**Fall 2004 Handout 8**

# Tcl

Mark Meretzky
*New York University School of Continuing and Professional Studies*

**mark.meretzky@nyu.edu**
**http://i5.nyu.edu/~mm64**

## Introduction

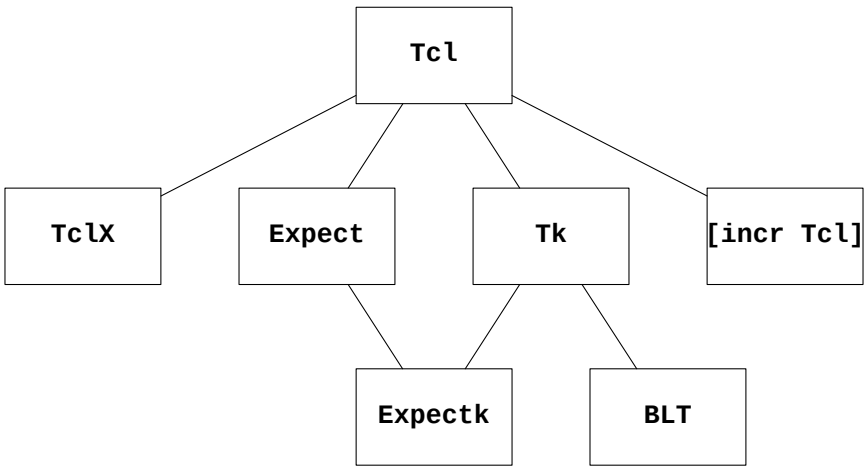
The ‘‘Tool Command Language’’ Tcl is a generic (i.e., plain vanilla) interpreted language, with variables, assignment statements, loops, conditionals, procedures, etc. What sets it apart is that it is *extensible* and *embeddable* .

Tcl can be extended with new commands and even new control structures by writing C functions to implement them. The two most popular extended forms of Tcl are the language Tk, for building graphical user interfaces (GUI’s) on top of the X window system, and the language Expect, for writing scripts to drive interactive programs such as **`ftp`** or **`telnet`**.

Tcl and its extensions can also be embedded in an application to endow the application with its own command language. For example, instead of using **`yacc`** to create new ad hoc languages to tell a debugger or a mail reader how to loop and search, the language Tcl can be built into both applications. A common language will also make it easier for applications to communicate with each other.

This paper accompanies a tutorial course to let programmers use Tcl, Tk, and Expect without reading through the 500-page books in the bibliography. Although the programming facilities of Tcl are blandly generic, the Tcl interpreter provides them in a nonstandard way. We therefore concentrate on the unusual features of Tcl: the order of parsing and substitution, normal vs. deferred substitution, repeated parsing with the **`eval`** command, and list composition.

## The most popular extended forms of Tcl



**Tcl, Tk, and Expect bibliography and links**

The best tutorial for beginners is

*Tcl and the Tk Toolkit*
by John K. Ousterhout; Addison-Wesley, 1994; ISBN 0-201-63337-X
**http://www.awl.com/cseng/titles/0-201-63337-X**
**http://www.scriptics.com/people/john.ousterhout/**

But Tcl has grown since the Osterhout book was published. An up-to-date handbook of commands and their options is

*Tcl/Tk in a Nutshell*
by Paul Raines and Jeff Tranter; O'Reilly & Associates, 1999;
ISBN 1-56592-433-9
**http://www.oreilly.com/catalog/tclnut/**

Advanced examples are shown in

*Effective Tcl/Tk Programming: Writing Better Programs with Tcl and Tk*
by Mark Harrison and Michael McLennan; Addison-Wesley, 1998; ISBN 0-201-63474-0
**http://www.awl.com/cseng/titles/0-201-63474-0**
**http://www.tcltk.com/itcl/mmc/index.html**

The largest number of examples are in

*Practical Programming in Tcl and Tk, 2nd ed.*
by Brent B. Welch; Prentice Hall, 1997; ISBN 0-13-616830-2
**http://www.prenhall.com/allbooks/ptr_0136168302.html**
**http://www.beedub.com**

| | |
|---|---|
| Tcl/Tk home page: | **http://www.scriptics.com/** |
| Tcl/Tk web browser plugin: | **http://www.scriptics.com/plugin/** |
| Tcl/Tk newsgroup: | **news:comp.lang.tcl** |
| Perl/Tk FAQ: | **http://w4.lns.cornell.edu/~pvhp/ptk/ptkFAQ.html** |

