`.

Chapter 10
String Manipulation

This chapter describes Tcl's facilities for manipulating strings. The string manipulation commands provide pattern matching in two different forms, one that mimics the rules used by shells for file name expansion and another that uses regular expressions as patterns. Tcl also has commands for formatted input and output in a style similar to the C procedures `scanf` and `printf`. Finally, there are several utility commands with functions such as computing the length of a string, extracting characters from a string, and case conversion. Tables10.1 and 10.2 summarize the Tcl commands for string processing.

10.1 Glob-style pattern matching

The simplest of Tcl's two forms of pattern matching is called "glob" style. It is named after the mechanism used in the `csh` shell for file name expansion, which is called "globbing". Glob-style matching is easier to learn and use than the regular expressions described in the next two sections, but it only works well for simple cases. For more complex pattern matching you will probably need to use regular expressions.

The command `string match` implements glob-style pattern matching. For example, the following script extracts all of the elements of a list that begin with "`Tcl`":

```
set new {}
foreach el $list {
    if [string match Tcl* $el] {
        lappend new $el
    }
}
```

format *formatString* ?*value value* ...?	Returns a result equal to *formatString* except that the *value* arguments have been substituted in place of % sequences in *formatString*.
regexp ?-indices? ?-nocase? ?--? *exp string* ?*matchVar*? \ ?*subVar subVar* ...?	Determines whether the regular expression *exp* matches part or all of *string* and returns 1 if it does, 0 if it doesn't. If there is a match, information about matching range(s) is placed in the variables named by *matchVar* and the *subVar*'s, if they are specified.
regsub ?-*all*? ?-*nocase*? ?--? *exp string subSpec varName*	Matches *exp* against *string* as for regexp and returns 1 if there is a match, 0 if there is none. Also copies *string* to the variable named by *varName*, making substitutions for the matching portion(s) as specified by *subSpec*.
scan *string format varName* ?*varName varName* ...?	Parses fields from *string* as specified by *format* and places the values that match % sequences into variables named by the *varName* arguments.
string compare *string1 string2*	Returns -1, 0, or 1 if *string1* is lexicographically less than, equal to, or greater than *string2*.
string first *string1 string2*	Returns the index in *string2* of the first character in the leftmost substring that exactly matches the characters in *string1*, or -1 if there is no such match.
string index *string charIndex*	Returns the *charIndex*'th character of *string*, or an empty string if there is no such character. The first character in *string* has index 0.
string last *string1 string2*	Returns the index in *string2* of the first character in the rightmost substring of *string2* that exactly matches *string1*. If there is no matching substring then -1 is returned.
string length *string*	Returns the number of characters in *string*.
string match *pattern string*	Returns 1 if *pattern* matches *string* using glob-style matching rules (*, ?, [], and \) and 0 if it doesn't.
string range *string first last*	Returns the substring of *string* that lies between the indices given by *first* and *last*, inclusive. An index of 0 refers to the first character in the string, and *last* may be end to refer to the last character of the string.

Table 10.1. A summary of the Tcl commands for string manipulation (continued in Table 10.2).

`string tolower` *string*	Returns a value identical to *string* except that all upper case characters have been converted to lower case.
`string toupper` *string*	Returns a value identical to *string* except that all lower case characters have been converted to upper case.
`string trim` *string* ?*chars*?	Returns a value identical to *string* except that any leading or trailing characters that appear in *chars* are removed. *Chars* defaults to the white space characters (space, tab, newline, and carriage return).
`string trimleft` *string* ?*chars*?	Same as `string trim` except that only leading characters are removed.
`string trimright` *string* ?*chars*?	Same as `string trim` except that only trailing characters are removed.

Table 10.2. A summary of the Tcl commands for string manipulation, cont'd.

The `string` command is actually about a dozen string-manipulation commands rolled into one. If the first argument is `match` then the command performs glob-style pattern matching and there must be two additional arguments, a pattern and a string. The command returns `1` if the pattern matches the string, `0` if it doesn't. For the pattern to match the string, each character of the pattern must be the same as the corresponding character of the string, except that a few pattern characters are interpreted specially. For example, a `*` in the pattern matches a substring of any length, so "`Tcl*`" matches any string whose first three characters are "`Tcl`". Here is a list of all the special characters supported in glob-style matching:

`*`	Matches any sequence of zero or more characters.
`?`	Matches any single character.
`[`*chars*`]`	Matches any single character in *chars*. If *chars* contains a sequence of the form *a*`-`*b* then any character between *a* and *b*, inclusive, will match.
`\`*x*	Matches the single character *x*. This provides a way to avoid special interpretation for any of the characters `*?[]\` in the pattern.

Many simple things can be done easily with glob-style patterns. For example, "`*.[ch]`" matches all strings that end with either "`.c`" or "`.h`". However, many interesting forms of pattern matching cannot be expressed at all with glob-style patterns. For example, there is no way to use a glob-style pattern to test whether a string consists entirely of digits: the pattern "`[0-9]`" tests for a single digit, but there is no way to specify that there may be more than one digit.

Character(s)	Meaning
`.`	Matches any single character.
`^`	Matches the null string at the start of the input string.
`$`	Matches the null string at the end of the input string.
`\x`	Matches the character *x*.
`[chars]`	Matches any single character from *chars*. If the first character of *chars* is ^ then it matches any single character not in the remainder of *chars*. A sequence of the form *a*−*b* in *chars* is treated as shorthand for all of the ASCII characters between *a* and *b*, inclusive. If the first character in *chars* (possibly following a ^) is] then it is treated literally (as part of *chars* instead of a terminator). If a − appears first or last in *chars* then it is treated literally.
`(regexp)`	Matches anything that matches the regular expression *regexp*. Used for grouping and for identifying pieces of the matching substring.
`*`	Matches a sequence of 0 or more matches of the preceding atom.
`+`	Matches a sequence of 1 or more matches of the preceding atom.
`?`	Matches either a null string or a match of the preceding atom.
regexp1 | *regexp2*	Matches anything that matches either *regexp1* or *regexp2*.

Table 10.3. The special characters permitted in regular expression patterns.

10.2 Pattern matching with regular expressions

Tcl's second form of pattern matching uses regular expressions like those for the `egrep` program. Regular expressions are more complex than glob-style patterns but more powerful. Tcl's regular expressions are based on Henry Spencer's publicly available implementation, and parts of the description below are copied from Spencer's documentation.

A regular expression pattern can have several layers of structure. The basic building blocks are called *atoms*, and the simplest form of regular expression consists of one or more atoms. For a regular expression to match an input string, there must be a substring of the input where each of the regular expression's atoms (or other components, as you'll see below) matches the corresponding part of the substring. In most cases atoms are single characters, each of which matches itself. Thus the regular expression `abc` matches any string containing `abc`, such as `abcdef` or `xabcy`.

A number of characters have special meanings in regular expressions; they are summarized in Table 10.3. The characters `^` and `$` are atoms that match the beginning and end of the input string respectively; thus `^abc` matches any string that starts with `abc`, `abc$` matches any string that ends in `abc`, and `^abc$` matches `abc` and nothing else. The atom

"`.`" matches any single character, and the atom `\x`, where `x` is any single character, matches `x`. For example, the regular expression "`.\$`" matches any string that contains a dollar-sign, as long as the dollar-sign isn't the first character.

Besides the atoms already described, there are two other forms for atoms in regular expressions. The first form consists of any regular expression enclosed in parentheses, such as "`(a.b)`". Parentheses are used for grouping. They allow operators such as `*` to be applied to entire regular expressions as well as atoms. They are also used to identify pieces of the matching substring for special processing. Both of these uses are described in more detail below.

The final form for an atom is a *range*, which is a collection of characters between square brackets. A range matches any single character that is one of the ones between the brackets. Furthermore, if there is a sequence of the form *a*`-`*b* among the characters, then all of the ASCII characters between *a* and *b* are treated as acceptable. Thus the regular expression `[0-9a-fA-F]` matches any string that contains a hexadecimal digit. If the character after the `[` is a `^` then the sense of the range is reversed: it only matches characters `not` among those specified between the `^` and the `]`.

The three operators `*`, `+`, and `?` may follow an atom to specify repetition. If an atom is followed by `*` then it matches a sequence of zero or more matches of that atom. If an atom is followed by `+` then it matches a sequence of one or more matches of the atom. If an atom is followed by `?` then it matches either an empty string or a match of the atom. For example, "`^(0x)?[0-9a-fA-F]+$`" matches strings that are proper hexadecimal numbers, i.e. those consisting of an optional `0x` followed by one or more hexadecimal digits.

Finally, regular expressions may be joined together with the `|` operator. The resulting regular expression matches anything that matches either of the regular expresssions that surround the `|`. For example, the following pattern matches any string that is either a hexadecimal number or a decimal number:

```
^((0x)?[0-9a-fA-F]+|[0-9]+)$
```

Note that the information between parentheses may be any regular expression, including additional regular expressions in parentheses, so it is possible to build up quite complex structures.

The `regexp` command invokes regular expression matching. In its simplest form it takes two arguments: the regular expression pattern and an input string. It returns `0` or `1` to indicate whether or not the pattern matched the input string:

```
   regexp {^[0-9]+$} 510
⇒  1
   regexp {^[0-9]+$} -510
⇒  0
```

Note: The pattern must be enclosed in braces so that the characters `$`, `[`, and `]` are passed through to the `regexp` command instead of triggering variable and command *substitution. In almost always a good idea to enclose regular expression patterns in braces.*

If `regexp` is invoked with additional arguments after the input string then each additional argument is treated as the name of a variable. The first variable is filled in with the substring that matched the entire regular expression. The second variable is filled in with the portion of the substring that matched the leftmost parenthesized subexpression within the pattern; the third variable is filled in with the match for the next parenthesized subexpression, and so on. If there are more variable names than parenthesized subexpressions then the extra variables are set to empty strings. For example, after executing the command

```
regexp {([0-9]+) *([a-z]+)} "Walk 10 km" a b c
```

variable `a` will have the value "`10 km`", `b` will have the value `10`, and `c` will have the value `km`. This ability to extract portions of the matching substring allows `regexp` to be used for parsing.

