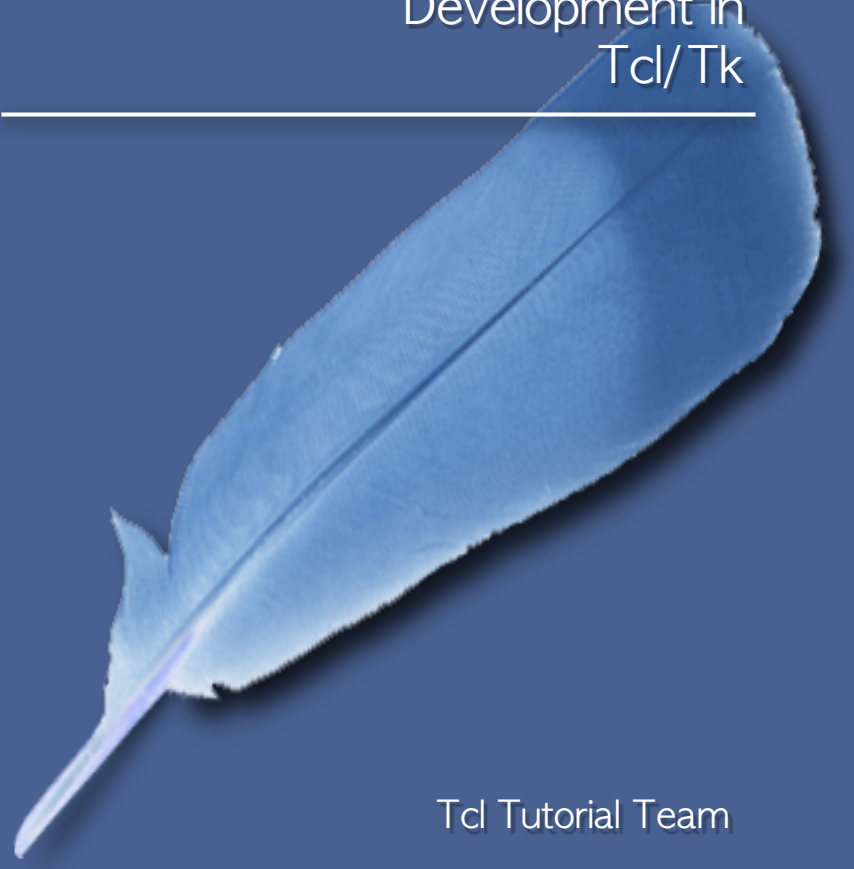
Modern Application Development in Tcl/Tk



Tcl Tutorial Team

Modern Application Development in Tcl/Tk

The Tutorial Team

March 2009

Contents

Preface

Introduction

Welcome to *Modern Application Development in Tcl/Tk*! This book aims to provide a complete introduction to developing applications in Tcl/Tk, and to emphasise modern software development methodology throughout. The book is intended as a free companion and eventual replacement for the existing Tcl Tutorial [5], written by Clif Flynt, Arjen Markus, David N. Welton and Neil Madden. Some material from that tutorial has been reused in this book, but the majority has been heavily edited or rewritten from scratch to reflect current best practices.

Tcl stands for the 'Tool Command Language'. It was originally developed in the 1980's by Dr. John Ousterhout, then a Professor of Computer Science at the University of California at Berkeley, as a lightweight embeddable *command language* for a number of software tools developed by Dr. Ousterhout and his students. The idea was that Tcl would provide some basic facilities common to all programming languages, such as variables, loops, and control structures, and then a host application could embed the language and extend it with new commands relevant to that tool. Users could then use Tcl to control the tool by issuing commands, either interactively or in batch scripts. You can read more about the history of Tcl at `http://www.tcl.tk/about/history.html`.

Tcl still excels as an embedded command language, but it has also developed into a fully fledged general programming language. The core principles that made it such a good command language: simplicity, extensibility, and ease of integrating different components, have also stood Tcl in good stead as a general purpose language. Today it is used for everything from modern desktop applications, to heavy-duty network servers, and lots in between.

Outline of the Book

The first part of the book introduces the Tcl language, from basic facilities through to advanced topics such as developing custom flow control constructs and event-based TCP/IP networking. Don't Panic! No previous knowledge of Tcl/Tk or even programming in general is assumed in this book. Feel free to skip the more advanced chapters to begin with, and then come back to them when you feel more comfortable with the material. The remaining parts of the book introduce a number of popular extensions to Tcl for developing sophisticated applications:

- Part II introduces *Tk*, the Graphical User Interface (GUI) ToolKit, and its newer "themed" companion, *Ttk*. Together, these extensions allow you to create polished desktop applications that integrate well with all major desktop environments: Microsoft Windows, Mac OS X, and Linux/X11 (KDE and Gnome).

- Part III then introduces a number of useful and popular extensions for handling real-world requirements: interfacing to database management systems (Chapter 13), processing XML (Chapter 14), interfacing with Web Services (Chapter 15), and writing powerful Web applications.

This book doesn't currently cover the implementation of Tcl or how to extend or embed Tcl from C or C++. These advanced topics are covered by the Tcl manual pages and a number of other books on Tcl.

Additional Resources

The Tcl community is an exceedingly friendly one. Here are some good places to get help:

- The Tcl Developer eXchange (`http://www.tcl.tk/`) is the main website for Tcl, and contains links to downloads, documentation, mailing lists, and the open source Tcl development project where you can also download the source-code of the Tcl interpreter.
- The `comp.lang.tcl` newsgroup. Accessible via a newsreader, or Google Groups.
- The Wiki (`http://wiki.tcl.tk/`) has a great deal of useful code, examples and discussions of the finer points of Tcl usage.
- If you need help right away, there is often someone on the `#tcl` channel on `irc.freenode.net` who can help you out, but please don't be impatient if no one can help you instantly—if you need that level of support, consider hiring a consultant.
- There are several other good books on Tcl/Tk if you wish to gain more in-depth knowledge of Tcl. Recommended titles include 'Tcl/Tk: A Developer's Guide'[4] and 'Practical Programming in Tcl/Tk'[12].

Contributors

The following people have contributed material to this book:

- Clif Flynt
- Neil Madden
- Arjen Markus
- David N. Welton

Part I

Tcl: The Tool Command Language

Chapter 1

Getting Started

1.1 Running Tcl

Tcl is an interpreted language. The main interpreter program is named `tclsh` or, depending on the installation, `tclsh8.6` or `tclsh86.exe` on Windows.[1] The `tclsh` program can be started in one of two modes: interactively or in batch mode. In batch mode, `tclsh` accepts a file name as an argument and will then load the file and execute each command in it before exiting. In interactive mode, `tclsh` presents the user with a prompt, much like a shell, from which you can type in commands to be evaluated one at a time. After evaluating each command, `tclsh` will print the result of the command, or any error message that was generated. To start `tclsh` in interactive mode, simply start it without any arguments.

As we mentioned in the introduction, Tcl was designed to be embedded into other applications. It should therefore come as no surprise that `tclsh` is not the only way of accessing a Tcl interpreter. Tcl is embedded into a number of different applications and devices, from CAD (Computer Aided Design) packages to network routers. However, there are also a number of other standard interpreter programs which might also be installed on your system. One example is the WIndowing SHell, `wish`. This is a version of `tclsh` that automatically loads the Tk graphical user interface extension. Other options are also available, providing more functional environments for developing and debugging Tcl code. One very popular choice is the TkCon (`http://tkcon.sf.net/`) enhanced interactive interpreter, written by Jeff Hobbs. The Eclipse IDE also offers good Tcl support, via the DLTK project (which also supports Python, Ruby and other 'dynamic languages'). For writing longer applications, a good programmer's text editor is worth acquiring. A number of IDEs and editors with support for Tcl are listed on the Tcl Wiki (see the Additional Resources section in the Preface).

1.2 Hello World!

The traditional starting place for a tutorial is the classic "Hello, World" program. Once you can print out a string, you're well on your way to using Tcl for fun and profit!

The command to output a string in Tcl is the **`puts`** command. A single text argument after the **`puts`** command will be printed to the standard output device (`stdout`; usually the console). The default behaviour is to print a newline character ("return") appropriate for the system after printing the text. Here's an example of using the **`puts`** command to display some text. You can run this either by start `tclsh` interactively and typing in the command at the `%` prompt, or by saving the command into a file called `hello.tcl` and then running `tclsh hello.tcl`.

```
puts Hello!
```

[1] Throughout this book we will assume that you have the 8.6 version of Tcl installed, which is the latest version as of this writing. Differences to previous versions will be pointed out in the text.

If the string has more than one word, you must enclose the string in double quotes (`""`) or braces (`{}`). A set of words enclosed in quotes or braces is treated as a single unit, while words separated by whitespace are treated as multiple arguments to the command. Quotes and braces can both be used to group several words into a single unit. However, they actually behave differently. In the next section, you'll start to learn some of the differences between their behaviours. Note that in Tcl, single quotes are not significant, as they are in some other programming languages, such as C, Java, Perl and Python.

Many commands in Tcl (including **`puts`**) can accept multiple arguments. If a string is not enclosed in quotes or braces, the Tcl interpreter will consider each word in the string as a separate argument, and pass each individually to the command. The **`puts`** command will try to evaluate the words as optional arguments, likely resulting in an error. Here is an example interactive session with `tclsh`, showing the correct ways of calling **`puts`** and an error that results from an incorrect call.

```
% puts "Hello, World!"
Hello, World!
% puts {Hello, World!}
Hello, World!
% puts Hello, World!
can not find channel named "Hello,"
```

A *command* in Tcl is a list of words terminated by a newline or semicolon. You can write a *comment* (i.e., a piece of text that documents the program) by starting a command with a `#` character. The comment continues until the end of the line, ignoring any semicolons or other punctuation. Tcl will skip any text in a comment and move on to the next command. You can use comments to describe the intention of your code to other programmers, or temporarily disable a command that you do not want to be evaluated. *Note:* In Tcl, comments can *only* appear where Tcl would expect a command to begin. You cannot, for instance, introduce a comment in the middle of a command, or in the middle of a quoted string. If you wish to add a comment to the end of an existing line, you can use a semicolon to first terminate the command.

```
# This is a comment on its own line
puts "Hello, World" ;# This is a comment after a command
puts "Bad" # This will cause an error - no semicolon!
puts "Line 1"; puts "Line 2"; # Prints two lines
puts "Semi-colons inside quotes are ignored: ; See?"
puts {... and within braces: ; See?}
```

1.3 Variables and Values

A *value*[2] is a piece of data that can be passed as the argument of a command, or returned as its result. In Tcl, every value (or term) is a string, i.e., a piece of text. This is in contrast to many other languages, in which values are sharply divided into different 'types', e.g., integers, strings, lists, and so on. In Tcl, it is up to individual commands to determine how to interpret the arguments that they are passed. For instance, one command might treat the string `{a b c}` as a list, while another treats it as a set (i.e., a collection without duplicates), while yet another might treat it simply as a string. Internally, Tcl remembers how commands have interpreted different values, so for instance if the string `3.14159` is always interpreted as a floating-point number, then Tcl will remember this interpretation and store an efficient representation of the number, identical to the representation used in most other languages. This allows Tcl to still be quite fast, while maintaining the flexibility that makes it such a convenient 'glue language'. This property is often referred to as 'everything-is-a-string', or EIAS, within the Tcl community.

[2] More correctly, a *term*.

While all Tcl values are strings, there are a number of other elements to the language which are not values, and are not strings. We have already encountered one such entity: commands. Another very important type of object are *variables*. A variable is a named location in memory that can be used to store values. All non-value resources in Tcl are accessed via names: commands, variables, channels, and so on. While the name is a string, the underlying resource that it names is usually a more complex entity. You can assign a value to a variable using the **set** command:

```
set fruit "Cauliflower"
```

When **set** is called with two arguments, as above, it places the second argument (*Cauliflower*) into the memory space referenced by the first argument (the variable; *fruit*), overwriting any value that was already there. If the variable does not already exist then a new one is created with the given name. The **set** command always returns the new value of the variable, in this case the string *Cauliflower*.

The **set** command can also be called with just a single argument. In this case it simply returns the current value of the variable:

```
% set fruit
Cauliflower
```

Because fetching the value of a variable is such a common task, Tcl has built-in syntax just for this operation: the $ symbol can be used in front of a variable name to retrieve the current value. This can even be used within a quoted string:

```
puts $fruit ;# Just prints "Cauliflower"
puts "The fruit is: $fruit"
```

You cannot however use this within braces. The following code will print the literal string "The fruit is: $fruit".

```
puts {The fruit is: $fruit}
```

This is the main different between quotes and braces: *braces prevent substitution*. By default, Tcl's $-substitution rules only allow variable names to consist of letters, digits, or underscores.[3] If you want to have more complicated variable names, perhaps including spaces or other punctuation characters, you can use braces to tell Tcl where the end of the variable is:

```
set "My long variable" "Some value"
puts ${My long variable}
```

A variable can be destroyed using the **unset** command, which takes a variable name as an argument and removes that variable, as if it had never been **set**:

```
% set x 1
1
% unset x
% set x
can't read "x": no such variable
```

You can also find out which variables have been created by using the **info vars** command[4]. This returns a list of all the variables that are currently defined in this scope. You can also pass a pattern to the command in order to restrict which variables are returned. Here we can see a number of special variables that Tcl defines automatically on startup:

```
% info vars tcl_*
tcl_rcFileName tcl_interactive tcl_version tcl_pkgPath tcl_patchLevel
tcl_library tcl_platform
```

[3] Two other forms are also allowed, which will be discussed in later chapters on namespaces and arrays.

[4] The **info** command contains lots of useful sub-commands for inspecting the state of the interpreter. We will discuss others as we encounter them.

1.4 Evaluation and Substitution

In Tcl, the evaluation of a command is done in 2 phases. The first phase is a single pass of subsitutions. The second phase is the evaluation of the resulting command. Note that only *one* pass of substitutions is made. Thus, in the following command, the contents of the proper variable are substituted for `$varName`, and then the command is executed.

```
puts $varName
```

Assuming we have set `varName` to `"Hello, World!"`, the sequence would look like this:

1. First `$varName` is substituted: `puts $varName` ⇒ `puts "Hello, World!"`.
2. Then, the command is evaluated, resulting in `"Hello, World!"` being output to the console.

During the subsitution phase, several types of subsitutions occur:

- A command within square brackets (`[]`) is replaced with the result of evaluating that command.
- Words within double quotes or braces are grouped into a single argument.
- Variable references (with a preceeding `$`) are replaced with the value of that variable.
- 'Backslash substitution' is performed (see Section 1.4.1).

Double-quotes (`""`) and braces (`{}`) both serve to group multiple words together into a single word in the command. The difference between them is that braces prevent subtitution being performed on their contents, whereas double quotes do not. Compare:

```
set a 10
puts "a = [set a] or $a" ;# prints "a = 10 or 10"
puts {a = [set a] or $a} ;# prints "a = [set a] or $a"
```

Command substitutions can be nested, for example, given fictitious 'readsensor' and 'selectsensor' commands, we can write:

```
puts [readsensor [selectsensor]]
```

This will first evaluate the `selectsensor` command and then pass the result to the `readsensor` command as a single argument. The result of that command will then become the argument to **`puts`**. Note that only a single round of substitution is ever done, despite multiple commands being evaluated. This means that, for instance, if the `selectsensor` command returns a string such as '`$someText`' this will be treated as a literal string and *not* substituted as if it was a variable reference. You can perform an extra round of substitution using the **`subst`** command if needed, but that is a rare requirement.

The result of a command or variable substitution is usually treated as a single word in the resulting command. For instance, in the example above, the string `"Hello, World!"` became a single argument to the **`puts`** command, despite having a space in it, because it was the result of a variable substitution. The only exception to this rule is the new `{*}` operator that was introduced in Tcl 8.5. This operator converts a list of words (such as the result of a command or variable substitution) into a set of individual arguments. Compare:

```
set x "a b c d"
puts $x    ;# => puts "a b c d" -> displays "a b c d"
puts {*}$x ;# => puts a b c d -> error
```

String	Output	ASCII Hex/Notes
\a	Audible Bell	0x07
\b	Backspace	0x08
\f	Form Feed (clear screen)	0x0c
\n	Newline	0x0a
\r	Carriage-return	0x0d
\t	Tab	0x09
\v	Vertical tab	0x0b
\<newline>whitespace	A single space.	Line continuation.
\\	Backslash	A literal backslash
\0oo	A Unicode character (octal escape)	*obsolete*
\xhh	A Unicode character (hexadecimal)	*obsolete*
\uhhhh	A Unicode character	The four hexadecimal digits ('h') give the sixteen-bit value for the Unicode character that will be inserted.

Table 1.1: Backslash escape sequences.

The {*} operator is an advanced feature that you generally won't have to use much when you are learning Tcl. The full behaviour of substitution and evaluation in Tcl is described in the main Tcl.n manual page [11], which you should read completely before starting any serious programming with Tcl. The 12 rules given in that manual page completely describe the operation of the Tcl interpreter, and you should try to read and understand them before moving on. The rules can be quite subtle, however, especially if you are used to other programming languages, so don't be surprised if it takes a little while to fully appreciate the details!

1.4.1 Backslash Substitution

In general, the backslash (\) disables substitution for the single character immediately following the backslash. For example, the string '\"' will be replaced with the literal character '"' (a double quote), and will not be regarded as starting a double-quote string for grouping. However, there are also a number of specific 'Backslash Sequence' strings which are replaced by special values during the substitution phase, shown in Table 1.1. Backslash-newline ('line continuation') substitutions are the only form that is performed within braces.

Here are some examples you can try in a tclsh interpreter session. Try to guess what the result of each will be using the rules before testing it.

```
set Z       Albany
set Z_LABEL "The Capitol of New York is: "

puts "$Z_LABEL $Z"
puts "$Z_LABEL \$Z"

puts "\nBen Franklin is on the \$100.00 bill"

set a 100.00
puts "Washington is not on the $a bill"
puts "Lincoln is not on the $$a bill"
puts "Hamilton is not on the \$a bill"
puts "Ben Franklin is on the \$$a bill"

puts "\n..................... examples of escape strings"
```

```
puts "Tab\tTab\tTab"
puts "This string prints out \non two lines"
puts "This string comes out\
on a single line"
```

1.5 Arithmetic

The Tcl command for evaluating mathematical expressions is **expr**. Many commands use **expr** behind the scenes in order to evaluate condition test expressions, such as **if**, **while**, and **for** loops. All of the advice given here for **expr** also holds for these other commands. The following sections are largely derived from the standard Tcl manual page for the **expr** command.

