An Introduction To Tcl and Tk

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 1993 by Addison-Wesley. 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.

Table of Contents

Chapter 1
Introduction

This book is about two systems called Tcl and Tk that provide a simple yet powerful programming system for developing and using windowing applications. Tcl stands for "tool command language" and is pronounced "tickle." It is a simple interpretive programming language. Tcl is implemented as a library of C procedures, so it can be included in many different applications and can be used for many different purposes. Tk is a toolkit for the X11 window system. Its name is pronounced "tee-kay." Tk is also implemented as a library of C procedures so it too can be used as part of many different windowing applications. More importantly, Tk is implemented using Tcl: its facilities can be invoked using Tcl commands.

If an application is based on Tcl and Tk, then both its functionality and its user interface can be modified at run-time by writing short Tcl scripts. This allows users to personalize and extend existing applications without having to recompile them. Many new windowing applications can be created without writing any C code at all, just by writing short scripts for a windowing shell called `wish`, which contains Tcl and Tk. In the same way that a script for a shell program like `csh` can usually be written much more quickly than a C program that does the same thing, many simple windowing applications can be written more quickly as `wish` scripts than as C programs that do the same thing.

Even more important, Tcl and Tk make it easy for different applications to communicate with each other. Tk provides a special command called `send`, which allows any Tk-based application to send Tcl commands to any other Tk-based application. `Send` provides a much more powerful form of communication than the window system's selection, which is the only mechanism available in most of today's X11 applications. With `send`, hypertext and hypermedia applications become easy to build; spreadsheets can query databases for values; user-interface editors can modify the interfaces of live applications as they run; and many other similar things become possible. Tcl and Tk are intended to stimulate the development of *hypertools*: specialized applications that can be plugged together in a variety of interesting ways.

This book provides a complete explanation of both Tcl and Tk. It contains five major parts:

- **Part I** introduces the features of the Tcl language. After reading this section you will be able to issue commands to Tcl-based applications and write scripts to extend those 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 will be able to modify the interfaces of existing applications, create new applications by writing Tcl scripts for existing applications, and use `send` to make Tk-based applications work together.
- **Part III** describes how to write applications that use Tcl. It discusses the C procedures provided by the Tcl library and how to use them to build applications. After reading this section you will be able to write C code for new Tcl-based applications.
- **Part IV** describes the C library procedures provided by Tk. After reading this section you will be able to write new widgets and geometry managers in C.
- **Part V** contains reference documentation for Tcl and Tk. It describes both the Tcl commands and the C library procedures. Whereas the rest of the book is intended to be introductory in nature, this section is intended as a reference manual, so it is terse but complete.

This book is intended for people who will be scripting existing applications or writing new ones. It assumes that you already know the C programming language and that you have some experience with UNIX and with X11. You need not know anything about either Tcl or Tk before reading this book: both of them will be introduced from scratch.

The remainder of this chapter provides a more thorough overview of the philosophy and structure of Tcl and Tk.

1.1 The philosophy behind Tcl

Every computer application has a command language of some sort. It may be as simple as the options that can be specified on the shell command line, or it may be a graphical language consisting of menus and buttons and mouse clicks, or it may be a full-fledged programming language, but there must be some way for a user to tell the application what to do.

Larger and more powerful applications generally need to have more powerful and flexible command languages. This is because it is hard for an application designer to predict all of the ways the application will be used. If the command language is powerful enough, individual users can tailor the application to their needs. If a user needs a function that wasn't present in the original application, he or she may be able to create that function by writing a short program in the command language. A good command language allows an application to be used for many tasks never considered by the application's designers. This greatly increases the value of the application.

Command languages are particularly important for interactive windowing applications. Windowing applications tend to have rich user interfaces with many different ways the user can tell the application what to do. A user might invoke an operation by pulling down a menu entry, or by clicking on a button-like object, or by dragging an object on the screen, or by typing keystrokes. It's important for interactive applications to be configurable. "Power users" may wish to create new operations that save them time by executing a sequence of actions in response to a single keystroke or mouse movement. Or , a user may wish to re-arrange the application's appearance to fit his or her particular needs (e.g. a left-handed user might prefer to have scrollbars on the left side instead of the right). Users may also wish to connect different applications together so that they can work cooperatively. For example, a debugger application might use an editor application to display the current line of execution, or a spreadsheet application might wish to retrieve values from a database application. All of these functions require a mechanism for telling an application what to do: a command language.

Unfortunately, today's applications don't usually have good command languages. Where good languages exist, they tend to be tied to specific programs. Each new application requires a new command language to be developed. In most cases application programmers do not have the time or inclination to implement a general-purpose facility, particularly if the application itself is simple. As a result, command languages tend to have insufficient power and clumsy syntax. This makes applications hard to use and even harder to reconfigure or extend; it is difficult to use most applications for anything that wasn't explicitly planned by the application's designers.

The guiding philosophy for Tcl is that every application, no matter how simple, should have a powerful and flexible command language that can be used to control and extend the application. Figure 1.1 shows how Tcl achieves this goal. The Tcl language exists as a library of C procedures that can be included easily in any application. The library procedures implement an interpreter for a simple but fully programmable language; this language is called *the Tcl core*. The Tcl core provides a collection of commonly used features such as variables, conditional and looping commands, procedures, associative arrays, lists, expressions, and file manipulation.

Each application can extend the Tcl core by implementing new commands that are specific to that application. These application-specific commands are indistinguishable from the commands in the Tcl core, but they are implemented by C procedures that are part of the application rather than the Tcl core. With this approach, an application need only implement a few new commands that provide the primitives for that application. Then the commands in the Tcl core can be used to assemble the application-specific primitives into more complex and powerful operations. For example, an application for reading

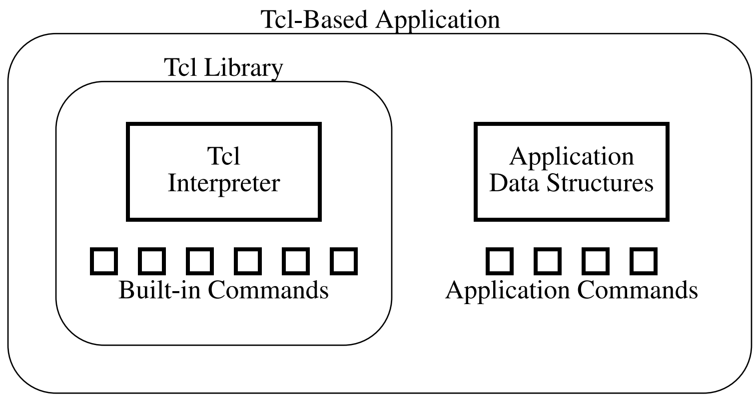

Figure 1.1. To create a new application based on Tcl, an application developer need only create the new data structures specific to that application, plus a few new Tcl commands to manipulate the data structures. The Tcl library provides everything else that is needed to produce a fully programmable command language.

electronic bulletin boards might provide a command to query a bulletin board for new messages and another command to retrieve a given message. Once these commands exist, Tcl scripts can be written to keep track of a collection of bulletin boards, or 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 wrtten as Tcl scripts, and users of the application could write additional Tcl scripts to add more functions to the application.

The Tcl approach has two advantages. First, Tcl makes it easy to build applications that have powerful command languages. Even the simplest application becomes fully programmable and extensible when it is built with Tcl. Second, Tcl makes it possible for the same language to be used in many different places, either to control different aspects of a single application or to control entirely different applications. This uniformity makes it easier for users since they can learn a single language and then be able to write scripts for many different applications. The uniformity also provides great power. If different parts of an application are all built as Tcl commands, then the different parts can work together by exchanging Tcl commands. If a mechanism is provided to exchange Tcl commands between applications (and Tk provides such a mechanism), then it becomes possible for groups of applications to work together in ways that wouldn't be possible if each application had a different command language.

1.2 The Tk toolkit

Tk is a toolkit for the X11 window system. It allows you to create user interfaces as collections of *widgets*, where each widget is a user-interface element such as a menu or scrollbar or text entry. Tk allows the widgets to be connected to the rest of the application so that actions on the widgets (such as invoking a menu entry or dragging the slider in a scrollbar) can cause things to happen in the application. Tk also provides mechanisms for arranging widgets into interesting groups of controls. The overall features of Tk are roughly similar to the features of other toolkits. What makes Tk unusual is that it is based on Tcl; Tk's features exist as a set of Tcl commands that supplement those in the Tcl core.

It might seem that a textual command language like Tcl is the wrong thing for a windowing environment, where most actions are invoked with the mouse and few users want to type textual commands. In fact, though, a textual language is extremely useful in this sort of environment; it just works behind the scenes. For example, menu entries and accelerator keys are bound to commands in the language: when a menu entry is invoked or an accelerator key is pressed, the corresponding command is invoked. The command language isn't visible in normal use but it provides flexibility and power nonetheless. A user can customize an application's interface by changing the commands associated with the elements of the user interface. A complex set of operations can be described with a script in the command language and then associated with a menu entry or accelerator key so that it can be invoked easily. Users can write scripts that are read automatically when an application starts up and reconfigure the application's interface to suit the user. And one application can control another by sending the target application a command in its command language.

The guiding philosophy for Tk is that all aspects of all interactive applications, including both their interfaces and their functions, should be controlled by a single interpretive command language. Since the language is interpretive, it can be used to modify any aspect of an application or its interface while the application is running. Entirely new applications can be created simply by writing Tcl scripts for a windowing shell based on Tcl and Tk; this allows application designers to work at a higher level of programming where new applications can be created more easily than if they had to be coded in C. The fact that a *single* language is used everywhere results in great power. It makes it easy to connect interface actions to application functions, and it makes it easy for an application to modify its interface while it runs. If many different application are all based on the same language then the benefit is even greater: users need only learn a single language and will then be able to personalize and extend all the applications; in addition, the applications can communicate directly with each other by sending commands back and forth in Tcl.

1.3 Reading this book

Most of this book (all but Part V) is intended to be introductory in nature. Each of Parts I-IV introduces one major aspect of Tcl and Tk, and the material is organized for smooth reading from start to finish within each part. Parts I-IV do not cover every feature of Tcl and Tk; instead, they focus on the major concepts and the philosophy of how to use Tcl and Tk.

Part V contains reference documentation. It is intended to be absolutely complete, but it is terse so it probably won't make sense until after you've read the corresponding material from Parts I-IV. Part V is organized for looking up individual pieces of information when you're building or modifying applications, rather than for reading from start to finish to learn the system.

Thus I recommend reading Parts I-IV to get a general feel for how things work, but I suggest that you refer to Part V whenever you have specific questions about any feature of Tcl or Tk.

Part I:

The Tcl Language

Chapter 2
Tcl Basics

Part I of this book is about the Tcl language. This chapter describes the basic language syntax. The other chapters in Part I describe the commands provided by the Tcl core, i.e. those commands that will be present in every Tcl-based application. Once you've read Part I you should be able to write scripts for existing Tcl-based applications.

Tcl has a simple syntax consisting of about a half-dozen rules that determine how commands are parsed. Control structures and other features that have special syntax in other languages are implemented as commands in Tcl, so they use the same simple syntax rules as all other commands. For example, `if` and `while` are implemented as commands in Tcl, and Tcl procedures are defined with a command named `proc`.

2.1 Simple commands

A Tcl command consists of one or more *words* separated by spaces or tabs. The first word is the name of the command and additional words (if present) are arguments to that command. Each command returns a string of zero or more characters as its result. For example, here are two simple commands:

```
set a 22
22
expr 4+6
10
```

In this example, as in all Tcl examples in this book, Tcl commands that you type are shown in a computer-like typeface and the results of commands are shown in an underlined typeface. Results may be omitted in examples if they are empty or unimportant to the example. The first command in the above example has three words: `set`, `a`, and `22`. It causes the `set` command to be invoked with two arguments, `a` and `22`. `Set` treats its first argument as the name of a variable and its second argument as a new value for that variable. It assigns the new value (`22`) to the variable (`a`) and returns the new value as result. The second command has two words, `expr` and `4+6`. It causes the `expr` command to be invoked with a single argument. `Expr` treats its argument as an arithmetic expression, evaluates that expression, and returns the value as a decimal string.

Each Tcl command is free to interpret its arguments in any way it pleases. For example, the `set` command expects its first argument to be the name of a variable while the `expr` command expects its first (and only) argument to be an arithmetic expression. It is possible to specify any string value as an argument for any command, but some commands expect their arguments to have particular forms. The `set` command allows either of its arguments to be an arbitrary string, whereas the `expr` command will generate an error if its argument isn't a proper expression.

Spaces and tabs are usually significant in commands since they act as word separators (later in this chapter you'll see how to prevent this effect). If the `expr` example had been typed as

```
expr 4 + 6
```

(with spaces on either side of the `+`) then the `expr` command would receive three arguments: `4`, `+`, and `6`. In this case `expr` would generate an error, since it expects to receive only a single argument. If there are multiple spaces or tabs in a row, they act together as a single word separator.

2.2 Command terminators

A Tcl script consists of one or more commands. Commands are normally separated by newline characters. For example,

```
set a abcd
set b efg
```

is a script with two commands separated by a newline. The first command sets the value of variable `a` and the second sets the value of variable `b`. Commands may also be separated by semi-colons; this allows multiple commands to be placed on a single line. For example, the script

```
set a abcd; set b efg
```

has the same effect as the preceding example. The newline and semi-colon characters are removed by the Tcl parser and are not included in the arguments passed to the commands.

There are times when you'll want to include newline and semi-colon characters as part of command words, and to do this you'll need to prevent them from being interpreted as command terminators. You'll see how to do this later in the chapter.

2.3 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, such as the case above where `expr` received more than one argument. When an error return occurs, it means that the command could not complete its intended function. The Tcl command is aborted and any commands that follow it in the script are skipped. An error return includes a string identifying what went wrong; this string is normally printed out for the user by the application.

The complete exceptional return mechanism for Tcl is discussed in Chapter 8. It includes a number of exceptional returns other than errors, provides additional information about errors other than 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.

2.4 Variable substitution

Tcl provides three forms of *substitution*, each of which causes the contents of a command word to be modified in some way. Substitutions may occur in any word of a command, including the command name. Tcl also provides mechanisms for preventing substitutions, which are described in Sections 2.6-2.8 below.

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 command word. For example, consider the following commands:

```
set a 8
set b $a
8
```

The first command sets the value of variable `a` to `8` (it returns the string `8`, which isn't shown). In the second command, the string `$a` causes variable substitution to occur. Instead of receiving `$a` as its second argument, the `set` command receives `8` (the value of variable `a`).

In the above example the variable was the only thing in the word where substitution occurred. However, it is also possible for variable substitution to affect only a part of a word, leaving the rest of the word unaffected. Here is a simple example:

```
set a 6
set b2 4
expr ($b2+2)*$a
36
```

In the `expr` command the string `$b2` is replaced with the value of variable `b2` and the string `$a` is replaced with the value of variable `a`, so that `expr` receives `(4+2)*6` as its argument. When variable substitution occurs, the variable name consists of everything after the dollar-sign up to the first character that isn't a number, letter, or underscore. In the example above, the first variable name ends just before the + character and the second variable name ends just before the newline that terminates the command. Any number of variable substitutions may occur within a single 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 permit characters other than numbers, letters, or digits in variable names. These other forms are discussed in Chapter 3.

2.5 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 returned by another Tcl command. Command substitution is invoked with square brackets:

```
set a 14
set a [expr $a+2]
16
```

When an open square bracket appears in a command word, the information following the open bracket must be a Tcl script followed by a close bracket. If the script contains more than one command, the commands are separated by newlines or semi-colons in the usual fashion. Thus in the example above the `expr` command is executed while parsing the words for `set`; when the set command is eventually executed, its second argument will be `16`.