It is also possible to specify two extra switches to `regexp` before the regular expression argument. A `-nocase` switch specifies that alphabetic atoms should match either upper-case or lower-case letters. For example:

```
regexp {[a-z]} A
```
⇒ *0*
```
regexp -nocase {[a-z]} A
```
⇒ *1*

The `-indices` switch specifies that the additional variables should not be filled in with the values of matching substrings. Instead, each should be filled in with a list giving the first and last indices of the substring's range within the input string. After the command

```
regexp -indices {([0-9]+) *([a-z]+)} "Walk 10 km" \
    a b c
```

variable `a` will have the value "`5 9`", `b` will have the value "`5 6`", and `c` will have the value "`8 9`".

10.3 Using regular expressions for substitutions

Regular expressions can also be used to perform substitutions using the `regsub` command. Consider the following example:

```
regsub there "They live there lives" their x
```
⇒ *1*

The first argument to `regsub` is a regular expression pattern and the second argument is an input string, just as for `regexp`. And, like `regexp`, `regsub` returns `1` if the pattern matches the string, `0` if it doesn't. However, `regsub` does more than just check for a match: it creates a new string by substituting a replacement value for the matching substring. The replacement value is contained in the third argument to `regsub`, and the new string is stored in the variable named by the final argument to `regsub`. Thus, after the above command completes `x` will have the value "`They live their lives`". If the pattern had not matched the string then `0` would have been returned and `x` would have the value "`They live there lives`".

Two special switches may appear as arguments to `regsub` before the regular expression. The first is `-nocase`, which causes case differences between the pattern and the string to be ignored just as for `regexp`. The second possible switch is `-all`. Normally `regsub` makes only a single substitution, for the first match found in the input string. However, if `-all` is specified then `regsub` continues searching for additional matches and makes substitutions for all of the matches found. For example, after the command

```
regsub -all a ababa zz x
```

`x` will have the value `zzbzzbzz`. If `-all` had been omitted then `x` would have been set to `zzbaba`.

In the examples above the replacement string is a simple literal value. However, if the replacement string contains a "`&`" or "`\0`" then the "`&`" or "`\0`" is replaced in the substitution with the substring that matched the regular expression. If a sequence of the form `\`*n* appears in the replacement string, where *n* is a decimal number, then the substring that matched the *n*-th parenthesized subexpression is substituted instead of the `\`*n*. For example, the command

```
regsub -all a|b axaab && x
```

doubles all of the `a`'s and `b`'s in the input string. In this case it sets `x` to `aaxaaaabb`. Or, the command

```
regsub -all (a+)(ba*) aabaabxab {z\2} x
```

replaces sequences of `a`'s with a single `z` if they precede a `b` but don't also follow a `b`. In this case `x` is set to `zbaabxzb`. Backslashes may be used in the replacement string to allow "`&`", "`\0`", "`\`*n*", or backslash characters to be substituted verbatim without any special interpretation.

Note: *It's usually a good idea to enclose complex replacement strings in braces as in the example above; otherwise the Tcl parser will process backslash sequences and the replacement string received by* `regsub` *may not contain backslashes that are needed.*

10.4 Generating strings with format

Tcl's `format` command provides facilities like those of the `sprintf` procedure from the ANSI C library. For example, consider the following command:

```
format "The square root of 10 is %.3f" [expr sqrt(10)]
⇒ The square root of 10 is 3.162
```

The first argument to `format` is a format string, which may contain any number of conversion specifiers such as "`%.3f`". For each conversion specifier `format` generates a replacement string by reformatting the next argument according to the conversion specifier. The result of the `format` command consists of the format string with each conversion specifier replaced by the corresponding replacement string. In the above example "`%.3f`" specifies that the next argument is to be formatted as a real number with three digits after the decimal point. `Format` supports almost all of the conversion specifiers defined for ANSI C `sprintf`, such as "`%d`" for a decimal integer, "`%x`" for a hexadecimal integer, and "`%e`" for real numbers in mantissa-exponent form.

The `format` command plays a less significant role in Tcl than `printf` and `sprintf` play in C. Many of the uses of `printf` and `sprintf` are simply for conversion from binary to string format or for string substitution. Binary-to-string conversion isn't needed in Tcl because values are already stored as strings, and substitution is already available through the Tcl parser. For example, the command

```
set msg [format "%s is %d years old" $name $age]
```

can be written more simply as

```
set msg "$name is $age years old"
```

The `%d` conversion specifier in the `format` command could be written just as well as `%s`; with `%d` `format` converts the value of age to a binary integer, then converts the integer back to a string again.

`Format` is typically used in Tcl to reformat a value to improve its appearance, or to convert from one representation to another (e.g. from decimal to hexadecimal). As an example of reformatting, here is a that script prints the first ten powers of *e* in a table:

```
puts "Number  Exponential"
for {set i 1} {i <= 10} {incr i} {
    puts [format "%4d %12.3f" $i [expr exp($i)]]
}
```

This script generates the following output on standard output:

```
Number  Exponential
   1        2.718
   2        7.389
   3       20.085
   4       54.598
   5      148.413
   6      403.429
   7     1096.630
   8     2980.960
   9     8103.080
  10    22026.500
```

The conversion specifier "`%4d`" causes the integers in the first column of the table to be printed right-justifed in a field four digits wide, so that they line up under their column header. The conversion specifier "`%12.3f`" causes each of the real values to be printed right-justified in a field 12 digits wide, so that the values line up; it also sets the precision at 3 digits to the right of the decimal point.

The second main use for `format`, changing the reprensentation of a value, is illustrated by the script below, which prints a table showing the ASCII characters that correspond to particular integer values:

```
puts "Integer  ASCII"
for {set i 95} {$i <= 101} {incr i} {
    puts [format "%4d       %c" $i $i]
}
```

This script generates the following output on standard output:

```
Integer  ASCII
  95       _
  96       `
  97       a
  98       b
  99       c
 100       d
 101       e
```

The value of `i` is used twice in the format command, once with `%4d` and once with `%c`. The `%c` specifier takes an integer argument and generates a replacement string consisting of the ASCII character whose represented by the integer.

10.5 Parsing strings with scan

The `scan` command provides almost exactly the same facilities as the `sscanf` procedure from the ANSI C library. `Scan` is roughly the inverse of `format`. It starts with a formatted string, parses the string under the control of a format string, extracts fields corresponding to `%` conversion specifiers in the format string, and places the extracted values in Tcl variables. For example, after the following command is executed variable `a` will have the value `16` and variable `b` will have the value `24.2`:

```
scan "16 units, 24.2% margin" "%d units, %f" a b
⇒ 2
```

The first argument to `scan` is the string to parse, the second is a format string that controls the parsing, and any additional arguments are names of variables to fill in with converted values. The return value of 2 indicates that two conversions were completed successfully.

`Scan` operates by scanning the string and the format together. Each character in the format must match the corresponding character in the string, except for blanks and tabs, which are ignored, and `%` characters. When a `%` is encountered in the format, it indicates the start of a conversion specifier: `scan` converts the next input characters according to the conversion specifier and stores the result in the variable given by the next argument to `scan`. White space in the string is skipped except in the case of a few conversion specifiers such as `%c`.

One common use for scan is for simple string parsing, as in the example above. Another common use is for converting ASCII characters to their integer values, which is done with the `%c` specifier. The procedure below uses this feature to return the character that follows a given character in lexicographic ordering:

```
proc next c {
    scan $c %c i
    format %c [expr $i+1]
}
next a
⇒ b
next 9
⇒ :
```

The `scan` command converts the value of the `c` argument from an ASCII character to the integer used to represent that character, then the integer is incremented and converted back to an ASCII character again with the `format` command.

10.6 Extracting characters: string index and string range

The remaining string manipulation commands are all implemented as options of the `string` command. For example, `string index` extracts a character from a string:

```
string index "Sample string" 3
⇒ p
```

The argument after `index` is a string and the last argument gives the index of the desired character in the string. An index of `0` selects the first character.

The `string range` command is similar to `string index` except that it takes two indices and returns all the characters from the first index to the second, inclusive:

```
string range "Sample string" 3 7
⇒ ple s
```

The second index may have the value `end` to select all the characters up to the end of the string:

```
string range "Sample string" 3 end
⇒ ple string
```

10.7 Searching and comparison

The command `string first` takes two additional string arguments as in the following example:

```
string first th "There is the tub where I bathed today"
⇒ 3
```

It searches the second string to see if there is a substring that is identical to the first string. If so then it returns the index of the first character in the leftmost matching substring; if not then it returns `-1`. The command `string last` is similar except it returns the starting index of the rightmost matching substring:

```
string last th "There is the tub where I bathed today"
⇒ 21
```

The command `string compare` takes two additional arguments and compares them in their entirety. It returns `0` if the strings are identical, `-1` if the first string sorts before the second, and `1` if the first string is after the second in sorting order:

```
string compare Michigan Minnesota
⇒ -1
string compare Michigan Michigan
⇒ 0
```

10.8 Length, case conversion, and trimming

The `string length` command counts the number of characters in a string and returns that number:

```
string length "sample string"
⇒ 13
```

The `string toupper` command converts all lower-case characters in a string to upper case, and the `string tolower` command converts all upper-case characters in its argument to lower-case:

```
string toupper "Watch out!"
⇒ WATCH OUT!
string tolower "15 Charing Cross Road"
⇒ 15 charing cross road
```

The `string` command provides three options for trimming: `trim`, `trimleft`, and `trimright`. Each option takes two additional arguments: a string to trim and an optional set of trim characters. The `string trim` command removes all instances of the trim characters from both the beginning and end of its argument string, returning the trimmed string as result:

```
string trim aaxxxbab abc
⇒ xxx
```

The `trimleft` and `trimright` options work in the same way except that they only remove the trim characters from the beginning or end of the string, respectively. The trim commands are most commonly used to remove excess white space; if no trim characters are specified then they default to the white space characters (space, tab, newline, and carriage return).