The **expr** command takes all of its arguments ('2 + 2' for example) and evaluates the result as a Tcl 'expression' (rather than a normal command), returning the value. The syntax of Tcl expressions resembles that of conventional mathematical notation, including various standard infix arithmetic and logical operators, bitwise operators, as well as functions like `rand()`, `sqrt()`, `cosh()` and so on. Expressions almost always yield numeric results (integer or floating-point values).

Tip: You should *always* enclose the arguments to **expr** in braces. While this is not strictly necessary, it results in faster code, and also is much safer when handling user-supplied data. The reason for this is that **expr** performs a round of subsititution in addition the the normal substitution phase performed by Tcl. The braces prevent the Tcl round from being performed, which is both faster and avoids the problems of *double substitution*. To illustrate the problems that can occur, consider this interactive session:

```
% set userinput {[puts DANGER!]} ;# example user input
[puts DANGER!]
% expr $userinput == 1
DANGER!
0
% expr {$userinput == 1}
0
```

In the first example, Tcl first substitutes the variable for its value. The **expr** command then evaluates the code that it contains, resulting in the 'DANGER!' message being displayed. Clearly, this could present a security problem if users can enter arbitrary code. In the second example, the braces prevent Tcl substituting the variable and so the code works correctly. As a general rule, always surround expressions with braces. The resulting code will be safe and will also run faster!

1.5.1 Operands

A Tcl expression consists of a combination of operands, operators, and parentheses. White space may be used between each of these; it is ignored by the expression processor. Where possible, operands are interpreted as integer values, otherwise it is treated as a floating-point number. Integer values may be specified in decimal (the normal case), in octal (using the prefix `0o`), in hexadecimal (using the prefix `0x`), or in binary (prefix `0b`). For compatibility with earlier versions of Tcl, you can also use just a leading zero to indicate octal, but this use is discouraged in new code. Some examples follow:

```
% expr {1234}
```

```
1234
% expr {0o1234} ;# 1234 octal = 668 decimal
668
% expr {01234} ;# 1234 octal (old-style notation)
668
% expr {0x123f} ;# 123f hexadecimal = 4671 decimal
4671
% expr {0b11110000}
240
```

Tip: The old octal format is a frequent cause of mistakes when dealing with user input. Whenever a user enters a decimal integer you must ensure to remove any leading zeros that they may have entered in order to avoid the octal interpretation. For instance, consider the following (pseudo)code:

```
set num [get-user-input]
puts "$num + 1 = [expr {$num + 1}]"
```

If the user enters a leading zero then unexpected results can occur, e.g., if the user enters `012` they will receive the unexpected result that '012 + 1 = 11'! Even worse, if the user enters an invalid octal number, such as `09`, the program will halt with an error. The best way to avoid such problems is to ensure that all user input is correctly converted to an appropriate format before being used. One particularly useful command here is the **scan** command that can perform lots of conversions. The `%d` conversion will ensure that our number is a decimal integer:

```
set num [scan [get-user-input] %d]
puts "$num + 1 = [expr {$num + 1}]"
```

If an operand does not have one of the integer formats given above, then it is treated as a floating-point number (i.e., an approximate real number), if that is possible. Floating point numbers may be specified in any of several common formats making use of decimal digits, the decimal point ('.'), the characters 'e' or 'E' indicating scientific notation, and the sign characters '+' or '−'. For example, all of the following are valid floating point numbers:

```
% expr {2.1}
2.1
% expr {3.}
3.0
% expr {6e4}
60000.0
% expr {7.91e+16}
79100000000000000.0
```

Note that the comma ',' is not a valid decimal point indicator in Tcl. The special values '`Inf`' and '`NaN`' (exact spelling) can also be used to represent floating point infinity and not-a-number, respectively.

If no numeric interpretation is possible, then an operand is left as a string, and only a limited set of operators may be applied to it. Some operations expect a Boolean truth value. Table 1.2 shows the valid Boolean values in Tcl.

Big Integers

Since Tcl version 8.5

	False	True
Numeric	0	all others
Yes/No	`no`	`yes`
On/Off	`off`	`on`
True/False	`false`	`true`

Table 1.2: Valid Boolean truth values.

Tcl can handle arbitrarily large integer values exactly. This is handled largely transparently for the Tcl programmer, with the internal representation of the number automatically upgraded to a format that is large enough to represent the value. For example, we can calculate 2^{1000} as follows:

```
% expr {2**1000}
10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251871452856923140435984577574698574803934567774824230985421074605062371141877954182153046474983581941267398767559165543946077062914571196477686542167660429831652624386837205668069376
```

1.5.2 Operators

Logical and Relational Operators

The **`expr`** command understands the following logical (Boolean) operators (in descending order of precedence):

`!` Logical negation (NOT). Converts any 'true' value (see Table 1.2) to 0, and any 'false' value to 1.

`<,<=,>,>=` Relational less than, less than or equal, greater than, and greater than or equal operators, respectively. Each operator produces 1 if the condition is true, or 0 otherwise. These operators can be applied to numeric operands or strings, in which case string comparison is used.

`==,!=` Equals and not-equal operators, respectively.

`&&` Logical conjunction (AND). Returns 1 if both operands are 'true' values, otherwise 0.

`||` Logical disjunction (OR). Returns 1 if either of the operands are 'true', otherwise 0.

`$x?$y:$z` The if-then-else ternary operator. If `$x` evaluates to a 'true' value then the result is the value of `$y`, otherwise it is the value of `$z`.

Arithmetic Operators

The following operators are used for standard arithmetic operations on numbers:

`–,+` Unary minus and plus operators, used to set the sign of the operand.

`**` Exponentiation operator: `[expr {$x**$n}]` means x^n.

`*,/,%` Multiply, divide, and remainder respectively. The remainder operations can only be applied to integers, and the result will always have the same sign as the divisor and an absolute value smaller than the divisor. If the divide operator is applied to two integer operands then integer division is performed, otherwise floating-point division is used.

`+,–` Binary addition and substraction, respectively. Valid for any numeric operands.

Bitwise Operators

The following operators are used to perform operations on the binary (bitwise) representation of integers:

~ Bit-wise NOT. Negates each bit in its operand (including the sign bit). For example[5]:

```
% format %#hb [expr {~0b0101010101010101}]
0b1010101010101010
```

<<,>> Left and right bit shift, respectively.

& Bitwise AND.

^ Bitwise exclusive OR.

| Bitwise OR.

String and List Operators

The following operators can be used to compare strings and to determine if a value is or is not an element of a list. Although listed separately, all of these operators share the same precedence.

eq,ne String equality/inequality operators. Causes an equality test to be done on the string representation of each value, even if they have a numeric interpretation.

in,ni Set operators: contains and doesn't contain operators, respectively. `[expr {$x in $xs}]` will return 1 if the value in `$x` is a member of the list `$xs`.

1.5.3 Math Functions

Tcl supports the following mathematical functions in expressions, which you can access using traditional mathematical notation, e.g., `[expr {sin($x)}]`.

abs	acos	asin	atan
atan2	bool	ceil	cos
cosh	double	entier	exp
floor	fmod	hypot	int
isqrt	log	log10	max
min	pow	rand	round
sin	sinh	sqrt	srand
tan	tanh	wide	

You can also access normal Tcl commands within **expr** using the usual square-bracket syntax, e.g., `[expr {[set a]+1}]`. Be aware, however, that any further sub-expressions contained within such command calls must include a further call to **expr**, as in:

```
[expr {[mycmd [expr {$x*2}]]+1}]
```

As of Tcl 8.5, you can now expose Tcl commands as functions within **expr** by creating a command in a `tcl::mathfunc` namespace[6], which would allow you to write the expression as `[expr {mycmd($x*2)+1}]`, which is much clearer.

[5] The **format** command here converts the decimal result into a 16-bit binary representation.
[6] Namespaces are discussed in Section 4.1.

1.5.4 Incrementing Integers

One extremely common arithmetic operation is to simply increment a variable containing an integer. You could accomplish this simply using **`expr`**, as:

```
set x [expr {$x + 1}]
```

However, the operation is so common that Tcl builds-in a convenient short-hand, which is also slightly more efficient: the **`incr`** command. The syntax of **`incr`** is as follows:

incr *varName ?amount?*

This increments the value stored in `varName` and stores the new value back into the variable. The command also returns the new value. The optional `amount` argument can be used to specify how much to increment the value, and can be any integer, including negative to indicate decrement.

Chapter 2

Handling Data

Like most programming languages, Tcl comes with facilities for handling many different sorts of data. We've already seen Tcl's facilities for handling numbers and basic arithmetic. In this chapter we will examine all the other sorts of data that can be used in a Tcl program: strings, lists, dictionaries, and associative arrays, and how they can be put to work. We'll also look at how you can use these basic data 'types' to build up more sophisticated data structures suited to your application.

Unlike most programming languages, Tcl does not have a built-in notion of 'type'. More precisely, all values in Tcl are of a single type: strings. In practice, most programming with data in Tcl is much like programming in other languages: you decide on how some data is to be interpreted (e.g., as a number, or a list) and then use commands that are appropriate for that sort of data. Tcl will take care of making sure that efficient representations are used under-the-hood (e.g., integers are represented as native machine integers).

Tcl's approach to data structures is to provide a few general purpose structures and to then optimise the implementation for typical uses. This is in contrast to lower-level languages, such as C++ or Java, that provide lots of different data structures for different purposes. Tcl instead aims to simplify the task of programming by picking a few powerful and general data structures. In this regard, Tcl is similar to high-level applications like spreadsheets or relational database systems, that allow the user to specify what they want to achieve and leave it up to the implementation to ensure that most operations are efficient. Of course, you can always design your own custom data structures in Tcl if you need to, but the defaults provided are sufficient for most tasks. In addition, Tcl takes care of memory management and index checking for you, and so is safe from whole classes of errors due to memory mismanagement or 'buffer overflows'.[1]

2.1 Strings

Tcl commands often have subcommands. We've already seen an example in the **`info`** command. The **`string`** command is another example, that contains subcommands for manipulating strings. While all values in Tcl are strings, the **`string`** command should be used only on data that you really want to be treated as a string. Examples might include names of people, or addresses. A command with subcommands is known as an *"ensemble"* command. Some useful **`string`** commands include:

`string length`: Returns the length of a string, e.g., `string length "Hello, World!"` returns 13.

`string index`: Returns the nth character from a string, counting from 0. For example:

[1] For the technically minded, Tcl uses a form of reference-counting for memory management. This suffices for the majority of cases, and simplifies the C interface.

```
% string index "Hello, World!" 0
H
% string index "Hello, World!" 5
,
```

You can also index strings from the end using the following syntax:

```
% string index "Hello, World!" end
!
% string index "Hello, World!" end-2
l
```

`string range:` Returns the range of characters between two indices:

```
% string range "Hello, World!" 2 end-2
llo, Worl
```

`string equal:` Compares two strings for equality. You can specify whether the comparison should be case sensitive (the default) or not, and also specify a length limit for the comparison (which defaults to the full length of the strings):

```
% string equal -nocase -length 5 "Hello, World!" "hElLo Tcl!"
1
```

`string compare:` Compares two strings for alphabetical ordering. Each character is compared one at a time, with with 'a' being less than 'b' and 'A' being less than 'a'. Note that numbers are also compared in this fashion, so that the string '10' will be considered less than '2' as the first character (1) is less than 2. As for **`string equal`**, you can specify `-nocase` and `-length` options. The command returns -1 if the first string is 'less' than the second, 0 if they are equal, or 1 if the second string is less than the first:

```
% string compare "Hello" "World"
-1
% string compare -nocase apple Apple
0
```

`string first:` Searches for a string within a larger string. If found, it returns the index of the first occurrence, otherwise it returns -1:

```
% string first "ll" "Hello, World!"
2
```

You can also specify an optional start index for the search:

```
% string first "ll" "Hello, World!" 5
-1
```

`string last:` Searches for the last occurrence of a search string:

```
% string last "ll" "Hello, World!"
2
% string last l "Hello, World!"
10
```

string wordend: Returns the index of the character just after the last one in the word that contains the given index. A word is any contiguous set of letters, numbers or underscore characters, or a single other character.

string wordstart: Returns the index of the first character in the word that contains the index given. You can use these two commands along to extract the word that a given index lies within:

```
% set str "Hello, World!"
Hello, World!
% set idx 2
2
% string range $str [string wordstart $str $idx]\
    [string wordend $str $idx]-1
Hello
```

string reverse: Reverses a string:

```
% string reverse "Hello, World!"
!dlroW ,olleH
```

string repeat: Generates a string consisting of n repetitions of the given string:

```
% string repeat "Hello! " 5
Hello! Hello! Hello! Hello! Hello!
```

string replace: Replace a substring within a larger string. The string to use as a replacement can be omitted, in which case the original string will be deleted. Note that this command, like most Tcl commands, does not alter the original string, but instead returns a new string with the replacement made:

```
% string replace "Hello, World!" 0 4 Goodbye
Goodbye, World!
```

string map: This very useful command performs a bulk search and replace operation on a string. The first argument specifies a list of mappings: a string to find and then a string to replace it with. The command then searches through the given string replacing each search string with its mapping and returns the resulting string:

```
% string map {0 "zero " 1 "one " 2 "two "} 01201201202
zero one two zero one two zero one two zero two
```

string tolower, string toupper, string totitle: Converts a string to all lower-case, all upper-case, or title-case, respectively. Title-case means that the first letter is capitalized, while all others are made lower-case.

`string trim`, `string trimleft`, `string trimright`: Trims any occurrences of a given set of characters from the start or end of a string. The default is to trim all whitespace characters:

```
% string trim "\t  Hello, World!   "
Hello, World!
% string trimleft "Hello, World!" "lHe!"
o, World!
```

Note that the second argument is considered as a set of characters rather than a string.

`string is`: Performs input validation on an input string to determine if it can be interpreted as the given class of value. This command is meant for validating user input, rather than for general type checking. In particular, the command will accept the empty string (`""`) as valid unless the `-strict` option is given. This command is mostly useful in Tk GUIs (Part II).

A number of other commands are useful for manipulating strings, but are not in the **`string`** ensemble:

`append`: Appends one or more strings to a *variable* that contains a string. Note that this command alters a variable rather than just returning a new string:

```
% set str "Hello,"
Hello,
% append str " World!"
Hello, World!
% set str
Hello, World!
```

Note that we do not use a dollar-sign when passing the variable `str` to this command. This is because the command expects the name of a variable containing a string, rather than the string itself. The general rule is that if a command returns a new value then you should use a dollar, but if it manipulates a variable then you should pass the variable name without a dollar sign.

`concat`: Concatenates zero or more strings together, inserting spaces between each one. Returns the resulting string:

```
% concat "Hello," "World!"
Hello, World!
```

`scan` and `format`: These commands can be used to perform conversions between strings and various different data types, while also specifying details of the format, such as maximum length and padding characters. The full details of these commands are too complex for this tutorial, so we will just show some examples of each. The full details are available in manual pages [9, 6]. C programmers will be familiar with these as the `scanf` and `sprintf` functions.

```
# Parse a simple colour specification of the form #RRGGBB in hex format
% scan #08D03F "#%2x%2x%2x" r g b
3
% puts "$r $g $b"
8 208 63
```

```
# Format the string back again
% format "#%02X%02X%02X" $r $g $b
#08D03F
```

These two commands are very useful for input validation and output formatting, and we will demonstrate their usage throughout the text. In particular, you should almost always use **scan** and **format** when processing numeric input and output.

2.1.1 Unicode

Tcl was one of the first programming languages to adopt Unicode-aware strings throughout the language. To this day, Tcl still has one of the most mature and well-integrated Unicode string implementations, roughly on a par with that of Java. Tcl strings are represented internally as a slightly modified version of UTF-8. Data entering or exiting the Tcl system is translated to and from UTF-8 largely transparently. (The details of how to control this conversion are discussed in Chapter 5). What this means for the Tcl programmer is that they can concentrate on the logic of their application, safe in the knowledge that they can handle input in most languages and character sets in use in the world. All standard Tcl commands that deal with strings (i.e., pretty much all of them!) can handle Tcl's UTF-8 format, as can all commands that deal with input and output. Most Tcl extensions in common use are also usually Unicode-aware, so, for instance, Tcl XML parsers do actually handle the Unicode requirements of XML, unlike some languages we might care to mention!

There are some current limitations to Tcl's otherwise excellent Unicode support. Firstly, Tcl currently only supports Unicode code-points up to `U+FFFF` (the 'Basic Multilingual Plane', or BMP). Support for Unicode characters beyond this range is a work-in-progress. The remaining limitations are largely within the Tk graphical user interface extension: for instance, support for rendering bidirectional ('bidi') scripts is currently missing. Overall, though, Tcl's support for Unicode and different character encodings is first class.

To enter arbitrary Unicode characters into a Tcl script, you can use the Unicode escape syntax described in Section 1.4.1:

```
set str "n\u00E4mlich"
```

This will result in `$str` containing the string `nämlich`. To manually convert between different encodings you can use the **encoding** command. *Note:* you should normally never have to use the **encoding** command directly, as all conversions are usually done by the I/O system automatically.

2.1.2 Pattern Matching

One particularly useful string operation is the **string match** command that takes a pattern, known as a 'glob', and determines if a given string matches the pattern. 'Globbing' is the wild-card matching technique that most Unix shells use. In addition to **string match**, globbing is also used by the **switch**, **lsearch** and **glob** commands, discussed later. The wildcards that **string match** accepts are:

***** Matches any quantity (including zero) of any characters.

? Matches one occurrence of any character.

[...] Matches one occurrence of any character between the brackets. A range of characters can also be specified. For example, `[a-z]` will match any lower-case letter.

**X* The backslash escapes a glob-special character, such as `*` or `?`, just as in Tcl. This allows you to write glob patterns that match literal glob-sensitive characters, which would otherwise be treated specially.

Some examples are as follows:

```
% string match f* foo
1
% string match f?? foo
1
% string match f foo
0
% string match {[a-z]*} foo
1
% string match {[a-z]*} Foo
0
```

The **string match** command is a great way of determining if some input string matches a particular format. If you have more complex pattern matching requirements, or need to simultaneously extract information as well as match it, then *regular expressions* (Section 2.6) provide a more powerful (but harder to use) facility.

2.2 Dates and Times

Tcl has sophisticated facilities for dealing with dates and times in the form of the **clock** command, another ensemble. Tcl represents dates and times as the number of seconds since the *epoch* time of 1st January, 1970, 00:00 UTC. Note that Tcl's clock times do not include leap seconds: each UTC day is considered to have exactly 86400 seconds. Instead, Tcl minutely adjusts its clock speed to account for leap seconds. The various clock-related commands are as follows:

clock seconds Returns the current time as an integer number of seconds since the epoch.

clock milliseconds Returns the current time as an integer number of milliseconds since the epoch.

clock microseconds Returns the current time as an integer number of microseconds (millionths of a second) since the epoch.

clock clicks Returns a system-dependent high-resolution timer value.

clock format Formats a time value into a human-readable form.

clock scan Parses a date/time value in some format into a Tcl time value.

clock add Adds two time values together, accepting a variety of ways of specifying the dates and times.