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. The square brackets determine the range of characters replaced in each command substitution: the command for a given substitution ends at the matching close square bracket. A single word may contain both command and variable substitutions, and nested commands may themselves contain additional substitutions of any form, as in the following example:

```
set frac 2
set int 4
set num [expr $int+2].[expr $frac+1]
6.3
```

Command and variable substitutions are always performed in order from left to right.

If an error or other exceptional return occurs within a nested command, then the entire chain of partially evaluated commands is aborted. For example, if the last command above had been

```
set num [expr $int + 2].[expr $frac+1]
```

then the first `expr` command would return an error (the extra spaces around + result in too many arguments to `expr`) and neither the second `expr` command nor the enclosing `set` command would be executed.

2.6 Backslash substitution

The final form of substitution in Tcl is *backslash substitution*. It is used to prevent special interpretation of characters like `[` and `$` and space so that they can be inserted into words. For example, consider the following command:

```
set a 1\ 2\$\ 3\[
1 2$ 3[
```

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. The backslash followed by dollar-sign is replaced with a single dollar-sign (no variable substitution is triggered) and the backslash followed by open square bracket is replaced in the word with the open square bracket (no command substitution is performed). Any character that has special interpretation in Tcl, including backslash, can be backslashed to prevent that special interpretation. This includes both the special characters discussed so far and those to be discussed in the remainder of this chapter.

Backslash substitution can also be used to insert non-printing characters into words. For example, `\n` is replaced with a newline character and `\b` is replaced with a backspace character. Tcl supports all of the backslash sequences supported by the C compiler for strings. See Table 2.1 for a complete listing of the backslash sequences supported by Tcl.

The sequence backslash-newline has special significance. When the last character on a line is a backslash then both the backslash and the following newline are ignored; the result is to join the line containing the sequence to the line following it, preventing the newline character from acting as a command terminator. For example, the script

```
set a A\ very\ \
long\ string
A very long string
```

is identical in effect to the command

```
set a A\ very\ long\ string
A very long string
```

Unlike other backslash sequences, backslash-newline is replaced by nothing. Backslash-newline is also special in that it is handled even when it occurs between braces, which are described in Section 2.8.

Backslash Sequence	Replaced By
`\b`	Backspace (0x8)
`\t`	Tab (0x9)
`\e`	Escape (0x1b)
`\n`	Newline (0xa)
`\r`	Carriage-return (0xd)
`\{`	Left brace (“`{`“)
`\}`	Right brace (“`}`”)
`\[`	Open bracket (“`[`“)
`\]`	Close bracket (“`]`”)
`\$`	Dollar sign (“`$`”)
`\`*`space`*	Space (“ ”)
`\;`	Semi-colon
`\"`	Double-quote (0x22)
`\\`	Backslash (“\”)
`\`*`newline`*	Nothing
`\`*`ddd`*	Octal value given by *`ddd`*

Table 2.1. Backslash substitutions supported by Tcl. When one of the given backslash sequences appears in a word of a Tcl command, the sequence is replaced by the corresponding string in the right column. The terms *`space`* and *`newline`* refer to the space and newline characters. *`ddd`* refers to any 1, 2, or 3 octal digits.

If a backslash is followed by one of the characters not in Table 2.1, then the backslash receives no special treatment: both the backslash and the following character will appear in the word.

2.7 Quoting with double-quotes

In addition to the substitutions described in the previous sections, Tcl supports two forms of *quoting*. When a word of a command is quoted then some or all of the special characters lose their special meaning: they are passed through to the command just like other characters. Tcl provides two forms of quoting: double-quotes and curly braces. Double-quotes only disable a few of the special characters, while curly braces disable almost all special characters.

If the first character of a word is a double-quote character then the word consists of everything after the double-quote up to the next double-quote character. Within the word, neither spaces, tabs, newlines, or semi-colons have special interpretation; they are treated just like other characters. Double-quotes provide a convenient way to specify words that contain white space without having to type lots of unsightly backslashes. For example, the following command sets variable `a` to a value containing several spaces:

```
set a "A long string with spaces"
A long string with spaces
```

Notice that the quotes themselves are not passed through to the command in the argument word.

Variable substitutions, command substitutions, and backslash substitutions are still performed within double-quotes, as in the following example:

```
set a 24
set b "if a is $a then a+4 is [expr $a+4]"
if a is 24 then a+4 is 28
```

To include a double-quote within a double-quoted word, use backslash substitution:

```
set a "word contains \" char."
word contains " char.
```

A double-quote character only has special interpretation when it is the first character of a word. If the first character of a word isn't a double-quote then double-quotes are treated like ordinary characters within that word. Thus the following example generates an error because it results in three arguments for the `set` command:

```
set a two" words"
```

In this case the three arguments are `a` and `two"` and `words"`.

2.8 Quoting with curly braces

Curly braces provide a more radical form of quoting. If the first character of a word is an open curly brace, then the word consists of everything up to the matching close curly brace (not including the braces themselves). There may be nested curly braces within the word. Within the word no substitutions or special interpretations occur whatsoever except that (a) backslashed curly braces are not considered in the search for the closing brace and (b) backslash-newline substitutions are made as described in Section 2.6. Curly braces provide a convenient way to specify arguments that contain characters like `$` and `[` without having to type backslashes.

Braces are most commonly used for lists and nested commands. For example, the following command sets variable `a` to a list containing three elements of which the middle element is itself a list with two elements:

```
set a {a {b c} d}
a {b c} d
```

Lists are discussed in detail in Chapter 5. The second common use for curly braces is specifying a Tcl program as an argument to a command. This is used for control structures like `if` and `while`, as in the following example:

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

This program computes the factorial of 5, leaving the value in variable `result`. The `while` command receives two arguments: `$i>0` and everything between the curly braces (an initial newline, two commands separated by newlines, and a final newline). The `while` command evaluates its first argument as an expression and if the result is non-zero then it executes its second argument as a nested Tcl script and repeats this process over and over until the first argument evaluates to zero. For this script to operate correctly it is essential that the variables and commands in the arguments not be evaluated before the `while` command is executed, but rather be evaluated repeatedly during the execution of the command. Curly braces achieve just this effect by passing the `$` and `[` characters through to the `while` command so they can be evaluated during the execution of the command.

For comparison, consider the following example where double-quotes are used instead of braces:

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

In this case, the first argument to `while` is

```
5 > 0
```

and the second argument is

```
set result 5
set i 4
```

In this case the substitutions have all been made before the `while` command is invoked. The loop will never terminate, since `while`'s first argument is a constant expression that always evaluates to non-zero. The body of the loop behaves exactly the same from iteration to iteration, since all the arguments to all the commands are now constants.

See Chapter 6 for more details on control structures.

2.9 Comments

The comment character in Tcl is the hash-mark (#). 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 comments 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.

Because of the way curly braces and hash-marks are processed, confusion can sometimes occur when comments appear within curly braces. For example, the following example cannot be parsed correctly by Tcl:

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

The problem with this example is that the hash-mark isn't treated as a comment character when the second argument to `while` is being processed; at the time the argument is processed Tcl doesn't even know that it contains a nested command. Because of this, the open curly brace in the comment is considered to be significant, and Tcl can't find enough close curly braces to complete the word; an error results. The solution in this case is to add a backslash before the brace in the comment so that it isn't counted when parsing the argument to `while`.

Chapter 3
Variables

Like virtually all programming languages, Tcl allows you to use variables for storing information. Tcl supports two kinds of variables: simple variables and arrays. Variable names and variable values are both strings. This chapter describes the basic Tcl commands for manipulating variables and arrays, and the substitution mechanism that allows variable values to be passed to commands. See Table 3.1 for a summary of the commands discussed in this chapter.

3.1 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. It will be easier to use the variable substitution mechanism if you restrict yourself to these characters.

Variables may be created, read, and modified with the `set` command. `Set` 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 "three word value"
three word value
set a
three word value
```

`append` *`varName value`* ?*`value`* ...?	Append 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 appending is done in an efficient way that avoids copying the variable's old value. The return value is the new value of *`varName`*.
`incr` *`varName`* ?*`increment`*?	Add *`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, set the value of variable *`varName`* to *`value`*. In any case, return the current value of the variable.
`unset` *`varName`* ?*`varName varName`* ...?	Remove the variables given by the *`varName`* arguments. Returns an empty string.

```
set a 44
44
```

The first command above creates a new variable `a` if it doesn't already exist and sets its value to the character sequence "`three word value`". The result of the command is the new value of the variable. Tcl variables are created automatically when they are assigned values; there is no mechanism for declaring variables in Tcl except to access global variables inside procedures (see Chapter 7).

The second `set` command has only one argument: `a`. In this form it simply returns the value of the named variable without changing its value.

The third `set` command changes the value of `a` to `44` and returns that new value. Although the value looks like a decimal integer, it is stored as an ASCII string. Tcl variables can be used to represent many things, such as integers, floating-point numbers, names, lists, and Tcl programs, but they are always stored as strings. This use of a single representation for all values is one of the sources of Tcl's power, since it allows all of these different values to be manipulated in the same way and interchanged.

An error will occur if you attempt to read the value of a non-existent variable. For example, if there is no variable `badName` then the following command produces an error:

```
set badName
can't read "badName": no such variable
```

3.2 Arrays

In addition to simple variables Tcl also provides *arrays*. An array is a collection of related variables. Each element of an array 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.

Arrays are similar to simple variables in that you can't use an array value until it has been set. Furthermore, each variable is either a simple variable or an array: an error will occur if you attempt to use a simple variable as an array or vice versa.

3.3 Variable substitution

Chapter 2 has already introduced the use of `$`-notation for substituting variable values into Tcl commands. This section describes the mechanism in more detail. Strictly speaking, variable substitution isn't necessary since you can achieve the same effect using command substitution with the `set` command. However, variable substitution is useful because it saves typing and makes Tcl programs more concise and readable.

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 command's word by the value of the variable. The program below shows a simple example of variable substitution:

```
set a 44
set b $a
```

```
44
```

Variable substitution can also occur in more complex situations where it is less obvious how it should behave. For example, consider the following command:

```
expr $a+2
```

Is the name of the variable `a`, which makes the most sense in this case, or `a+2`? There are actually three forms of variable substitution, each with slightly different behavior.

The commands above are all examples of the first form, which is the simplest and most common of the forms. In this form the `$` is followed by a sequence of letters and digits and underscores; the variable name consists of all the characters up to the first one that isn't a letter or digit or underscore. This means that the variable name `a` is used in the expr example above, and the argument to the `expr` command is `44+2` (assuming that `a` has the value `44`).

The second form allows array values to be substituted. This form is like the first one except that the character just after the variable name is an open parenthesis. In this case all of the characters up to the next close parenthesis are taken as the name of an element within the array, and the value of that element is substituted:

```
set earnings(January) 87966
set x "--- $earnings(January) ---"
--- 87966 ---
```

The element name (everything between the parentheses) is parsed in the same way as a command word in double-quotes: variable substitution, command substitution, and backslash substitution are performed, and there may be spaces in the element name. This means, for example, that you can compute the name of an array element and insert that computed value in the element name during substitution:

```
set earnings(January) 87966
set month January
set x $earnings($month)
87966
```

The above rules for parsing elements lead to an unfortunate inconsistency. A space in an element name is not treated as a word separator during variable substitution, so the following command is perfectly legitimate, assuming that there exists an array `currency` with an element named `Great Britain`:

```
set x $currency(Great Britain)
```

However, if the same element name is used as the target in a `set` command then the space is significant and an error occurs:

```
set currency(Great Britain) pound
wrong # args: should be "set varName ?newValue?"
```

The error occurs because the Tcl parser uses its normal rules for parsing the first agument to `set`. It has no idea that this argument is the name of an array element, and the argument isn't enclosed in quotes or braces, so it treats the space character as a word separator. As a result, the `set` command receives three arguments and generates an error. The solution in this case is to surround the first argument with braces or quotes.

The last form of variable substitution is intended for situations where you wish to substitute a variable value in the middle of a string of letters or digits, or just before an open parenthesis. For example, suppose that you wish to substitute the value of variable `z` just after the `x` in `xxxyyy`. The following command won't work because it includes too many characters in the variable name:

```
set y xxx$zyyy
can't read "zyyy": no such variable
```

To get around this problem Tcl allows you to enclose the variable name in curly braces in variable substitution. When this happens the variable name is exactly what is between the braces. No substitutions of any sort are made on the characters between the braces and no special interpretation is given to the characters between the braces. Braces provide a simple solution to the problems above:

```
set z 123
set y xxx${z}yyy
xxx123yyy
```

Curly brace notation can only be used for simple variables, but it shouldn't be needed for arrays anyway, since the parentheses already indicate where the variable name ends.

Tcl's variable substitution mechanism is only intended to handle the most common situations; it's possible to imagine scenarios where none of the above forms of substitution achieves the desired effect. Fortunately, these situations can be handled by using command substitution in conjunction with the `set` command. Tcl also provides many other ways to deal with these situations, such as the `eval` and `format` commands; these techniques will be described in later chapters.

3.4 Removing variables: unset

It is possible to remove variables using the `unset` command. This command takes any number of arguments, each of which is a variable name. Each of the named variables is destroyed: 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. Each of the variables named in an `unset` command must exist at the time the command is invoked or else an error occurs.

One convenient use of `unset` is to convert a simple variable into an array or vice versa. For example, consider the following program:

```
set a 44
set a(12) 100
can't set "a(12)": variable isn't array
unset a
set a(12) 100
100
```

The first attempt to set `a(12)` fails because `a` is a simple variable rather than an array. After the variable has been `unset`, the second `set` succeeds, re-creating `a` as an array variable. Of course, at this point it is no longer possible to reference `a` as a simple variable.

3.5 Multi-dimensional arrays

The implementation of arrays in Tcl uses only a single element name in each reference, but it is easy to make Tcl arrays behave as if they are multi-dimensional. To do this, just use element names that consist of two or more independent parts concatenated together. 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
expr $matrix($i,$j)+12
230
```

In this example `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.

This example illustrates the power that derives from using textual strings everywhere in Tcl. Even though the basic language facilities are very simple, it is possible to achieve powerful effects by composing strings in interesting ways. In this case, element names are composed in a way that simulates multi-dimensional arrays. You'll see other interesting ways of composing strings later in conjunction with commands such as `eval`.

3.6 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 a number; it adds the number to the variable, 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:

```
set x cat
append x dog
catdog
```

The `append` command doesn't add any new functionality to Tcl, since the same effect can be achieved with a `set` command:

```
set x "${x}dog"
catdog
```

The main reason for `append` is efficiency. `Append` is implemented in a particularly efficient way that avoids copying the value of the variable. In contrast, the `set` approach requires the current contents of the variable to be copied twice: once when creating the argument to `set` and again to store it into the variable. For normal small variables the copying costs are insignificant, but when manipulating variables with thousands or tens of thousands of characters `append` will be substantially faster than other approaches.

3.7 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 `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 `trace` command can be used to monitor a variable so that a Tcl program gets invoked whenever the variable is set or read or unset. Variable tracing is convenient 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 discussion of variables so far has assumed a single global space of variables with every variable visible in all the Tcl code of an application. After procedures are introduced in Chapter 7 you'll see that Tcl actually provides two kinds of variables: global ones and local variables for procedures. Chapter 7 will show how a procedure can access variables other than its own local variables.