Chapter 11
Accessing Files

This chapter describes Tcl's commands for dealing with files. The commands allow you to read and write files sequentially or in a random-access fashion. They also allow you to retrieve information kept by the system about files, such as the time of last access. Lastly, they can be used to manipulate file names; for example, you can remove the extension from a file name or find the names of all files that match a particular pattern. See Table 11.1 for a summary of the file-related commands.

Note: *The commands described in this chapter are only available on systems that support the kernel calls defined in the POSIX standard, such as most UNIX workstations. If you are using Tcl on another system, such as a Macintosh or a PC, then the file commands may not be present and there may be other commands that provide similar functionality for your system.*

11.1 File names

File names are specified to Tcl using the normal UNIX syntax. For example, the file name `x/y/z` refers to a file named `z` that is located in a directory named `y`, which in turn is located in a directory named `x`, which must be in the current working directory. The file name `/top` refers to a file `top` in the root directory. You can also use tilde notation to specify a file name relative to a particular user's home directory. For example, the name `~ouster/mbox` refers to a file named `mbox` in the home directory of user `ouster`, and `~/mbox` refers to a file named `mbox` in the home directory of the user running the Tcl script. These conventions (and the availability of tilde notation in particular) apply to all Tcl commands that take file names as arguments.

`cd ?dirName?`	Changes the current working directory to *dirName*, or to the home directory (as given by the `HOME` environment variable) if *dirName* isn't given. Returns an empty string.
`close ?fileId?`	Closes the file given by *fileId*. Returns an empty string.
`eof fileId`	Returns `1` if an end-of-file condition has occurred on *fileId*, `0` otherwise.
`file option name ?arg arg ...?`	Performs one of several operations on the filename given by *name* or on the file that it refers to, depending on *option*. See Table 11.3 for details.
`flush fileId`	Writes out any buffered output that has been generated for *fileId*. Returns an empty string.
`gets fileId ?varName?`	Reads the next line from *fileId* and discards its terminating newline. If *varName* is specified, places the line in that variable and returns a count of characters in the line (or `-1` for end of file). If *varName* isn't specified, returns line as result (or an empty string for end of file).
`glob ?-nocomplain? ?--? pattern ?pattern ...?`	Returns a list of the names of all files that match any of the *pattern* arguments (special characters `?`, `*`, `[]`, `{}`, and `\`). If `-nocomplain` isn't specified then an error occurs if the return list would be empty.
`open name ?access?`	Opens file *name* in the mode given by *access*. Access may be `r`, `r+`, `w`, `w+`, `a`, or `a+` or a list of flags such as `RDONLY`; it defaults to `r`. Returns a file identifier for use in other commands like `gets` and `close`. If the first character of *name* is "|" then a command pipeline is invoked instead of opening a file (see Section 12.2 for more information).
`puts ?-nonewline? ?fileId? string`	Writes *string* to *fileId*, appending a newline character unless `-nonewline` is specified. *FileId* defaults to `stdout`. Returns an empty string.
`pwd`	Returns the full path name of the current working directory.

Table 11.1. A summary of the Tcl commands for manipulating files (continued in Table 11.2).

`read ?-nonewline? `*`fileId`*	
	Reads and returns all of the bytes remaining in *`fileId`*. If `-nonewline` is specified then the final newline, if any, is dropped.
`read `*`fileId numBytes`*	
	Reads and returns the next *`numBytes`* bytes from *`fileId`* (or up to the end of the file, if fewer than *`numBytes`* bytes are left).
`seek `*`fileId offset`*` ?`*`origin`*`?`	
	Position *`fileId`* so that the next access starts at *`offset`* bytes from *`origin`*. *`Origin`* may be `start`, `current`, or `end`, and defaults to `start`. Returns an empty string.
`tell `*`fileId`*	
	Returns the current access position for *`fileId`*.

Table 11.2. A summary of the Tcl commands for manipulating files, cont'd.

11.2 Basic file I/O

The Tcl commands for file I/O are similar to the procedures in the C standard I/O library, both in their names and in their behavior. Here is a script called `tgrep` that illustrates most of the basic features of file I/O:

```
#!/usr/local/bin/tclsh
if {$argc != 2} {
    error "Usage: tgrep pattern fileName"
}
set f [open [lindex $argv 1] r]
set pat [lindex $argv 0]
while {[gets $f line] >= 0} {
    if [regexp $pat $line] {
        puts stdout $line
    }
}
close $f
```

This script behaves much like the UNIX `grep` program: you can invoke it from your shell with two arguments, a regular expression pattern and a file name, and it will print out all of the lines in the file that match the pattern.

When `tclsh` processes evaluates the script it makes the command-line arguments available as a list in variable `argv`, with the length of that list in variable `argc`. After making sure that it received enough arguments, the script invokes the `open` command on the file to search, which is the second argument. `Open` takes two arguments, the name of a file and an access mode. The access mode provides information such as whether you'll be reading the file or writing it, and whether you want to append to the file or access it from the beginning. The access mode may have one of the following values:

`r`	Open for reading only. The file must already exist. This is the default if the access mode isn't specified.
`r+`	Open for reading and writing; the file must already exist.
`w`	Open for writing only. Truncate the file if it already exists, otherwise create a new empty file.
`w+`	Open for reading and writing. Truncate the file if it already exists, otherwise create a new empty file.
`a`	Open for writing only and set the initial access position to the end of the file. If the file doesn't exist then create a new empty file.
`a+`	Open the file for reading and writing and set the initial access position to the end of the file. If the file doesn't exist then create a new empty file.

The access mode may also be specified as a list of POSIX flags like `RDONLY`, `CREAT`, and `TRUNC`. See the reference documentation for more information about these flags.

The `open` command returns a string such as `file3` that identifies the open file. This *file identifier* is used when invoking other commands to manipulate the open file, such as `gets`, `puts`, and `close`. Normally you will save the file identifier in a variable when you open a file and then use that variable to refer to the open file. You should not expect the identifiers returned by `open` to have any particular format.

Three file identifiers have well-defined names and are always available to you, even if you haven't explicitly opened any files. These are `stdin`, `stdout`, and `stderr`; they refer to the standard input, output, and error channels for the process in which the Tcl script is executing.

After opening the file to search, the `tgrep` script reads the file one line at a time with the `gets` command. `Gets` normally takes two arguments: a file identifier and the name of a variable. It reads the next line from the open file, discards the terminating newline character, stores the line in the named variable, and returns a count of the number of characters stored into the variable. If the end of the file is reached before reading any characters then `gets` stores an empty string in the variable and returns `-1`.

Note: Tcl also provides a second form of `gets` *where the line is returned as the result of the command, and a command* `read` *for non-line-oriented input.*

For each line in the file the `tgrep` script matches the line against the pattern and prints it using `puts` if it matches. The `puts` command takes two arguments, which are a file identifier and a string to print. `Puts` adds a newline character to the string and outputs the line on the given file. The script uses `stdout` as the file identifier so the line is printed on standard output.

When `tgrep` reaches the end of the file `gets` will return `-1`, which ends the `while` loop. The script then closes the file with the `close` command; this releases the resources associated with the open file. In most systems there is a limit on how many files may be open at one time in an application, so it is important to close files as soon as you are finished reading or writing them. In this example the close is unnecessary, since the file will be closed automatically when the application exits.

11.3 Output buffering

The `puts` command uses the buffering scheme of the C standard I/O library. This means that information passed to `puts` may not appear immediately in the target file. In many cases (particularly if the file isn't a terminal device) output will be saved in the application's memory until a large amount of data has accumulated for the file, at which point all of the data will be written out in a single operation. If you need for data to appear in a file immediately then you should invoke the `flush` command:

```
flush $f
```

The `flush` command takes a file identifier as its argument and forces any buffered output data for that file to be written to the file. `Flush` doesn't return until the data has been written. Buffered data is also flushed when a file is closed.

11.4 Random access to files

File I/O is sequential by default: each `gets` or `read` command returns the next bytes after the previous `gets` or `read` command, and each `puts` command writes its data immediately following the data written by the previous `puts` command. However, you can use the `seek`, `tell`, and `eof` commands to access files non-sequentially.

Each open file has an *access position*, which is the location in the file where the next read or write will occur. When a file is opened the access position is set to the beginning or end of the file, depending on the access mode you specified to `open`. After each read or write operation the access position increments by the number of bytes transferred. The `seek` command may be used to change the current access position. In its simplest form `seek` takes two arguments, which are a file identifier and an integer offset within the file. For example, the command

```
seek $f 2000
```

changes the access position for the file so that the next read or write will start at byte number 2000 in the file.

`Seek` can also take a third argument that specifies an origin for the offset. The third argument must be either `start`, `current`, or `end`. `Start` produces the same effect as if the argument is omitted: the offset is measured relative to the start of the file. `Current` means that the offset is measured relative to the file's current access position, and `end` means that the offset is measured relative to the end of the file. For example, the following command sets the access position to 100 bytes before the end of the file:

```
seek $f -100 end
```

If the origin is `current` or `end` then the offset may be either positive or negative; for `start` the offset must be positive.