The `format` and `scan` sub-commands accept a `-format` option that specifies the expected format of the output/input. If not specified, **clock format** defaults to a format of '%a %b %d %H:%M:%S %z %Y', while **clock scan** uses 'free form scan' mode, in which it attempts to guess the format of the given input string. The possible format specifiers are too numerous to list in this tutorial. Instead, we refer the reader to the official manual page [2]. The `-locale` and `-timezone` options can be used to specify which locale and timezone to process time values. The `-base` option to **clock scan** specifies a base time to consider all values as relative to. In older versions of Tcl you can use the `-gmt 1` option to specify that all processing should be done in UTC. Since Tcl 8.5, this usage is deprecated in favour of `-timezone :UTC`. Some examples of the use of the clock command:

```
% clock format [clock seconds] -format "%Y-%m-%d %T"
2009-03-22 20:50:42
% clock format [clock scan "now + 1 year"]
Mon Mar 22 00:00:00 GMT 2010
```

The **clock clicks** command returns a system-dependent high-resolution timer. This timer is not guaranteed to be relative to any particular epoch, unlike the other clock commands, and so should only be used for relative timing when the highest-resolution timer that a system supports is needed (such as for benchmarks).

If you want to time how long a particular piece of Tcl code is taking to run (for *profiling*), then you can use the **time** command that is designed for just this purpose:

time *script ?iterations?*

This command runs the given script for the given number of iterations (defaults to 1) and then returns a human-readable string of the form: `N microseconds per iteration`.

2.3 Lists

The *list* is a fundamental data structure in Tcl. A list is simply an ordered collection of elements: numbers, words, strings, or other lists. As Tcl is untyped, lists can contain elements of multiple different kinds. Even commands in Tcl are just lists, in which the first element is the name of the command to call, and subsequent elements are the arguments to that command. Tcl lists are implemented in a similar manner to arrays or vectors in languages such as C. That is, Tcl list elements are held in contiguous ranges of memory. This makes indexing and iterating over elements very efficient (O(1) indexing in 'big-Oh notation' [13]). Other operations, such as appending and deleting elements are also efficient for most usage patterns due to the way Tcl handles list memory.

Lists can be created in several ways:

As a literal string: As with all values, Tcl lists are just strings in a certain format. If your list is simple you can just write it as a literal string:

```
set myList {"Item 1" "Item 2" "Item 3"}
```

Note that in this tutorial we often use braces to delimit lists and quotes to delimit other strings. This is just a convention: Tcl will accept either as either a string or a list, so long as it is in the right format.

Using the `list` command: The **list** command takes an arbitrary number of arguments and returns a list whose elements are those arguments:

```
set myList [list "Item 1" "Item 2" "Item 3"]
```

Using the `split` command: The **split** command converts an arbitrary string into a valid list by splitting the string on one of a given set of *delimiter* characters (which defaults to any whitespace):

```
set myList [split "Item 1,Item 2,Item 3" ","]
```

As for strings, there are a number of commands for accessing and manipulating elements of lists. Unlike **string**, these list commands are not part of a single ensemble, but are separate commands using a 'l' prefix, e.g., **lindex**, **lrange** etc.:[2]

llength: Returns the length of the list:

[2]The reason for this discrepancy is largely historical, as the **list** command was already taken as a list constructor.

```
% llength {"Item 1" "Item 2" "Item 3"}
3
```

lindex: Returns the nth element of a list, counting from 0:

```
% lindex {a b c d} 2
c
```

As for strings, you can also use the 'end-n' syntax to access elements from the end.

lrange: Returns all the elements between two indices as a new list:

```
% lrange {"Item 1" "Item 2" "Item 3"} 1 2
{Item 1} {Item 2}
```

Note that Tcl leaves off the outer braces when displaying lists, and prefers to use braces rather than quotes to delimit elements. Most Tcl list commands will 'normalize' lists in this way. Don't be fooled though: Tcl always ensures that the elements of lists can be retrieved in exactly the format that they were created. In other words, the following law holds for all list operations:

```
[lindex [list $x]] == $x
```

You should always use the standard Tcl list commands for creating and dismantling lists to ensure that this law continues to hold. Don't be tempted to use string commands to try and create strings that 'look like lists'. While this can work, the details are much trickier than they appear at first. All Tcl's list commands ensure that the lists they produce are always properly formed.

linsert: Inserts one or more elements into a list, returning the new list with the added elements:

```
% linsert {b c d} 0 a
a b c d
% linsert {a b c} end d
a b c d
```

lreplace: Replaces one or more elements of a list with zero or more substitute elements. You can also use this to delete elements by specifying no replacements:

```
% lreplace {a b c d} 1 2 "Bee" "Cee"
a Bee Cee d
% lreplace {a b c d} 1 2
a d
```

lreverse: Reverses a list:

```
% lreverse {a b c d}
d c b a
```

lrepeat: Creates a list by repeating a set of elements n times. Note that the order of the elements and the count is reversed from that of the **string repeat** command, due to historical reasons:

```
% lrepeat 3 a b c
a b c a b c a b c
```

join: The inverse of the **split** operation, this creates a string by joining together elements of a list with the given joining string:

```
% join {a b c d} ", "
a, b, c, d
```

Note that **join** and **split** aren't complete inverses. In particular, it is not guaranteed that joining a list by a certain delimiter and then splitting that list on the same delimiter will result in the original list. Consider this example:

```
% set x [list a b "c,d" e]
a b c,d e
% set xstr [join $x ,]
a,b,c,d,e
% split $xstr ,
a b c d e
```

foreach The **foreach** command can be used to loop through each element of a list, performing an action at each step. The full syntax of this command is detailed in Section 3.2.4:

```
% set xs {a b c d}
a b c d
% foreach x $xs { puts $x }
a
b
c
d
```

2.3.1 List Variable Commands

A number of commands operate on *variables containing lists*, rather than directly on lists, for reasons of efficiency and convenience, much like the **append** command for strings. These commands also follow the l-prefix convention:

lappend: Appends zero or more elements to a list contained in a variable, storing the new list in the same variable:

```
% set xs [list a b]
a b
% lappend xs c d
a b c d
% set xs
a b c d
```

lset: Updates a list in a variable, replacing the element at the given index with a new value. This command will throw an error if the index given is outside of the range of the list (i.e., less than 0 or greater than the final index in the list):

```
% set xs [list a b c d]
a b c d
% lset xs 1 Bee
a Bee c d
```

lassign: Assigns the elements of a list to the local variables given, returning any unassigned extra elements:

```
% set xs [list a b c d]
a b c d
% lassign $xs first second
c d
% puts "$first, $second"
a, b
```

2.3.2 Nested Lists

As well as string and number elements, Tcl lists can also contain other lists as elements. Some Tcl commands, such as **lindex**, can automatically extract elements from nested lists in a single operation. For instance, we might represent a person as a list of three elements: a name, an address, and a date of birth. The address field is itself a list of strings. For example:

```
% set jon [list "Jon Doe" [list "A House" "Somewhere"] "1-Jan-1970"]
{Jon Doe} {{A House} Somewhere} 1-Jan-1970
```

We can then retrieve the first line of Jon's address using **lindex** and specifying multiple indices. The indices act like a 'path' through the nested list: the first index specifies where to look in the outer list, then the next index where to look in that list, and so on. In this case, the address is the second element of the `jon` list (i.e., element at index 1), and the first line is then element 0 of the address list, so the complete path is `1 0`:

```
% lindex $jon 1 0
A House
```

The **lset** command also recognises nested lists, and can be used to alter variables containing them. For instance, if we wanted to change Jon's house name, we can use:

```
% lset jon 1 0 "Jon's House"
{Jon Doe} {{Jon's House} Somewhere} 1-Jan-1970
% lindex $jon 1 0
Jon's House
```

2.3.3 Sorting and Searching

Tcl builds in powerful facilities for *sorting* and *searching* lists of elements, in the form of the **lsort** and **lsearch** commands:

lsort *?-option value . . . ? list*

lsearch *?-option value . . . ? list pattern*

Sorting a list rearranges the elements into sequence based on some ordering relation. Tcl's **lsort** command is based on an efficient *merge sort* algorithm, which has O(n log n) performance characteristics (i.e., for a list of length n, **lsort** is able to sort that list in time proportional to n log n). The sort is *stable*, meaning that the ordering of elements that compare as equal will be preserved after the sort. By default, **lsort** sorts elements by comparing their string values, using the same ordering as **string compare**:

```
% lsort {b c e g f a d}
a b c d e f g
```

The comparison function can be changed by specifying one of the following options:

-ascii Use string comparison with Unicode code-point collation order (the name is historical). This is the default.

-dictionary Sort using 'dictionary' comparison. This is the same as `-ascii`, except that case is ignored and embedded numbers within the strings are compared as integers rather than as character strings. For example, 'x10y' will sort after 'x2y'. Note though, that negative integers sort without regard for the leading minus sign, so for instance '-10' will be considered 'greater' than '-5'.

-integer Treats the elements of the list as integers, and sorts using integer comparisons.

-real Treats the elements of the list as floating-point numbers.

-command cmd Allows specifying an arbitrary comparison command. This command will be called with pairs of values (as two separate arguments), and should return an integer: less than 0 to mean that the first argument is less than the second, 0 if equal, and greater than 0 otherwise. For instance, `lsort -command {string compare}` is equivalent to the default ASCII sorting order.

Tip: The `-command` option is very slow, as it has to call a Tcl command multiple times during the sort. It is often much faster to massage the list into a format where it can be sorted using one of the built-in comparison functions, typically by creating a *collation key*. The wiki article "Custom sorting" [3] provides a general solution to this problem, along with much useful discussion of performance optimisations.

A number of other options can be used to change the sorting order:

-increasing Sorts the elements from smallest to largest (the default).

-decreasing Sorts the elements from largest to smallest.

-indices Returns the indices of elements in the list, rather than their values. This is useful if you are sorting one list based on information contained in another list.

-index indexList This useful option allow you to sort a list of lists based on one particular element in each sub-list. You can even pass multiple indices, in which case they will be treated as a path through each nested sub-list, exactly as for **lindex** (see Section 2.3.2). For example, if we have a list of people, where each person is represented a list of name, age, we can sort based just on the age using the following command:

```
% set people {{Jon 32} {Mary 24} {Mike 31} {Jill 20}}
{Jon 32} {Mary 24} {Mike 31} {Jill 20}
% lsort -integer -index 1 $people
{Jill 20} {Mary 24} {Mike 31} {Jon 32}
```

-stride strideLength Since Tcl version 8.6 This option can be used in conjunction with `-index` to specify that each sub-list is not a nested list, but instead the entire list should be taken as being grouped into strideLength lists of elements. For example, if our list of people was instead represented as a flat list of name-age-name-age..., then we could still sort it using the following command:

```
% set people {Jon 32 Mary 24 Mike 31 Jill 20}
Jon 32 Mary 24 Mike 31 Jill 20
% lsort -integer -index 1 -stride 2 $people
Jill 20 Mary 24 Mike 31 Jon 32
```

-nocase Causes ASCII/Unicode comparisons to be case-insensitive. Only meaningful with `-ascii` sort mode.

-unique Causes duplicate elements to be eliminated from the resulting list. Duplicate elements are those which compare as equal. The last duplicate element from the input list will be the one that is preserved. For example, if `-index 0` is specified and the two elements `{1 a}` and `{1 b}` are present, in that order, then only `{1 b}` will be present in the output.

As an example, we can retrieve a list of all commands defined in a Tcl interpreter, sort them, and then print them out, one per line, using the following 'one-liner':

```
puts [join [lsort [info commands]] \n]
```

As well as sorting lists, you can also search for elements within them using the **lsearch** command. Like **lsort**, this command comes with a somewhat bewildering array of options to control its operation. However, its basic usage is very straight-forward: given a list and a pattern to search for, it will return the index of the first element that matches the pattern, or -1 if it cannot be found. By default this uses 'glob'-style matching, as described in Section 2.1.2:

```
% lsearch {foo bar jim} b*
1
```

You can control the way that matching is performed using the following options:

-exact Use exact string matching.

-glob Use glob-style matching (the default).

-regexp Use regular-expression matching (see Section 2.6).

-sorted Specfies that the elements of the list are in sorted order. This allows the search to use a more efficient algorithm. This option cannot be used with the `-glob` or `-regexp` options.

-inline Causes the command to return the actual value rather than the index:

```
% lsearch -inline {foo bar jim} b*
bar
```

-all Causes the command to return a list of every matching index (or value in the case of `-inline`). Returns an empty list for no matches.

-not Negates the match, returning the index of the first element that does *not* match the pattern.

-start index Causes the search to start from `$index`.

The **lsearch** command also supports options for specifying the contents of each element (as for **lsort**), e.g., `-ascii`, `-integer` etc., as well as specifying the sort order of sorted lists, finding the nearest element to a given pattern (if it is not exactly matched), or searching within nested lists.

2.4 Dictionaries

Since Tcl version 8.5

While a list is an ordered collection of elements, a dictionary is an unordered[3] mapping from keys to values. Other languages might refer to such data structures as *maps* or *(hash-)tables*. Rather than each element having a position and an integer index, as in a list, each element in a dictionary is given a string name. The elements of a Tcl dictionary can be any string value, allowing them to also be used in a similar manner to records or structures in typed languages. The format of a Tcl dictionary is that of a list with an even number of elements: the even indexed elements (0, 2, 4, ...) corresponding to the *keys*, and the odd indexed elements correspond to the values for those keys. For instance, we can represent a person in the following format:

```
set jon {
    name    "Jon Doe"
    address {{A House} {Somewhere}}
    dob     1-Jan-1970
}
```

Viewed as a list, the `$jon` variable contains 6 elements. Viewed as a dictionary, it contains 3 mappings for the keys `name`, `address`, and `dob`. As well as creating dictionaries using a literal string, you can also use the **`dict create`** constructor command, that works like the **`list`** command, but creates a dictionary. We can retrieve the value associated with a key using the **`dict get`** command:

```
% dict get $jon dob
1-Jan-1970
```

You can check whether a dictionary contains a mapping for a given key using the **`dict exists`** command:

```
% dict exists $jon dob
1
% dict exists $jon some-other-key
0
```

Both of these commands can take a series of keys (as separate arguments), allowing you to access elements in nested dictionaries, in a similar manner to that for nested lists. Some other useful dictionary commands are as follows:

`dict size` Returns the number of key-value mappings in the dictionary.

`dict keys` Returns a list of the keys contained in the dictionary. You can optionally specify a glob-pattern to restrict which keys are returned.

`dict values` Returns a list of the values contained in the dictionary. Again, you can specify a glob-pattern to restrict which values are returned (note: this pattern matches the value, not the associated key).

`dict replace` Returns a new dictionary value with some key/value pairs altered, and possibly new entries add to the dictionary:

```
% set jane [dict replace $jon name "Jane Doe" gender female]
name {Jane Doe} address {{A House} {Somewhere}} dob 1-Jan-1970
gender female
```

[3] Actually, dictionaries preserve the order of elements as they are given, adding any new keys to the end.

dict set Manipulates a dictionary-holding *variable* to change the value under one (possibly nested) key:

```
# Change Jon's date-of-birth
dict set jon dob 2-Jan-1970
```

dict merge Returns a new dictionary that combines the entries from each dictionary given as arguments. If two or more dictionaries contain entries for the same key, then the value of the last dictionary containing that key will be used. For instance, you could use this to implement something like CSS's cascading style options:

```
set default {
    font-weight: normal
    font-shape: roman
}
set user {
    font-family: sans-serif
}
set author {
    font-family: serif
    font-weight: bold
}
# Merge the styles, preferring author over user over defaults
set style  [dict merge $default $user $author]
# Extract final values from the combined dictionary
set family [dict get $style font-family:]
set weight [dict get $style font-weight:]
set shape  [dict get $style font-shape:]
puts "Paragraph font: $family $shape $weight"
```

dict remove Returns a new dictionary with entries for the given keys removed. Note that this command will not error if the given keys do not exist in the dictionary.

dict unset Removes a (possibly nested) key from a dictionary contained within a variable, updating the variable in-place.

dict for Loops through each key/value pair in the dictionary. This is similar to the **foreach** loop for lists, but specialised (and optimised) for dictionary values:

```
dict for {key value} $jon {
    puts "$key = $value"
}
```

In addition to these commands, the **dict** command also includes a number of convenience commands for manipulating common data in-place within a dictionary. For example, the **dict incr** command allows efficient in-place incrementing of an integer value stored within a dictionary. For example, we can print a summary of word counts within a text using the following code (ignoring punctuation for now):

```
set word_counts [dict create]
foreach word [split $text] {
    dict incr word_counts $word
```

```
}
# Display the results
dict for {word count} $word_counts {
    puts [format "%-30s : %d" $word $count]
}
```

You can sort a dictionary using **lsort**, but not with the `-dictionary` option! (The name clash is accidental). Instead, you can use the `-stride` and `-index` options, taking advantage of the fact that all dictionaries are also lists:

```
set sorted_words [lsort -integer -stride 2 -index 1 $word_counts]
```

2.4.1 Filtering a Dictionary

The **dict** command comes with powerful functionality for filtering a dictionary value to select just those key-value pairs that match some criteria. The **dict filter** command supports three different forms of filtering:

- *key* and *value* filtering returns those key-value pairs whose key or value (respectively) matches one of a set of glob patterns, similar to **lsearch**.
- *script* filtering allows a script to be run on each key-value pair of the dictionary, and includes only those elements for which the script returns a true value.

For example, given our dictionary of word counts from the previous section, we can return the counts of all words beginning with 'a' using:

```
set a_words [dict filter $word_counts key a*]
```

Or we could return all words with a count in double figures:

```
set d_words [dict filter $word_counts value ??]
```

Finally, we could return all words with a count greater than 15 using:

```
set frequent [dict filter $word_counts script {key value} {
  expr {$value > 15}
}]
```

2.4.2 Updating a Dictionary

The **dict** command also includes a very convenient feature for updating the contents of a dictionary variable in-place using an arbitrary script. The **dict update** and **dict with** commands unpack a dictionary value into a set of local variables with the names and values given by the contents of the dictionary. A script can then be executed that manipulates these variables. At the end of the script, any changes are read back from the variables and the corresponding changes are made to the original dictionary. For example, if we consider our 'jon' dictionary from the start of Section 2.4:

```
set jon {
    name    "Jon Doe"
    address {{A House} {Somewhere}}
    dob     1-Jan-1970
}
```

We can use the **dict with** command to unpack this data and manipulate it more concisely than by using individual **dict get** and **dict set** commands:

```
dict with jon {
    puts "Name: $name"
    puts "Addr: [join $address {, }]"
    puts "DOB : $dob"

    # Change Jon's name
    set name "Other Jon"
}
puts "Jon's name is now: [dict get $jon name]"
```

The **dict update** command works in a similar fashion, except that it allows you to specify exactly which entries should become variables and also to specify what those variable names should be (rather than just using the key name). This approach can be used to make batch manipulations to a dictionary value, using the full range of Tcl commands for manipulating ordinary variables.