Chapter 4
Expressions

Tcl is a typeless string-oriented language, which means that in general you can pass any sequence of characters as an argument to any command. Many of the commands, such as `set`, are perfectly happy with any argument value whatsoever. However, there are many other commands that expect some of their arguments to have particular forms. Three forms are particularly common in Tcl: expressions, lists, and Tcl scripts. This chapter shows how to write expressions, Chapter 5 discusses lists, and Chapter 6 describes how Tcl scripts are embedded in commands like `if`.

4.1 Introduction and the expr command

Expressions combine values (or *operands*) with *operators* to produce new values. For example, the expression `4+2` evaluates to `6` and the expression `(8+4)*6.2` evaluates to `74.4`. Many Tcl commands expect one or more of their arguments to be expressions. The simplest such command is `expr`, which does nothing but evaluate an expression and return the result as a string:

```
expr (8+4)*6.2
74.4
```

Many other Tcl commands also take expressions as arguments. For example, the `if` command evaluates an expression and uses the result to determine whether or not to execute a nested Tcl script:

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

In this example the command `set x 2` will be executed if the expression `$x<2` evaluates to a non-zero result.

In most cases the operands for expressions are numbers (either integers like 24 or floating-point numbers like 16.5). A few of the operators allow their operands to be arbitrary strings; this allows string comparisons to be performed easily inside expressions (see Section 4.3.2).

4.2 Operands and substitutions

Operands for Tcl expressions may be specified in several ways. The simplest form consists of integers. Integers are normally specified in decimal, but if the first character of the number is `0` then the number is read in octal (base 8) and if the first two characters are `0x` then the number is 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.

Operands may also be specified as floating-point numbers. Tcl accepts any of the forms defined for the ANSI C standard except that the f, F, l, and L suffixes are not supported. All of the following are valid floating-point numbers:

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

Tcl treats numbers as integers whenever possible. To force an integer value to be represented in floating-point form, add a decimal point as in the last example above. Floating-point numbers are manipulated using double-precision format.

It is possible to perform variable substitutions and command substitutions on expressions using the standard Tcl facilities for command-line substitution:

```
set x 2.0
expr $x-0.8
1.2
expr [set x]-0.8
1.2
```

The actual argument received by the `expr` command is `1.2-0.8` in both cases.

The Tcl expression evaluator also performs variable substitution and command substitution on its own, as in the following examples:

```
set x 2.0
expr {$x-0.8}
1.2
expr {[set x] - 0.8}
```

```
1.2
```

In these examples the curly braces prevent any substitutions before the `expr` command is invoked; the substitutions are carried out by the expression evaluator.

It's important for the expression evaluator to perform substitutions because in many cases expressions are not evaluated immediately, but rather saved for later evaluation or evaluated repeatedly. When this happens it is usually important to evaluate variables or execute embedded commands at the time the expression is evaluated rather than the time the command containing the expression is invoked. For example, consider the following program, which computes a power of a particular base:

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

The `while` statement repeatedly evaluates its first argument as an expression then executes the second argument as a Tcl command, as long as the first argument evaluates to non-zero (see Chapter 6 for details). It is essential that the first argument to `while` be enclosed in braces so that the argument received by the `while` command is `$power>0`. Without the braces the value of `$power` would be substituted before the `while` command is invoked and the value of the expression would not change from iteration to iteration; an infinite loop would occur if the initial value of `$power` were greater than 0. On the other hand, braces are not needed in the `expr` command inside the loop: the braces around the loop body prevent substitutions before `while` is invoked and it doesn't matter whether the variable substitutions occur before the `expr` command is invoked or while it computes its result.

One of the most important things in learning Tcl is to understand when and how substitutions occur. Substitutions occur before each command is invoked, as described in Chapter 2, but some commands perform additional substitutions after they are invoked. In the case of the `expr` command there are two different times when substitutions can occur. One round occurs when the command is broken up into words before invocation, and the second round occurs for the second word of the command when it is evaluated as an expression. In most cases the second round of substitutions is sufficient for an expression and the first round is likely to cause more harm than good, so it is common to see expressions enclosed in braces when they appear in Tcl commands.

Expression operands may also be specified as character strings enclosed in double-quotes or curly braces. If an operand is enclosed in quotes then variable substitutions, command substitutions, and backslash substitutions are performed on the information between the quotes just as for commands. If the operand is enclosed in braces then no substitutions are performed on the characters between the braces, again just as for commands. Double-quotes and braces are most useful when performing string comparisons as described in Section 4.3.2 but they can also be used to achieve exotic arithmetic effects, as in the following example:

```
set tens 2
set ones 1
set fraction 6
expr {"$tens$ones.$fraction" + 3.0}
24.6
```

Again, these substitutions are performed by the expression evaluator in addition to any substitutions that occurred when the command was parsed.

When the expression evaluator performs variable substitutions, or when it processes double-quotes or curly braces, each such operation yields a single operand. The results of the substitution or quoting are not rescanned for additional substitutions or embedded operators. Because of this behavior, both of the following programs produce errors:

```
set x 4+2
expr {$x+4}
can't use non-numeric string as operand of "+"
expr {"2".4 + 3}
syntax error in expression ""2".4 + 3"
```

In the first case the substitution for `$x` yields `4+2`, which isn't a valid numeric operand. In the second case, the quoted string yields `2`, which is a valid numeric operand, but it is treated as a complete operand by itself and is not combined with the characters following it to produce `2.4`. Instead the expression parser sees two numbers in a row (`2` and `.4`) with no intervening operator, which is an error.

4.3 Operators and precedence

Table 4.1 lists all of the operators supported in Tcl expressions; they are similar to the operators in C expressions. 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, I recommend that you 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 left to right. For example, `4*5%2` evaluates to `0`.

4.3.1 Arithmetic operators

Tcl expressions support the standard arithmetic operators including `+`, `-`, `*`, `/`, and `%`. The - operator may be used either as a binary operator for subtraction, as in `4-2`, or as a unary

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	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 4.1. Summary of the operators allowed in Tcl expressions. These operators have the same behavior as in 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.

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. The behavior of `/` and `%` for negative operands is the same in Tcl as in C. This means that the sign of the remainder and the direction of truncation are machine-dependent if either operand is negative. However, `/` and `%` are always consistent: if `a/b` is `c` and `a%b` is `d`, then `(b*c)+d` will be equal to `a`.

4.3.2 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.

The relational operators may be applied not only to numbers but also to arbitrary strings. If both of the operands are numbers then the comparison is done numerically. If either or both of the operands doesn't make sense as a number then the operands are compared as strings using lexicographic ordering, as in the following examples:

```
expr {"abc" < "abcd"}
1
expr {2.4 >= "abcd"}
0
```

4.3.3 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 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.

The operators `&&` and `||` are special in that they always evaluate their left operand first and only evaluate the right operand if needed to determine the result (e.g. if the left operand is non-zero for `&&` or zero for `||`). This behavior is useful in situations where the right operand sometimes generates errors during evaluation. For example, consider the following command:

```
expr {[info exists x] && ($x<2)}
```

This command returns `1` if the variable `x` is defined and has a value less than `2`, and it returns `0` otherwise. Command substitution causes the left operand of `&&` to be `1` if `x` exists and `0` if it doesn't. In the case where `x` doesn't exist, an error would occur if

`($x<2)` were evaluated, but this is guaranteed never to happen. This example shows once again the importance of performing substitutions during expression evaluation instead of during command parsing: if the braces were replaced with double-quotes then both the command substitution and the variable substitution would be performed by the Tcl command parser before invoking the `expr` command and an error would be generated if `x` doesn't exist.

4.3.4 Bitwise operators

Tcl provides six operators that manipulate the individual bits of integers: `&`, `|`, `^`, `<<`, `>>`, and `~`. These operators require both of 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 zeroes 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 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.

4.3.5 Choice operator

The operator pair `?:` may be used with three operands 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.

4.4 Types and conversions

All of the expression operators accept integers as operands. The arithmetic operators `+`, `-`, `*`, and `/` accept floating-point operands as well. If one operand is an integer and the other is a floating-point number, then the integer is converted to floating-point and the result will be in floating-point. Floating-point numbers are manipulated using double-precision representation.

The logical operators `!`, `&&`, and `||` also accept floating-point arguments as well as integers. They test their arguments to see if they are equal to zero and set their results accordingly.

The relational operators <, <=, >=, >, ==, and != accept operands of any form. If both operands are integers then the comparison is done using integer arithmetic. If both operands are numeric but at least one of them is floating-point then the comparison is done using floating-point arithmetic. When one or both of the operands cannot be read as either an integer or a floating-point number, then the comparison is done using lexicographic string comparison. In this case numeric operands are converted back to strings using `%d` format for integers and `%g` format for floating-point numbers (see the `format` command in Section 9.1 for a description of these formats).

All operators other than the ones mentioned above require their arguments to be integers. An error will occur if an argument does not have the proper form for an integer.

When including non-numeric strings in an expression you must use variable substitution, double-quotes, or braces. If you attempt to enter a non-numeric string directly then an error will occur, as in the following example:

```
expr {$x != red}
syntax error in expression "$x != red"
```

Tcl treats this as an error because in most cases when it happens the user has forgotten to type the `$` in front of a variable name; if the name were accepted as a literal string then it would result in confusing errors in many cases.

Chapter 5
Lists

Lists in Tcl provide a simple mechanism for dealing 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. With lists you can easily 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 particular structure; this means that you can store lists in variables, type them to commands, and even 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 5.1 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.

5.1 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 Bob
```

`concat` *list list* ...	Concatenate multiple lists into a single list (each element of each *list* becomes an element of the result list) and return the new list.
`join` *list* ?*joinString*?	Concatenate list elements together with *joinString* as separator and return the result.
`lappend` *varName value value* ...	Append each *value* to variable *varName* as a list element, and return the new value of the variable.
`lindex` *list index*	Return *index*'th element from *list*.
`linsert` *list index value value* ...	Insert *values* as list elements before *index*'th element of *list*.
`list` *value value* ...	Create and return a new list whose elements are the *value* arguments.
`llength` *list*	Return the number of elements in *list*.
`lrange` *list first last*	Return a list consisting of elements *first* through *last* of *list*.
`lreplace` *list first last ?value value* ...?	Return 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` *list pattern*	Return the index of the first element in *list* that matches *pattern* (using the rules for `string match`), or `-1` if none.
`lsort` *list*	Return a new list formed by sorting the elements of *list* in alphabetical order.
`split` *string ?splitChars?*	Return a list formed by splitting *string* at instances of *splitChars* and turning the characters between these instances into list elements.

Table 5.1. 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 may be used to index into a list and extract a single element:

```
lindex {John Anne Mary Bob} 1
Anne
```

`Lindex` takes two arguments: a list and an index. It returns the element of the list selected by the index. 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 so that the entire list is passed to the command as a single word even though it contains spaces. When lists are stored in variables or printed out, there are no braces around the list:

```
set x {John Anne Mary Bob}
John Anne Mary Bob
```

When the list commands are processing lists, they treat curly braces and backslashes specially in the same way that the Tcl command parser treats these characters specially. If the first character of a list element is an open curly brace then the element is not terminated by spaces or tabs. Instead, it consists of all the characters up to the matching close curly brace (but not including the braces themselves). For example:

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

One of the most common uses for braces is to create lists that contain other lists as elements, as in the above example. There is no limit on how deeply lists may be nested:

```
lindex [lindex {top {middle {bottom 1 2 3} next}} 1] 2
next
```

Backslashes may also be used within list elements to prevent special interpretation of characters such as spaces and braces, or to insert special characters such as newline:

```
lindex {a \{x\ y c d} 1
{x y
```

The same backslash substitutions are available in lists as in commands (see Table 2.1). As with commands, backslashes receive no special treatment when they occur in elements enclosed in braces, except that backslashed braces are ignored in the search for the element's matching close brace:

```
lindex {a {b c \} d} e} 1
b c \} d
```

It's important to remember that the special treatment of backslashes and braces in list elements is carried out by the list commands themselves as they process lists. This special treatment is independent of the substitutions made by the Tcl command parser when it is preparing the command for execution. If a list is passed to a command without enclosing it in braces, then the list will be processed twice: once by the Tcl command parser and again by the list command. This potential for double-substitution in lists is similar to that in expressions. As with expressions, it is usually a good idea to enclose lists in braces to prevent substitutions by the Tcl command parser.

5.2 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 and command-line substitution:

```
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 works a little differently than `concat`: its arguments need not be lists themselves, and it joins them 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, three of which are themselves lists with more than one element. The `list` command will always produce a list with proper structure, regardless of the structure of its arguments, 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:

```
set x [list "{a" "Unmatched brace: {" "Short phrase."]
\{a Unmatched\ brace:\ \{ {A short phrase.}
lindex $x 0
{a
lindex $x 1
Unmatched brace: {
lindex $x 2
A short phrase.
```

Notice that the result of the `list` command doesn't always look exactly like the input arguments. In the example above `list` added backslashes and braces in order to generate a well-formed list from which the original elements could be extracted with `lindex`.

Properly-formed lists such as those produced by `list` have another important property. If such a list is executed as a Tcl command then the words of the command, after all substitutions, will be exactly the arguments passed to `list`. The first argument to `list` will be the command's name, the next argument to `list` will be the first argument for the command, and so on. This property is very important because it allows you to generate commands that are guaranteed to parse in a particular fashion, even if some of the command's arguments contain characters like $ and [that normally cause substitutions to be performed (the `list` command quotes $ and [in list elements so that they won't cause substitutions if the list is executed as a Tcl command). You'll hear more about this feature of `list` in later chapters.

The `llength` command may be used to query the number of elements in a list:

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

Llength takes a single argument, which must be a well-formed list, and returns a decimal string. As you can see from the last example above, an empty string is considered to be a proper list with zero elements.

5.3 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 one or more elements from a list and replaces them with zero or more new elements:

```
set x {a b {c d} e}
a b {c d} e
lreplace $x 3 3
a b {c d}
lreplace $x 1 2 {W X} Y Z
a {W X} Y Z e
```

`Lreplace` 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, as in the first `lreplace` command above, then the result is a new list produced by deleting the given range of elements from the original list. If additional arguments are specified to `lreplace` as in the second example, then they are inserted into the list in place of the elements that were deleted.

The `lrange` command is used to extract 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:

```
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 a particularly efficient way to append new elements to an existing list. Instead of taking a list as argument, it takes the name of a variable whose value is a list. The variable's value is modified by appending additional arguments to it as new list elements. The return value from the command is the new value of the variable:

```
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 isn't strictly necessary; for example, the same effect as the lappend command above could be produced with the following command:

```
set x [linsert $x 100000 XX {YY ZZ}]
a b {c d} e XX {YY ZZ}
```

However, the set+linsert combination copies the entire list four times: from the variable into the linsert command, from linsert's argument to its result, from linsert's result to set's argument, and from set's argument to the variable's value. In contrast, lappend does at most one copy (from the old variable value to a new larger one) and in repeated lappend operations it avoids even this copy by allocating extra space for the variable when it grows it. For small lists the difference in peformance probably won't be noticeable, but if you're building a very large list a piece at a time then lappend is much more efficient than set+linsert.

5.4 Searching lists: lsearch

The lsearch command may be used to search for a particular element within a list. It takes two arguments, the first of which is a list and second of which is a pattern:

```
lsearch {ab ac bc} bc
2
lsearch {ab ac bc} ?c
1
lsearch {ab ac bc} cb
-1
```

Lsearch returns the index of the first element in the list that matches the pattern, or -1 if there was no matching element. Matching is determined with the rules used by the string match command described in Section 9.5. The first lsearch command above checks for an element that is exactly bc. The second command searches for an element containing two characters with c as the second character, and the third command searches for an element that is exactly cb.