Note: It is possible to seek past the current end of the file, in which case the file may contain a hole. Check the documentation for your operating system for more information on what this means.

The `tell` command returns the current access position for a particular file identifier:

```
tell $f
⇒ 186
```

This allows you to record a position and return to that position later on.

The `eof` command takes a file identifier as argument and returns `0` or `1` to indicate whether the most recent `gets` or `read` command for the file attempted to read past the end of the file:

```
eof $f
⇒ 0
```

11.5 The current working directory

Tcl provides two commands that help to manage the current working directory: `pwd` and `cd`. `Pwd` takes no arguments and returns the full path name of the current working directory. `Cd` takes a single argument and changes the current working directory to the value of that argument. If `cd` is invoked with no arguments then it changes the current working directory to the home directory of the user running the Tcl script (`cd` uses the value of the `HOME` environment variable as the path name of the home directory).

11.6 Manipulating file names: glob and file

Tcl has two commands for manipulating file *names* as opposed to file contents: `glob` and `file`. The `glob` command takes one or more patterns as arguments and returns a list of all the file names that match the pattern(s):

```
glob *.c *.h
⇒ main.c hash.c hash.h
```

`Glob` uses the matching rules of the `string match` command (see Section 10.1). In the above example `glob` returns the names of all files in the current directory that end in `.c` or `.h`. `Glob` also allows patterns to contain comma-separated lists of alternatives between braces, as in the following example:

```
glob {{src,backup}/*.[ch]}
⇒ src/main.c src/hash.c src/hash.h backup/hash.c
```

Glob treats this pattern as if it were actually multiple patterns, one containing each of the strings, as in the following example:

```
glob {src/*.[ch]} {backup/*.[ch]}
```

Note: The extra braces around the patterns in these examples are needed to keep the brackets inside the patterns from triggering command substitution. They are removed by the Tcl parser in the usual fashion before invoking the command procedure for `glob`.

If a `glob` pattern ends in a slash then it only matches the names of directories. For example, the command

```
glob */
```

will return a list of all the subdirectories of the current directory.

If the list of file names to be returned by `glob` is empty then it normally generates an error. However, if the first argument to `glob`, before any patterns, is `-nocomplain` then `glob` will not generate an error if its result is an empty list.

The second command for manipulaing file names is `file`. `File` is a general-purpose command with many options that can be used both to manipulate file names and also to retrieve information about files. See Tables 11.3 and 11.4 for a summary of the options to `file`. This section discusses the name-related options and Section 11.7 describes the other options.The commands in this section operate purely on file names. They make no system calls and do not check to see if the names actually correspond to files.

`File dirname` returns the name of the directory containing a particular file:

```
file dirname /a/b/c
⇒ /a/b
file dirname main.c
⇒ .
```

`File extension` returns the extension for a file name (all the characters starting with the last `.` in the name), or an empty string if the name contains no extension:

```
file extension src/main.c
⇒ .c
```

`File rootname` returns everything in a file name except the extension:

```
file rootname src/main.c
⇒ src/main
file rootname foo
⇒ foo
```

Lastly, `file tail` returns the last element in a file's path name (i.e. the name of the file within its directory):

```
file tail /a/b/c
⇒ c
file tail foo
⇒ foo
```

`file atime` *`name`*	Returns a decimal string giving the time at which file *`name`* was last accessed, measured in seconds from 12:00 A.M. on January 1, 1970.
`file dirname` *`name`*	Returns all of the characters in *`name`* up to but not including the last `/` character. Returns `.` if *`name`* contains no slashes, `/` if the last slash in *`name`* is its first character.
`file executable` *`name`*	Returns `1` if *`name`* is executable by the current user, `0` otherwise.
`file exists` *`name`*	Returns `1` if *`name`* exists and the current user has search privilege for the directories leading to it, `0` otherwise.
`file extension` *`name`*	Returns all of the characters in *`name`* after and including the last dot. Returns an empty string if there is no dot in *`name`* or no dot after the last slash in *`name`*.
`file isdirectory` *`name`*	Returns `1` if *`name`* is a directory, `0` otherwise.
`file isfile` *`name`*	Returns `1` if *`name`* is an ordinary file, `0` otherwise.
`file lstat` *`name arrayName`*	Invokes the `lstat` system call on *`name`* and sets elements of *`arrayName`* to hold information returned by `lstat`. This option is identical to the `stat` option unless *`name`* refers to a symbolic link, in which case this command returns information about the link instead of the file it points to.
`file mtime` *`name`*	Returns a decimal string giving the time at which file *`name`* was last modified, measured in seconds from 12:00 A.M. on January 1, 1970.
`file owned` *`name`*	Returns `1` if *`name`* is owned by the current user, `0` otherwise.
`file readable` *`name`*	Returns `1` if *`name`* is readable by the current user, `0` otherwise.
`file readlink` *`name`*	Returns the value of the symbolic link given by *`name`* (the name of the file it points to).

Table 11.3. A summary of the options for the `file` command (continued in Table 11.4).

`file rootname` *`name`*	Returns all of the characters in *`name`* up to but not including the last `.` character. Returns *`name`* if it doesn't contain any dots or if it doesn't contain any dots after the last slash.
`file size` *`name`*	Returns a decimal string giving the size of file `name` in bytes.
`file stat` *`name arrayName`*	Invokes `stat` system call on *`name`* and sets elements of *`arrayName`* to hold information returned by `stat`. The following elements are set, each as a decimal string: `atime`, `ctime`, `dev`, `gid`, `ino`, `mode`, `mtime`, `nlink`, `size`, and `uid`.
`file tail` *`name`*	Returns all of the characters in *`name`* after the last `/` character. Returns *`name`* if it contains no slashes.
`file type` *`name`*	Returns a string giving the type of file *`name`*. The return value will be one of `file`, `directory`, `characterSpecial`, `blockSpecial`, `fifo`, `link`, or `socket`.
`file writable` *`name`*	Returns `1` if *`name`* is writable by the current user, `0` otherwise.

Table 11.4. A summary of the options for the `file` command, cont'd.

11.7 File information commands

In addition to the options already discussed in Section 11.6 above, the `file` command provides many other options that can be used to retrieve information about files. Each of these options except `stat` and `lstat` has the form

```
file option name
```

where *`option`* specifies the information desired, such as `exists` or `readable` or `size`, and *`name`* is the name of the file. Table 11.3 summarizes all of the options for the `file` command.

The `exists`, `isfile`, `isdirectory`, and `type` options return information about the nature of a file. `File exists` returns `1` if there exists a file by the given name and `0` if there is no such file or the current user doesn't have search permission for the directories leading to it. `File isfile` returns `1` if the file is an ordinary disk file and `0` if it is something else, such as a directory or device file. `File isdirectory` returns `1` if the file is a directory and `0` otherwise. `File type` returns a string such as `file`, `directory`, or `socket` that identifies the file type.

The `readable`, `writable`, and `executable` options return `0` or `1` to indicate whether the current user is permitted to carry out the indicated action on the file. The `owned` option returns `1` if the current user is the file's owner and `0` otherwise.

The `size` option returns a decimal string giving the size of the file in bytes. `File mtime` returns the time when the file was last modified. The time value is returned in the standard POSIX form for times, namely an integer that counts the number of seconds since 12:00 A.M. on January 1, 1970. The `atime` option is similar to `mtime` except that it returns the time when the file was last accessed.

The `stat` option provides a simple way to get many pieces of information about a file at one time. This can be significantly faster than invoking `file` many times to get the pieces of information individually. `File stat` also provides additional information that isn't accessible with any other file options. It takes two additional arguments, which are the name of a file and the name of a variable, as in the following example:

```
file stat main.c info
```

In this case the name of the file is `main.c` and the variable name is `info`. The variable will be treated as an array and the following elements will be set, each as a decimal string:

`atime`	Time of last access.
`ctime`	Time of last status change.
`dev`	Identifier for device containing file.
`gid`	Identifier for the file's group.
`ino`	Serial number for the file within its device.
`mode`	Mode bits for file.
`mtime`	Time of last modification.
`nlink`	Number of links to file.
`size`	Size of file, in bytes.
`uid`	Identifier for the user that owns the file.

The `atime`, `mtime`, and `size` elements have the same values as produced by the corresponding `file` options discussed above. For more information on the other elements, refer to your system documentation for the `stat` system call; each of the elements is taken directly from the corresponding field of the structure returned by `stat`.

The `lstat` and `readlink` options are useful when dealing with symbolic links, and they can only be used on systems that support symbolic links. `File lstat` is identical to `file stat` for ordinary files, but when it is applied to a symbolic link it returns information about the symbolic link itself, whereas `file stat` will return information about the file the link points to. `File readlink` returns the contents of a symbolic link, i.e. the name of the file that it refers to; it may only be used on symbolic links. For all of the other `file` commands, if the name refers to a symbolic link then the command operates on the target of the link, not the link itself.

11.8 Errors in system calls

Most of the commands described in this chapter invoke calls on the operating system, and in many cases the system calls can return errors. This can happen, for example, if you invoke `open` or `file stat` on a file that doesn't exist, or if an I/O error occurs in reading a file. The Tcl commands detect these system call errors and in most cases the Tcl commands will return errors themselves. The error message will identify the error that occurred:

```
open bogus
```

∅ *couldn't open "bogus": no such file or directory*

When an error occurs in a system call Tcl also sets the `errorCode` variable to provide more precise information. You may find this information useful as part of error recovery so that, for example, you can determine exactly why the the file wasn't accessible (Was there no such file? Was it protected to prevent access? ...). If a system call error has occurred then `errorCode` will consist of a list with three elements:

```
set errorCode
```

⇒ *POSIX ENOENT {no such file or directory}*

The first element is always `POSIX` to indicate that the error occurred in a POSIX system call. The second element is the official name for the error (`ENOENT` in the above example). Refer to your system documentation or to the include file `errno.h` for a complete list of the error names for your system. These names adhere to the POSIX standard as much as possible. The third element is the error message that corresponds to the error. This string usually appears in the error message returned by the Tcl command. Tcl uses the standard list of error messages provided by your system, if there is one, and adheres to the POSIX standard as much as possible.