Tip: While this update facility is very useful and convenient, there are a few 'gotchas' to be aware of. Firstly, you should avoid using the **dict with** command with dictionary values that may come from user input as this could allow a malicious user to override private variables with incorrect information. Instead, always use the **dict update** command so that you can control exactly which variables will be created or updated. Secondly, be aware that entries from the dictionary could accidentally overwrite the dictionary variable itself, causing an error. For example, suppose our 'jon' dictionary happened to contain a key called 'jon'. In that case, **dict with** would overwrite our initial dictionary with the contents of the jon key, either creating an error or (even worse) silently destroying some information:

```
% dict set jon jon "garbage"
...
% dict with jon { }
missing value to go with key
% set jon
garbage
```

2.5 Arrays

Tcl's *arrays* are similar in some respects to dictionaries: they are unordered mappings from string keys to some value.[4] In contrast to dictionaries, however, arrays map from string keys to *variables* rather than values. This means that arrays are not themselves first-class values in Tcl and cannot be passed to and from procedures without some extra work. On the other hand, arrays provide a convenient syntax and can be updated in-place just like other variables. It is an error to have an array variable and a normal variable with the same name.

Array variables can be created and manipulated just like regular variables, but using the special array syntax, which consists of a normal variable name followed by a string key in parentheses. For example, to create an array variable named 'jon' containing name, address, and date-of-birth fields, we would write:

```
set jon(name)       "Jon Doe"
set jon(address)    {"A House" "Somewhere"}
set jon(dob)        1-Jan-1970
```

[4] This may be confusing to users from other languages where an 'array' is usually indexed by an integer rather than a string. Tcl's arrays are really 'associative arrays' or hashtables.

The **array set** command can also be used to create an array in one go:

```
array set jon {
    name            "Jon Doe"
    address         {"A House" Somewhere}
    dob             1-Jan-1970
}
```

Once an array has been created, we can use the array syntax to manipulate its elements using normal variable commands. An array entry can be used pretty much everywhere that a normal variable can be:

```
puts "Name: $jon(name)"
set jon(name) "Other Jon"
lappend jon(address) "Planet Earth"
```

The **parray** command can be used to display the contents of an array variable:

```
% parray jon
jon(address) = {A House} Somewhere {Planet Earth}
jon(dob)     = 1-Jan-1970
jon(name)    = Other Jon
```

Given the apparent similarity between arrays and dictionaries, it may not seem obvious when you should use one or the other. In fact, the two are quite different things and are useful in different situations. A dictionary should be used when your primary objective is to create a data structure which is naturally modelled as a collection of named fields. Using a dictionary gives you all the advantages of a normal Tcl value: it can be passed to and from procedures and other commands, it can be sent over communication channels (covered in Chapter 5), and it can be easily inserted into other data structures. Tcl also automatically manages the memory for dictionaries, as for other values, so you do not need to worry about freeing up resources when you have finished with it. An array, on the other hand, is a complex stateful entity, and is best used when you want to model a long-lived stateful component in your application. For instance, it can be very useful to associate an array with a graphical user interface component, to hold current information on the state of the display and user interactions. Such information typically lasts a long time and is constantly changing in response to user inputs. A dictionary could also be used in this situation, but an array as a collection of variables brings advantages such as variable traces that really shine in these kinds of situations.

2.6 Regular Expressions

Regular expressions are a compact means of expressing complex patterns for matching strings and extracting information. If you are not already familiar with regular expressions from other languages, such as Perl, then they can be quite daunting at first, with their terse and cryptic syntax. However, once you have mastered the basics they soon become an indispensable tool that you will wonder how you ever managed without! This tutorial will give only a very brief introduction to the basic features of regular expressions and the support provided for them in Tcl. There are many tutorials and good books devoted to the subject available on the Web and in technical bookstores if you wish to learn more. Readers who have come across regular expressions in other languages will find that Tcl uses the familiar POSIX-style syntax for basic regular expressions. Note that Tcl does not use the popular Perl-Compatible Regular Expression (PCRE) syntax used in several modern programming languages, so some advanced features may be slightly different.[5]

[5] It is also worth noting that Tcl's regular expression engine, written by Henry Spencer, is of a different design to most other languages, with different performance characteristics. Patterns that are slow in some implementations may be faster in Tcl, and vice-versa.

`^`	Matches the beginning of the string.
`$`	Matches the end of the string.
`.`	Matches any single character.
`[...]`	Matches any character in the set between the brackets.
`[a-z]`	Matches any character in the range a..z.
`[^...]`	Matches any character *not* in the set given.
`(...)`	Groups a pattern into a sub-pattern.
`p|q`	Matches pattern p or pattern q.
`*`	Matches any count (0–n) of the previous pattern.
`+`	Matches at least 1 occurrence of the previous pattern.
`?`	Matches 0 or 1 occurrence of the previous pattern.
`{n}`	Matches exactly n occurrences of the previous pattern.
`{n,m}`	Matches between n and m occurrences of the previous pattern.

Table 2.1: Basic regular expression syntax.

Before getting into the technicalities of regular expressions (REs) in Tcl, it is worth pointing out a little of the theoretical origins and practical limitations. A regular expression is so-called because it is capable of matching a *regular language*. Such languages are relatively simple in the grand scheme of things: no popular computer programming language has a grammar that simple, for instance. As an example, no regular expression is capable of matching a string only if the braces within it are *balanced* (i.e., each open brace is paired with exactly one close brace). It is therefore good to be aware of the limits of regular expressions, and when a more appropriate technology should be used. A classic example is extracting information from XML or HTML documents: at first it can look as if a simple RE will do the trick, but it usually then becomes apparent that XML is more complicated than it looks. Soon you end up with a large and unwieldy RE that *still* doesn't match every example. The solution is to use a proper XML parser. However, this is not to say that regular expressions are not a powerful tool. Firstly, almost all regular expression engines—and Tcl is no exception here—implement more than the basic regular expressions (BREs) of theory. Various extensions to REs allow writing quite complex patterns quite succinctly, making them a handy tool to have in your toolbox. Even when your problem requires more than a RE can provide, regular expressions can still help when building a more comprehensive parser. In terms of parsing theory, regular expressions are most useful for *lexical analysis* while more sophisticated tools (such as recursive descent parsers) are more suited to *syntactic analysis*.

Tcl supports regular expressions primarily through two commands:

regexp *?options? RE string ?matchVar ...?*

regsub *?options? RE string subSpec ?varName?*

The **`regexp`** command searches `string` for the pattern `RE`. If it matches, then the command returns the index at which it matched, and optionally copies the matched string, and any sub-matches, into one or more named variables supplied at the end. The **`regsub`** command also matches a regular expression against a string, but in this case is substitutes the matching string with the string given in `subSpec`. Table 2.1 gives the basic elements of regular expression syntax.

Regular expressions are similar to the glob-matching that was discussed in Section 2.1.2. The main difference is the way that sets of matched characters are handled. In globbing the only way to select sets of unknown text is the `*` symbol, which matches any quantity of any character. In regular expression matching, this is much more precise. Instead of using such 'wild-card' matching, you instead can say exactly which characters you wish to match and then apply a modifier to this pattern to say how many times you wish to match it. For example, the RE `a*` will match 0, 1 or any number of `a` characters in a sequence. Modifiers can also be applied to sub-patterns, not

just individual characters, so that `[abc]*` will match strings of any length containing just the characters `a`, `b`, or `c` (e.g., 'bbacbaababc'). Another way of writing this pattern would be to use a *character range*: `[a-c]*`, which again matches strings of any length (including empty strings) consisting of just characters between `a` and `c` in the Unicode alphabet. You can also create *negated patterns* which match anything that *isn't* in a certain set of characters, for example: `[^a-c]` will match any character that isn't `a`, `b`, or `c`. As elsewhere in Tcl, a backslash character can be used to escape any of this special syntax to include literal asterisks and so on in a pattern. For instance, to match any number of dots, we can use `\.*`. Here are some examples of regular expressions:

```
# Match an Internet Protocol (IP4) address in dotted-form:
set re(ip)  {([0-9]{1,3}\.){3}[0-9]{1,3}}
# Example:
regexp $re(ip) 192.168.0.1 ;# returns 1 (i.e., matched)

# Match a Web URL
set re(web) {^https?://([^:/]+)(:[0-9]+)?(/.*)?$}
regexp $re(web) http://www.tcl.tk:80/docs/ ;# matches
regexp $re(web) www.slashdot.org ;# no match

# Match an email address
set re(email) {^([^@]+)@(.*\.(com|org|net))$}
regexp $re(email) jeff@example.org ;# matches
```

As you can see from these examples, regular expressions are both difficult to read, but also capable of performing quite complex matching. Let's examine one of these patterns in more depth to see what is going on. If we take the Web URL pattern, we can break it apart step by step:

1. Firstly, the entire pattern is surrounded by a `^...$` pair of *anchors*. These ensure that the pattern only matches the entire string that is given, as `^` will only match at the start of the string, and `$` will only match at the end. This is a common pattern which you will see on many regular expressions.

2. The first part of the pattern then matches against the *protocol* part of the URL: the initial `http://` bit that tells us that this is a HyperText Transfer Protocol (HTTP) link. The pattern here is just a literal string, with the exception that we also allow secure HTTPS URLs using the optional pattern `s?`. Note that as this pattern follows the initial `^` anchor, it will only match as the very first characters of the input string.

3. The next part of a link is the host name of the server we wish to connect to, for instance `wiki.tcl.tk`. We could try and specify exactly which characters can appear in this part of the URL, but it is often easier to specify those which cannot: in this case, we know that the colon and forward-slash characters are forbidden in host names, and so we can use a negated pattern to match anything but these: `[^:/]`. As the host name is not optional, we want to match at least one such character, but with no upper limit on the size. Therefore we use the `+` modifier. Finally, we group this pattern into a sub-pattern to allow us to capture the host name when matching: `([^:/]+)`.

4. A HTTP URL can have an optional *port* number following the host name. This is simply a colon followed by a simple positive integer, which we can match as just `:[0-9]+` (a colon followed by 1 or more digits). We again make this into a sub-pattern for capture and also make the entire sub-pattern optional: `(:[0-9]+)?`.

5. Finally, we also match the *path* of the requested document on the target server. This part is also optional, and we define it as simply anything else remaining at the end of the URL following an initial forward-slash: `(/.*)?`.

6. Putting everything together, we arrive at our final pattern:

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

Beyond simple matching, REs are also capable of *extracting* information from a string while it is being matched. In particular, any sub-pattern in a RE surrounded by parentheses can be extracted into a variable by the **regexp** command:

```
regexp $re(web) https://example.com:8080/index.php match host port path
```

After this command, the `match` variable will contain the full string that was matched (i.e., `https://example.com:8080/index.php`), while the `host`, `port`, and `path` variables will each contain their respective parts of the URL: i.e., `example.com`, `8080` and `/index.php` respectively. The **regexp** command can also be used to count the number of occurrences of a given pattern, by passing the `-all` option as an initial argument:

```
puts "Number of words: [regexp -all {[^ ]+} $text]"
```

The **regsub** command allows substitutions to be performed on a string based on the matching of a regular expression pattern, either returning the modified string or saving it in a new variable. The substitution string can itself refer to elements of the matched RE pattern, by using one or more substitution escapes of the form `\N` where N is a number between 0 and 9: `\0` will be replaced with the string that matched the entire RE, `\1` with the string that matched the first sub-pattern, and so on. You can also use the symbol `&` in place of `\0`. For instance, if you want to take a plain text file and convert it to HTML by making every instance of your own name highlighted in bold, you could achieve that with a single **regsub** command:

```
regsub -all {Neil Madden} $text {<b>\0</b>} html
```

2.6.1 Crafting Regular Expressions

Regular expressions provide a very powerful method of defining a pattern, but they are a bit awkward to understand and to use properly. In this section we will examine some more examples in detail to help the reader develop an understanding of how to use this technology to best effect. We start with a simple yet non-trivial example: finding *floating-point numbers* in a line of text. Do not worry: we will keep the problem simple than it is in its full generality. We only consider numbers like 1.0 and not 1.00e+01.

How do we *design* our regular expression for this problem? By examining typical examples of the strings we want to match:

- Examples of valid numbers are: $1.0, .02, +0., 1, +1, -0.0120$.
- Examples of invalid numbers (that is, strings we do not want to recognise as numbers but which superficially look like them): $-, +, 0.0.1, 0..2, ++1$.
- Questionable numbers are: $+0000$ and 0001. We will accept them—because they normally are accepted and because excluding them makes our pattern more complicated.

A pattern is beginning to emerge:

- A number can start with a sign ($-$ or $+$) or with a digit. This can be captured with the pattern `[-+]?`, which matches a single $-$, and single $+$ or nothing.
- A number can have zero or more digits in front of a single period (.) and it can have zero or more digits following the period. Perhaps `[0-9]*\.[0-9]*` will do ...
- A number may not contain a period at all. So, revise the previous expression to: `[0-9]*\.?[0-9]*`

The complete expression is:

```
[-+]?[0-9]*\.?[0-9]*
```

At this point we can do three things:

1. Try the expression with a bunch of examples like the ones above and see if the proper ones match and the others do not.
2. Try to make the expression look nicer, before we start testing it. For instance, the class of characters `[0-9]` is so common that it has a shortcut: `\d`. So, we could settle for `[-+]?\d*\.?\d*` instead. Or we could decide that we want to capture the digits before and after the period for special processing: `[-+]?(\d*)\.?(\d*)`
3. Or (and this is a good strategy in general!), we can carefully examine the pattern before we start actually using it.

You may have noticed a problem with the pattern created above: all the parts are optional. That is, each part can match a null string—no sign, no digits before the period, no period, no digits after the period. In other words, *our pattern can match an empty string!* Our questionable numbers, like '+000' will be perfectly acceptable and we (grudgingly) agree. But, more surprisingly, the strings '–1' and 'A1B2' will be accepted too! Why? Because the pattern can start anywhere in the string, so it would match the substrings '-1' and '1' respectively. We need to reconsider our pattern—it is too simple, too permissive:

- The character before a minus or a plus, if there is any, cannot be another digit, period or a minus or plus. Let is make it just a space or a tab or the beginning of the string: `^|[ \t]`. This may look a bit strange, but what it says is: either the beginning of the string (`^` outside the square brackets) *or* (the vertical bar) a space or tab (remember: the string `\t` represents the tab character).
- Any sequence of digits before the period (if there is one) is allowed: `\d+\.?`
- There may be zero digits in front of the period, but then there must be at least one behind it: `\.\d+`
- And, of course, digits in front and behind the period: `\d+\.\d+`
- The character after the string (if any) can not be a +, − or '.' as that would get us into the unacceptable number-like strings: `$|[^+-.]` (the dollar sign signifies the end of the string).

Before trying to write down the complete regular expression, let us see what different forms we have:

- No period: `[-+]?\d+`
- A period without digits before it: `[-+]?\.\d+`
- Digits before a period, and possibly digits after it: `[-+]?\d+\.\d*`

Now the synthesis:

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

Let's put it to the test:

```
set pattern {(^|[ \t])([-+]?(\d+|\.\d+|\d+\.\d*))($|[^+-.])}
set examples {
    "1.0"    "  .02"      "  +0."
    "1"      "+1"         " -0.0120"
    "+0000" " - "         "+."
```

```
    "0001"  "0..2"      "++1"
    "A1.0B" "A1"
}
foreach e $examples {
    if {[regexp $pattern $e whole _ number digits _]} {
        puts "PASS: >>$e<<: $number ($whole)"
    } else {
        puts "FAIL: >>$e<<: Does not contain a valid number"
    }
}
```

The result is:

```
PASS: >>1.0<<: 1.0 (1.0)
PASS: >>  .02<<: .02 ( .02)
PASS: >>  +0.<<: +0. ( +0.)
PASS: >>1<<: 1 (1)
PASS: >>+1<<: +1 (+1)
PASS: >> -0.0120<<: -0.0120 ( -0.0120)
PASS: >>+0000<<: +0000 (+0000)
FAIL: >> - <<: Does not contain a valid number
FAIL: >>+.<<: Does not contain a valid number
PASS: >>0001<<: 0001 (0001)
FAIL: >>0..2<<: Does not contain a valid number
FAIL: >>++1<<: Does not contain a valid number
FAIL: >>A1.0B<<: Does not contain a valid number
FAIL: >>A1<<: Does not contain a valid number
```

Our pattern correctly accepts the strings we intended to be recognised as numbers and rejects the others. See if you can adapt it to accept more valid forms of numbers, such as scientific notation (`1.03e-2`) or hexadecimal (`0x12`).

Here are a few other examples of uses for regular expressions:

- Text enclosed in quotes: *This is 'quoted text'*. If we know what character will be used for the quotes (e.g., a double-quote) then a simple pattern will do: `"([^"]*)"`. If we do not know the enclosing character (it can be `"` or `'`) then we can use a so-called 'back-reference' to the first captured sub-string:

  ```
  regexp {(["']).[^"']*\1} $string
  ```

  The `\1` reference is replaced with whichever character matched at the start of the string.

- You can use this technique to see if a word occurs twice in the same line of text[6]:

  ```
  set string "Again and again and again ..."
  if {[regexp {(\y\w+\y).+\1} $string -> word]} {
      puts "The word $word occurs at least twice"
  }
  ```

- Suppose you need to check the parentheses in some mathematical expression, such as $(1 + a)/(1 - b \times x)$. A simple check is to count the open and close parentheses:

[6] This RE uses some advanced features: the
`\y` anchor matches only at the beginning or end of a word, and
`\w` matches any alpha-numeric character. The `->` symbol is not a new bit of syntax but just a handy name for the variable that captures the entire match. This idiom is used frequently in Tcl code to indicate that the whole match was not required.

```
if {[regexp -all {(} $exp] != [regexp -all {)} $exp]} {
    puts "Parentheses unbalanced!"
}
```

Of course, this is just a rough check. A better one is to see if at any point while scanning the string there are more close parentheses than open ones. We can easily extract the parentheses into a list and then check that (the `-inline` option helps here):

```
set parens [regexp -all -inline {[()]} $exp]
set counts [string map {( 1 ) -1} $parens]
set balance 0
foreach c $counts {
    if {[incr balance $c] < 0} { puts "Unbalanced!" }
}
if {$balance != 0} { puts "Unbalanced!" }
```

We use a trick here by replacing the parentheses in the list with either +1 or -1. This allows us to quickly sum the list to check for balanced parentheses.