The standard book on Expect is

*Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs*
by Don Libes; O'Reilly & Associates, 1994; ISBN 1-56592-090-2
**http://www.oreilly.com/catalog/expect/**
**http://www.mel.nist.gov/msidstaff/libes/**

The Expect home page is **http://expect.nist.gov/**

Other extended forms of Tcl are

| | |
|---|---|
| BLT: | **http://www.tcltk.com/blt** |
| [incr Tcl]: | **http://www.tcltk.com/itcl** |
| TclX: | **http://www.neosoft.com/tcl/TclX.html** |

To print the documentation,

| | |
|---|---|
| **1$ man -t tclsh \| grops \| lpr** | *Tcl shell* |
| **2$ man -t wish \| grops \| lpr** | *Tk shell ("window shell")* |
| **3$ man -t tk \| grops \| lpr** | *Tk colormodels* |

**Tcl commands and comments**

See **execve**(2) for the Unix **#!**. Like the **puts** function in C, the **puts** command in Tcl outputs a string followed by a newline.

The end of the line terminates each command in a Tcl program. A semicolon can also terminate a command, so you can write two commands on the same line.

The **#** is a comment delimiter only when it is the first character of a command. To write a comment alongside a command, you must therefore terminate the command with a semicolon. Write the semicolon immediately before the comment delimiter.

A long command may be continued onto the next line with a backslash. Start a new line between words, not in the middle of a word.

```
#!/usr/local/bin/tclsh

puts hello
puts hello; puts goodbye

#This is a comment.
puts ##################    ;#Output a row of #'s.

puts \
    supercalifragilisticexpialidocious

exit 0                      ;#Return exit status; zero is the default.
```

```
hello
hello
goodbye
##################
supercalifragilisticexpialidocious
```

**Tcl variables**

Tcl has only one data type: string. A value that looks like a number is really only a string of one or more digits, and maybe a minus sign and/or decimal point. It is an error to attempt to use an uninitialized variable, or one which has been **unset**.

Tcl and the shell share the following two rules for dollar signs in front of variables. Don't put a **$** in front of a variable name when you're giving it a new value. Put the **$** there only when you're using the variable's existing value:

```
#!/usr/local/bin/tclsh

set x capital        ;#This is an assignment.
puts $x
append x ism
puts $x

set y 10
puts $y
incr y 1             ;#1 is the default; -1 will decrement
puts $y

puts y               ;#What happens if you forget the dollar sign?

unset x y            ;#Unnecessary in this program.
exit 0
```

```
capital
capitalism
10
11
y
```

**File identifiers**

A **puts** command must have one or two arguments. If there is only one argument, it is written to the standard output. If there are two arguments, the second one is written to the destination specified by the first, which must be an output *file identifier*: a one-word string specifying the destination of the output. A file identifier is also used to specify a source of input. Every Tcl program is born with the right to use three file identifiers: **stdin**, **stdout**, and **stderr**. Additional file identifiers may be created with the **open** command, below.

```
puts hello
puts stdout hello          ;#does the same thing
puts stderr hello
```

**Make several words count as a single word**

The following commands are therefore illegal:

```
#Don't worry: comma and exclamation point are not special characters in Tcl.

puts Hello there, world!   ;#too many arguments
puts                       ;#too few arguments: attempt to output an empty line
puts Hello, world!         ;#Hello, is not a file identifier
```

To make two or more words count as a single word enclose them in **"**double quotes**"** or **{**curly braces**}**. The following are now legal because **puts** has one argument:

```
puts "Hello there, world!"
puts ""                    ;#output an empty line
puts "Hello, world!"       ;#to stdout by default
```

You also need double quotes or curly braces around an argument that begins or ends with a white space:

```
puts "   This line is indented three spaces."
```

**Double quotes vs. curly braces**

| *Shell* | *Tcl* |
|---|---|
| **$***variable* | **$***variable* |
| **"***double quotes***"** | **"***double quotes***"** |
| **'***single quotes***'** | **{***curly braces***}** |
| **`***back quotes***`** | **[***square brackets***]** |

(1) Double quotes in Tcl are like double quotes in the shell: you can use all of the substitution characters **$ \ [ ]** inside of them. Note that ***** is not a special character in Tcl.

```
#!/bin/sh

w=world                 #no space around the equal sign
echo "Hello, $w!"       #quotes unnecessary
echo "Hello, \$w!"

exit 0
```

```
Hello, world!
Hello, $w!
```

```
#!/usr/local/bin/tclsh

set w world

puts "Hello, $w!"       ;#quotes necessary
puts "Hello, \$w!"

exit 0
```

```
Hello, world!
Hello, $w!
```

```
#!/usr/local/bin/tclsh

puts "Hello there,\nworld!"
exit 0
```

```
Hello there,
world!
```