Chapter 12
Processes

Tcl provides several commands for dealing with processes. You can create new processes with the `exec` command, or you can create new processes with `open` and then use file I/O commands to communicate with them. You can access process identifiers with the `pid` command. You can read and write environment variables using the `env` variable and you can terminate the current process with the `exit` command. Like the file commands in Chapter 11, these commands are only available on systems that support POSIX kernel calls. Table 12.1 summarizes the commands related to process management.

12.1 Invoking subprocesses with exec

The `exec` command creates one or more subprocesses and waits until they complete before returning. For example,

```
exec rm main.o
```

executes `rm` as a subprocess, passes it the argument `main.o`, and returns after `rm` completes. The arguments to `exec` are similar to what you would type as a command line to a shell program such as `sh` or `csh`. The first argument to `exec` is the name of a program to execute and each additional argument forms one argument to that subprocess.

To execute a subprocess, `exec` looks for an executable file with a name equal to `exec`'s first argument. If the name contains a `/` or starts with `~` then `exec` checks the single file indicated by the name. Otherwise `exec` checks each of the directories in the `PATH` environment variable to see if the command name refers to an executable file in that directory. `Exec` uses the first executable that it finds.

`exec ?-keepnewline? ?--? arg ?arg ...?`	Executes command pipeline specified by *arg*'s using one or more subprocesses and returns the pipeline's standard output or an empty string if output is redirected (the trailing newline, if any, is dropped unless `-keepnewline` is specified). I/O redirection may be specified with <, <<, and > and several other forms and pipes may be specified with |. If the last *arg* is `&` then the pipeline is executed in background and the return value is a list of its process ids.
`exit ?code?`	Terminates process, returning *code* to parent as exit status. *Code* must be an integer. *Code* defaults to 0.
`open |command ?access?`	Treats *command* as a list with the same structure as arguments to `exec` and creates subprocess(es) to execute command(s). Depending on *access*, creates pipes for writing input to pipeline and reading output from it.
`pid ?fileId?`	If *fileId* is omitted, returns the process identifier for the current process. Otherwise returns a list of all the process ids in the pipeline associated with *fileId* (which must have been opened using |).

Table 12.1. A summary of Tcl commands for manipulating processes.

`Exec` collects all of the information written to standard output by the subprocess and returns that information as its result, as in the following example:

```
exec echo wc tcl.h
⇒      618    2641   21825 tcl.h
```

If the last character of output is a newline then `exec` removes the newline. This behavior may seem strange but it makes `exec` consistent with other Tcl commands,which don't normally terminate the last line of the result; you can retain the newline by specifying `-keepnewline` as the first argument to `exec`.

`Exec` supports I/O redirection in a fashion similar to the UNIX shells. For example, if one of the arguments to `exec` is "`>foo`" (or if there is a "`>`" argument followed by a "`foo`" argument), then output from the process is placed in file `foo` instead of returning to Tcl as `exec`'s result. In this case `exec`'s result will be an empty string. `Exec` also supports several other forms of output redirection, such as `>>` to append to a file, `>&` to redirect both standard output and standard error, and `2>` to redirect standard error independently from standard output.

Standard input may be redirected using either < or <<. The < form causes input to be taken from a file. In the << form the following argument is not a file name, but rather an immediate value to be passed to the subprocess as its standard input. The following command uses << to write data to a file:

```
exec cat << "test data" > foo
```

The string “`test input`” is passed to `cat` as its standard input; `cat` copies the string to its standard ouput, which has been redirected to file `foo`. If no input redirection is specified then the subprocess inherits the standard input channel from the Tcl application.

You can also invoke a pipeline of processes instead of a single process using |, as in the following example:

```
exec grep #include tclInt.h | wc
⇒       8      25     212
```

The `grep` program extracts all the lines containing the string “`#include`” from the file `tclInt.h`. These lines are then piped to the `wc` program, which computes the number of lines, words, and characters in the `grep` output and prints this information on its standard output. The `wc` output is returned as the result of `exec`.

If the last argument to `exec` is `&` then the subprocess(es) will be executed in background. `Exec` will return immediately, without waiting for the subprocesses to complete. Its return value will be a list containing the process identifiers for all of the processes in the pipeline; standard output from the subprocesses will go to the standard output of Tcl application unless redirected. No errors will be reported for abnormal exits or standard error output, and standard error for the subprocesses will be directed to the standard error channel of the Tcl application.

If a subprocess is suspended or exits abnormally (i.e., it is killed or returns a non-zero exit status), or if it generates output on its standard error channel and standard error was not redirected, then `exec` returns an error. The error message will consist of the output generated by the last subprocess (unless it was redirected with >), followed by an error message for each process that exited abnormally, followed by the information generated on standard error by the processes, if any. In addition, `exec` will set the `errorCode` variable to hold information about the last process that terminated abnormally, if any (see the reference documentation for details).

Note: *Many UNIX programs are careless about the exit status that they return. If you invoke such a program with* `exec` *and it accidentally returns a non-zero status then the* `exec` *command will generate a false error. To prevent these errors from aborting your scripts, invoke* `exec` *inside a* `catch` *command.*

Although `exec`’s features are similar to those of the UNIX shells there is one important difference: `exec` does not perform any file name expansion. For example, suppose you invoke the following command with the goal of removing all `.o` files in the current directory:

```
exec rm *.o
∅ rm: *.o nonexistent
```

Rm receives "`*.o`" as its argument and exits with an error when it cannot find a file by this name. If you want file name expansion to occur you can use the `glob` command to get it, but not in the obvious way. For example, the following command will not work:

```
exec rm [glob *.o]
∅ rm: a.o b.o nonexistent
```

This fails because the list of file names that `glob` returns is passed to `rm` as a single argument. If, for example, there exist two `.o` files, `a.o` and `b.o`, then rm's argument will be "`a.o b.o`"; since there is no file by that name `rm` will return an error. The solution to this problem is the one described in Section 7.5: use `eval` to reparse the `glob` output so that it gets divided into multiple words. For example, the following command will do the trick:

```
eval exec rm [glob *.o]
```

In this case `eval` concatenates its arguments to produce the string

```
exec rm a.o b.o
```

which it then evaluates as a Tcl script. The names `a.o` and `b.o` are passed to `rm` as separate arguments and the files are deleted as expected.

12.2 I/O to and from a command pipeline

You can also create subprocesses using the `open` command; once you've done this you can then use commands like `gets` and `puts` to interact with the pipeline. Here are two simple examples:

```
set f1 [open {|tbl | ditroff -ms} w]
set f2 [open |prog r+}
```

If the first character of the "file name" passed to `open` is the pipe symbol | then the argument isn't really a file name at all. Instead, it specifies a command pipeline. The remainder of the argument after the | is treated as a list whose elements have exactly the same meaning as the arguments to the `exec` command. `Open` will create a pipeline of subprocesses just as for `exec` and it will return an identifier that you can use to transfer data to and from the pipeline. In the first example the pipeline is opened for writing, so a pipe is used for standard input to the `tbl` process and you can invoke `puts` to write data on that pipe; the output from `tbl` goes to `ditroff`, and the output from `ditroff` goes to the standard output of the Tcl application. The second example opens a pipeline for both reading and writing so separate pipes are created for `prog`'s standard input and standard output. Commands like `puts` can be used to write data to `prog` and commands like `gets` can be used to read the output from `prog`.

Note: When writing data to a pipeline, don't forget that output is buffered: it probably will not actually be sent to the child process until you invoke the `flush` command to force the buffered data to be written.

When you close a file identifier that corresponds to a command pipeline, the `close` command flushes any buffered output to the pipeline, closes the pipes leading to and from the pipeline, if any, and waits for all of the processes in the pipeline to exit. If any of the processes exit abnormally then `close` returns an error in the same way as `exec`.

12.3 Process ids

Tcl provides three ways that you can access process identifiers. First, if you invoke a pipeline in background using `exec` then `exec` returns a list containing the process identifiers for all of the subprocesses in the pipeline. You can use these identifers, for example, if you wish to kill the processes. Second, you can invoke the `pid` command with no arguments and it will return the process identifier for the current process. Third, you can invoke `pid` with a file identifier as argument, as in the following example:

```
set f [open {| tbl | ditroff -ms} w]
pid $f
⇒ 7189 7190
```

If there is a pipeline corresponding to the open file, as in the example, then the `pid` command will return a list of identifiers for the processes in the pipeline.

12.4 Environment variables

Environment variables can be read and written using the standard Tcl variable mechanism. The array variable `env` contains all of the environment variables as elements, with the name of the element in `env` corresponding to the name of the environment variable. If you modify the `env` array, the changes will be reflected in the process's environment variables and the new values will also be passed to any child process created with `exec` or `open`.

12.5 Terminating the Tcl process with exit

If you invoke the `exit` command then it will terminate the process in which the command was executed. `Exit` takes an optional integer argument. If this argument is provided then it is used as the exit status to return to the parent process. `0` indicates a normal exit and non-zero values correspond to abnormal exits; values other than `0` and `1` are rare. If no argument is given to `exit` then it exits with a status of `0`. Since `exit` terminates the process, it doesn't have any return value.

Chapter 13
Managing Tcl Internals

This chapter describes a collection of commands that allow you to query and manipulate the internal state of the Tcl interpreter. For example, you can use these commands to see if a variable exists, to find out what entries are defined in an array, to monitor all accesses to a variable, to rename or delete a command, or to handle references to undefined commands. Tables 13.1 and 13.2 summarize the commands.