A number of tools are available to help experimenting with regular expressions, and a number of books cover the topic in depth, such as J. Friedl's "Mastering Regular Expressions" [7].

Chapter 3

Control Structures

In this chapter we introduce the various commands that Tcl includes for controlling how the program executes. These include conditional branches for making decisions, loops for counting and iterating over data structures, constructs for dealing with error conditions, and procedures for creating new commands.

3.1 Branching with `if` and `switch`

3.1.1 The `if` Command

Like most languages, Tcl supports an **`if`** command for branching based on a simple Boolean condition. The syntax is as follows:

if *cond* ?**then**? *body1* ?**elseif** *cond2 body2* ...? ?**else** *bodyN*?

The **`then`** and **`else`** keywords are optional. Typically, **`then`** is left out whereas **`else`** is usually used, as in the following example:

```
if {$x < 0} {
    puts "$x is negative"
} elseif {$x > 0} {
    puts "$x is positive"
} else {
    puts "$x is zero"
}
```

The test expression should return a value that can be interpreted as representing 'true' or 'false'. Valid Boolean values are shown in Table 1.2. The string constants 'yes'/'no' and 'true'/'false' are case-insensitive, so 'True' and 'FALSE' and 'YeS' etc. are all valid truth values. If the test condition expression evaluates to *true* then *body1* is executed, otherwise any further conditional checks, specified in **elseif** clauses are tested in turn. Finally, if no conditional expression evaluates to true, then the final body following an **`else`** clause is executed (if supplied). All test expressions are evaluated in the same manner as the **`expr`** command, discussed in Section 1.5. Each `body` is an arbitrary Tcl script (sequence of commands separated by newlines or semi-colons). As for **`expr`**, it is good style to surround all test expressions with braces. The **`if`** command returns the result of the last command executed in the selected code body, or the empty string if no condition matched (and no else clause was supplied).

Tip: The **`if`** command in Tcl is just another command, and is not a special piece of syntax as in other languages. This is a feature of Tcl, and part of what makes the language so powerful. However, you should be aware that this also means that **`if`** statements have to follow the same syntax rules as all other commands. In particular, each word in the command must be separated by whitespace, and you must be careful where newlines are inserted in the command.

```
# INCORRECT examples of if
if {$x < 0}{ ;# no space between the braces!
    ...
}
if {$x < 0} { ... }
else { ... } ;# "else" cannot go on a line by itself!
```

3.1.2 The `switch` Command

An alternative to a long sequence of **`if`**/**`elseif`**/**`else`** branches is to use the **`switch`** command. Tcl's **`switch`** is similar to the `switch` statements found in languages such as C or Java, except that it allows switching on arbitrary strings rather than just integers and builds in some sophisticated pattern matching machinery. The **`switch`** command matches a value against a list of patterns; the first pattern that matches the value is chosen and an associated code body is executed. A **`switch`** command can often be easier to read than a long **`if`** command, but is more restricted in the types of conditions it can match. The syntax of the command takes one of two forms:

switch *?options? value pattern1 body1 ?pattern2 body2? ... ?patternN bodyN?*
— or —
switch *?options? value* { *pattern1 body1 ?pattern2 body2? ... ?patternN bodyN?* }

The second form is the one most commonly used, with the first form only being used in specific situations (usually only when the patterns themselves are held in variables or otherwise computed at runtime). The *value* argument is the string that you wish to match, and the *pattern* arguments are the patterns to match it against. If the value matches a pattern then the associated *body* argument will be executed. The return value of that body becomes the return value of the **`switch`** command. Only a single pattern will be matched. If the last pattern is the word `default`, then that pattern will match any value that is not matched by a previous pattern. This is similar to the `else` clause in the **`if`** command. If there is no default pattern, and no other pattern matches the value, then **`switch`** returns an empty string. Unlike in other languages, you do not have to use a **`break`** statement to terminate the body of a switch branch, as Tcl will not fall through to the next branch anyway. Instead, you may use the special body '-' (a single hyphen/dash) to indicate that the next branch should be used for multiple conditions:

```
# Use the same branch body for multiple conditions
switch $x {
    0   -   1   -   2   -
    3       { puts "$x is between 0 and 3" }
    default { puts "$x > 3 or $x < 0" }
}
```

Tip: The syntax of **`switch`** *can not contain comments!* This is a frequently made mistake for newcomers to Tcl from other languages. Remember that a comment is only valid where Tcl expects the start of a new command. The **`switch`** command however expects a list of patterns and bodies rather than a sequence of commands, so you cannot insert arbitrary comments:

```
switch $x {
    # This is not a comment!
    default {
        # But this is, because Tcl expects a command here
        puts "Test"
    }
}
```

By default, Tcl uses exact string matching when determining which pattern matches the input value[1]. A number of *options* can be given to the switch command to change the matching process:

`-exact` Use exact string matching (the default).

`-glob` Use 'glob-style' pattern matching (see Section 2.1).

`-regexp` Use regular-expression pattern matching (see Section 2.6).

`-nocase` Use case-insensitive matching.

`--` Indicates 'end of options', and should always be specified if the value being matched may start with a dash.

A handful of other options can also be specified to further control the matching process. These are listed in the switch manual page [10].

3.2 Loops: `while` and `for`

Tcl includes two general commands for looping, the **`while`** and **`for`** commands. Both of these commands implement *imperative* loops, as found in languages such as C or Java. That is, these loops work by making updates to some state at each iteration until some termination condition has been satisfied. In addition to the general looping commands, Tcl also has a number of specialised loop commands, such as **`foreach`**, which loop over the elements of a particular collection, e.g., a list. The **`foreach`** loop is discussed in Section 3.2.4, while other loops are introduced elsewhere in the book.

3.2.1 The `while` Loop

The **`while`** command evaluates a script repeatedly while some condition remains true. The syntax of the command is as follows:

while *test body*

The `test` argument is an expression, in the same format as for **`expr`** (see Section 1.5), which must evaluate to a Boolean value. If the test evaluates to true, then the `body` script is executed, and then the loop is repeated. If the test evaluates to false, then the loop terminates and control

[1] In earlier versions of Tcl, glob-matching was the default, so it is good to get into the habit of always specifying what sort of matching you require.

moves on to the next command in the program. For example, we can use a **while** loop to count from 1 to 10[2]:

```
set i 1
while {$i <= 10} {
    puts "i = $i"
    incr i
}
puts "Done!"
```

It is important to note that changes made to variables within the loop persist once the loop has terminated. In this case, the variable `i` will be set to 10 after the loop has finished. If the `test` condition evaluates to false when the **while** loop is first executed, then the `body` is never executed, as in this example:

```
set x "no"
if {$x} { puts "Cannot happen!" }
```

The `test` condition to **while** *must* be placed within braces. The reason for this is that the **while** command, like every statement in Tcl, is just another command and the same syntax rules apply. In particular, only a single round of substitution is done on the command, If braces were not used then Tcl would substitute the current values for any variables in the expression when the command is originally called, and these would then never be updated, resulting in the test condition never evaluating to false, which would then cause an *infinite loop*.

3.2.2 The `for` Loop

Tcl supports an iterated loop construct similar to the `for` loop found in C or Java. The **for** command in Tcl takes four arguments: an initialisation, a test, a 'next' script, and the body of code to evaluate on each pass through the loop:

for *init test next body*

During evaluation of the **for** command, firstly the `init` argument is evaluated before any other evaluation takes place. This can be used to initialize any variables and other resources used in the loop body. After the start code has been evaluated, the `test` expression is evaluated. As for the **while** loop, this is a braced expression of any format acceptable to **expr**. If the `test` evaluates to true, then the `body` script is evaluated, just like in **while**. However, once the loop body has completed, **for** then evaluates the `next` argument to prepare the loop for the next iteration. The loop then repeats from evaluating the `test`. The complete cycle is as follows:

1. Evaluate the `init` script.
2. Evaluate the `test` expression: if *false* then terminate; otherwise continue.
3. Evaluate the loop `body` script.
4. Evaluate the `next` script.
5. Go to step 2.

By including `init` and `next` arguments, the **for** loop helps to separate those parts of the code that are part of the application logic (i.e., everything in the loop body), from those parts that are just there to control the loop. Compare the following **for** loop for counting from 1 to 10 to the example shown in Section 3.2.1:

[2]It is conventional to name loop variables as `i` or `j`. Any name could be used, but experienced programmers will recognise i and j as loop variables.

```
for {set i 1} {$i <= 10} {incr i} {
    puts "i = $i"
}
puts "Done!"
```

Notice how all references to the loop variable, `i` are now entirely contained within the **`for`** loop, and clearly separated from the loop body. Please note, though, that this does not mean that the scope of the `i` variable is confined to the loop. In fact, like a **`while`** loop, the variable will still exist after the loop with the value it was last assigned (10 in this case).

The `init` and `next` arguments can contain arbitrary Tcl scripts. In particular, you can initialise and update multiple loop variables in these scripts, or perform other resource initialisation. For instance, here is a program to display all the lines in a text file[3] with line numbers:

```
for {set i 1; set in [open myfile.txt]} {[gets $in line] != -1} {incr i} {
    puts "$i: $line"
}
close $in
```

While such code is possible, it is not always a good idea. In this example, for instance, the `test` expression is also being used to actually read a line of the file into the `line` variable, which may not be immediately obvious to a reader of this code. The choice of when to use a **`for`** loop and when to use a plain **`while`** loop is largely one of determining which form is most readable. If your loop can be cleanly broken into initialisation, iteration, and test invariants, then a **`for`** loop can be a good choice, otherwise a **`while`** loop may be clearer. Ultimately, what matters most is whether the resulting code is understandable to somebody reading it later.

As for the **`while`** loop, the `test` expression should *always* be enclosed in braces. It is good style to also enclose the other arguments in braces.

3.2.3 Early termination: `break` and `continue`

Sometimes even the **`for`** command is not flexible enough to accurately capture all the termination and iteration conditions of your loop (i.e., when to stop and when to start the next cycle through the loop). Tcl provides two further commands that can be used with **`for`** and **`while`** to allow finer control over these conditions: the **`break`** and **`continue`** commands. These commands can be used within the loop body, and when encountered, they cause the loop to stop processing that iteration and to perform a special action:

`break`: Causes the loop to terminate completely.

`continue`: Causes the loop to stop this iteration and begin on the next.

The **`continue`** command essentially just ends the current loop iteration early. Any test conditions and `next` steps are still evaluated before the next iteration. The **`break`** command stops the loop altogether. One common use of these commands is to code apparently *infinite loops*, in which the termination condition is actually hidden within the body of the loop. Here is an example, in which we want to read user input until they enter a special 'quit' message:

```
# An infinite loop
while 1 {
    set input [gets stdin]
    if {$input eq "quit"} { break }
    puts "You said: $input"
}
puts "Goodbye!"
```

[3] The commands for reading files are covered in more detail in Chapter 5

Tip: Unlike some other languages, where `break` and `continue` are special syntax that applies only to loops, in Tcl even they are just ordinary commands! They use Tcl's very powerful and general *exception* mechanism to indicate special conditions to the loop command. You will learn more about other exceptions in Section 3.4, and how to create your own exceptions and even your own loop commands in Chapter 7. These and other capabilities allow you to design programming interfaces that look and behave just like the built-in commands, creating 'little languages' that are specialised to a particular problem domain or application. Such *domain-specific languages*, or DSLs, are a powerful way of structuring complex software, and Tcl excels at them. We'll encounter more examples of this approach in later sections, and show how you can build your own.

3.2.4 The `foreach` Loop

The **`foreach`** loop at first glance appears to be much more limited than the other loops we have discussed. It's purpose is to simplify iterating over the elements of a list (Section 2.3) in order:

foreach *varName list body*

However, iterating over the elements of a list is such a common and useful operation that we can really view the **`foreach`** command as being a general loop. Indeed, many operations that you might use a **`for`** loop for can often be expressed quite succinctly using a **`foreach`** loop. Let's take a look at **`foreach`** in action:

```
puts {The authors of "Practical Programming in Tcl/Tk" are:}
foreach x {"Brent Welch" "Ken Jones" "Jeff Hobbs"} {
    puts $x
}
```

The **`foreach`** loop is actually much more flexible than the simple version we've just presented. It can iterate over multiple lists simultaneously, and it can also extract more than one element from a list at a time. The full syntax is as follows:

foreach *varlist1 list1 ?varlist2 list2 . . . ? body*

Here's an example using these advanced features:

```
set people {
    John    24
    Mary    39
    Simon   33
}
set salaries {
    10000
    20000
    30000
}
foreach {name age} $people salary $salaries {
    puts "$name ($age) earns \$$salary"
}
```

3.3 Procedures

In Tcl there is actually no distinction between commands (often known as 'functions' in other languages), and 'syntax'. There are no reserved words (like `if` or `while`) as exist in C, Java, Python, Perl, etc.. When the Tcl interpreter starts up there is a list of known commands that the interpreter uses to parse a line. These commands include **`while`**, **`for`**, **`set`**, **`puts`**, and so on. They are, however, still just regular Tcl commands that obey the same syntax rules as all Tcl commands, both built-in, and those that you create yourself. New commands can be created in a Tcl script using the **`proc`** command, short for 'procedure'. The syntax of this command is as follows:

proc *name params body*

When the **`proc`** command is evaluated, it creates a new command with the name `name` that takes the arguments given in `params`. When the procedure is called, by invoking the command `name`, it assigns any arguments it is given to the variables named in the `params` and then executes the `body` script, returning the result of the last command as its result. Here is an example that returns the sum of two numbers:

```
proc sum {x y} {
    expr {$x + $y}
}
puts "9 + 13 = [sum 9 13]"
```

Procedures can be used to package up parts of a script into a new command. This can be used to clearly structure your code into separate functional areas, and also to allow reuse, as the same procedure can be called many times with different arguments from different parts of your program. For instance, imagine we were designing a program to fetch stock quotes from the internet, process them in some manner, and then display them. We could write a single long script that performed each step in turn. However, this would be difficult to read and understand. By using procedures we can break the task up into independent chunks that can be understood in isolation. Our program would then look something like the following:

```
# Define the parts of our program
proc FetchQuotes {} { ... }
proc Process {quotes} { ... }
proc Display {quotes} { ...}
# Now use them:
set quotes [FetchQuotes]
set processed [Process $quotes]
Display $processed
# Alternatively:
Display [Process [FetchQuotes]]
```

The only constraint on using procedures is that you *must* define the procedure before you use it. Some programming languages allow functions to be defined in a later part of the program to where they are used. Tcl forbids this as the **`proc`** command needs to have been evaluated to create the procedure before it can be used. One way to get around this is to put the main part of your code into a procedure too, and then call it at the end:

```
proc Main {} { Display [Process [FetchQuotes]] }
proc FetchQuotes {} { ... }
proc Process {quotes} { ... }
proc Display {quotes} { ... }
# Now call our main procedure
Main
```

The reason this works is that Tcl doesn't look inside a procedure body until it is actually called, so the `Display` and so on procedures are all created by the time the `Main` procedure is called.

Tip: As well as making your code easier to read and more structured, procedures in Tcl also offer a further advantage: speed. The current implementation of Tcl includes a *byte-code compiler* in addition to the standard interpreter. This byte-code compiler can convert Tcl scripts from simple strings of source-code into a more efficient byte-code format. This byte-code can then be very efficiently interpreted, resulting in a considerable speed-up over normal scripts. The use of the byte-code compiler is entirely hidden to the author of a script: it is just something that the Tcl interpreter does behind the scenes to improve performance. Traditionally, only code contained in procedures was sent to the byte-code compiler as this was code that was likely to be executed multiple times. In more recent Tcl versions, other code, such as loop bodies, may also be byte-compiled, but it is good style to put most of your code into procedures, especially if you are targetting older Tcl versions. The technically-minded can find more details of the byte-code compiler in [8], which also discusses other important elements of the current interpreter design.

You can actually define procedures with the same name as existing Tcl commands. In this case, Tcl will silently replace the existing command with the new version. This can be useful in certain advanced situations (such as when building a debugger), but can also cause unexpected behaviour. In general you should avoid overwriting standard Tcl commands, or commands from any packages that you use. One simple way to ensure this is to adopt a consistent naming convention for your procedures that is different from the standard naming convention. For example, you could choose to start all of your procedure names with a capital letter, or better yet, use a namespace (Section 4.1).

3.3.1 Parameters

In addition to just specifying a name for each parameter, Tcl also allows some parameters to be given a *default value*. This is done by specifying the parameter as a list of two elements: the name and the default value:

```
proc sum {x {y 1}} { expr {$x + $y} }
```

The following `sum` command can be called with either one or two arguments. If only one argument is given then the `y` variable will be given the default value of 1, much like in the built-in **`incr`** command. Default parameters should only appear after ordinary parameters to avoid confusion as to which argument will be assigned to which parameter.

You can also define commands which take an arbitrary number of arguments. This is done by using the special **`args`** parameter name. When this name is used as the name of the last parameter in the parameter list then Tcl will treat that parameter differently. Instead of assigning it a single argument value, it will instead assign it a *list* of any further arguments that are given. To illustrate:

```
% proc test {a b args} { puts "a = $a, b = $b, args = {$args}" }
% test 1 2
a = 1, b = 2, args = {}
% test 1 2 3
a = 1, b = 2, args = {3}
% test 1 2 3 4 5 6
a = 1, b = 2, args = {3 4 5 6}
```

The `args` parameter can only be the last parameter and it cannot have a default value. The name is not special anywhere else in the parameter list (it will be treated as just an ordinary parameter elsewhere).

3.3.2 Global and Local Variables

As mentioned in the previous section, when a procedure is called, Tcl first creates new variables with the names given in the procedure's *parameter list* and assigns them the values of any arguments passed, in order. If the number of arguments doesn't match the number of parameters then an error will be produced:

```
% proc sum {x y} { expr {$x + $y} }
% sum 1
wrong # args: should be "sum x y"
```

Unlike the loop variables used in the **for** and **while** commands, variables created inside procedures—either as parameters, or by using the **set** command—do *not* continue to exist after the command exits:

```
% sum 1 2
3
% set x
can't read "x": no such variable
```

In fact, when a procedure is called, Tcl creates a new variable *scope* that is distinct from the variable scope that we have previously been using. This new variable scope contains only the variables that were created for the parameters. Note that in Tcl only variables defined within a procedure, or explicitly imported (as described later), are visible to the body of the procedure. This is in contrast to some other languages in which variables defined outside of a procedure are also visible. If you wish to make use of a variable defined outside of a procedure, you can *import* it using the **global** command[4]:

```
proc foo {} {
    global x
    puts "x = $x"
    incr x
}
set x 12
foo
puts "Now, x = $x" ;# displays "Now, x = 13"
```

Another way of referring to global variables within a procedure is by using a *fully-qualified variable name*. This is done by prefixing the variable name with two colon characters, `::`. This tells Tcl that the variable being referred to is a global variable:

```
proc foo {} {
  puts "x = $::x"
  incr ::x
}
```

In both cases, the variable being referred to doesn't have to exist when it is referred to: you can also create global variables from within a procedure, using the usual **set** command:

```
proc initialise {} {
    global name lang
    set name "John Ousterhout"
    set lang "Tcl"
}
initialise
puts "$name invented $lang!"
```

[4]Variables defined at the top level of a script are known as *global variables*, and the top level itself is known as the *global scope*.