5.5 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 Bob}
Anne Bob John Mary
```

5.6 Converting between strings and lists: split and join

The `split` and `join` commands are useful for converting between lists and strings that contain elements separated by characters other than spaces. For example, suppose a variable contains a UNIX file name with elements separated by slashes, and you want to convert it to a list with one element for each component of the file name. This would then permit you to process the elements using commands described in this chapter, or other commands like `foreach`, which is described in Section 6.2. The conversion to a list can be done with the `split` command:

```
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 `split` command takes two arguments. The first is the string to be split up and the second is a string containing one or more *split characters*. The result is a list generated by finding all the split characters in the string and creating one list element from the information between each pair of 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 (e.g. the second example above) then empty elements are generated in the result list. The split characters themselves are discarded.

If an empty string is specified for the split characters in split, 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 list, concatenating list elements together with a given separator string between them:

```
join {a b c} /
a/b/c
join {{} usr include sys types.h} /
```

```
/usr/include/sys/types.h
```

`Join` takes two arguments: a list and a separator string. It generates its result by extracting all of the elements from the list and concatenating them together with the separator string between each pair of elements. The separator string can contain any number of characters, including zero:

```
join {a b c} .tmp/
a.tmp/b.tmp/c
join {a b c} {}
abc
```

In this respect `join`'s behavior is not exactly the inverse of `split`'s, since `split` treats multiple split characters as independent separators.

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:

```
set x [split {Now is the time
for all good men
to come to the aid
of their country} \n]
{Now is the time} {for all good men} {to come to the
aid} {of their country}
join $x \n
Now is the time
for all good men
to come to the aid
of their country
```

Chapter 6
Control Structures

The Tcl language provides a number of facilities that you can use to generate, sequence, and conditionally execute commands. These control structures mimic most of the control structures found in the C programming language and the `csh` shell, including `if`, `while`, `for`, `case`, `foreach`, and `eval`. Table 6.1 summarizes the Tcl control structures.

Control structures in Tcl are just commands, and they have the same form as all other Tcl commands. However, the commands that implement control structures, like `if` and `while`, are unusual in that one or more of their arguments are themselves Tcl scripts. The control structure commands examine some of their arguments to determine what to do, then execute other arguments one or more times by passing them to the Tcl interpreter. The arguments executed in this way may themselves include additional control structures or any other Tcl commands.

6.1 The if command

The `if` command in Tcl behaves like `if` in C: it evaluates an expression, tests its result, and conditionally executes a script based on the result. For example, consider the following command:

```
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, which spans three lines here. The expression can have any of the forms for expres-

`break`	Terminate innermost nested looping command (`for`, `foreach`, or `while`).
`case` *`string`* ?in? *`patList body`* ?*`patList body`* ...? `case` *`string`* ?in? {*`patList body`* ?*`patList body`* ...?}	Match *`string`* against each *`patList`* in order until a match is found, then execute the *`body`* corresponding to the matching *`patList`*. A *`patList`* of `default` matches any *`string`*. Returns the result of the *`body`* executed, or an empty string if no pattern matches.
`continue`	Terminate the current iteration of the innermost looping command and go on to the next iteration of that command.
`eval` *`arg`* ?*`arg arg`* ...?	Concatenate all of the *`arg`*'s with spaces as separators, then execute the result as a Tcl script and return its result.
`for` *`init test reinit body`*	Execute *`init`* as a Tcl script. Then evaluate *`test`* as an expression. If it evaluates to non-zero then execute *`body`* as a Tcl script, execute *`reinit`* as a Tcl script, and re-evaluate *`test`* as an expression. Repeat until *`test`* evaluates to zero. Returns an empty string.
`foreach` *`varName list body`*	*`List`* must be a valid Tcl list. For each element of *`list`*, in order, set variable *`varName`* to that value and execute *`body`* as a Tcl script. Returns an empty string.
`if` *`test`* ?then? *`trueBody`* ?else? ?*`falseBody`*?	Evaluate *`test`* as an expression. If its value is non-zero then execute *`trueBody`* as a Tcl script. If its value is zero then execute *`falseBody`* as a Tcl script (if *`falseBody`* is specified). Returns the result of *`trueBody`* or *`falseBody`*, or an empty string if *`test`* evaluates to zero and there is no *`falseBody`*.
`source` *`fileName`*	Read the file whose name is *`fileName`* and execute its contents as a Tcl script. Returns the result of the script.
`while` *`test body`*	Evaluate *`test`* as an expression. If its value is non-zero then execute *`body`* as a Tcl script and re-evaluate *`test`*. Repeat until *`test`* evaluates to zero. Returns an empty string.

Table 6.1. A summary of the Tcl commands that implement control structures.

sions described in Chapter 4. The `if` command tests the value of the expression; if the value is non-zero, then `if` executes the Tcl script. If the value is zero then `if` returns without taking any further action. The example above sets the variable `x` to zero if it was previously negative.

`If` commands can also receive an additional argument containing a Tcl script to execute if the expression evaluates to zero (an “else clause”). In addition, `if` allows the noise words `then` and `else` to precede the corresponding script arguments. The following commands are all identical in effect:

```
if {$x < 0} then {set x 0} else {set x [expr $x+2]}
if {$x < 0} {
    set x 0
} else {
    set x [expr $x+2]
}
if {$x < 0} {set x 0} {set x [expr $x+2]}
```

The result of an `if` command is the result of the “then” or “else” clause, whichever is executed. If neither a “then” clause nor an “else” clause is executed (because the command contained no “else” clause and the expression evaluated to 0), then the command returns an empty string:

```
set x −2
−2
if {$x < 0} {set x 0}
0
```

In the examples in this book the expression and script arguments to `if` are almost always enclosed in braces. This is usually a good idea in commands that implement control structures, and in some cases it is absolutely necessary. The reason for this is the same as the reason given in Chapter 4 for expressions: double substitution will occur if the arguments aren’t enclosed in braces. For the expression argument, substitutions will be performed when the argument is evaluated as an expression, so there is generally no need to have an earlier round of substutions while parsing the `if` command. For the script arguments, substitutions will be performed when the arguments are executed as Tcl commands, so there is generally no need to have an earlier round of substitutions while parsing the `if` command. If the braces are omitted, then the double substitution an undesirable effect. For example, consider the following script:

```
set b "A test string"
A test string
if {$a == ""} "set a $b"
wrong # args: should be "set varName ?newValue?"
```

The second argument to `if` wasn't enclosed in braces, so the value of variable `b` was substituted as part of invoking the `if` command. Thus the second argument to `if` was

```
set a A test string
```

When this string was subsequently executed as a Tcl script `set` returned an error because it received four arguments when it was expecting only one or two. If the `set` command had been enclosed in braces instead of quotes, then `b`'s value wouldn't have been substituted until the `set` command was executed and the entire value of `$b` would have formed a single argument to `set`.

6.2 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 constructs, and `foreach` is similar to the corresponding feature in 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 the elements from the list stored in variable `a` to variable `b` in reverse order:

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

The `for` command is similar to `while` except that it also takes two additional script arguments, which perform once-only initialization before the first iteration of the loop and reinitialization after each execution of the loop body. The above 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 the initialization script, the second is an expression that determines when to terminate the loop, the third (which is on the second line of the command) is the reinitialization script, 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 reevaluates 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 or 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 be operated upon (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 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` iterates over all of the elements of a list. For example, the following script provides yet another implemenation 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 named variable to hold the corresponding 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`". As with the other looping commands, `foreach` always returns an empty string.

6.3 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 constructs 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]
}
```

In this example the `continue` isn't absolutely necessary since the same effect could be achieved by enclosing the "`set b ...`" command in an `if` command. `Continue` commands are most often used to eliminate a deeply nested set of `if` commands that would result if `continue` weren't used. Typically this occurs when the main loop iterates over a superset of the desired elements and a complex set of tests must be performed on the individual elements to determine whether they should be acted upon. Each of these tests can be coded as an `if` command that invokes `continue` if the test determines that the element is inappropriate.

6.4 The case command

The `case` 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 `case` can be achieved with a nested set of `if` commands, but `case` provides a more compact encoding. Tcl's `case` command has two forms; here is an example of the first form:

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

The first argument to `case` is the value to be tested (the contents of variable `x` in the command above). The second argument is the "noise word" `in`; this argument can be omitted. After that come one or more pairs of arguments; 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 case 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 `case` 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 for `case` is similar to the first except that all of the pattern-script pairs are combined into a single list argument instead of being separate arguments. In the second form the above command looks like this:

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

This form is convenient because it allows the patterns and scripts to be spread across multiple lines: the braces around the list prevent the newlines from being treated as command separators. If the first form spills over onto multiple lines then backslashes have to be placed at the ends of lines to quote the newlines. In addition, variable and command substitutions never occur in the patterns of the second form because of the braces around the list of patterns and scripts. In the first form, variable and command substitutions will be performed unless the individual patterns are enclosed in braces. Most people seem to find the second form easier to use in most cases.

The `case` command has three other features that aren't used in the above examples. First, each pattern is actually a list of patterns (in the above examples the pattern lists only had a single element each). It is sufficient for any element of the list to match the value. For example, the command below increments `t1` if `x` is `a` or `b`, `t2` if `x` is `c` or `d` or `e`, and does nothing otherwise:

```
case $x in {{a b} {incr t1} {c d e} {incr t2}}
```

The second additional feature is that patterns may contain a variety of wild-card matching characters. For example, a `?` in a pattern matches any single character of the value and a `*` matches any substring (zero or more characters). The special characters follow the style of the `csh` shell and include `?`, `*`, `[]`, and `\`. See the description of the `string match` command in Section 9.5 for full details. As an example of the use of wild-cards, the following command increments `t1` if `x` contains the letter `a` and it increments `t2` if `x` contains a `b` or a `c` but no `a`:

```
case $x in {*a* {incr t1} {*b* *c*} {incr t2}}
```

The final feature of `case` is that a pattern of `default` matches any value. It is equivalent to a pattern of `*` and is typically the last pattern in the `case` command. 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 have a `b` or `c` but no `a`. The third, `t3`, counts the number of elements that have neither an `a`, a `b`, nor a `c`:

```
set t1 0
set t2 0
set t3 0
foreach i $x {
    case $x in {
        *a*        {incr t1}
```

```
            {*b* *c*}   {incr t2}
            default     {incr t3}
        }
    }
```

6.5 Generating commands on the fly: eval

`Eval` is a command that makes it easy to synthesize scripts on-the-fly, save them around in variables, and eventually execute them. It takes one or more arguments. If it receives only one argument then it simply executes that argument as a Tcl script. If `eval` is given two or more arguments then it concatenates them together with spaces between them and executes the result as a Tcl script. The command

```
eval a b c d
```

is exactly equivalent to the command

```
if 1 [concat a b c d]
```

for any values of `a`, `b`, `c`, and `d`.

One possible use for `eval` is in a macro facility where a user's actions are recorded for later replay. This can be implemented by appending the Tcl command for each user action to a variable before executing it. Then the sequence of commands can be replayed by `eval`-ing the variable.

It's important to realize that two rounds of parsing occur in the arguments to `eval`. They are parsed (and substituted) once when the `eval` command is parsed, and again when they are executed as a script. It's easy to run into troubles with `eval` if you aren't aware of the two levels of parsing. For example, suppose that the variables `a`, `b`, and `c` contain the name of a command and two arguments for it, and you want to execute the command defined by the variables. An obvious but incorrect way to do it is like this:

```
eval $a $b $c
```

The problem with this is that the variables are concatenated to form the script and the script is re-parsed when it is executed. There is no guarantee that the script will be parsed so that each of the variables ends up as a single word of the resulting command. For example, suppose `a` has the value `set`, `b` has the value `x` and `c` has the value "`A test string`". The `eval` command will concatenate these variables to produce the string "`set x A test string`", then it will execute this string. Unfortunately, this string will be parsed into a command with five words, not three. The `set` command will receive four arguments and it will generate an error because of this.

There are two ways to solve this problem. One approach that works in this particular instance (but not in more complex situations) is to enclose all of the variables in braces:

```
eval {$a $b $c}
```

The braces prevent variable substitution from occuring when the eval command is parsed, so the command that eval executes is "`$a $b $c`". The variables get substituted when this command is executed, and each variable becomes one word of the command as desired.

A more general approach to this problem is to take advantage of the fact that Tcl commands have list structure. If a proper Tcl list is executed as a command then each element of the list will become one word of the resulting command. You can use the list commands to generate a list with a particular structure and be absolutely certain that the list will parse in a particular way when executed as a command. For example, the above problem can be eliminated with the following command:

```
eval [list $a $b $c]
```

In this case `eval` will receive a single argument, "`set x {A test string}`", which it will then execute. This command will be parsed into three words and achieve the desired effect. It is guaranteed to work regardless of the values of the three variables. Using lists is more general than the braces approach because you can generate lists with arbitrary structure. For example, suppose that you have the same three variables but `c` is now a list itself and you want each of `c`'s elements to form a distinct argument to the generated command. `A` and `b` are each to form one word as before. You can handle this situation with the following command:

```
eval [concat [list $a $b] $c]
```

or, equivalently,

```
eval [list $a $b] $c
```

In each of these cases the generated command will be a list whose first two elements are `a` and `b` and whose remaining elements are the elements of `c`.

6.6 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 test.tcl
```

will execute the contents of the file `test.tcl`. The return value from `source` will be the value returned when the file contents are executed, which will normally be 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 7.1 for more information on `return`.

Chapter 7
Procedures

A procedure in Tcl is a command that looks and behaves like the built-in commands, but 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 extend the functions of a Tcl-based application and to package up the extensions in a clean and easy-to-use fashion. 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 as built-in commands written 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.

The procedure mechanism also provides some unusual and sophisticated commands for dealing with variable scopes. Among other things, these commands allow you to implement new Tcl control structures as procedures. Table 7.1 summarizes the Tcl commands related to procedures.

7.1 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 (two arguments, `a` and `b`, here). The third argument to `proc` is a Tcl script that forms the body of the new procedure. Proc creates a new command for the procedure's name. It also arranges that

`global` *`name1`* ?*`name2`* ...?	Bind variable names *`name1`*, *`name2`*, etc. to global variables. References to variables with 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`*	Define 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.
`rename` *`oldName newName`*	Rename the command that used to be called *`oldName`* so that it is now called *`newName`*. There must not currently be a command named *`newName`*. If *`newName`* is an empty string then *`oldName`* is deleted. Returns an empty string.
`return` ?*`value`*?	Return from the innermost nested procedure with *`value`* as the result of the procedure. *`Value`* defaults to an empty string.
`uplevel` ?*`level`*? *`arg`* ?*`arg arg`* ...?	Concatenate all of the *`arg`*'s with spaces as separators, then execute the resulting Tcl script in the variable context of stack level *`level`*. *`Level`* consists of a number optionally preceded by `#`, and defaults to `#0`. Returns the result of the script.
`upvar` ?*`level`*? *`otherVar`*1 *`myVar`*1 ?*`otherVar`*2 *`myVar`*2 ...?	Bind 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`* at *`level`* instead. Additional bindings may be specified with *`otherVar2`* and *`myVar2`*, etc. *`Level`* has the same syntax and meaning as for `uplevel`. Returns an empty string.

Table 7.1. A summary of the Tcl commands related to procedures and variable scoping.

whenever the command is invoked the procedure's body will be executed. In this case the new command will have the name `plus`; whenever `plus` is invoked it must receive two arguments. While the 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"
```

The `return` command may be used to force an immediate return from a procedure. When it is invoked inside a procedure, the procedure returns immediately and the argument to `return` becomes the return value from the procedure. The following script defines a procedure `less` that returns `1` if its first argument is less than its second and `0` otherwise:

```
proc less {a b} {
    if {$a < $b} {
        return 1
    }
    return 0
}
```

If `return` is invoked with no arguments then the enclosing procedure returns with a result that is an empty string. The `return` command may also be used in a script file to terminate a `source` command. Any other use of `return` (e.g. when there is no active procedure and no active `source` command) generates an error.