13.1 Querying the elements of an array

The `array` command provides information about the elements currently defined for an array variable. It provides this information in several different ways, depending on the first argument passed to it. The command `array size` returns a decimal string indicating how many elements are defined for a given array variable and the command `array names` returns a list whose entries are the names of the elements of a given array variable:

```
set currency(France) franc
set "currency(Great Britain)" pound
set currency(Germany) mark
array size currency
⇒ 3
array names currency
⇒ {Great Britain} France Germany
```

For each of these commands the final argument must be the name of an array variable. The list returned by `array names` does not have any particular order.

`array anymore` *name searchId*	Returns `1` if there are any more elements to process in search *searchId* of array *name*, `0` if all elements have already been returned.
`array donesearch` *name searchId*	Terminates search *searchId* of array *name* and discard any state associated with the search. Returns an empty string.
`array names` *name*	Returns a list containing the names of all the elements of array *name*.
`array nextelement` *name searchId*	Returns the name of the next element in search *searchId* of array *name*, or an empty string if all elements have already been returned in this search.
`array size` *name*	Returns a decimal string giving the number of elements in array *name*.
`array startsearch` *name*	Initializes a search through all of the elements of array *name*. Returns a search identifier that may be passed to `array nextelement`, `array anymore`, or `array donesearch`.
`auto_mkindex` *dir pattern*	Scans all of the files in diretory *dir* whose names match *pattern* (using the glob-style rules of `string match`) and generates a file `tclIndex` in *dir* that allows the files to be auto-loaded.
`info` *option* ?*arg arg* ...?	Returns information about the state of the Tcl interpreter. See Table 13.3.
`rename` *old new*	Renames command *old* to *new*, or deletes *old* if *new* is an empty string. Returns an empty string.
`time` *script* ?*count*?	Executes *script count* times and returns a string giving the average elapsed time per execution. *Count* defaults to 1.

Table 13.1. A summary of commands for manipulating Tcl's internal state (continued in Table 13.2).

The `array names` command can be used in conjunction with `foreach` to iterate through the elements of an array. For example, the code below deletes all elements of an array with values that are `0` or empty:

```
foreach i [array names a] {
    if {($a($i) == "") || ($a($i) == 0)} {
        unset a($i)
    }
}
```

`trace variable` *`name ops command`*	Establishes a trace on variable *`name`* such that *`command`* is invoked whenever one of the operations given by *`ops`* is performd on *`name`*. *`Ops`* must consist of one or more of the characters `r`, `w`, or `u`. Returns an empty string.
`trace vdelete` *`name ops command`*	If there exists a trace for variable *`name`* that has the operations and command given by *`ops`* and *`command`*, removes that trace so that its command will not be executed anymore. Returns an empty string.
`trace vinfo` *`name`*	Returns a list with one element for each trace currently set on variable *`name`*. Each element is a sub-list with two elements, which are the *`ops`* and *`command`* associated with that trace.
`unknown` *`cmd`* ?*`arg arg`* ...?	This command is invoked by the Tcl interpreter whenever an unknown command name is encountered. *`Cmd`* will be the unknown command name and the *`arg`*'s will be the fully-substituted arguments to the command. The result returned by `unknown` will be returned as the result of the unknown command.

Table 13.2. Commands for manipulating Tcl's internal state, cont'd.

Note: *The `array` command also provides a second way to search through the elements of an array, using the `startsearch`, `anymore`, `nextelement`, and `donesearch` options. This approach is more general than the `foreach` approach given above, and in some cases it is more efficient, but it is more verbose than the `foreach` approach and isn't needed very often. See the reference documentation for details.*

13.2 The info command

The `info` command provides information about the state of the interpreter. It has more than a dozen options, which are summarized in Tables 13.3 and 13.4.

13.2.1 Information about variables

Several of the `info` options provide information about variables. `Info exists` returns a `0` or `1` value indicating whether or not there exists a variable with a given name:

```
set x 24
info exists x
⇒ 1
```

`info args` *`procName`*	Returns a list whose elements are the names of the arguments to procedure *`procName`*, in order.
`info body` *`procName`*	Returns the body of procedure *`procName`*.
`info cmdcount`	Returns a count of the total number of Tcl commands that have been executed in this interpreter.
`info commands` ?*`pattern`*?	Returns a list of all the commands defined for this interpreter, including built-in commands, application-defined commands, and procedures. If *`pattern`* is specified then only the command names matching *`pattern`* are returned (`string match`'s rules are used for matching).
`info default` *`procName argName varName`*	Checks to see if argument *`argName`* to procedure *`procName`* has a default value. If so, stores the default value in variable *`varName`* and returns `1`. Otherwise, returns `0` without modifying *`varName`*.
`info exists` *`varName`*	Returns `1` if there exists a variable named *`varName`* in the current context, `0` if no such variable is currently accessible.
`info globals` ?*`pattern`*?	Returns a list of all the global variables currently defined. If *`pattern`* is specified, then only the global variable names matching *`pattern`* are returned (`string match`'s rules are used for matching).
`info level` ?*`number`*?	If *`number`* isn't specified, returns a number giving the current stack level (`0` corresponds to top-level, `1` to the first level of procedure call, and so on). If *`number`* is specified, returns a list whose elements are the name and arguments for the procedure call at level *`number`*.
`info library`	Returns the full path name of the library directory in which standard Tcl scripts are stored.
`info locals` ?*`pattern`*?	Returns a list of all the local variables defined for the current procedure, or an empty string if no procedure is active. If *`pattern`* is specified then only the local variable names matching *`pattern`* are returned (`string match`'s rules are used for matching).

Table 13.3. A summary of the options for the `info` command (continued in Table 13.4).

`info procs ?pattern?`	Returns a list of the names of all procedures currently defined. If *pattern* is specified then only the procedure names matching *pattern* are returned (`string match`'s rules are used for matching).
`info script`	If a script file is currently being evaluated then this command returns the name of that file. Otherwise it returns an empty string.
`info tclversion`	Returns the version number for the Tcl interpreter in the form *major.minor*, where *major* and *minor* are each decimal integers. Increments in *minor* correspond to bug fixes, new features, and backwards-compatible changes. *Major* increments only when incompatible changes occur.
`info vars ?pattern?`	Returns a list of all the names of all variables that are currently accessible. If *pattern* is specified then only the variable names matching *pattern* are returned (`string match`'s rules are used for matching).

Table 13.4. A summary of the options for the `info` command, cont'd.

```
unset x
info exists x
```
⇒ *0*

The options `vars`, `globals`, and `locals` return lists of variable names that meet certain criteria. `Info vars` returns the names of all variables accessible at the current level of procedure call; `info globals` returns the names of all global variables, regardless of whether or not they are accessible; and `info locals` returns the names of local variables, including arguments to the current procedure, if any, but not global variables. In each of these commands an additional pattern argument may be supplied. If the pattern is supplied then only variable names matching that pattern (using the rules of `string match`) will be returned.

For example, suppose that global variables `global1` and `global2` have been defined and that the following procedure is being executed:

```
proc test {arg1 arg2} {
    global global1
    set local1 1
    set local2 2
    ...
}
```

Then the following commands might be executed in the procedure:

```
   info vars
⇒ global1 arg1 arg2 local2 local1
   info globals
⇒ global2 global1
   info locals
⇒ arg1 arg2 local2 local1
   info vars *al*
⇒ global1 local2 local1
```

13.2.2 Information about procedures

Another group of `info` options provides information about procedures. The command `info procs` returns a list of all the Tcl procedures that are currently defined. Like `info vars`, it takes an optional pattern argument that restricts the names returned to those that match a given pattern. `Info body`, `info args`, and `info default` return information about the definition of a procedure:

```
   proc maybePrint {a b {c 24}} {
       if {$a < $b}{
           puts stdout "c is $c"
       }
   }
   info body maybePrint
⇒
       if {$a < $b} {
           puts stdout "c is $c"
       }

   info args maybePrint
⇒ a b c
   info default maybePrint a x
⇒ 0
   info default maybePrint c x
⇒ 1
   set x
⇒ 24
```

`Info body` returns the procedure's body exactly as it was specified to the `proc` command. `Info args` returns a list of the procedure's argument names, in the same order they were specified to `proc`. `Info default` returns information about an argument's default value. It takes three arguments: the name of a procedure, the name of an argument to that procedure, and the name of a variable. If the given argument has no default value (e.g. `a` in the above example), `info default` returns `0`. If the argument has a default value (`c` in the above example) then `info default` returns `1` and sets the variable to hold the default value for the argument.

As an example of how you might use the commands from the previous paragraph, here is a Tcl procedure that writes a Tcl script file. The script will contain Tcl code in the form of `proc` commands that recreate all of the procedures in the interpreter. The file can then be `source`'d in some other interpreter to duplicate the procedure state of the original interpreter. The procedure takes a single argument, which is the name of the file to write:

```
proc printProcs file {
    set f [open $file w]
    foreach proc [info procs] {
        set argList {}
        foreach arg [info args $proc] {
            if [info default $proc $arg default] {
                lappend argList [list $arg $default]
            } else {
                lappend argList $arg
            }
        }
        puts $f [list proc $proc $argList \
                [info body $proc]]
    }
    close $f
}
```

`Info` provides one other option related to procedures: `info level`. If `info level` is invoked with no additional arguments then it returns the current procedure invocation level: `0` if no procedure is currently active, `1` if the current procedure was called from top-level, and so on. If `info level` is given an additional argument, the argument indicates a procedure level and `info level` returns a list whose elements are the name and actual arguments for the procedure at that level. For example, the following procedure prints out the current call stack, showing the name and arguments for each active procedure:

```
proc printStack {} {
    set level [info level]
    for {set i 1} {$i < $level} {incr i} {
        puts "Level $i: [info level $i]"
    }
}
```

13.2.3 Information about commands

`Info commands` is similar to `info procs` except that it returns information about all existing commands, not just procedures. If invoked with no arguments, it returns a list of the names of all commands; if an argument is provided, then it is a pattern in the sense of `string match` and only command names matching that pattern will be returned.