The **info vars** command, introduced in Section 1.3 can be used to get a list of all the variables that are visible in a procedure, either *local* variables (that have been defined in the procedure), or those that have been imported. You can list just the local variables with the **info locals** command, and you can list all global variables with the **info globals** command.

3.3.3 The `return` Command

A procedure usually returns the value of the last command that executed in that procedure. You can force a procedure to return a specific value by using the **return** command. In its most basic form, this command takes a single value argument. When executed it will cause the current procedure to exit, returning the value as its result:

```
proc test a {
    if {$a > 10} { return "big" }
    return "small"
}
```

The **return** command can actually do much more than simply returning a result, as it is the basis of Tcl's exception mechanism. See Section 3.4.3 for details.

3.3.4 The Procedure Call Stack

Each time you call a procedure, a new variable scope is created, and when that procedure returns, the scope is destroyed, destroying any variables that were created inside it. Two calls to the same procedure will result in two separate variable scopes being created. If the body of one procedure calls another procedure then another new variable scope is created and pushed on top of the previous one. When that procedure returns its scope is destroyed and the previous scope on this *stack* is reinstated. As with global variables, you can also import variables that have been defined in other procedures that are on the stack: a form of explicit *dynamic scoping*. To do this, you use the **upvar** command that creates a new local variable that is linked to a local variable defined in another procedure scope. The syntax of the command is:

upvar *?level? otherVar localVar ?otherVar localVar ...?*

The first argument specifies the *level* of the target procedure in the current procedure stack. Each level (or *frame*) in the stack is numbered: the global scope is level `#0`, the first procedure called is `#1`, and the procedure it called is `#2`, and so on. You can also specify the level relative to the current level: the currently executing scope is then level `0` (without the `#`), the procedure that called it is level `1`, and so on. Consider the following program:

```
proc foo {} { puts "Here!" }
proc bar {} { foo }
proc jim {} { bar }
jim
```

At the point at which the 'Here!' message is displayed, there will be four frames on the stack, labelled as follows:

Procedure	Absolute	Relative
`foo`	#3	0
`bar`	#2	1
`jim`	#1	2
global	#0	3

Therefore, in order to import a variable into the `foo` procedure from it's caller (in this case, `bar`), we could use either **upvar** `#2` or **upvar** `1`. Note, though, that we usually cannot be guaranteed that `bar` will be the only procedure that will call `foo`, so it is generally bad style to make assumptions about what variables will be available in a particular scope. In fact, most Tcl

code only makes use of **upvar** in a few very limited situations. The most important of these is to mimick the pass-by-reference style of calling available in some other languages. In this style, we explicitly pass a reference to a variable in one scope as an argument to another procedure which then manipulates that variable using **upvar**. For instance, we can write our own version of the **incr** command that doubles the variable as follows:

```
proc double varName {
    upvar 1 $varName n
    set n [expr {$n * 2}]
}
set x 12
double x
puts "x = $x" ;# displays "x = 24"
```

The **upvar** command is almost always used with a level of 0, 1 or #0 in real code. If you find yourself using other levels frequently then it may indicate a design problem, and your code may be difficult to read later on. Generally, good Tcl style is to avoid the use of global variables as much as possible.

As with **global**, the **upvar** command can also be used to create variables. The **info level** command can be used to examine the procedure call stack at runtime.

3.3.5 Recursion: Looping without loops!

Note: The next few sections cover some relatively advanced material. You may wish to skip ahead to Section 3.4 on a first reading, or if you are short of time.

As well as calling other commands, a procedure may also call itself, either directly or indirectly (via another procedure). Such circular calls are known as *recursion*, and can be used to implement lots of interesting control logic. For instance, we can implement loops without using any of the loop commands by just using procedure calls:

```
proc count {i end} {
    if {$i > $end} { return }
    puts "i = $i"
    count [expr {$i + 1}]
}
count 1 10
```

This example prints the numbers from 1 to 10, just like our earlier examples with **for** and **while**, but without using an explicit loop. While recursion can be difficult to grasp, especially for problems that have a simple solution using an iterative **for** loop, many problems can often be concisely expressed using recursive procedures, including many mathematical *inductive* definitions. For instance, given the following inductive definition of the factorial function ($n!$), we can easily define a recursive implementation:

$$n! = \begin{cases} 1 & n \leq 1 \\ n(n-1)! & n > 1 \end{cases}$$

```
proc fac n {
    if {$n <= 1} { return 1 }
    if {$n >  1} { expr {$n * [fac [incr n -1]]} }
}
```

In the case of mathematical functions, we can make this look a bit nicer by making the procedure a function as described in Section 1.5.3:

```
proc tcl::mathfunc::fac n { ::fac $n }
proc fac n {
    if {$n <= 1} { return 1}
    if {$n >  1} { expr {$n * fac($n-1)} }
}
```

Generally, recursive definitions will run slower and use more memory than their iterative counterparts. This is due to the construction and deletion of the stack frames that are needed for each recursive call. From Tcl 8.6, you can use the new **`tailcall`** command to convert a recursive definition into an iterative version almost automatically. The **`tailcall`** command effectively turns a procedure call into a simple jump: instead of creating a new stack frame, the tailcall command simply reuses the current one for the new procedure. This eliminates the extra memory required for recursive calls as only a single stack frame is required, but it can only appear as the very last operation in a procedure (it is similar to the **`return`** command in this regard). This means that our original definition cannot be simply reused, as the recursive call to `fac` is not the final operation: its result is then used in an expression. Instead, we must add a new *accumulator parameter* to transform the procedure into *tail-recursive form*:

```
proc fac {n {accum 1}} {
    if {$n <= 1} { return $accum }
    if {$n >  1} { tailcall fac [expr {$n-1}] [expr {$n*$accum}] }
}
```

The main advantage of the **`tailcall`** command is that it enables recursive definitions to run in constant memory space, and to avoid running out of stack memory (which can happen if you try to create too many stack frames). Try both recursive definitions of `fac` with an argument of 1000: the first will error, whereas the second will produce the correct result.

While the **`tailcall`** command solves the problem of memory usage for recursive functions, procedures using it will still run more slowly than their iterative counterparts. Most recursive procedures can be naturally rewritten into an iterative form, using **`for`** or **`while`**. For example:

```
proc fac n {
    for {set accum 1} {$n > 1} {incr n -1} {
        set accum [expr {$accum*$n}]
    }
    return $accum
}
```

This version also uses constant space, but runs much quicker than the recursive definitions. Notice how this version also uses an accumulator parameter, but here it is used as the loop variable, rather than as an extra argument.

3.3.6 Higher-Order Procedures and Callbacks

All of the procedures we have seen so far have taken simple string values as arguments. As Tcl procedures are commands, and commands have string names, we can also pass the name of a procedure as an argument to a command. Commands in Tcl are therefore *first-class*: i.e., they can be treated just like other values in the language, with the caveat that you are always passing the *name* of a command, rather than the command itself. This means that we can write procedures that take other procedures and commands as arguments. Such procedures are known as *higher-order* procedures in Computer Science.[5] We have already seen some examples of 'higher-order' commands, such as the **`lsort`** command, which can take a `-command` option. For instance, we can sort a list of strings by size using the following approach:

[5]The origin of this term is in mathematical logic. You might like to think of higher-order commands as being a bit like adverbs in natural languages such as English.

```
proc comp-len {a b} { expr {[string length $a] - [string length $b]} }
lsort -command comp-len $strings
```

Here the `comp-len` ('compare length') procedure is being passed as an argument to the **`lsort`** command. Many other Tcl commands also work in this manner. The `comp-len` argument is sometimes referred to as a 'callback command', or just 'callback', as a frequent use of such arguments is to allow a command to 'call back' or notify the application when something interesting happens.

As well as supplying some examples of higher-order commands, Tcl also makes it very simple to write your own. Such procedures are very useful for packaging up commonly-used patterns of code ('design patterns') into reuseable commands. For example, when processing lists a common operation is to iterate through each element of the list applying some function and building up a new list as we go. For instance, we may want to extract the names from a list of people (where each person is represented as a dictionary):

```
set people [list {name Neil age 28} {name Mary age 37}]
proc name {record} { dict get $record name }
foreach person $people { lappend names [name $person] }
puts $names ;# displays "Neil Mary"
```

We can use the same pattern to get the ages, or addresses, and so on. The code is mostly identical, with just a different operation being applied to each element to extract the appropriate field. We can package up this pattern of code into a reuseable procedure, **`map`**[6]:

```
proc map {f xs} {
    set result [list]
    foreach x $xs { lappend result [{*}$f $x] }
    return $result
}
```

This procedure takes a command `f` and a list `xs` and then applies the command to each element of the list, returning a list of the results. The `{*}` operator is used to expand the command before we call it, so that we can pass in commands such as **`string length`** and Tcl will know to look for the `length` subcommand of the **`string`** command, rather than for a (non-existent) `'string length'` command. We can use this to implement our original loop in very little code:

```
set names [map name $people]
```

If we expand the definition of `map` by replacing the arguments with their values, we can see that this performs exactly the same loop as before:

```
# Expanded definition of map: $f = "name"
set result [list]
foreach x $people { lappend result [name $x] }
```

The advantage of this style of coding is that we can then re-use this `map` procedure for selecting other elements, or indeed for completely unrelated tasks:

```
proc age person { dict get $person age }
set ages [map age $people]
proc plus {a b} { expr {$a + $b} }
set future-ages [map {plus 10} $ages]
```

Other common higher-order operations on lists include filtering a list to select just those that satisfy some predicate condition:

[6] Named after the same function in functional programming languages.

```
proc filter {p xs} {
    set result [list]
    foreach x $xs { if {[{*}$p $x]} { lappend result $x } }
    return $result
}
proc less-than {amount x} { expr {$x < $amount} }
filter {less-than 30} $ages
```

A particularly useful operation is known as a 'fold' or 'reduce' operation. This operation takes a list of data and *folds* an operator between each pair of elements:

$$\text{fold } + \ \{1\,2\,3\,4\,5\} \Rightarrow 1+2+3+4+5 = 15$$

As well as specifying the operator to apply, we can also specify a 'zero' or 'identity' value that is used when the list is empty. We name the procedure `foldl` to indicate that this version is 'left-associative', i.e., that it calculates the result from left-to-right $(((1+2)+3)+\ldots)$:

```
proc foldl {f z xs} {
    foreach x $xs { set z [{*}$f $z $x] }
    return $z
}
```

We can use this for instance to provide general `sum` and `product` operations on lists of integers:

```
proc + {a b} { expr {$a + $b} }
proc * {a b} { expr {$a * $b} }
proc sum xs { foldl + 0 $xs }
proc product xs { foldl * 1 $xs }
sum {1 2 3 4 5} ;# results in 15
```

A `foldr` command can be defined similarly, and is a useful exercise (hint: you may wish to use a **`for`** loop instead of **`foreach`**). We can even use this definition to write our factorial example! First, we define a helper procedure for generating all natural numbers from 1 up to a given limit (giving it a cute name):

```
proc 1.. n {
    set range [list]
    for {set i 1} {$i <= $n} {incr i} { lappend range $i }
    return $range
}
```

We can then simply define the factorial function as the product of all integers from $1..n$:

```
proc fac n { product [1.. $n] }
```

This definition is as efficient as the iterative version, as it reduces to just a pair of iterative loops, yet it is clearer even than the original recursive definition.[7] The main remaining inefficiency is that it must generate the complete list of integers up to n before computing the result. We can eliminate this inefficiency by creating a specialised version of `fold` especially for operating on integer ranges, $n..m$[8]:

```
proc fold-range {f z n m} {
    for {set i $n} {$i <= $m} {incr i} {
        set z [{*}$f $z $i]
```

[7] See http://www.willamette.edu/~fruehr/haskell/evolution.html for a humourous look at the many ways of defining the factorial function.

[8] Another way of removing this inefficiency would be to use a *lazy list* or *stream* that only computes its elements as they are needed. Such an extension is left as an exercise for the curious reader.

```
    }
    return $z
}
proc fac n { fold-range * 1 1 $n }
```

3.3.7 Anonymous Procedures

Since Tcl version 8.5

In addition to creating named procedures, Tcl also allows you to create un-named *anonymous procedures* (sometimes referred to as 'lambdas' after the anonymous functions of the Lambda Calculus). An anonymous procedure in Tcl is simply a list consisting of two elements[9]: a parameter list and a procedure body. These take exactly the same form as they do for normal procedures. As an anonymous procedure has no command name, you cannot call it directly. Instead, it can be called with the **apply** command:

apply *anonymousProc ?arg ...?*

For example, to define an anonymous `plus` command that sums two numbers, we can use:

```
set plus {{x y} { expr {$x + $y} }}
apply $plus 3 4 ;# returns 7
```

Many standard Tcl commands that take callbacks will also accept anonymous procedures. The reason for this is that these commands actually accept *command prefixes*—a list consisting of a command name followed by some initial arguments. Any further arguments are then appended to this command prefix before it is called as a command (recall that a command in Tcl is a list of words). We can pass an anonymous procedure to these commands by making a list consisting of the command name **apply** followed by our anonymous procedure. For example, we can sort a list by string length in a single command using an anonymous procedure:

```
lsort -command {apply {{a b} {
    expr {[string length $a] - [string length $b]}
}}} $strings
```

Such in-line anonymous procedures can be useful for short procedures, as they keep the logic of the callback in the same place in the source-code as it is used. For longer procedures, it is still generally better to use a named procedure, so long as you choose a descriptive name. We can make the use of the anonymous procedure easier to read by using a *constructor function*. This is simply a procedure that constructs some data (in this case, another procedure) and returns it. We will call our anonymous procedure constructor 'lambda' in honour of similar constructs in other languages:

```
proc lambda {params body} { list apply [list $params $body] }
lsort -command [lambda {a b} {
    expr {[string length $a] - [string length $b]}
}] $strings
```

It is generally good style to always use a constructor function for complex data like anonymous procedures. You can even define specialised constructors that use different behaviour. For instance, here is a version that uses the **expr** command to evaluate the body of the procedure (see if you can work out how it works):

```
proc func {params body} { list apply [list $params [list expr $body]] }
set plus [func {x y} {$x + $y}]
```

[9] A third element is also permitted: a namespace. See Section 4.1.

The higher-order procedures given in Section 3.3.6 will all work with anonymous procedures, due to the use of the `{*}` operator:

```
proc fac n { fold-range [func {x y} {$x * $y}] 1 1 $n }
set future_ages [map [func x { $x + 10 }] $ages]
```

3.3.8 Command Aliases

Tcl also provides a way to convert a command prefix (i.e., a command name plus a list of initial arguments) into a new named command, using the **`interp alias`** command. In basic usage, the syntax is as follows:

interp alias {} *newCommand* {} *oldCommand ?arg ...?*

The `{}` arguments can be ignored for now—they are explained in Section 7.4. This command creates a new command called `newCommand`, which, when called, will immediately call `oldCommand` passing any initial arguments defined in the alias, followed by any other arguments passed in this call. For example, we can define a `sumRange` command that sums all of the integers in a given range, using our previous `fold-range` command, as follows:

```
proc plus {x y} { expr {$x + $y} }
interp alias {} sumRange {} fold-range plus 0
sumRange 1 5 ;# returns 15
```

When this command is called, the alias is expanded as follows:

```
sumRange 1 5 => fold-range plus 0 1 5 => 15
```

Command aliases can be used to convert anonymous procedures back into named procedures:

```
interp alias {} add {} apply {{x y} { expr {$x + $y} }}
```

Indeed, with these components we can even define our own version of the **`proc`** command!

```
proc myproc {name params body} {
    interp alias {} $name {} apply [list $params $body]
}
myproc add {x y} { expr {$x + $y} }
```

This new `myproc` command acts virtually identically to the built-in **`proc`** command. This should give you some indication of the flexibility of Tcl, and we have just started to scratch the surface!

Tip: Command names, parameter names and argument values in Tcl are just ordinary strings, just like everything else. This can be used to good effect to create natural looking 'syntax' for commands, by just using interesting choices for various names. For instance, we can wrap up the **`interp alias`** command to remove those ugly braces and to make our source-code look more elegant and readable, simply by defining a little wrapper procedure:

```
proc define {name = cmd args} {
    interp alias {} $name {} $cmd {*}$args
}
proc + {x y} { expr {$x + $y} }
define sumRange = foldl + 0
```

The use of `+` and `=` as names here is not special: to Tcl they are just more strings. However, by carefully choosing symbols which are widely recognised, we can make our definition of `sumRange` more readable. In fact, Tcl already defines command versions of all the **`expr`** operators in the `tcl::mathop` namespace.

3.3.9 Examining Procedures

As for variables, Tcl provides methods for listing the procedures (and other commands) that are defined in the interpreter, through the **`info`** command. The **`info commands`** and **`info procs`** commands can be used to get a list of all commands (including procedures), and just procedures, respectively:

info commands *?pattern?*

info procs *?pattern?*

Beyond just listing the names of defined procedures, you can also examine their definitions, using the **`info args`**, **`info default`**, and **`info body`** commands, which return the argument (parameter) list, the default values of any arguments, and the body of a procedure, respectively:

info args *procName*

info default *procName param varName*

info body *procName*

The **`info default`** command doesn't directly return the default value, but instead returns 1 or 0 to indicate whether the argument has a default value, and then sets the given variable *varName* to the default value, if it exists. This allows you to determine whether a default value has been specified, even if it is the empty string (`""`). To illustrate how to use these commands, here is a small procedure that can recreate the source-code of any procedure defined in an interpreter (inspired by the functionality in the `tkcon` application):

```
# dump -- dumps the source code of a procedure
proc dump procName {
    set body   [info body $procName]
    set params [list]
    foreach param [info args $procName] {
        if {[info default $procName $param default]} {
            lappend params [list $param $default]
        } else {
            lappend params [list $param]
        }
    }
    return [list proc $procName $params $body]
}
```

Entering this into an interactive interpreter session and then running it on itself (i.e., `dump dump`) results in the same source code being displayed. Such a procedure is a useful tool when developing procedures interactively and debugging.