7.2 Local and global variables

When the body of a Tcl procedure is executed, it behaves exactly the same as if it were invoked outside of the procedure (with `eval`, for example) except for one thing: it has a different *variable context*. Each procedure invocation has its own private set of variables, called *local variables*, and these variables are different from the *global variables* that are accessible outside any procedure. When a variable name is used inside a procedure, it refers by default to a local variable. Local variables are created the first time they are set, and they are all deleted when the procedure returns. If one procedure calls another, the callee's local variables are disjoint from the caller's local variables.

The arguments to a procedure are just local variables whose values are set from the command-line arguments at the time the procedure is invoked. When execution begins in a procedure, the only local variables with values are those corresponding to arguments.

A procedure can reference global variables by invoking 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.

If you're used to programming in a language with declarations like C, it's important to realize that `global` is *not* a declaration; it's a command. This means that the global binding doesn't take effect until the command is actually executed. Also, the precedence of local and global variables is different in Tcl from what it is in most other programming languages. If a procedure first accesses a variable as a local variable and then invokes `global`, the global variable takes precedence over the local one and the local variable will not be accessible for the rest of the procedure. The script below is probably not very useful but it demonstrates this behavior:

```
proc add1 a {
    global a
    expr $a+1
}
set a 44
add1 33
45
add1 21
45
```

If `unset` is invoked within a procedure on a global variable, it unsets the global variable but does not remove the binding for that name within the procedure. The global variable continues to be accessible for the rest of the procedure, and it can be re-`set` by the procedure if desired.

7.3 More on arguments: 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 stdout "a is $a, b is $b"
}
```

The second additional feature for argument lists 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 wasn'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 that one 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 `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 may be invoked with any number of arguments and it 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 always its default.

7.4 Exotic scoping facilities: upvar and uplevel

By default, all of the variables used by a procedure are local to that procedure. With the `global` command a procedure can access global variables. This section describes two additional commands, `upvar` and `uplevel`, that allow a procedure to access the variable context of any procedure that is currently active as well as global variables. These commands are useful for implementing call-by-reference argument semantics, and they can also be used to define new control structures as Tcl procedures.

The `upvar` command binds one or more names in the local variable context to other variables at global level or in the context of some other active procedure. `Upvar` has the form

```
upvar ?level? otherVar1 myVar1 ?otherVar2 myVar2 ...?
```

The *`level`* argument selects a variable context. If it is `1`, it selects the context of the procedure that invoked the current one (or global context if the command that invoked this procedure was at global level). If *`level`* is `2` it selects the context of the caller's caller, and so on. Alternatively, level may be specified as `#0` to specify global level, `#1` to specify the context of the first-level procedure invoked from global level, and so on. *`Level`* may be omitted (unless the first character of *`otherVar1`* is `#` or a digit), in which case it defaults to 1.

The *`otherVar1`* argument to `upvar` specifies the name of a variable in the context selected by *`level`*. The `upvar` command will make this variable accessible by the name *`myVar1`* in the current procedure. If additional arguments are specified, they give the names of other variables in the context selected by *`level`*, along with the names by which those variables will be accessible in the current procedure. The effect of an `upvar` command lasts until the procedure returns, and the general behavior of `upvar` is the same as `global` except that a wider range of variables may be acessed through `upvar`.

One of the most common uses of `upvar` is to implement call-by-reference argument semantics, where a procedure receives as one its arguments the name of a variable in the caller's context. It can then use `upvar` to read or modify the variable. For example, consider the following procedure:

```
proc squares {varName n} {
    upvar 1 $varName v
    set v {}
    for {set i 1} {$i <= $n} {incr i} {
        lappend v [expr $i*$i]
    }
}
```

The `squares` procedure takes two arguments: the name of a variable in the caller's context and a number `n`. It sets the variable to a list whose elements are the squares of the first `n` integers:

```
squares x 5
set x
1 4 9 16 25
```

The `squares` procedure will work equally well whether its caller is a command at global level or another procedure; whatever variable was accessible to the caller by the given name will be modified. The *`otherVar`* variable in an `upvar` command may itself be the *`myVar`* name of an `upvar` command; in this case a chain of variable names can arise with all of them referring to the same original variable.

The `uplevel` command may be used to execute a script in another variable context. It is a cross between `upvar` and `eval`, and it has the following form:

```
uplevel ?level? arg ?arg arg ...?
```

`Level` has the same forms as for `upvar`. `Uplevel` concatenates all of its *`arg`* arguments (with spaces separating them) and executes the resulting string as a Tcl script just as `eval` does. However, the execution is carried out in the variable context given by *`level`* rather than the current context as for `eval`. The result of the nested script will be returned by `uplevel` as its result.

`Uplevel` and `upvar` can be used to create new control structures as Tcl procedures. For example, if there were no `for` command in Tcl, it could be defined with the following procedure:

```
proc for {init test reinit body} {
    uplevel 1 $init
    while {[uplevel 1 [list expr $test]]} {
        uplevel 1 $body
        uplevel 1 $reinit
    }
}
```

Actually, this code isn't a perfect emulation of the built-in `for` command because it doesn't handle `break`, `continue`, and errors in the same way as the built-in command, but it could easily be extended to do so using the facilities described in Chapter 8. The `uplevel` command is essential to this script; without it `test`, `init`, `reinit`, and `body` would not be able to access variables in the calling procedure's variable context. The use of the `list` command in this example is also essential for the procedure to work with arbitrary `test` arguments; the reasons for this are the same as those discussed in Section 6.5 for the `eval` command.

The `do` procedure below is another example that uses `upvar` and `uplevel` to create a new control structure:

```
proc do {varName first last body} {
    upvar 1 $varName v
    for {set v $first} {$v <= $last} {incr v} {
        uplevel 1 $body
```

```
        }
    }
```

The first argument to `do` is the name of a variable. `Do` sets that variable to each number in the range between its second and third arguments, inclusive, 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
```

7.5 Replacing, renaming, and deleting commands

If you invoke `proc` at a time when there is already a command with the name specified in the command, then the existing command is replaced with the new procedure. This is true regardless of whether the existing command is a built-in command or a procedure. This means, for example, that you can redefine procedures at any time in the life of a Tcl-based application. It also means that you can redefine built-in commands like `if` (or even `proc`!) if you wish (this can occasionally be useful, but it may also cause your scripts to misbehave in confusing ways).

If you wish to redefine or re-arrange the command structure of an application, you may find the `rename` command useful. It takes two arguments:

```
rename oldName newName
```

`Rename` does just what its name implies: it renames the command that used to have the name *`oldName`* so that it now has the name *`newName`*. *`NewName`* 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 `newName` argument. For example, the following script disables all file I/O from an application by deleting the relevant commands:

```
rename open {}
rename read {}
rename gets {}
rename puts {}
```

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 very useful. For example, the `exit` command as defined by Tcl just exits the process immediately (see Section 11.4). 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.

Chapter 8
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 receive the right number of arguments, or because the arguments have the wrong form (e.g. a string with improper list structure passed to a list command), 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.

Tcl also allows errors to be "caught" by scripts so that only part of the work in progress in unwound. After catching an error, the script can either ignore the error or take steps to recover from it. If it can't recover then the script can then reissue the error. The error-handling facilities in Tcl also apply to a collection of *exceptions* including the `break`, `continue`, and `return` commands.The facilities for catching and reissuing errors are not needed very often in Tcl scripts, but when they are needed they can be used to achieve powerful effects. This chapter is organized with the most basic error facilities first and the more esoteric features at the end. Table 8.1 summarizes the Tcl commands related to errors.

8.1 What happens after an error?

When a Tcl error occurs, the command being processed 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

`catch` *command* ?*varName*?	Execute *command* as a Tcl script and return an integer code that identifies the completion status of the command. If *varName* is specified, it gives the name of a variable; the variable will be modified to hold the return value or error message generated by *command*.
`error` *message* ?*info*? ?*code*?	Generate 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.

Table 8.1. A summary of the Tcl commands related to errors.

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 error occurred and a human-readable error message. It is up to the application to decide how to handle this situation, but a typical response for an interactive application is to 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 commands accordingly, the application might print the error message into a log and abort.

As an 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}]
}
```

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 specified for the `foreach` command. If the script is executed in its current form an error will occur in the `expr` command: as it is processing its expression argument, it will attempt to substitute the value of variable `element`; it 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 execute 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 passed along with the error, and will probably be displayed for the user.

When an error occurs, three pieces of information about the error are available afterwards. The first piece of information is the error message. In simple cases this will provide enough information for you to pinpoint where and why the error occurred so you can avoid the problem in the future.

The second piece of information about errors is the global variable `errorInfo`, which is set by Tcl after each error. If a complex script was being executed when the error occurred, the message alone may not provide enough information for you to figure out where the error occurred. This is particularly true if nested procedure calls were active at the time of the error. To help you pinpoint the context of the error, Tcl stores information in the `errorInfo` variable as it unwinds the commands that were in progress. This information 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}]
}"
```

The third piece of information that is available after errors is the global variable `errorCode`. `ErrorCode` provides information in a form that is easy to process with Tcl scripts; it is most commonly used after catching errors as 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 `UNIX` then it means that an error occurred in a UNIX system call. `ErrorCode` will contain two additional elements giving the UNIX 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 10 and Chapter 11.

The `errorCode` variable is a relative late-comer to Tcl and is only filled in with useful information by a few commands, mostly dealing with file access and child processes. When an error occurs without any useful information available for `errorCode`, Tcl fills it in with the value `NONE`.

8.2 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)"
}
```

In this case `error` takes a single argument, which is the error message. The `error` command simply generates an error and uses its argument as the error message. `Error` can also have one or two additional arguments, which are used when reissuing errors (see Section 8.5 below).

As a matter of programming style, you should only use the `error` command in situations where the correct action is usually to abort the script being executed. If you think that an error is likely to be recovered from by the script in which it occurred 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 8.3 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.

8.3 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 processing Tcl commands 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` executes. If the script completes normally then `catch` returns 0. If an error occurs in the script, the `catch` command traps the error (so that the `catch` command itself is not aborted by the error) and it returns 1 to indicate that an error occurred. In the above example the `catch` command ignores any errors in `unset`; thus `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 succesfully, 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 succesfully. It's also useful if you need to do something with the error message after an error, such as logging it to a file.

8.4 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. These exceptions cause active scripts to be aborted just like errors, 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 implement the expected behavior by terminating the loop or going on 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.

If `break` or `continue` is invoked at a time when none of the enclosing commands is prepared to catch the exception then unwinding occurs just as for errors. After all of the active commands have been aborted the Tcl interpreter turns the exception into an error:

```
set x 22
if {$x < 30} {
    break
}
invoked "break" outside of a loop
```

`Break` and `continue` exceptions are also caught and turned into errors if they occur inside a procedure and are not caught within that procedure. If `return` is invoked at a point outside a procedure or `source`'d file then all the active commands are aborted and the Tcl interpreter turns the exception into a normal return:

```
set x 22
if {$x < 30} {
    return "all done"
}
all done
```

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 which 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 8.2). For example:

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

As an example of how `catch` might be used to deal with exceptions other than errors, consider the `for` command. In Section 7.4 you saw how `for` can be emulated with a Tcl procedure using `uplevel`. However, the example in Section 7.4 did not properly handle `break` or `continue` commands within the loop body. Here is a new implementation of the `for` procedure that uses `catch` to deal with them:

```
proc for {init test reinit body} {
    uplevel 1 $init
    while {[uplevel 1 [list expr $test]]} {
        set code [catch {uplevel 1 $body} string]
        case $code {
            0 {uplevel 1 $reinit; continue}
            2 return
            3 return
            4 {uplevel 1 $reinit; continue}
        }
        error $string
    }
}
```

This new implemenation of `for` executes the loop body inside a `catch` command so that exceptions in the body don't unwind past the `for` procedure. If no exception occurs, or if the exception is a continue, then `for` just goes on to the next iteration. If a break or return

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 from procedure or file `source`.	`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

Table 8.2. A summary of Tcl exceptions. The first column indicates the value returned by `catch` in each instance. The third column describes when the exception occurs and what is the value 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

exception occurs then `for` terminates the loop and returns. Lastly, if an error occurs then `for` reflects that error upwards using the error command. `For`'s handling of return isn't quite correct, since it should cause a return from the procedure in which `for` was invoked, not just from `for`. Unfortunately there is currently no way to achieve the desired behavior with the current Tcl implementation (this will be fixed soon).

8.5 Reissuing errors

The implementation of `for` as a procedure in the previous section has one remaining problem, which occurs when an error is generated by the loop body. The `for` procedure catches the exception, sees that it is an error, and reissues the error by invoking the `error` command. Unfortunately, neither `errorInfo` or `errorCode` will be set properly in this case. The variables will reflect the state of execution when `error` is invoked, whereas they should really reflect the state of execution at the time the original error occurred in the loop body. For example, suppose that the following command is typed after the `for` procedure has been defined as above:

```
set sum 0
for {set i 1} {$i <= 10} {incr i} {
    incr sum [expr i*i]
}
```

When this script is executed an error will be generated by the `expr` command because the dollar signs were accidentally omitted from the references to variable `i`. After the error is reissued and unwinding completes, `errorInfo` will have the following value:

```
syntax error in expression i*i
    while executing
"error $string"
    ("while" body line 9)
    invoked from within
"while {[uplevel 1 [list expr $test]]} {
    set code [catch {uplevel 1 $body} string]
    case $code {
        0 {uplevel 1 $reinit; continue}
        2 return
        ..."
"case [catch {uplevel 1 $body} string] {
            1 {error $string}
            2 return
            3 return
        }"
    ("while" body line 2)
    invoked from within
"for {set i 1} {$i <= 10} {incr i} {
    incr sum [expr $i * $i]
}"
```

Note that the error is attributed to the `error` command in the `for` procedure, not to the `expr` command that originally generated it. A similar problem occurs with `errorCode`: the `error` command will set it to `NONE`, thereby losing any information left there by the original error (in this particular case, there was no useful information anyway).

To solve both these problems, the `error` command may be given two additional arguments. The first of these is an initial value for `errorInfo`, which is used instead of the information that would have been recorded for the `error` command. This initial value is extended with additional entries as unwinding continues up through higher levels of active commands. The second additional argument is a value to place in the `errorCode` variable instead of the default `NONE`. In the `for` example both of these arguments can simply be supplied from the current values in the variables, which are the values left there by the original error:

```
proc for {init test reinit body} {
    global errorInfo errorCode
    uplevel 1 $init
    while {[uplevel 1 [list expr $test]]} {
        set code [catch {uplevel 1 $body} string]
        case $code {
            0 {uplevel 1 $reinit; continue}
            2 return
            3 return
            4 {uplevel 1 $reinit; continue}
        }
        error $string $errorInfo $errorCode
    }
}
```

When this new version of `for` is used with the erroneous `expr` command, `errorInfo` has the following value at the time the `error` command is executed in `for`:

```
syntax error in expression "i*i"
    while executing
"expr i*i"
    invoked from within
"incr sum [expr $i*$i]"
    ("uplevel" body line 2)
    invoked from within
"uplevel 1 $body"
```

This value describes all the active commands nested inside (but not including) the `catch` command. As the reissued error unwinds, more information gets added to `errorInfo` so that it has the following result when the error has been completely unwound:

```
syntax error in expression "i*i"
    while executing
"expr i*i"
    invoked from within
"incr sum [expr i*i]"
    ("uplevel" body line 2)
    invoked from within
"uplevel 1 $body"
    ("while" body line 1)
    invoked from within
"while {[uplevel 1 [list expr $test]]} {
    set code [catch {uplevel 1 $body} string]
    case $code {
            0 {uplevel 1 $reinit; continue}
            2 return
            ..."
    (procedure "for" line 4)
```

```
        invoked from within
    "for {set i 1} {$i <= 10} {incr i} {
        incr sum [expr $i * $i]
    }"
```

This information is still not completely perfect, since there is no mention in `errorInfo` of the `set` or `catch` commands that were active when the error occurred. However, information about these commands could be appended to `errorInfo` before passing its value to the `error` command.

Chapter 9
String Manipulation

This chapter describes Tcl's facilities for manipulating strings. The string commands mimic the behavior of C library procedures such as `scanf`, `printf`, and `strcmp`, plus they provide a few additional features not present in the C library. They allow you to generate formatted strings, parse strings to extract values, compare strings using any of several pattern-matching techniques, and modify strings (e.g. by removing trailing blanks or converting upper case characters to lower case). Table 9.1 summarizes the Tcl commands for string processing.

9.1 Generating strings with format

Tcl's `format` command provides almost exactly the same facilities as the `sprintf` procedure from the ANSI C library. It takes any number of arguments, of which the first is a format string and the others are values to convert and substitute into the format string. The `format` command combines the format string with the values to generate a new string, which it returns as result. A simple example follows below:

```
format "There are %d days in a week" 7
There are 7 days in a week
```

In this example the characters "`%d`" in the format string are replaced with the decimal value of the next argument to produce the result.

The `format` command operates by scanning the format string from left to right. Each character from the format string is appended to the result string unless it is a percent sign. If it is, then it is not copied to the result string. Instead, the characters following the `%`

`format` *`formatString`* ?*`value value`* ...?	Returns a result that is equal to *`formatString`* except that the *`value`* arguments have been substituted in place of `%` sequences in *`format-String`*.
`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 `0` if there is no match. If there is a match, then the command returns `1` and 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 the `%` sequences in *`format`* into the 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.

`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 9.1, cont'd. A summary of the Tcl commands for string manipulation.

character are treated as a *conversion specifier*. The conversion specifier controls the conversion of the next successive argument to a particular format and the result is appended to the result string. If there are multiple conversion specifiers in the format string, then each one controls the conversion of one additional argument. The `format` command must be given enough arguments to meet the needs of all of the conversion specifiers in the format string.

Each conversion specifier may contain up to five different parts: a set of flags, a minimum field width, a precision, a length modifier, and a conversion character. Any of these fields may be omitted except for the conversion character. The fields that are present must appear in the order given above. The paragraphs below discuss each of these fields in turn.

The flags portion of a conversion specifier may contain any of the following characters, in any order:

`-`	Specifies that the converted argument should be left-justified in its field (numbers are normally right-justified with leading spaces if needed).
`+`	Specifies that a number should always be printed with a sign, even if positive.
`space`	Specifies that a space should be added to the beginning of the number if the first character isn't a sign.
`0`	Specifies that the number should be padded on the left with zeroes instead of spaces.

`#` Requests an alternate output form. For `o` and `O` conversions it guarantees that the first digit is always `0`. For `x` or `X` conversions, `0x` or `0X` (respectively) will be added to the beginning of the result unless it is zero. For all floating-point conversions (`e`, `E`, `f`, `F`, `g`, and `G`) it guarantees that the result always has a decimal point. For `g` and `G` conversions it specifies that trailing zeroes should not be removed.

The second portion of a conversion specifier is a number giving a minimum field width for this conversion. It is typically used to make columns line up in tabular printouts. If the converted argument contains fewer characters than the mnimum field width, then it will be padded so that it is as wide as the minimum field width. Padding normally occurs by adding extra spaces on the left of the converted argument, but the `0` and `−` flags may be used to specify padding with zeroes on the left or with spaces on the right, respectively. If the minimum field width is specified as `*` rather than a number, then the next argument to the `format` command determines the minimum field width; it must be a numeric string.

The third portion of a conversion specifier is a precision, which consists of a period followed by a number. The number is used in different ways for different conversions. For `e`, `E`, `f`, and `F` conversions it specifies the number of digits to appear to the right of the decimal point. For `g` and `G` conversions it specifies the total number of digits to appear, including those on both sides of the decimal point (however, trailing zeroes after the decimal point will still be omitted unless the `#` flag has been specified). For integer conversions, it specifies a mimimum number of digits to print (leading zeroes will be added if necessary). For `s` conversions it specifies the maximum number of characters to be printed; if the string is longer than this then the trailing characters will be dropped. If the precision is specified as `.*` then the next argument to the `format` command determines the precision; it must be a numeric string.

The fourth part of a conversion specifier is a length modifier, which must be `h` or `l`. If it is `h` it specifies that the numeric value should be truncated to a 16-bit value before converting. If it is `l` it specifies that the numeric value should be extended to 32-bits before converting. Almost all machines that Tcl runs on use 32 bits by default, so the `l` modifier is seldom useful. For that matter, the `h` modifier is rarely useful either.

The last thing in a conversion specifier is an alphabetic character that determines what kind of conversion to perform. Table 9.2 lists the conversion characters that are available in the `format` command and the kind of conversion performed by each. For the numerical conversions the argument being converted must be an integer or floating-point string; `format` converts the argument to binary and then converts it back to a string according to the conversion specifier.

Here are a few examples of complete conversion specifiers and the results that they produce:

```
format %10d −243
```

```
       -243
format %010d -243
-000000243
format %-10d -243
-243      
format %10s "Two words"
 Two words
format %10.5s "Two words"
```

Character	Type of Conversion
`d`	Convert integer to signed decimal string.
`u`	Convert integer to unsigned decimal string.
`o`	Convert integer to unsigned octal string.
`x, X`	Convert integer to unsigned hexadecimal string (use `abcdef` for `x` and `ABCDEF` for `X`).
`c`	Convert integer to single ASCII character.
`s`	No conversion; just insert string.
`f`	Convert floating-point number to signed decimal string of the form *xx.yyy*, where the number of *y*'s is determined by the precision (default: 6). If the precision is 0 then no decimal point is output.
`e, E`	Convert floating-point number to scientific notation in the form *x.yyy*`e` *zz*, where the number of *y*'s is determined by the precision (default: 6). If the precision is 0 then no decimal point is output. If the `E` form is used then `E` is printed instead of `e`.
`g, G`	If the exponent is less than -4 or greater than or equal to the precision, then convert floating-point number as for `%e` or `%E`. Otherwise convert as for `%f`. Trailing zeroes and a trailing decimal point are omitted.
`%`	No conversion: just insert `%`.

Table 9.2. Conversion characters for the `format` command. The value being converted must have a proper integer or floating-point syntax as specified in the table.

```
      Two w
format %.2f -243
-243.00
format %.2e -243
-2.43e+02
```

The `format` command is really only needed if you're using relatively fancy conversion specifiers. If you want to do a simple substitution, you can already do it easily in Tcl without using `format`. For example, if the variable `days` has the value `30`, the following commands all produce the same result:

```
set x [format "There are %d days in April" $days]
There are 30 days in April
set x [format "There are %s days in April" $days]
There are 30 days in April
set x "There are $days days in April"
There are 30 days in April
```

Note that in this case `%d` behaves exactly the same as `%s` (except that it does more work and hence is probably slower); this would not be true if `days` had the hexadecimal value `0x30`.

9.2 Extracting characters: string index and string range

The `string` command provides a number of features for general-purpose string manipulation. It is actually about a dozen commands rolled into one; the first argument to `string` selects one of many options. For example, consider the command

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

When the first argument to `string` is `index`, as in this case, then there must be two additional arguments. The first of these may be any string value, and the last argument must be a number; `string index` uses the last argument as an index into the string and returns the indexed character as result. 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
```

In both `string range` and `string index` an empty string will be returned if the index or indices are completely outside the range of the string.

There are a number of places in Tcl where related commands are grouped together into a single Tcl command like `string` with a first argument that chooses among the various options. I did this to avoid polluting the Tcl command space with lots of tiny commands. If you build collections of related commands yourself, I recommend using this same approach for the commands you write.

9.3 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 `scanf` 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. If the next character in the format is a blank or tab then it is ignored. Otherwise, if it isn't a `%` character then it must match the next non-white-space character of the string. When a `%` is encountered in the format, it indicates the start of a conversion specifier. A conversion specifier contains three fields after the `%`: a `*`, which indicates that the converted value is to be discarded instead of assigned to a variable; a number indicating a maximum field width; and a conversion character. All of these fields are optional except for the conversion character.

When `scan` finds a conversion specifier in the format, it first skips any white-space characters in the input string. Then it converts the next input characters according to the conversion specifier and stores the result in the variable given by the next argument to `scan`. See Table 9.3 for a list of the conversion characters and their meanings. The number of characters read from the input for a conversion is the largest number that makes sense for that particular conversion (e.g. as many decimal digits as possible for `%d`, as many octal digits as possible for `%o`, and so on). The input field for a given conversion terminates either when a white-space character is encountered or when the maximum field width has been reached, whichever comes first. If a `*` is present in the conversion specifier then no variable is assigned and the next `scan` argument is not consumed.

For example, consider the following command:

```
scan 12345678 %*2d%3o%d a b
2
```

Character	Type of Conversion
`d`	The input field must be a decimal integer. It is read in and the value is stored in the variable as a decimal string.
`o`	The input field must be an octal integer. It is read in and the value is stored in the variable as a decimal string.
`x`	The input field must be a hexadecimal integer. It is read in and the value is stored in the variable as a decimal string.
`c`	A single character is read in and its ASCII value is stored in the variable as a decimal string. Initial white space is not skipped in this case, so the input field may be a white-space character. This conversion is different from the ANSI standard in that the input field always consists of a single character and no field width may be specified.
`s`	The input field consists of all the characters up to the next white-space character; the characters are copied to the variable.
`e, f, g`	The input field must be a floating-point number consisting of an optional sign, a string of decimal digits possibly containing a decimal point, and an optional exponent consisting of an `e` or `E` followed by an optional sign and a string of decimal digits. It is read in and stored in the variable as a floating-point string.
`[`*`chars`*`]`	The input field consists of any number of characters in *`chars`*. The matching string is stored in the variable. If the first character between the brackets is a `]` then it is treated as part of *`chars`* rather than the closing bracket for the set.
`[^`*`chars`*`]`	The input field consists of any number of characters not in *`chars`*. The matching string is stored in the variable. If the character immediately following the `^` is a `]` then it is treated as part of the set rather than the closing bracket for the set.

Table 9.3. Conversion characters for the `scanf` command.

The return value of 2 indicates that two values were successfully converted and assigned to variables (three conversions were performed but the first converted value was discarded because of the * in its conversion specifier). After the command completes variable a has the value 229 (the decimal equivalent of the octal value 345) and b has the value 678.

9.4 Simple searching and comparison

This section and the next two that follow describe several ways to search strings for particular substrings or compare strings using various pattern-matching techniques. This section presents three simple mechanisms that are available as options of the string command.

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

```
string first th "In 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 "In the tub where I bathed today"
21
```

The command string compare takes two additional arguments and compares them in their entirety. It returns −1 if the first string is lexicographically less than the second, 0 if they are identical, and 1 if the first is lexicographically greater than the second:

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

9.5 Glob-style pattern matching

Tcl offers two different kinds of pattern matching: “glob” style, which is named after the file name matching used in shells, and regular expressions. This section describes the glob style and the next section describes regular expressions.

The command string match implements glob-style pattern matching. It takes two additional arguments, a pattern and a string, and 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 (differences in case are significant), except that the following pattern characters are interpreted specially:

`?`	Matches any single character.
`*`	Matches any sequence of zero or more characters.
`[`*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.

Glob-style matching is similar to that used by the shells for file names. The following commands illustrate some of the features of glob-style matching:

```
string match a*b*a abracadabra
1
string match a?[1234567890A-Z] abX
1
string match a?[1234567890A-Z] abbX
0
```

9.6 Pattern matching with regular expressions

The glob style of matching described in the previous section is simple and easy to work with, but it is limited in the kinds of patterns that can be expressed. Tcl's second form of pattern matching uses regular expressions like those available in the `egrep` program. Regular expressions are more complex than glob-style patterns but much more powerful. Tcl's regular expressions are based on Henry Spencer's publically 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 9.4. 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

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 9.4. The special characters permitted in regular expression patterns.

`.` 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 in the `regexp` and `regsub` commands 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 `]`. It is possible to specify a `-`, `^`, or `]` as one of the acceptable or unacceptable characters of the range, but only with special care about where the character appears in the range; see Table 9.4 for details.

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 `|`. Thus `^((0x)?[0-9a-fA-F]+|[0-9]+)$` matches any string that is either a hexadecimal number or a decimal number. 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 is used to invoke regular expression matching in Tcl. 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
```

Note that the pattern had to be enclosed in braces so that the characters `$`, `[`, and `]` are passed through to the `regexp` command instead of triggering variable and command substitution.

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- 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". If there are extra variables specified with the -indices option, they are set to "-1 -1".

In general there may be more than one way to match a regular expression to an input string. For example, consider the following command:

```
regexp (a*)b* aabaaabb x y
1
```

Considering only the rules given so far, x and y could have the values aabb and aa, aaab and aaa, ab and a, or any of several other combinations. To resolve this potential ambiguity the regular expression parser chooses among alternatives using the rule "first then longest." In other words, it considers the possible matches in order working from left to right across the input string and the pattern, and it attempts match longer pieces of the input string before shorter ones. More specifically, the following rules apply in decreasing order of priority:

1. If a regular expression could match two different parts of an input string then it will match the one that begins earliest.

2. If a regular expression contains | operators then the leftmost matching sub-expression is chosen.

3. In *, +, and ? constructs, longer matches are considered before shorter ones.

4. In sequences of expression components the components are considered from left to right.

In the example from above, (a*)b* matches aab (the (a*) portion of the pattern is matched first, and it consumes the leading aa; then the b* portion of the pattern consumes the next b). After the command

```
regexp (ab|a)(b*)c abc x y z
```

x will be abc, y will ab, and z will be an empty string. Rule 4 specifies that (ab|a) gets first shot at the input string and Rule 2 specifies that the ab sub-expression is checked before the a sub-expression. Thus the b has already been claimed before the (b*) component is checked and (b*) must match an empty string.

9.7 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 not have been modified.

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 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*. Backslashes may be used in the replacement string to allow &, \0, *n*, or backslash characters to be substituted verbatim without any special interpretation. 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. In general it's 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.

9.8 Length, case conversion, and trimming

The string command provides three additional features that haven't yet been discussed: length counting, 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 10
Accessing Files

This chapter describes the built-in Tcl 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 10.1 for a summary of the file-related commands.

The commands described in this chapter are only available on UNIX-like systems and systems that support the kernel calls defined in the POSIX standard. 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; talk to your local Tcl wizards to see what is available.

10.1 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 named `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?`	Change 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?`	Close 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 ...?`	Perform one of several operations on the filename given by *name* or on the file that it refers to, depending on *option*. See Table 10.2 for details.
`flush fileId`	Write out any buffered output that has been generated for *fileId*. Returns an empty string.
`gets fileId ?varName?`	Read the next line from *fileId* and discard its terminating newline. If *varName* is specified, place the line in that variable and return a count of characters in the line (or `-1` for end of file). If *varName* isn't specified, return line as result (or an empty string for end of file).
`glob ?-nocomplain? ?pattern pattern...?`	Return a list of all file names that match any of the *pattern* arguments, using `csh` rules for pattern matching (special characters `?`, `*`, `[]`, `{}`, and `\`). If `-nocomplain` isn't specified then an error occurs if the return list would be empty.
`open name ?access?`	Open file *name* in the mode given by *access*. Access must be `r`, `r+`, `w`, `w+`, `a`, or `a+` and 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 11.2 for more information).
`puts fileId string ?nonewline?`	Write *string* to *fileId*, appending a newline character unless `nonewline` is specified. Returns an empty string.
`pwd`	Returns the full path name of the current working directory.

Table 10.1. A summary of the Tcl commands for manipulating files (continued on next page).

`read` *`fileId`* ?`nonewline`?	Read and return all of the bytes remaining in *`fileId`*. If `nonewline` is specified then the final newline, if any, is dropped.
`read` *`fileId numBytes`*	Read and return 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 10.1, cont'd. A summary of the Tcl commands for manipulating files.

10.2 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. To access a file you must first open it with the `open` command:

```
open main.c r
file3
```

The `open` command takes as 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 to access it from the beginning. The access mode must have one of the following values:

`r`	Open for reading only. The file must already exist.
`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.

If you don't specify an access mode then it defaults to `r`.

The `open` command returns a string that identifies the open file, such as `file3` in the above example. 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. Right now they have the format `file`*x* where *x* is the UNIX descriptor number for the file, but this format might change in the future.

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.

Once you've opened a file you can read and write it using the `gets`, `read`, and `puts` commands. `Gets` is used for reading files a line at a time, and it has two forms. In the most common form, you invoke `gets` with two arguments, which are a file identifier and the name of a variable:

```
gets file3 line
18
set line
#include <stdio.h>
```

In this case `gets` 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 an empty string is stored into the variable and `-1` is returned.

You can also omit the variable name when invoking `gets`. In this case the contents of the line (minus the newline, of course) are returned as the command's result. If the end of the file is reached before reading any characters then an empty string is returned. An empty string is also returned for a line with no characters except the newline, but you can use the `eof` command described in Section 10.3 below to tell the difference between these two cases.

The `read` command may be used for non-line-oriented input. It takes either two or three arguments, of which the first is always a file identifier. If `read` is invoked with only a single argument then it reads all of the remaining bytes from the file and returns them as result. If `nonewline` is specified as the second argument, then the last character of the file is discarded if it is a newline. Otherwise the second argument must be a number telling how many bytes to read: `read` will read this many bytes from the file and return them as its result. If there are fewer bytes left in the file than the number requested, then all of the remaining bytes will be returned. For example, the command

```
set buffer [read file3 1000]
```

will read the next 1000 bytes from `file3` and place them in the variable `buffer`.

The `puts` command writes data to an open file. It takes two or three arguments, of which the first is a file identifier and the second is a string to output. If only two arguments are provided, as in

```
puts stdout "Hello, world"
```

then `puts` appends a newline character to the string and outputs it to the file. In the above example, "`Hello, world`" is printed to standard output followed by a newline character. If a third argument is specified to `puts` then it must be the keyword `nonewline` or an abbreviation of it. This causes `puts` not to append a newline character to the string.

`Puts` 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 most cases it 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 file3
```

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.

When you are finished reading or writing a file you should invoke the `close` command, giving it the identifier for the file as its argument:

```
close file3
```

`Close` will flush any buffered data for the open file and release the resources associated with it. In most systems there is a limit on how many files may be open at one time, so it is important to close files as soon as you are finished reading or writing them.

10.3 Random access to files

File I/O is sequential by default: each `read` or `gets` command returns the next bytes after the previous `gets` or `read` command, and each `puts` command writes the bytes immediately following those 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 determines 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 is incremented 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 file3 2000
```

changes the access position for `file3` so that the next read or write will start at byte number 2000 in the file. The command

```
seek file3 0
```

resets the file's access position to the beginning of 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. For example, the following command moves the access position forward 10 bytes, skipping over the intervening data:

```
seek file3 10 current
```

If the origin is `end` then 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 file3 -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. It is possible to seek past the current end of the file, in which case the file will 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 file3
186
```

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

`Seek` and `tell` may only be used on files that support random-access I/O, such as ordinary disk files. If you attempt to use `seek` with a file identifier that doesn't support random access I/O, such as a terminal or other sequential device, then `seek` will generate an error. If you invoke `tell` on such a file then it will return `-1`.

The `eof` command indicates whether an open file is currently positioned at the end of the file. It takes a file identifier as argument and returns `1` if the current access position is at the end of the file, `0` otherwise:

```
eof file3
0
```

10.4 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).

10.5 Manipulating file names

Tcl contains two built-in commands that you can use to manipulate file *names* as opposed to file contents. These commands don't provide any new functionality, since you could produce the same effects using other Tcl commands, but they make it easy to perform several common operations on file names.

The first of these commands 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 Table 10.2 for a summary of all the options to `file`. This section discusses the name-related options and Section 10.6 describes the other options.

`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
```

`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`*.
`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` *`varName`*	Invokes the `lstat` kernel 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 10.2. A summary of the options for the `file` command (continued on next page).

`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.
`file size` *name*	Returns a decimal string giving the size of file `name` in bytes.
`file stat name` *varName*	Invokes `stat` kernel 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 10.2, cont'd. A summary of the options for the `file` command.

```
foo
```

These `file` commands all operate purely on file names. They make no system calls and don't check to see if the names actually correspond to files.

The `glob` command also operates on file names. It mimics the behavior of file name globbing in `csh`, taking one or more patterns as arguments and returning a list of all the file names that match the pattern(s):

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

It uses the same matching rules as the `string match` command (see Section 9.5) and returns the names of all of the matching files. In the above example `glob` returned all of the file names in the current directory that end in `.c` or `.h`.

If a pattern contains an open-brace, then the brace should be followed by one or more strings separated by commas and terminated with a close-brace, such as `{src,backup}/*.c`. Glob treats such a pattern as if it were actually multiple patterns, one containing each of the strings between the braces (`src/*.c` and `backup/*.c` in this example). For example, the following command returns a list of all of the `.c` or `.h` files in two subdirectories:

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

The extra set of braces around the glob pattern is needed to keep the Tcl parser from applying its usual interpretation to the {} and [] characters within the pattern.

`Glob` patterns may contain multiple sets of `{}` elements, or any combination of the various special characters. If a pattern contains any of the string matching characters `[]?*\` then `glob` only returns the names of actual files that match the pattern. If a pattern doesn't contain any of the string matching characters then `glob` returns names without checking to be sure that the corresponding files actually exist. This behavior may seem strange but is similar to what occurs in `csh`.

If the list of file names to be returned by `glob` is empty then it normally generates an error, as in the following command:

```
glob *.x *.y
no file matched glob patterns "*.x *.y"
```

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.

10.6 File information commands

In addition to the options already discussed in Section 10.5 above, the `file` command provides many other options that can be used to retrieve information about files. Each of these options except `stat` 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 10.2 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. `File exists` also returns `0` if the file exists but 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` results 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 UNIX 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.

10.7 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
UNIX ENOENT {no such file or directory}
```

The first element is always `UNIX` to indicate that the error occurred in a `UNIX` 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 11
Processes

Tcl provides simple facilities for dealing with processes. You can create new processes with the `exec` command, or you can create new processes with `open` and then use the file I/O commands to communicate with the process(es). In addition, 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 10, these commands are only available on UNIX systems and systems that support the kernel calls defined in the POSIX standard. Table 11.1 summarizes the commands related to process management.

11.1 Invoking subprocesses with exec

The `exec` command creates one or more subprocesses and, in the normal case, 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 subprocess to execute and each additional argument forms one argument to that subprocess. It's important to realize that each argument to `exec` forms a single argument to the subprocess. For example, consider the following command:

```
exec "rm main.o"
couldn't find "rm main.o" to execute
```

`exec` *`arg`* ?*`arg`* ...?
: Execute command pipeline specified by *`arg`*'s as a subprocess. I/O redirection may be specified with <, <<, and >, pipes may be specified with |, and background execution may be specified with a final arg of `&`. Returns standard output produced by command (without trailing newline, if any) or an empty string if output is redirected.

`exit` ?*`code`*?
: Terminate process, returning *`code`* to parent as exit status. *`Code`* must be an integer. If *`code`* isn' t specified, return 0 as exit status.

`open` |*`command`* ?*`access`*?
: Treat *`command`* as a list with the same structure as arguments to `exec` and create subprocess(es) to execute command(s). Depending on *`access`*, create pipes for writing input to pipeline and reading output from it.

Table 11.1. A summary of Tcl commands for manipulating processes.

An error occurred because `exec` tried to use "`rm main.o`" as the name of the subprocess and couldn't find an executable file by that name.

To execute a subprocess, `exec` looks for an executable file with a name equal to `exec`'s first argument. If the name starts with `/` or ~ then `exec` checks the single file indicated by the name. If the name doesn't start with `/` or ~ then `exec` checks each of the directories in the `PATH` environment variable to see if the command name refers to an executable reachable from that directory. `Exec` uses the first executable that it finds.

Under normal conditions `exec` collects all of the information written to standard output by the subprocess and returns that information as its result. If the last character of output is a newline then `exec` removes the newline from what it returns as result (this behavior may seem strange but it makes `exec` consistent with other Tcl commands,which don't normally terminate the last lines of their results). For example:

```
exec echo foo bar
foo bar
```

The arguments to `exec` may specify input and output redirection in a fashion similar to the UNIX shells. If one of the arguments to `exec` is > then the following argument is taken as the name of a file. The subprocess's standard output will be redirected to that file and `exec` will return an empty string as result.

Standard input may be redirected using either < or <<. If one of `exec`'s arguments is < then the following argument is taken as a file name and the subprocess's standard input is taken from that file. If one of `exec`'s arguments is << then the following argument is taken as an immediate value to be passed to the subprocess as its standard input:

```
exec cat << "test input"
test input
```

If no input redirection is specified then the subprocess inherits the standard input channel from the process executing the `exec` command.

The arguments to `exec` may also specify a pipeline of proceses to execute instead of a single process. This is done in the standard fashion with the | character. If one or more of the arguments to `exec` are | then the | arguments separate the specifications for the different subprocesses. The first agument in each subprocess specification is the name of the file to execute for that subprocess and the remaining arguments are arguments to that subprocess. The standard output of each subprocess is piped to the standard input of the next subprocess. I/O redirection may be specified using <, <<, or > anywhere among `exec`'s arguments; it will apply to the first subprocess for input redirection and to the last process for output redirection.

If any of the subprocesses exits abnormally (i.e. it was killed or suspended or returned a non-zero exit status), or if any of them generates output on its standard error channel, 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 (if any), followed by the information generated on standard error by the processes, if any. If the last character of standard error output is a newline, then it is deleted. In addition, `exec` will set the `errorCode` variable to hold information about the last process that terminated abnormally, if any (see Table 11.2 for details).

If the last argument to `exec` is `&` then the subprocess(es) will be executed in background. `Exec` will return an empty result immediately, without waiting for the subprocesses to complete. Standard output from the subprocesses will go to the standard output of the process in which `exec` was executed, 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 process in which `exec` was executed.

Although `exec`'s mechanisms for I/O redirection, pipelines, and background execution are similar to those of the UNIX shells, there are a few differences. Special characters like <, |, and `&` must appear as distinct arguments to `exec` if they are to receive special treatment (i.e. they must be surrounded by white space). The shells are generally less particular about requiring white space. In addition, `exec` doesn't perform all of the substitutions performed by shells. In particular, `exec` doesn't perform file name "globbing" in response to characters like `*` and `?`. If you want globbing to occur you must request it explicitly using the `glob` command described in Section 10.5. For example, to remove all of the `.o` files in the current directory you can't use the command

```
exec rm *.o
```

Instead, use the (admittedly more complicated) command

CHILDKILLED *pid sigName msg*	Used when a child process has been killed because of a signal. The second element of `errorCode` is the process's identifier (in decimal). The third element is the symbolic name of the signal that caused the process to terminate; it will be one of the names from the include file `signal.h`, such as `SIGPIPE`. The fourth element is a short human-readable message describing the signal, such as "`write on pipe with no readers`" for `SIGPIPE`.
CHILDSTATUS *pid code*	Used when a child process exits with a non-zero exit status. The second element of `errorCode` is the process's identifier in decimal and the third element is the exit status returned by the process, in decimal.
CHILDSUSP *pid sigName msg*	Used when a child process has been suspended because of a signal. The second, third, and fourth elements of `errorCode` have the same meaning as for `CHILDKILLED` above.

Table 11.2. Values placed in the `errorCode` variable by the `exec` command. The top line of each entry in the table gives the value of the first element of the list that comprises `errorCode` and provides symbolic names for the remaining elements of `errorCode`. The text describes the conditions under which that format for `errorCode` is used and explains the meaning of the

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

11.2 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 read the subprocesses' standard output and write their standard input. To create subprocesses with `open`, invoke it with the pipe symbol | as the first character of the file name. In this case the file name 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. If writing was requested in the access mode to `open` then a pipe will be used for standard input to the first process in the pipeline and you can invoke `puts` to write data on that pipe (remember that the data may not become visible to the process until you invoke `flush`). If reading was requested in the access mode then a pipe will be used for the standard output of the last process in the pipeline and you can use `gets` and `read` to retrieve the output generated by that process.

Here is an example of opening a command pipeline:

```
open {|tbl | ditroff -ms} w
file4
```

This command creates a pipeline containing two processes running the document formatting programs `tbl` and `ditroff`. Any data written to `file4` with `puts` will be passed to the `tbl` process; `tbl`'s output will be passed to `ditroff` as input; and `ditroff`'s output, if any, will go to the standard output file of the process executing the Tcl script.

If a command pipeline is opened for writing then it is an error to redirect the pipeline's standard input. If the pipeline isn't opened for writing then its input will be taken by default from the standard input of the process that executed the `open` command, but it may be redirected as part of the `open` command. If a command pipeline is opened for reading then it is an error to redirect the pipeline's standard output. If the pipeline isn't opened for reading (as in the above example) then the pipeline's standard output goes by default to the standard output of the process that executed the `open` command, but it may be redirected.

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`. If there is unread output from the pipeline at the time of the `close` command then it is lost when the output pipe is closed.

11.3 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 envionrment 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 child process created with `exec` or `open`.

11.4 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 a single 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 12
History

This chapter describes Tcl's history mechanism. The history mechanism keeps track of commands that you have typed recently 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.

Tcl's facilities provide the same general features as the history mechanism in `csh`. However, in order to keep the Tcl language syntax simple I didn't add all of `csh`'s history syntax into the Tcl language syntax. Instead, history is implemented with a `history` command that has several options summarized in Table 12.1 . The `history` command requires you to type more characters than the super-concise `csh` syntax, but you can always use the `history` command to build your own short-hands (or re-implement the `csh` syntax) if you wish. In fact, the `unknown` command described in Section 13.5 already implements some of the `csh` short-hands such as `!!`, `!`*`event`*, and `^`*`old`*`^`*`new`*.

History is an optional feature in Tcl and is only present in applications that request it. It's really only useful in applications where you type Tcl commands interactively, such as Tcl-based shells, and it tends to be available only in these applications.

12.1 The history list

In applications that use the history mechanism, each command that you type interactively is entered into a *history list*. The application arranges for this to happen before it executes the command. Only the commands that you actually type are saved in the history list. Commands that are executed by Tcl procedures or read from script files are not recorded.