The command `info cmdcount` returns a decimal string indicating how many commands have been executed in this Tcl interpreter. It may be useful during peformance tuning to see how many Tcl commands are being executed to carry out various functions.

The command `info script` indicates whether or not a script file is currently being processed. If so then the command returns the name of the innermost nested script file that is active. If there is no active script file then `info script` returns an empty string. This command is used for relatively obscure purposes such as disallowing command abbreviations in script files.

13.2.4 Tclversion and library

`Info tclversion` returns the version number for the Tcl interpreter in the form *`major.minor`*. Each of *`major`* and *`minor`* is a decimal string. If a new release of Tcl contains only backwards-compatible changes such as bug fixes and new features, then its minor version number increments and the major version number stays the same. If a new release contains changes that are not backwards-compatible, so that existing Tcl scripts or C code that invokes Tcl's library procedures will have to be modified, then the major version number increments and the minor version number resets to 0.

The command `info library` returns the full path name of the Tcl library directory. This directory is used to hold standard scripts used by Tcl, such as a default definition for the `unknown` procedure described in Section 13.6 below.

13.3 Timing command execution

The `time` command is used to measure the performance of Tcl scripts. It takes two arguments, a script and a repetition count:

```
time {set a xyz} 10000
⇒ 92 microseconds per iteration
```

`Time` will execute the given script the number of times given by the repetition count, divide the total elapsed time by the repetition count, and print out a message like the above one giving the average number of microseconds per iteration. The reason for the repetition count is that the clock resolution on most workstations is many milliseconds. Thus anything that takes less than tens or hundreds of milliseconds cannot be timed accurately. To make accurate timing measurements, I suggest experimenting with the repetition count until the total time for the `time` command is a few seconds.

13.4 Tracing operations on variables

The `trace` command allows you to monitor the usage of one or more Tcl variables. Such monitoring is called *tracing*. If a trace has been established on a variable then a Tcl command will be invoked whenever the variable is read or written or unset. Traces can be used for a variety of purposes:

- monitoring the variable's usage (e.g. by printing a message for each read or write operation)
- propagating changes in the variable to other parts of the system (e.g. to ensure that a particular widget always displays the picture of a person named in a given variable)
- restricting usage of the variable by rejecting certain operations (e.g. generate an error on any attempt to change the variable's value to anything other than a decimal string) or by overriding certain operations (e.g. recreate the variable whenever it is unset).

Here is a simple example that causes a message to be printed when either of two variables is modified:

```
trace variable color w pvar
trace variable a(length) w pvar
proc pvar {name element op} {
    if {$element != ""} {
        set name ${name}($element)
    }
    upvar $name x
    puts "Variable $name set to $x"
}
```

The first `trace` command arranges for procedure `pvar` to be invoked whenever variable `color` is written: `variable` specifies that a variable trace is being created, `color` gives the name of the variable, `w` specifies a set of operations to trace (any combination of `r` for read, `w` for write, and `u` for unset), and the last argument is a command to invoke. The second trace command sets up a trace for element `length` of array `a`.

Whenever `color` or `a(length)` is modified, Tcl will invoke `pvar` with three additional arguments, which are the variable's name, the variable's element name (if it is an array element, or an empty string otherwise), and an argument indicating what operation was actually invoked (`r` for read, `w` for write, or `u` for unset). For example, if the command "`set color purple`" is executed, Tcl will evaluate the command "`pvar color {} purple`" because of the trace. If "`set a(length) 108`" is invoked, the trace command "`pvar a length w`" will be evaluated.

The `pvar` procedure does three things. First, if the traced variable is an array element then `pvar` generates a complete name for the variable by combining the array name and the element name. Second, the procedure uses `upvar` to make the variable's value accessible inside the procedure as local variable `x`. Finally, it prints out the variable's name and value on standard output. For the two accesses in the previous paragraph the following messages will be printed:

```
Variable color set to purple
Variable a(length) set to 108
```

The example above set traces on individual variables. It's also possible to set a trace on an entire array, as with the command

```
trace variable a w pvar
```

where a is the name of an array variable. In this case pvar will be invoked whenever any element of a is modified.

Write traces are invoked after the variable's value has been modified but before returning the new value as the result of the write. The trace command can write a new value into the variable to override the value specified in the original write, and this value will be returned as the result of the traced write operation. Read traces are invoked just before the variable's result is read. The trace command can modify the variable to affect the result returned by the read operation. Tracing is temporarily disabled for a variable during the execution of read and write trace commands. This means that a trace command can access the variable without causing traces to be invoked recursively.

If a read or write trace returns an error of any sort then the traced operation is aborted. This can be used to implement read-only variables, for example. Here is a script that forces a variable to have a positive integer value and rejects any attempts to set the variable to a non-integer value:

```
trace variable size w forceInt
proc forceInt {name element op} {
    upvar $name x ${name}_old x_old
    if ![regexp {^[0-9]*$} $x] {
        set x $x_old
        error "value must be a postive integer"
    }
    set x_old $x
}
```

By the time the trace command is invoked the variable has already been modified, so if forceInt wants to reject a write it must restore the old value of the variable. To do this it keeps a shadow variable with a suffix "_old" to hold the previous value of the variable. If an illegal value is stored into the variable, forceInt restores the variable to its old value and generates an error:

```
  set size 47
⇒ 47
  set size red
∅ can't set "size": value must be a postive integer
  set size
⇒ 47
```

Note: The forceInt *procedure only works for simple variables, but it could be extended to handle array elements as well.*

It is legal to set a trace on a non-existent variable; the variable will continue to appear to be unset even though the trace exists. For example, you can set a read trace on an array and then use it to create new array elements automatically the first time they are read. Unsetting a variable will remove the variable and any traces associated with the variable, then invoke any unset traces for the variable. It is legal, and not unusual, for an unset trace to immediately re-establish itself on the same variable so that it can monitor the variable if it should be re-created in the future.

To delete a trace, invoke `trace vdelete` with the same arguments passed to `trace variable`. For example, the trace created on `color` above can be deleted with the following command:

```
trace vdelete color w pvar
```

If the arguments to `trace vdelete` don't match the information for any existing trace exactly then the command has no effect.

The command `trace vinfo` returns information about the traces currently set for a variable. It is invoked with an argument consisting of a variable name, as in the following example:

```
trace vinfo color
⇒ {w pvar}
```

The return value from `trace vinfo` is a list, each of whose elements describes one trace on the variable. Each element is itself a list with two elements, which give the operations traced and the command for the trace. The traces appear in the result list in the order they will be invoked. If the variable specified to `trace vinfo` is an element of an array, then only traces on that element will be returned; traces on the array as a whole will not be returned.

13.5 Renaming and deleting commands

The `rename` command can be used to change the command structure of an application. It takes two arguments:

```
rename old new
```

`Rename` does just what its name implies: it renames the command that used to have the name *`old`* so that it now has the name *`new`*. *`New`* must not already exist as a command when `rename` is invoked.

`Rename` can also be used to delete a command by invoking it with an empty string as the `new` name. For example, the following script disables file I/O from an application by deleting the relevant commands:

```
foreach cmd {open close read gets puts} {
    rename $cmd {}
}
```

Any Tcl command may be renamed or deleted, including the built-in commands as well as procedures and commands defined by an application. Renaming or deleting a built-in command is probably a bad idea in general, since it will break scripts that depend on the command, but in some situations it can be useful. For example, the `exit` command as defined by Tcl just exits the process immediately (see Section 12.5). If an application wants to have a chance to clean up its internal state before exiting, then it can create a "wrapper" around `exit` by redefining it:

```
rename exit exit.old
proc exit status {
    application-specific cleanup
    ...
    exit.old $status
}
```

In this example the `exit` command is renamed to `exit.old` and a new `exit` procedure is defined, which performs the cleanup required by the application and then calls the renamed command to exit the process. This allows existing scripts that call `exit` to be used without change while still giving the application an opportunity to clean up its state.

13.6 Unknown commands

The Tcl interpreter provides a special mechanism for dealing with unknown commands. If the interpreter discovers that the command name specified in a Tcl command doesn't exist, then it checks for the existence of a command named `unknown`. If there is such a command then the interpreter invokes `unknown` instead of the original command, passing the name and arguments for the non-existent command to `unknown`. For example, suppose that you type the following commands:

```
set x 24
createDatabase library $x
```

If there is no command named `createDatabase` then the following command is invoked:

```
unknown createDatabase library 24
```

Notice that substitutions are performed on the arguments to the original command before `unknown` is invoked. Each argument to `unknown` will consist of one fully-substituted word from the original command.

The `unknown` procedure can do anything it likes to carry out the actions of the command, and whatever it returns will be returned as the result of the original command. For example, the procedure below checks to see if the command name is an unambiguous abbreviation for an existing command; if so, it invokes the corresponding command:

```
proc unknown {name args} {
    set cmds [info commands $name*]
    if {[llength $cmds] != 1} {
        error "unknown command \"$name\""
    }
    uplevel $cmds $args
}
```

Note that when the command is re-invoked with an expanded name, it must be invoked using `uplevel` so that the command executes in the same variable context as the original command.

The Tcl script library includes a default version of `unknown` that peforms the following functions, in order:

1. If the command is a procedure that is defined in a library file, source the file to define the procedure, then re-invoke the command. This is called *auto-loading*; it is described in the next section.