3.4 Dealing with Errors

Up until now, the Tcl programs we have shown have all assumed that every command succeeds without problems. In the real world, however, things are rarely that simple: a file doesn't exist, or a network connection cannot be made, or an input that should have been a number turns out to be a name. These are just some examples of the sorts of errors that can occur in everyday programs. Whenever Tcl encounters an error it reacts by generating an error condition, or *exception*. If any procedures are currently executing, then by default, Tcl will immediately cause them to terminate one-by-one, unwinding the call stack until the top level is reached. At this point an error message is printed. If Tcl is running in interactive mode, you are then presented with the familiar prompt

and can continue entering commands. If Tcl was processing a script file, then the interpreter will exit and additionally print a *stack trace* showing the procedures that were active at the time of the call. For instance:

```
proc a {} { b }
proc b {} { c }
proc c {} { d }
proc d {} { some_command }
a
```

When executed, this script will produce the following error:

```
invalid command name "some_command"
    while executing
"some_command "
    (procedure "d" line 1)
    invoked from within
"d "
    (procedure "c" line 1)
    invoked from within
"c "
    (procedure "b" line 1)
    invoked from within
"b "
    (procedure "a" line 1)
    invoked from within
"a"
```

This stack trace is built up as the call stack is unwound, and is additionally recorded in the `errorInfo` global variable. You can print the contents of this variable from the Tcl interactive prompt to help with debugging errors. In addition to this stack trace, a Tcl error also produces two additional pieces of information:

- A simple string message that describes the error in human-readable terms.
- An 'error code', which is a list designed to be easily processed by a Tcl script (an *exception handler*). For example, a divide-by-zero error in **`expr`** results in an error code of {`ARITH DIVZERO {divide by zero}`}. The first element gives the general class of error (ARITHmetic), and subsequent elements give more specific details. The error code of an error is available in the `errorCode` global variable.

3.4.1 The `try` Command

Since Tcl version 8.6

Tcl also allows a script to intercept error exceptions before they completely unwind the call stack. The programmer can then decide what to do about the error and act accordingly, instead of having the entire application come to a halt. Dealing with errors appropriately is an important part of writing robust software. In fact, Tcl offers two commands for handling errors: the newer **`try`** command, and an older **`catch`** command. The **`try`** command is simpler to use for most cases, whereas the **`catch`** command can provide more flexibility in certain circumstances. Most new code should use the newer **`try`** command, unless you have very complex requirements or need to support Tcl versions prior to 8.6.

The syntax of the **`try`** command is as follows:

try *body ?handler . . . ? ?***finally** *script?*

The first argument to the command is a normal Tcl script. The script is first evaluated just like a normal script. If the script completes normally, then the **try** command returns its result. If an error or other exception occurs, however, then the **try** command behaves differently. Instead of unwinding the call stack all the way back to the top level, Tcl instead unwinds the stack only until the **try** command. At that point the error is matched against any specified handlers, one at a time. An error handler is specified with the following syntax:

trap *pattern varList script*

The arguments are as follows:

pattern: A pattern used to match against the error code of the error. The pattern can specify the full error code, or any prefix of it. For instance, we can trap all arithmetic errors by using the pattern `ARITH`. An empty list (`{}`) will match all errors.

varList: A list of variables to contain information about the error. The list takes the following format: `?msgVar? ?detailVar?`. The first element, if specified, names a variable into which to copy the human-readable error message of the error. The second element names a variable into which to copy the additional detail of the error, in the form of a *dictionary* value (see Section 2.4), the elements of which are shown below.

script: A script to run if an error matches this pattern. The script will be run in the same variable scope that the **try** command was originally called from. Any variables defined in that scope, including those specified in the `varList` are visible. Note that at this point the main `body` script of the **try** command has exited, along with any procedures it may have called.

More than one **trap** clause may be specified in a single **try** command. When an error is detected, each clause will be tried in the order they are specified until one matches the error code. If no clauses match then the error is propagated as if it had never been caught. If more than one clause matches the error, then only the first in the list will be executed.

Once an error has been trapped, the associated script can perform actions to either recover from the error, or to simply record it and carry on. Typical actions might include logging the error and then ignoring it, or returning a default value. The result of the handler script becomes the new result of the **try** command. Any errors that occur within the error handler script itself are treated as normal errors: in particular, they are not handled by the error handlers in that **try** command, but they may be caught by other **try** commands either within the error handler, or defined in scopes that are still on the call stack.

To illustrate how to use the **try** command, here is a function that performs integer division on two numbers, x and y, i.e., x/y. In the case when y is zero, this would result in a divide-by-zero error. In this example, we convert such errors into a default value of 0:

```
proc div {x y} {
    try {
        expr {$x/$y}
    } trap {ARITH DIVZERO} {} {
        return 0
    }
}
```

Of course, we could have achieved the same thing in this case by testing the value of y before performing the division, but in many cases errors are not so simple to prevent.

Error Options Dictionary

If both variable names are given in a **trap** clause of the **try** command, then the second will be filled with a special value called the *error options dictionary*. This is a dictionary value consisting

of a series of attributes and associated values that give extra information about the error that occurred. The keys available in the options dictionary are as follows:

-errorcode: Contains the full error code of the error, as also found in the `errorCode` global variable.

-errorinfo: Contains the stack trace of the error up to this point, as found in the `errorInfo` global variable.

-errorline: An integer indicating which line of the original script passed to the **try** command was executing when the error occurred.

Two other keys are also available: `-code` and `-level`. These will be discussed later, in Section 3.4.1. The values of each of these attributes can be extracted using the **dict get** command, described in Section 2.4. Here is an example procedure that shows some information about any errors that occur within a script:

```
proc show-error script {
    try $script trap {} {msg opts} {
        puts "Error: $msg"
        puts " Code: [dict get $opts -errorcode]"
        puts " Line: [dict get $opts -errorline]"
        puts "Stack:"
        puts [dict get $opts -errorinfo]
    }
}
```

For example, here is the output from a script that produces a divide-by-zero error:

```
% show-error { expr {1/0} }
Error: divide by zero
 Code: ARITH DIVZERO {divide by zero}
 Line: 1
Stack:
divide by zero
    invoked from within
"expr {1/0} "
    ("try" body line 1)
```

The `finally` Clause

In addition to catching errors, the **try** command also allows you to specify a script that will be run regardless of if any errors occur or not. This is the purpose of the **finally** clause. The finally clause can be used to clean up any resources that were acquired prior to the **try** command being executed. The script supplied to a finally clause will be run after the main body script, and after any error handlers have run. It will always be run, even if an error handler itself produces an error. This is very useful for guaranteeing that certain actions always get performed no matter what happens (short of the Tcl interpreter terminating). For example, we can write a procedure to read and return the contents of a file, and guarantee that if the file is opened at all, it will always be closed before the procedure returns[10]:

```
proc readfile file {
    set in [open $file]
    try { read $in } finally { close $in }
}
```

[10]The commands for reading the file are described in Chapter 5.

Code	Exception
0	OK
1	Error
2	Return
3	Break
4	Continue
...	User-defined

Table 3.1: Tcl exception codes

The finally clause in this example ensures that the file is always closed after its contents have been read, regardless of any errors that may occur in the process. These errors are still passed on to the caller of the `readfile` command, but they do not prevent the file being closed.

Tip: It is important to note that a **`finally`** clause is most useful for cleaning up resources that were acquired *before* the **`try`** command was executed. While it may be tempting to also move the acquiring of the resources into the **`try`** command (to handle any errors), this is not correct in most cases. The reason for this is that if an error occurs in acquiring the resource then it does not need to be cleaned up. In this case, the **`finally`** clause will probably produce an error, masking the original error. In such cases, Tcl will add a `-during` option to the error options dictionary for the new error that contains the options dictionary of the original error. To properly handle errors occurring when acquiring resources, you should use two separate **`try`** commands.

Other Exceptions

As well as handling errors, the **`try`** command can also catch other types of exceptions. In Section 3.2.3 we explained that the **`break`** and **`continue`** commands were also types of exceptions. The **`return`** command of Section 3.3.3 is another type of exception! The **`try`** command can handle all of these types of exceptions, and any more that might be defined. This is done using a different clause in the list of exception handlers:

on *code varList script*

The `varList` and `script` arguments to this clause are exactly as they were for the **`trap`** clause. The 'message' variable will contain the result value in the case of a `return` exception, or an empty string in most other cases. The options dictionary for a non-error exception typically contains just the `-code` and `-level` options. The `code` argument specifies an *exception code* rather than the error code pattern of a trap. In Tcl, each different type of exception is assigned a unique integer code, listed in Table 3.1: errors have code 1, the **`return`** command has code 2, **`break`** has code 3, and **`continue`** has code 4. You can even define your own exception types and assign them their own codes, but this is not frequently done. The **`on`** clause also accepts the following literal words for the corresponding standard exception codes: **`error`**, **`return`**, **`break`**, and **`continue`**. You can even catch the 'normal' (i.e., non-exceptional) condition, in which the script given to **`try`** completes without any exceptions, using the code 0 or the keyword **`ok`**.

Catching non-error exceptions is mostly useful in the context of defining custom control structures, such as new loops. These uses are discussed in Section 7.1.

3.4.2 The `catch` Command

In older versions of Tcl, the only way to catch errors and exceptions was with the simple **`catch`** command. This command will catch all exceptions (including normal/ok completion) that are

raised in the script it is given, returning the code of the exception and optionally setting variables to contain the message/result and options dictionary of the exception:

catch *script ?resultVar? ?optionsVar?*

The main disadvantage of **catch** when compared to **try** is that it catches absolutely everything, and the programmer then has to perform extra steps to discover if this is an exception that can be dealt with. The **try** command, on the other hand, allows the programmer to specify exactly which errors and exceptions can be handled, and all others will be left untouched. Consider the following version of the `div` command, reimplemented using **catch**:

```
proc div {x y} {
    set code [catch { expr {$x / $y} } result options]
    if {$code == 1} {
        # An error occurred
        set errorcode [dict get $options -errorcode]
        if {[string match {ARITH DIVZERO *} $errorcode]} { return 0 }
    }
    return -options $options $result
}
```

This example is harder to understand than the equivalent with **try**.

3.4.3 Throwing Errors and Exceptions

Tcl scripts can throw their own exceptions using one of three separate commands: **throw**, **error**, and **return**.

The `throw` Command

Since Tcl version 8.6

The **throw** command takes the following form:

throw *errorcode message*

When evaluated, this command generates a new error with the given error-code and message. For instance, to generate an error identical to **expr**'s divide-by-zero error, you can use:

```
throw {ARITH DIVZERO {divide by zero}} "divide by zero"
```

The `error` Command

The **error** command is similar to the **throw** command:

error *message ?errorinfo? ?errorcode?*

The difference between the two commands is in the order of arguments and that the **error** command also allows specifying an initial stack trace to fill in the errorinfo field in the options dictionary. The reason for the new **throw** command is to try to encourage the use of the error-code field, which has traditionally not been well supported by Tcl code. This limits the usefulness of the **trap** clause in the **try** command. New code should use the **throw** command for errors where possible, and try to specify meaningful error-codes. In older versions of Tcl the **throw** command can be emulated using **error**:

```
proc throw {code message} { error $message "" $code }
```

Throwing Other Exceptions

The most general method of throwing an exception is to use the **return** command, which is the Swiss Army Knife of exception generation. In addition to simply returning a result, **return** can also generate any exception, and specify values for the options dictionary. The full syntax of **return** is as follows:

return *?-option value . . . ? ?result?*

The options that can be specified are as follows:

-code: Specifies an exception code for a new exception to be generated. Can be one of: **ok**, **return**, **error**, **break**, **continue** or an integer code.

-errorcode: Specifies the error code for a new exception.

-errorinfo: Specifies the stack trace for a new exception.

-level: Specifies how many levels of the call stack to unwind before generating the exception. This defaults to 1, which means that the current procedure scope will be executed before the exception is generated. A value of 2 would cause the caller of this procedure to also exit, and so on.

-options: Allows specifying the complete options dictionary for the new exception.

Note that `return -code error` and `error` are not the same, as the former causes the current procedure to exit before the error is generated, whereas the latter immediately generates the error. The latter behaviour can be achieved by specifying a `-level 0` argument to the return, or by wrapping the return in a separate procedure:

```
proc myError msg { return -code error $msg }
proc myThrow {code msg} {
    return -code error -errorcode $code $msg
}
```

The **return** command is also useful in error handlers when you wish to re-throw an exception that could not be handled:

```
try {
    ...
} trap SOME_ERROR {msg opts} {
    ...
    return -options $opts $msg
}
```

Chapter 4

Program Organisation

4.1 Namespaces

4.1.1 Abstract Data Types

4.1.2 Example: Stacks and Queues

4.1.3 Example: Algebraic Types

4.2 Packages

4.3 Object Oriented Tcl

Chapter 5

Input and Output

Chapter 6

The Event Loop

Chapter 7

Advanced Topics

7.1 Custom Control Structures

eval and uplevel.

7.2 Unknown Command Handlers

7.3 Coroutines

7.4 Interpreters

7.5 Threads

Part II

Tk: Graphical User Interfaces

Chapter 8

Basic Concepts

8.1 Introduction

Tk (short for “toolkit”) is the most widely used GUI toolkit for Tcl. There are several others, like Gnocl, based on Gtk, but in this tutorial we will only look at Tk.

Here is a simple program to illustrate the style of programming:

```
#
# For generality: load Tk
#

package require Tk

#
# Define the callback procedure used by the pushbutton
#
proc handleMsg {} {
    tk_messageBox -title Message -message $::msg -type ok
    set ::msg ""
}

#
# Create the widgets
#

label  .label -text "Enter text:"
entry  .entry -textvariable msg
button .button -text Run -command handleMsg

#
# Make the widgets visible
#

grid .label  .entry -sticky news
grid .button -

#
# We want to be able to resize the entry ...
#
grid columnconfigure . 1 -weight 1
```

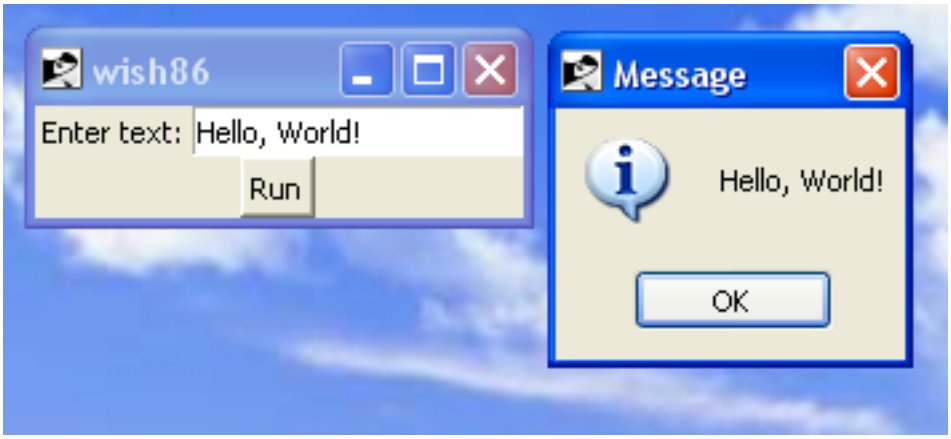


Figure 8.1: Example Tk GUI.

This program, when run in a Tk-enabled shell like `wish`, will show the window shown in Figure 8.1 (with details depending on the operating system and windowing environment).

In fact, this program will run in the command-line shell tclsh as well, thanks to the **`package require`** command.

You can fill in text in the text entry box at the right. If you press the pushbutton, a message appears and the entry box is cleared. You can resize it and only the entry box will become wider or narrower.

With just a few lines of code we have created a program that has a surprising amount of functionality. Even though it is small and does not do much of any use, the code does show many aspects you will find in actual GUI programs:

- Creating the widgets (the elements of your GUI) is separated from arranging them in a window: the label command create a label widget (just some text that does not interact with the user), but you put it into its parent window using a *geometry manager*, in this case the **`grid`** geometry manager.
- Widgets are represented by commands whose names start with a dot (.) Warning: after the dot you must have a lower-case letter. This has to do with the handling of display options.
- Widgets often take options that connect variables to them, like the `-textvariable` option for the **`entry`** widget. Such variables should persist outside any procedure, so they are global by default, but you can also specify namespace variables. If you change the value of the variable, see the `handleMsg` procedure, then this is immediately shown in the connected widget.
- Some types of widgets take a `-command` option that tells Tk to run a particular script when the user activates the widget. For the button widget that is: the user clicks in it with the mouse pointer. Since these scripts are not invoked in the context of any procedure, they run in the global namespace. It is good practice to use separate procedures (handleMsg for instance) instead of a complete script.
- The geometry manager controls the placement of the widgets but also the resizing — if any. The grid geometry manager arranges widgets in rows and columns. More on geometry management can be found in Chapter 11.

Here are a few more aspects not shown in this simple example, but they will be discussed later on:

- Widgets can be arranged in a hierarchy to make the geometrical management easier

- You can create new independent windows (toplevel windows) that can be filled with their own widgets
- Tk relies on options for the type of widget and on options that are set per widget. This gives you the freedom to change the appearance significantly without altering the code.
- Besides the 'classical' widgets presented in the example, Tk as of version 8.5 also has so-called themed widgets. They are discussed in Chapter 12.

One very important aspect that is very much hidden in the wish shell is this: when the shell reaches the end of the script, it does not stop, but instead starts an event loop. Only when the event loop is running is the GUI alive: then it will display the windows, respond to actions from the user and so on.

As Tk is at least as dynamic as Tcl itself, you can easily experiment with widgets in an interactive shell. Just start `wish` and try commands like:

```
text .t
pack .t
```

These two commands bring up what is essentially a complete text editor!

Chapter 9

Managing toplevel windows

At start-up Tk creates a window named '.'. This is the most important window when working with Tk: closing it — either by the user or programmatically — means the program stops. It is also most often used to hold the main part of the application's graphical interface.

You can create any number of new windows via the **toplevel** command:

```
toplevel .draw
wm title .draw Drawing
```

creates a new toplevel window with the title 'Drawing' and an associated command `.draw`. If you want to add a widget to this window, for instance a canvas widget (a widget suitable for displaying graphs and images and so on), include this name in the name of the new widget:

```
canvas .draw.canvas
pack .draw.canvas -fill both ;# Make it visible
```

As you can imagine, the widget names/commands can get pretty long. Storing the name in a variable makes it easier to set up or reorganise the GUI:

```
set c [canvas .draw.canvas]
pack $c -fill both ;# Make it visible
```

Toplevel windows can be resized, destroyed, brought to the front, minimized and so on programmatically with various commands. The command to do some of these things is **wm**, as these actions require interaction with the windowing environment, the *window manager*.