`history`	Same as `history info`.
`history add` *`command`* ?*`exec`*?	Add *`command`* to the history list as a new event. If `exec` is specified (or abbreviated) then also execute *`command`* and return its result. Otherwise an empty string is returned.
`history change` *`newValue`* ?*`event`*?	Replace the value recorded for *`event`* with *`newValue`*. *`Event`* defaults to the current event, not `-1`. Returns an empty string.
`history event` ?*`event`*?	Returns the value of *`event`*. History revision occurs.
`history info` ?*`count`*?	Returns a human-readable string giving the event number and command for each event in the history list. If *`count`* is specified then only the *`count`* most recent events are returned.
`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-execute the command recorded for *`event`* and return its result. History revision occurs.
`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 return its result. History revision occurs. Both *`old`* and *`new`* are simple strings. The substitution uses simple equality checks: no wild cards or regular expression features are supported.
`history words` *`selector`* ?*`event`*?	Retrieve from the command recorded for *`event`* the words given by *`selector`*, and return those words in a string separated by spaces. *`Selector`* can consist of a single number (`0` for the first word, `1` for the next, and so on), `$` to select the last word, two numbers separated by a dash to select a range of words (`$` may be used as the second "number"), or a pattern to select all words that match that pattern (the rules for `string match` are used in pattern matching).

Table 12.1. A summary of the options for the `history` command.

The idea behind history is to save typing; commands in procedures and script files are already recorded so they can be re-executed without re-typing them. The examples in this chapter assume that you've typed each of the commands, so that they are entered into the history list.

Each entry in the history list is referred to as 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
     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. Notice that the history command itself generates a fourth event on the list. 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 this chapter, such as `history event`.

The command `history info` provides a more selective way to print out events. For example, suppose you typed the following command instead of the `history` command above:

```
history info 3
     2 set y [expr $x*2.6]
     3 incr x
     4 history
```

The argument to `history info` determines how many events wll be returned from the history list; only information for that number of the most recent commands will be returned. If the last argument is omitted then `history info` behaves the same as `history` with no arguments.

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.

12.2 Specifying events

Several of the options of the `history` command require you to select an entry from the history list; the symbol *`event`* is used for such arguments in Table 12.1. 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 event just prior to the current event, `-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 109 above. If the next command refers to a history event as `-1` or `3` or `inc` then it selects the command `incr x`. If a history event is referred to as `-2` or `2` or `*2*` then it selects the command `set y [expr $x*2.6]`. If an event specifier is omitted then it defaults to `-1` for all options except `history change`.

12.3 Re-executing commands from the history list

Two of the options to `history` may be used to replay commands from the history list. `History redo` retrieves a command and re-executes it just as if you had typed the entire command in place of the `history redo` command. For example, after typing the three commands from page 109, 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 12.2; for example,

```
history redo 1
24
```

replays the first command, `set x 24`.

In the examples above it takes more keystrokes to type the `history` commands than it would take to simply retype the command from the history list. Given that the whole purpose of the history mechanism is to save typing, the commands above probably don't seem very useful. However, there are a number of shortcuts you can use to reduce your typing. First, `history`, like all Tcl commands, accepts unique abbreviations for its options, so you can just type `r` instead of `redo` as the option. Second, any application that uses the history mechanism should also allow abbreviations for commands typed interactively (this is implemented using the `unknown` procedure described in Section 13.5). Thus you should be able to replay the most recent command simply by typing

```
h r
```

which requires only four keystrokes including the return.

In addition, the same `unknown` mechanism that implements command abbreviations also simulates the `!!` and `!`*`event`* history mechanisms from `csh` using the `history redo` command. Thus you can type "`!!`" instead of "`history redo`" and "`!13`" instead of "`history redo 13`."

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.

The `history substitute` command above also takes more keystrokes than retyping the original command, but again there are shortcuts. One possibility is to abbreviate the words `history` and `substitute` and omit the event specifier:

```
h s ill mill
200 millimeters
```

Another possibility is take advantage of the fact that the `unknown` mechanism also simulates the `^`*`old`*`^`*`new`* syntax of the `csh` history mechanism using `history substitute`, so you can just type the following:

```
^ill^mill
200 millimeters
```

12.4 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
history
     1 set x 24
     2 history nextid
     3 history
```

By the time `history nextid` was executed the command had already been inserted into the history list as event 2, so the command returned `3`, the number of the next event.

`History nextid` 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.

12.5 Retrieving without re-executing

The commands `history event` and `history words` allow you to retrieve information from the history list without necessarily re-executing it. `History event` returns the command from an indicated event:

```
set x 24
set y [expr 2*$x]
history event -2
set x 24
```

As with other `history` options, the event can be omitted, in which case it defaults to `-1`.

The `history words` command returns one or more words from a command on the history list. It takes two additional arguments. The first indicates which words are wanted and the second is an optional event specifier:

```
set x 24
history words 0
set
```

In this case the first word of the preceding command was returned. The word specifier may have any of the following forms:

number	Selects the word given by number, with `0` corresponding to the first word, `1` to the next, and so on.

`$`	Selects the last word of the event.
`first-last`	Selects all of the words from *`first`* through *`last`*, inclusive. *`First`* must be a number; *`last`* may be a number or `$`.
`pattern`	Selects all the words that match *`pattern`* using the rules for `string match`.

When `history words` returns multiple words, it does not return them as a proper Tcl list. It simply concatenates the values with spaces between them. This approach is used because `history words` will most commonly be used as part of generating a new Tcl command that will be executed immediately. If the result of `history words` were made into a proper list, it would quote all of the special characters like `[]` and `$` inside the words, which would probably cause the command to do the wrong thing when executed.

12.6 History revision

The `history` options `event`, `redo`, `susbstitute`, and `words` all perform *history revision*. What this means is that these options modify the history list as part of their execution. To see the reason for history revision, consider the following command sequence:

```
incr x
history redo
history redo
```

Suppose there were no history revision. Then when the second `history redo` command is executed, the history list will be as follows:

```
1 incr x
2 history redo
3 history redo
```

The second `history redo` command will replay event 2, which is another `history redo` command, and an infinite loop will occur. The problem is that `history redo` is context sensitive: it only makes sense at a particular point in time and won't produce the same effect if it is replayed later.

History revision avoids this problem and several others by replacing `history` commands on the history list with the information that they return or replay. In the above example, the first `history redo` command replaces its entry in the history list with `incr x`, so that the history list looks like this when the second `history redo` command is executed:

```
1 incr x
2 incr x
3 history redo
```

The second `history redo` then replaces event 3 on the history list with `incr x` as part of its execution.

Similar history revision occurs for the `event`, `substitute`, and `word` options. For example, suppose the following command has just been executed:

```
set a [expr $b+2]
```

The table below shows a number of commands that might be typed after the above command and what will be recorded on the history list after the command carries out its history revision:

Command typed	Command recorded
`history redo`	`set a [expr $b+2]`
`history s a b`	`set b [expr $b+2]`
`set c [history w 2]`	`set c [expr $b+2]`
`set d [history event -1]`	`set d {set a [expr $b+2]}`

One final (obscure) note about history revision: it occurs even when `history` isn't the top-level command typed by the user. For example, if a user types `foo` and `foo` is a procedure that invokes `history redo`, then `foo` is replaced on the history list with the command that is replayed. This behavior turns out to do the right thing in most cases. In cases where this isn't the right behavior you can use `history event` to save the old contents of the event and `history change`, described below, to restore its value later.

12.7 Modifying the history list

The last two `history` options allow you to change the contents of the history list. The `history change` command modifies an event on the history list. It takes as arguments a new value to record for an event and an optional event specifier. In this command the event specifier defaults to the *current event* rather than to the previous event. For example, the `history change` command in the following sequence replaces its own entry in the history list:

```
set x 24
history change "strange value"
history
     1 set x 24
     2 strange value
     3 history
```

The `history add` command adds a new event to the history list and optionally executes it. For example, the following command adds `set x 24` to the history list as a new event:

```
history add "set x 24"
```

If an additional `exec` argument (or any abbreviation of it) is specified then the command will be executed as well as being added to the list.

Chapter 13
Accessing 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, or to handle references to undefined commands. Table 13.1 summarizes 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 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*	Terminate search *searchId* of array *name* and discard any state associated with the search. Returns an empty string.
`array names` *name*	Returns a list whose entries are 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`.
`info` *option* ?*arg arg* ...?	Provides information about the internal state of the Tcl interpreter, depending on *option* and *arg*'s. See Table 13.2 for details.
`time` *command* ?*count*?	Executes *command count* times and returns a string indicating the average elapsed time per execution. *Count* defaults to 1.
`trace variable` *name ops command*	Establish 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*, remove 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 isn't implemented by Tcl, but if it is defined then it 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 13.1. A summary of commands for manipulating Tcl's internal state

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)
    }
}
```

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. Using this approach, the example above looks like this:

```
set id [array startsearch a]
while [array anymore a $id] {
    set i [array nextelement a $id]
    if {($a($i) == "") || ($a($i) == 0)} {
        unset a($i)
    }
}
array donesearch a $id
```

The `array startsearch` command initiates a search through all of the elements of an array. It returns an identifier for that search, which the above code saves in variable `id`. This identifier must be passed to the `anymore`, `nextelement`, and `donesearch` options to identify the search. It's legal to call `array startsearch` several times with the same variable so that several searches are underway simultaneously; each will have a different identifier. The exact format of the search identifier isn't important; all you need to know is that it is returned by `array startsearch` and must be passed into the other searching commands.

The `array anymore` command indicates whether there are any more elements left in a search. It returns `1` if there are and `0` if all of the element names have already been returned in this search. If there are elements left, `array nextelement` will return the name of the next element. The element names are not returned in any particular order. If there are no elements left in a search then `array nextelement` returns an empty string. However, it may be dangerous to use the return value from `array nextelement` to detect the end of the search, since it is possible for an array element to have an empty string for its name.

When you are finished with a search you must invoke `array donesearch` to tell Tcl that you're done; this allows Tcl to free up all of its state associated with the search. If you forget to call `array donesearch` then Tcl's state will remain allocated; if you do this often then it will result in wasted memory and inefficient operation of future array searches.

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 Table 13.2.

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
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. Suppose also that a procedure is being executed with arguments named `arg1` and `arg2`, and that the procedure has executed a `global` command to make `global2` accessible, and that the procedure has also created local variables named `local1` and `local2`. Then the following commands might be executed in the procedure:

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

`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`*. *`Number`* may have any of the formats accepted by `uplevel`.
`info library`	Returns the full path name of the library directory in which standard Tcl scripts are stored.

Table 13.2. A summary of the options for the `info` command (continued on next page).

`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).
`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. Increases in *minor* correspond to bug fixes, new features, and backwards-compatible changes. *Major* increases 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 13.2, cont'd. A summary of the options for the `info` command.

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 could 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. The level argument may be specified in any of the forms described in Section 7.4 for the `uplevel` command. For example, the following procedure prints out the current call stack, showing the name and value for each argument of each active procedure:

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

In principle, of course, there is no such thing as a perfectly compatible change. Adding a new command to Tcl might break scripts that define a procedure with the same name, and fixing a bug might break a script that only works because of the bug. But in practice you should be able to upgrade to new versions with little or no effort as long as the major version number hasn't changed.

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 13.5 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).

To create a trace, invoke the `trace` command with the `variable` option:

```
trace variable x rwu xtrace
```

The first argument to `trace variable` is the name of the variable to trace (`x` in the example). The next argument is a string whose characters indicate the operations to be traced, `r` for reads, `w` for writes, and `u` for unsets. In the example above all operations will be traced, but that need not be the case in general. The last argument to `trace vari-`

`able` is a Tcl command to invoke whenever one of the selected operations occurs; typically this is the name of a procedure.

The variable name specified in `trace variable` may take any of three forms. First, it may be the name of a scalar variable, in which case a trace will be established on that variable. Second, it may be the name of an array element, in the usual form, such as `a(b)`. This results in a trace on the given element; other elements of the same array will not be affected by the trace. Third, the variable may be specified as the name of an array without any element specification. In this case the trace applies to all of the elements of the array, including new elements created after the trace is established.

When a traced operation occurs, Tcl invokes the trace command by appending three additional arguments to the command specified for the trace:

```
command name1 name2 op
```

The first part of the command will be exactly the same as the command specified in the `trace variable` command. *`Name1`* and *`name2`* give the name of the variable being accessed and *`op`* gives the operation being performed (`r` for read, `w` for write, or `u` for unset). If the variable is a scalar then *`name1`* is the variable's name and *`name2`* is an empty string. If the variable is an element of an array then *`name1`* is the name of the array and *`name2`* is the name of the element within the array. If an unset trace exists for an entire array and the array is deleted, then the trace will be invoked with *`name1`* equal to the array's name and *`name2`* an empty string.

For example, after a trace is set on variable `x` in the example above, the following command will be invoked in response to each read of variable `x` (assuming that `x` is a scalar variable):

```
xtrace x {} r
```

The command specified for a trace need not be a single word as in the above example. For example, if the trace had been set with the following command:

```
trace variable x rwu {xtrace 24 $x}
```

then reads of `x` would cause the following trace comand to be invoked:

```
xtrace 24 $x x {} r
```

The trace command is invoked in the execution context where the variable access occurred. Thus if the variable is accessed in a Tcl procedure then the trace command will have access to the same local variables as the code of the procedure. This context may be different than the context where the trace was created. In the normal case where the trace command invokes a Tcl procedure, the commands in the trace procedure will have to use `upvar` or `uplevel` to access the traced variable. Note also that *`name1`* and *`name2`* as passed to the trace command are the names used to access the variable. They may not be the same as the names under which the trace was created; differences occur if the access is made through a variable defined with `upvar`.

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. If the trace command returns an error of any sort then the traced operation is aborted with an error message saying that the trace command denied access; otherwise the result returned by the trace command is ignored.

Write traces are invoked after the variable's value has been modified but before reading the new value to return 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. The trace command can return an error in the same way as for read traces to deny access; this can be used to implement read-only variables, for example (however, the trace command will have to restore the old value of the variable, since the value will already have been modified before the trace command is invoked). As with read traces, the result of the trace command is ignored unless it is an error.

Tracing is temporarily disabled for a variable during the execution of read and write trace commands. This means that the trace commands can access the variable without causing traces to be invoked recursively. If there are multiple traces for a variable, all of them are disabled when any of them is executing.

Unset traces are invoked after the variable has already been deleted. From the standpoint of the trace command, the variable will appear to be undefined with no traces. If an unset occurs because of a procedure return then the trace will be invoked in the variable context of the procedure being returned to; the variable context of the returning procedure will no longer exist. If a variable is unset because its interpreter is deleted then no trace commands will be invoked, since there is no context in which to execute them. Traces are not disabled during unset traces, so if an unset trace command creates a new trace and accesses the varable then the trace will be invoked.

If multiple traces are set for the same variable, then each trace is invoked on each variable access. The most recently created trace is invoked first. If an array element has a trace set and there is also a trace set for the whole array, then array traces are invoked before element traces. If one trace returns an error then no additional traces are invoked for that access.

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 any traces on that 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 original trace created on `x` above can be deleted with the following command:

```
trace vdelete x rwu xtrace
```

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

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

```
trace vinfo x
{rwu xtrace}
```

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 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` as its arguments. For example, suppose that you type the following commands:

```
set x 24
createDatabase library $x
```

If there is no command named `createDatabase` but there is a command named `unknown`, 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 cmd [info commands $name*]
    if {[llength $cmds] != 1} {
        error "unknown command \"$name\""
    }
    uplevel [list $cmd] $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 expands abbreviations and performs many other functions, such as *auto-loading* script files when procedures defined in them are first invoked, automatically executing subprocesses, and performing simple history substitutions (see Chapter 12 for details). You're free to write your own `unknown` procedure or modify the library version to provide additional functions.