2. If there exists a program with the name of the command, use the `exec` command to invoke the program. This feature is called *auto-exec*. For example, you can type "`ls`" as a command and `unknown` will invoke "`exec ls`" to list the contents of the current directory. If the command doesn't specify redirection then auto-exec will arrange for the command's standard input, standard output, and standard error to be redirected to the corresponding channels of the Tcl application. This is different than the normal behavior of `exec` but it allows interactive programs such as `more` and `vi` to be invoked directly from a Tcl application.

3. If the command name has one of several special forms such as "`!!`" then compute a new command using history substitution and invoke it. For example, the if the command is "`!!`" then the previous command is re-invoked. See Chapter 14 for more information on history substitution.

4. If the command name is a unique abbreviation for an existing command, then the abbreviated command name is expanded and the command is re-invoked.

The last three actions are intended as conveniences for interactive use, and they only occur if the command was invoked interactively. You should not depend on these features when writing scripts. For example, you should not try to use auto-exec in scripts: always use the `exec` command explicitly.

If you don't like the default behavior of the `unknown` procedure then you can write your own version or modify the library version to provide additional functions. If you don't want any special actions to be taken for unknown commands you can just delete the `unknown` procedure, in which case errors will occur whenever unknown commands are invoked.

13.7 Auto-loading

One of the most useful functions performed by the `unknown` procedure is *auto-loading*. Auto-loading allows you to write collections of Tcl procedures and place them in script files in library directories. You can then use these procedures in your Tcl applications without having to explicitly `source` the files that define them. You simply invoke the procedures. The first time that you invoke a library procedure it won't exist, so `unknown` will be called. `Unknown` will find the file that defines the procedure, source the file to define the procedure, and then re-invoke the original command. The next time the procedure is invoked it will exist so the auto-loading mechanism won't be triggered.

Auto-loading provides two benefits. First, it makes it easy to build up large libraries of useful procedures and use them in Tcl scripts. You need not know exactly which files to `source` to define which procedures, since the auto-loader takes care of that for you. The second benefit of auto-loading is efficiency. Without auto-loading an appliation must `source` all of its script files when it starts up. Auto-loading allows an application to start up without loading any script files at all; the files will be loaded later when their procedures are needed, and some files may never be loaded at all. Thus auto-loading reduces startup time and saves memory.

Using the auto-loader is straightforward and involves three steps. First, create a library as a set of script files in a single directory. Normally these files have names that end in "`.tcl`", for example `db.tcl` or `stretch.tcl`. Each file can contain any number of procedure definitions. I recommend keeping the files relatively small, with just a few related procedures in each file. In order for the auto-loader to handle the files properly, the `proc` command for each procedure definition must be at the left edge of a line, and it must be followed immediately by white space and the procedure's name on the same line. Other than this the format of the script files doesn't matter as long as they are valid Tcl scripts.

The second step is to build an index for the auto-loader. To do this, start up a Tcl application such as `tclsh` and invoke the `auto_mkindex` command as in the following example:

```
auto_mkindex . *.tcl
```

`Auto_mkindex` isn't a built-in command but rather a procedure in Tcl's script library. Its first argument is a directory name and the second argument is a glob-style pattern that selects one or more script files in the directory. `Auto_mkindex` scans all of the files whose names match the pattern and builds an index that indicates which procedures are defined in which files. It stores the index in a file called `tclIndex` in the directory. If you modify the files to add or delete procedures then you should regenerate the index.

The third step is to set the variable `auto_path` in the applications that wish to use the library. The `auto_path` variable contains a list of directory names. When the auto-loader is invoked it searches the directories in `auto_path` in order, looking in their `tclIndex` files for the desired procedure. If the same procedure is defined in several libraries then the auto-loader will use the one from the earliest directory in `auto_path`. Typically `auto_path` will be set as part of an application's startup script. For example, if an application uses a library in directory `/usr/local/tcl/lib/shapes` then it might include the following command in its startup script:

```
set auto_path \
        [linsert $auto_path 0 /usr/local/tcl/lib/shapes]
```

This will add `/usr/local/tcl/lib/shapes` to the beginning of the path, retaining all the existing directories in the path such as those for the Tcl and Tk script libraries but giving higher priority to procedures defined in `/usr/local/tcl/lib/shapes`. Once a directory has been properly indexed and added to `auto_path`, all of its procedures become available through auto-loading.

Chapter 14
History

This chapter describes Tcl's history mechanism. In applications where you type commands interactively, the history mechanism keeps track of recent commands and makes it easy for you to re-execute them without having to completely re-type them. You can also create new commands that are slight variations on old commands without having to completely retype the old commands, for example to fix typos. Tcl's history mechanism provides many of the features available in `csh`, but not with the same syntax in all cases. History is implemented by the `history` command, which is summarized in Table 14.1 . Only a few of the most commonly used history features are described in this chapter; see the reference documentation for more complete information.

14.1 The history list

Each command that you type interactively is entered into a *history list*. Each entry in the history list is called an *event*; it contains the text of a command plus a serial number identifying the command. The command text consists of exactly the characters you typed, before the Tcl parser peforms substitutions for `$`, `[]`, etc. The serial number starts out at `1` for the first command you type and is incremented for each successive command.

Suppose you type the following sequence of commands to an interactive Tcl program:

```
set x 24
set y [expr $x*2.6]
incr x
```

At this point the history list will contain three events. You can examine the contents of the history list by invoking `history` with no arguments:

`history`	Returns a string giving the event number and command for each event on the history list.
`history keep` *count*	Changes the size of the history list so that the *count* most recent events will be retained. The initial size of the list is 20 events.
`history nextid`	Returns the number of the next event that will be recorded in the history list.
`history redo` ?*event*?	Re-executes the command recorded for *event* and returns its result.
`history substitute` *old new* ?*event*?	Retrieve the command recorded for *event*, replace any occurrences of *old* by *new* in it, execute the resulting command, and returns its result. Both *old* and *new* are simple strings. The substitution uses simple equality checks: no wild cards or regular expression features are supported.

Table 14.1. A summary of some of the options for the `history` command. Several options have been omitted; see the reference documentation for details.

```
history
⇒     1 set x 24
      2 set y [expr $x*2.6]
      3 incr x
      4 history
```

The value returned by `history` is a human-readable string describing what's on the history list, which also includes the `history` command. The result of `history` is intended for printing out, not for processing in Tcl scripts; if you want to write scripts that process the history list, you'll probably find it more convenient to use other `history` options described later in the reference documentation, such as `history event`.

The history list has a fixed size, which is initially 20. If more commands than that have been typed then only the most recent commands will be retained. The size of the history list can be changed with the `history keep` command:

```
history keep 100
```

This command changes the size of the history list so that in the future the 100 most recent commands will be retained.

14.2 Specifying events

Several of the options of the `history` command require you to select an event from the history list; the symbol *`event`* is used for such arguments in Table 14.1. Events are specified as strings with one of the following forms:

Positive number:	Selects the event with that serial number.
Negative number:	Selects an event relative to the current event. `-1` refers to the last command, `-2` refers to the one before that, and so on.
Anything else:	Selects the most recent event that matches the string. The string matches an event either if it is the same as the first characters of the event's command, or if it matches the event's command using the matching rules for `string match`.

Suppose that you had just typed the three commands from page 131 above. The command "`incr x`" can be referred to as event `-1` or `3` or `inc`, and "`set y [expr $x*2.6]`" can be referred to as event `-2` or `2` or `*2*`. If an event specifier is omitted then it defaults to `-1`.

14.3 Re-executing commands from the history list

The `redo` and `substitute` options to `history` will replay commands from the history list. `History redo` retrieves a command and re-executes it just as if you had retyped the entire command. For example, after typing the three commands from page 131, the command

```
history redo
```

replays the most recent command, which is `incr x`; it will increment the value of variable `x` and return its new value (`26`). If an additional argument is provided for `history redo`, it selects an event as described in Section 14.2; for example,

```
history redo 1
⇒ 24
```

replays the first command, `set x 24`.

The `history substitute` command is similar to `history redo` except that it modifies the old command before replaying it. It is most commonly used to correct typographical errors:

```
set x "200 illimeters"
⇒ 200 illimeters
history substitute ill mill -1
⇒ 200 millimeters
```

`History substitute` takes three arguments: an old string, a new string, and an event specifier (the event specifier can be defaulted, in which case it defaults to `-1`). It retrieves the command indicated by the event specifier and replaces all instances of the old string in that command with the new string. The replacement is done using simple textual comparison with no wild-cards or pattern matching. Then the resulting command is executed and its result is returned.

14.4 Shortcuts implemented by unknown

The `history redo` and `history substitute` commands are quite bulky; in the examples above it took more keystrokes to type the `history` commands than to retype the commands being replayed. Fortunately there are several shortcuts that allow the same functions to be implemented with fewer keystrokes:

`!!`	Replays the last command: same as "`history redo`".
`!`*`event`*	Replays the command given by *`event`*; same as "`history redo` *`event`*".
`^`*`old`*`^`*`new`*	Replay the last command, substituting new for old; same as "`history substitute` *`old new`*".

All of these shortcuts are implemented by the `unknown` procedure described in Section 13.6. `Unknown` detects commands that have the forms described above and invokes the corresponding `history` commands to carry them out.

Note: If your system doesn't use the default version of `unknown` provided by Tcl then these shortcuts may not be available.

14.5 Current event number: history nextid

The command `history nextid` returns the number of the next event to be entered into the history list:

```
history nextid
⇒ 3
```

It is most commonly used for generating prompts that contain the event number. Many interactive applications allow you to specify a Tcl script to generate the prompt; in these applications you can include a `history nextid` command in the script so that your prompt includes the event number of the command you are about to type.