- Bring the window to the front (this may or may not work directly, as the window manager may intervene):

  ```
  raise $window
  ```

- Bring it to the background:

  ```
  lower $window
  ```

- Destroy the window and all the widgets it contains:

  ```
  destroy $window
  ```

- Get the size and position of the window:

  ```
  set geometry [wm geometry $window]
  ```

This command returns a string like: `200x200+132+174` — width and height and the coordinates of the upper left corner

- Put the window in the lower right corner:

```
wm geometry $window -0-0
```

- Iconfiy the window and later show it again (after 2 seconds):

```
wm iconify $window
after 2000 {wm deiconify $window}
```

An important interaction with the window manager is to make sure that if the main window (.) is about to be closed, your program knows about it and can ask the user for confirmation. If you do not do that, the program will simply stop and the user may lose all his or her data. Here is the command to take care of that:

```
wm protocol . WM_DELETE_WINDOW {okayToExit}
```

In the procedure `okayToExit` you can ask the user for confirmation, save all the data your program is dealing with and then destroy the window. This command works on any toplevel window, not just the main window. The `WM_DELETE_WINDOW` signal is sent by the window manager *before* the window is destroyed, allowing your application a chance to prevent it. If you just want to execute some action when a window is actually destroyed (e.g., to clean up some associated resources), then you can bind to the `<Destroy>` event on the window. This is called *after* the window is destroyed[1]:

```
bind $window <Destroy> [list myCleanupProc $window]
```

The **bind** manual page [1] lists a number of other useful events that Tk will generate when a toplevel window changes state (e.g., visibility, size, etc.).

A common functionality is the so-called splash screen: a window without decoration that the user can not move around and that displays a logo or a short message and then disappears. This is used to show the user the application is starting up. The code fragment that follows is a trivial example:

```
toplevel .splash
wm overrideredirect .splash 1 ;# Make sure there is no title etc.

label .splash.msg -text "Tk tutorial\nversion 1.0" -font "Times 16"
grid  .splash.msg             ;# Make it visible

after 5000 {
    destroy .splash
}
```

After 5 seconds the splash screen is removed and it is assumed that the application has been initialised.

To be really useful, we will need to centre the window. So try this:

```
toplevel .splash -bg green    ;# Make it stick out (on most screens)

wm overrideredirect .splash 1 ;# Make sure there is no title etc.

set screenwidth  [winfo screenwidth  .]
```

[1] Technically, the window still exists, but most operations on it will fail at this point.

```
set screenheight [winfo screenheight .]

set width        400
set height       300

wm minsize .splash $width $height  ;# Make sure it has the right size
wm maxsize .splash $width $height

set xsplash      [expr {($screenwidth  - $width)  / 2}]
set ysplash      [expr {($screenheight - $height) / 2}]

wm geometry      .splash +$xsplash+$ysplash

label .splash.msg -text "Tk tutorial\nversion 1.0" -font "Times 16" \

.splash.msg configure -bg blue ;# We want contrast!

grid  .splash.msg              ;# Make it visible

after 5000 {
    destroy .splash
}
```

In the process we have used the **winfo** command to get information about the screen the window is displayed on: its size in pixels. Also we have made sure, via **wm minsize** and **wm maxsize** that we know in advance how large the window will be: Tk windows generally adjust to the size of their contents or to the size the user requests.

If you want a window with a size the user can not change, use:

```
wm resizable $window 0 0
```

(the two zeros indicate: not resizable in horizontal or vertical direction)

The key commands to manipulate the behaviour of the window or finding out all manner of information are **wm** and **winfo**.

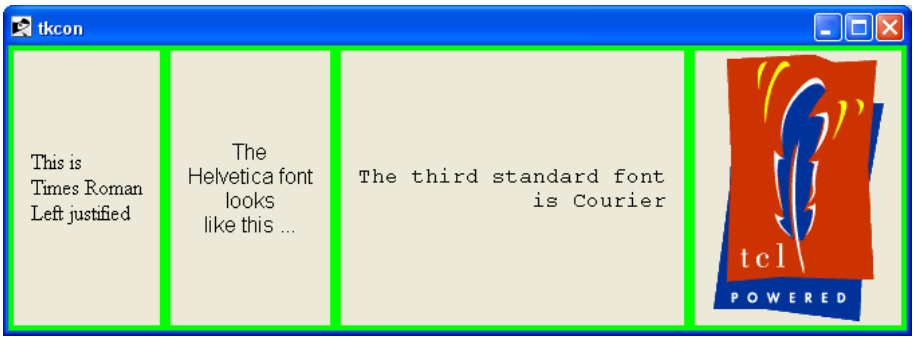
Chapter 10

Basic Widgets

Some of the widgets available in Tk are very simple: labels, entry widgets and various types of buttons. They are quite common in user-interfaces — they can be regarded as basic building blocks.

10.1 Labels

Label widgets are intended to display (static) text or images. They do not allow interaction with the user, but they have quite a few possibilities. To illustrate, see the screenshot below:



The code to create this window with its labels in various fonts and an image is this[1]:

```
#
# Show some possibilities of labels
#

. configure -bg green

label .label1 -text "This is\nTimes Roman\nLeft justified" \
        -font "Times" -justify left
label .label2 -text "The\nHelvetica font\nlooks\nlike this ..." \
        -font "Helvetica" -justify center
label .label3 -text "The third standard font\nis Courier" \
        -font "Courier" -justify right

set labelImage [image create photo -file pwrdLogo200.gif]
```

[1] The 'Tcl-Powered!' logo is available from http://www.demailly.com/tcl/about/logos.html.

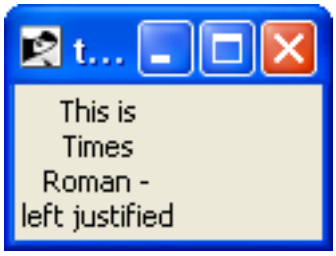
```
label .label4 -image $labelImage

grid  .label1 .label2 .label3 .label4 -padx 4 -pady 4 -ipadx 10 -sticky news
```

As you can see, labels can display long text spread out over a number of lines. The `-justify` option determines how the lines are justified with respect to each other. Using the `-wraplength` option you can let the widget decide the wrapping for itself:

```
# Wraplength is in pixels ...
label .label -text "This is Times Roman - left justified" -wraplength 60
grid  .label
```

gives:



Besides static text, the label widget also accepts the name of a (global or namespace) variable:

```
#
# Use a fixed-width font - otherwise the clock time seems to pulse
#
label .time -textvariable time -font Courier
grid  .time

proc showTime {colon} {

    #
    # time must be a _global_ or a _namespace_ variable
    #
    global time

    if { $colon } {
        set time [clock format [clock seconds] -format "%H:%M"]
        after 1000 showTime 0
    } else {
        set time [clock format [clock seconds] -format "%H %M"]
        after 1000 showTime 1
    }
}

showTime 1
```

presents a simple clock:

10.2 Entry widgets

10.2.1 Validation

10.3 Buttons

10.4 Checkbuttons

10.5 Radio buttons

Radio buttons are a bit special in the sense that a number of widgets work on the *same* variable:

```
set ::fruit none
radiobutton .r1 -variable fruit -value melon  -text "Melon"
radiobutton .r2 -variable fruit -value apple  -text "Apple"
radiobutton .r3 -variable fruit -value pear   -text "Pear"
radiobutton .r4 -variable fruit -value banana -text "Banana"

label .choice -text "Chosen:"
label .chosen -textvariable fruit

grid .r1 -sticky w
grid .r2 -sticky w
grid .r3 -sticky w
grid .r4 -sticky w
grid .choice .chosen -sticky w
```

which results in this window:

Each radiobutton is associated with the variable `fruit` and each has a string value that is assigned to that variable when the button is checked. Only one button at a time is set, but if the variable has a value that is not among the values of the buttons, none is set.

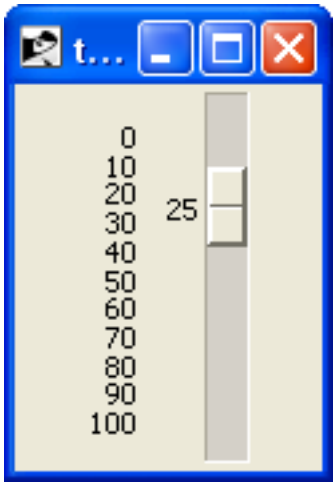
You can set the radiobuttons programmatically by setting the variable it is associated with. The buttons are automatically updated.

10.6 Scales

To select a value from a range you can use the scale widget. For instance:

```
scale .scale -from 0 -to 100 -tickinterval 10 -variable percentage
grid  .scale
```

gives the following window:



You can set the position of the slider programmatically by setting the value of the associated variable. You can also specify a command to be run whenever the value is changed, for instance to zoom in or out in a canvas widget:

```
scale .scale -from -5 -to 5 -tickinterval 1 -command {zoom}

canvas .c -width 200 -height 200

grid   .c .scale -sticky news

.c create polygon {100 100 100 110 110 110} -fill red

set prevFactor 1.0
proc zoom {value}
    if { abs($value) < 0.01 } {
        set value 0.01 ;# Avoid division by zero!
    }
    set factor [expr {$value/$::prevFactor}]
    .c scale all 100 100 $factor $factor
    set ::prevFactor $value
}
```

10.7 Scrollbars

Scrollbars are designed to cooperate with other widgets, such as the text widget and the listbox. They come in two flavours: horizontal and vertical scrollbars. Here is how you use them:

```
text .text -xscrollcommand {.hscroll set} \
           -yscrollcommand {.vscroll set} \
```

```
            -width 20 -height 10 \
            -wrap none
scrollbar .hscroll -orient horizontal -command {.text xview}
scrollbar .vscroll -orient vertical   -command {.text yview}

grid .text .vscroll -sticky news
grid .hscroll       -sticky news
grid rowconfigure    . 0 -weight 1
grid columnconfigure . 0 -weight 1

#
# Fill the text widget with some text ...
#
set string ""
for { set i 0 } { $i < 20 } { incr i } {
    append string " $i"
    .text insert end "$string\n"
}
```

This cooperation is completely automatic: both widgets know how to deal with positioning the "thumb" and how to react to the manipulation.

If you want to create a scrollable widget yourself, say to scroll a very large canvas, then you need to implement the scrolling protocol yourself. The chapter on the canvas widget contains an example.

10.8 Spinboxes

10.9 Comboboxes

10.10 Progressbars

10.11 Separators

Chapter 11

Geometry Management

11.1 Placing the widgets

As indicated before widgets only become visible when they are placed within their parent window via a *geometry manager*. Tk has three different geometry managers: **grid**, **pack** and **place**.

The simplest of these is **place**. It is also the least used, as you are yourself responsible for determining the coordinates of your widget. But in rare cases it is useful, for instance if you want to create a "tooltip" — a window with explanatory text that appears and disappears near a pushbutton or another widget. Here is an example, with the result shown in Figure 11.1:

```
button .help -text Help
bind .help <Enter> {showHelpTip %x %y}
bind .help <Leave> {removeHelpTip}

proc showHelpTip {xcoord ycoord} {
    #
    # Show the tooltip after half a second
    #
    set ::after [after 500 [list displayHelpTip $xcoord $ycoord]]
}
proc removeHelpTip {} {
    if { [winfo exists .helpTip] } {
        destroy .helpTip
    }
    after cancel $::after ;# We do not want to have the tooltip appear
}
proc displayHelpTip {xcoord ycoord} {
    label .helpTip -text "This button brings up\nthe online help" -bg green
    place .helpTip -x $xcoord -y $ycoord
    after 2000 removeHelpTip
}
grid .help
```

Note that we use the **bind** command to make Tk perform an action when the mouse enters the area occupied by the Help button (the `Enter` event) and when it leaves that area again. (You will probably want to enlarge the window when running this example — otherwise the message remains hidden.)

The **pack** geometry manager puts widgets inside their parent either to the one of the side of that parent or to other widgets already placed there:

```
label  .label -text "Some text:"
```

Figure 11.1: An example 'tooltip' for a button.

```
entry  .entry -textvariable X
button .ok    -text OK

pack .label .entry .ok -side left -fill x
```

will put the label on the left side of the window, then the entry widget on the left side of the label and finally the pushbutton on the left side of the entry widget. If there is not enough space **pack** willl increase the size of the widnow. Due to the way this geometry manager works, it can be tough to achieve the layout you want. For instance: try to put the pushbutton below the other two with code like this:

```
pack .label .entry -side left -fill x
pack .ok -side top
```

This brings us to the most versatile geometry manager: **grid**. As suggested by its name, it uses a grid of rows and columns to place the widgets. Here is a simple example:

```
text .text -xscrollcommand {.hscroll set} \
           -yscrollcommand {.vscroll set} \
           -width 20 -height 10 \
           -wrap none
scrollbar .hscroll -orient horizontal -command {.text xview}
scrollbar .vscroll -orient vertical   -command {.text yview}

grid .text .vscroll -sticky news
grid .hscroll       -sticky news
grid rowconfigure    . 0 -weight 1
grid columnconfigure . 0 -weight 1
```

This layout has two rows and two columns:

- The first grid command defines that there are two widgets on the first row
- The second grid command defines that there is only a single widget — the horizontal scrollbar. The second cell is left empty.
- By carefully setting the weight for the rows and columns we achieve that the text widget can grow in both directions and the scrollbars only in the direction of their long axes.

We can vary this design by using '–' and '^' to indicate that the widget to the left (–) or above (^) extends into the next cell:

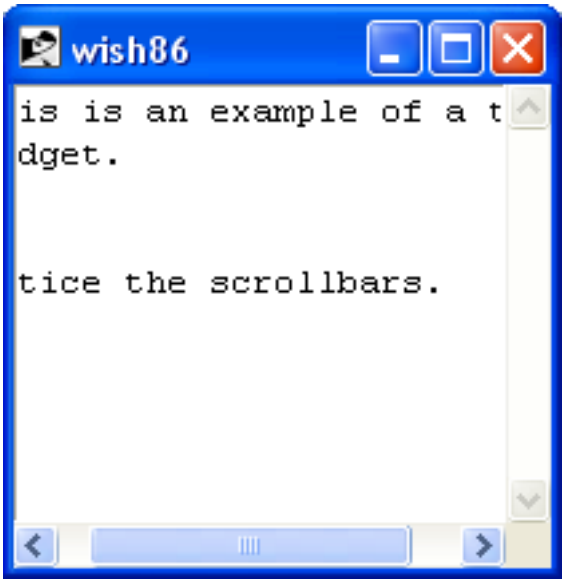


Figure 11.2: The **grid** geometry manager.

```
grid .text    .vscroll -sticky news
grid .hscroll ^        -sticky news
```

gives this layout (the vertical scrollbar occupies the lower right corner too):
/screenshot/
and

```
grid .text    .vscroll -sticky news
grid .hscroll -        -sticky news
```

gives this layout (the horizontal scrollbar is elongated):
/screenshot/
We can also vary the amount of space between widgets by using the `-padx` and `-pady` options:

```
grid .text    .vscroll -sticky news -padx 20 -pady 20
grid .hscroll          -sticky news -padx 20 -pady 20
```

creates this window:
/screenshot/
The `-sticky` option lets us position and size the widgets within the cell they occupy. Leaving them out like this:

```
grid .text    .vscroll
grid .hscroll
```

gives:
/screenshot/
The scrollbars get a default length, as they are not in any geometrical way coupled to the text widget.
This is not very attractive for scrollbars, but it is for pushbuttons:

```
label  .label -text "The computation has finised"
button .ok -text "OK"
grid .label -sticky news
grid .ok
```

/screenshot/

The string “news” means ‘north-east-west-south’. Sometimes it is useful to stick the widgets to one side only:

```
label  .label1 -text "List of options"
radiobutton .r1 -text "Ice cream with chocolate" -variable dessert -value ice
radiobutton .r2 -text "Cheese" -variable dessert -value cheese

grid   .label1
grid   .r1
grid   .r2

label  .label2 -text "List of options"
radiobutton .r3 -text "Ice cream with chocolate" -variable dessert2 -value ice
radiobutton .r4 -text "Cheese" -variable dessert2 -value cheese

grid   .label2 -sticky w
grid   .r3     -sticky w
grid   .r4     -sticky w

set dessert  ice
set dessert2 ice
```

/screenshot/

If you want to vary the spacing, for instance indent the radio buttons, you can use the -ipadx and -ipady options, these reserve space within the cells, instead of between rows and columns:

```
label  .label -text "List of options"
radiobutton .r1 -text "Ice cream with chocolate" -variable dessert -value ice
radiobutton .r2 -text "Cheese" -variable dessert -value cheese

grid   .label -sticky w
grid   .r1     -sticky w -ipadx 10
grid   .r2     -sticky w -ipadx 10

set dessert ice
```

/screenshot/

Chapter 12

Themed Tk

Part III

Real-World Application Development

Chapter 13

Tcl Database Connectivity (TDBC)

Chapter 14

XML

Chapter 15

Web Applications and Services

Bibliography

[1] *bind(n)—Tk Built-In Commands*. http://www.tcl.tk/man/tcl8.6/TkCmd/bind.htm.

[2] *clock(n)—Tcl Built-In Commands*. http://www.tcl.tk/man/tcl8.6/TclCmd/clock.htm.

[3] Richard Suchenwirth et al. Custom sorting, 2008. http://wiki.tcl.tk/4021.

[4] Clif Flynt. *Tcl/Tk: A Developer's Guide*. Morgan Kauffman, May 2003.

[5] Clif Flynt, Neil Madden, Arjen Markus, and David N. Welton. The Tcl tutorial, 2009. http://www.tcl.tk/man/tcl8.5/tutorial/tcltutorial.html.

[6] *format(n)—Tcl Built-In Commands*. http://www.tcl.tk/man/tcl8.6/TclCmd/format.htm.

[7] Jeffrey E. F. Friedl. *Mastering Regular Expressions*. O'Reilly, third edition, August 2006.

[8] Brian T. Lewis. An on-the-fly bytecode compiler for Tcl. In *Proceedings of 4th annual Tcl/Tk Workshop*, pages 103–114. USENIX, 1996. http://www.usenix.org/publications/library/proceedings/tcl96/full_papers/lewis/index.html.

[9] *scan(n)—Tcl Built-In Commands*. http://www.tcl.tk/man/tcl8.6/TclCmd/scan.htm.

[10] *switch(n)—Tcl Built-In Commands*. http://www.tcl.tk/man/tcl8.6/TclCmd/switch.htm.

[11] *Tcl—Tool Command Language*. http://www.tcl.tk/man/tcl8.6/TclCmd/Tcl.htm.

[12] Brent B. Welch, Ken Jones, and Jeffrey Hobbs. *Practical Programming in Tcl and Tk*. Prentice Hall, 4th edition, June 2003.

[13] Wikipedia. Big O notation. http://en.wikipedia.org/wiki/Big_O_notation.

Index