The function of double quotes is merely to group two or more words into a single word. They are therefore never necessary around one word, but we often write them anyway to remind the reader which words are conventional strings. For example, both of the arguments of the following **`puts`** command are strings *everything* is a string in Tcl), but only the **`hello`** would be a string in a normal programming language:

```
puts stderr "hello"          ;#The double quotes are just documentation.
```

(2) Curly braces in Tcl are like single quotes in the shell: you can't use any of the special characters **`$ \ [ ] ;`** inside of them:

```
#!/bin/sh

w=world
echo 'Hello, $w!'

exit 0
```

```
Hello, $w!
```

```
#!/usr/local/bin/tclsh

set w world
puts {Hello, $w!}

exit 0
```

```
Hello, $w!
```

```
#!/usr/local/bin/tclsh

puts {Hello there,\nworld!}
exit 0
```

```
Hello there,\nworld!
```

Curly braces not only group words together: they also turn off the meanings of the special characters they enclose. Curly braces are never necessary around a single word containing no special characters, but we often write them anyway to remind the reader which words are commands:

```
#The double quotes and curly braces are just documentation.
button .exitbutton -text "exit" -command {exit}
```

(3) Double quotes and curly braces can therefore be used interchangeably if they enclose no special characters. For example, the empty string can be written as either **""** or **{}**, but please write it as **""** because that's what human beings expect.

```
#!/usr/local/bin/tclsh
puts "Here are {curly braces} within a double-quoted string."
puts {Here are "double quotes" within a curly braced string.}

exit 0
```

```
Here are {curly braces} within a double-quoted string.
Here are "double quotes" within a curly braced string.
```

**The set command with two arguments**

A **set** command that gives a new value to a variable must have exactly two arguments. If the new value is more (or less) than a single word, it must be enclosed in double quotes or curly braces:

```
set name "John Doe"       ;#could have used braces, but quotes are clearer
set name "$first $last"   ;#had to use quotes, not braces
set name ""               ;#could have used braces, but quotes are clearer
```

**Run a command within a command**

‘Back quotes‘ in the shell language let you use the standard output of a program as part of a larger command line:

```
#!/bin/sh
#Just to remind you of back quotes in the shell.

a=9
b=16
c=‘expr $a + $b‘
echo The sum is $c.

#Of course, if all you want to do is print the sum,
#there is no need to deposit it into a variable:

echo The sum is ‘expr $a + $b‘.
exit 0
```

```
The sum is 25.
The sum is 25.
```

Every Tcl command produces a *result* string. **[**Square brackets**]** in Tcl let you use the result of a command as part of a larger command. Some commands return the null string as their result, but the **expr** command returns a number:

```
#!/usr/local/bin/tclsh
#The same program, in Tcl.

set a 9
set b 16
set c [expr $a + $b]
puts "The sum is $c."

#Once again, if all you want to do is print the sum,
#there is no need to deposit it into a variable:

puts "The sum is [expr $a + $b]."
exit 0
```

```
The sum is 25.
The sum is 25.
```

The Tcl **expr** command is different from (and is not implemented by calling) the Unix **expr** program. For example, Tcl **expr** does floating point arithmetic, while Unix **expr** does only integer arithmetic. And Tcl **expr** does not require that each token of the expression be a separate argument.

The shell lets you use back quotes within double quotes but not within single quotes:

```
#!/bin/sh

a=9
b=16

echo "The sum is ‘expr $a + $b‘."
echo ’The sum is ‘expr $a + $b‘.’

exit 0
```

```
The sum is 25.
The sum is ‘expr $a + $b‘.
```

Similarly, Tcl lets you use square brackets within double quotes but not within curly braces:

```
#!/usr/local/bin/tclsh

set a 9
set b 16

puts "The sum is [expr $a + $b]."
puts {The sum is [expr $a + $b].}

exit 0
```

```
The sum is 25.
The sum is [expr $a + $b].
```

**Square brackets around a command with no arguments**

The **-nonewline** argument of the **puts** command is just like the **-n** option of the Unix **echo** program:

```
#!/usr/local/bin/tclsh

puts -nonewline "My process ID number is [pid] "
puts "and my current directory is [pwd]."

exit 0
```

```
My process ID number is 20412 and my current directory is /usr/bin.
```

**Another square bracket example**

```
 1 /* C fragment */
 2 #include <stdio.h>
 3
 4     int red = 0;
 5     int green = 127;
 6     int blue = 255;
 7     char rgb[256];
 8
 9     sprintf (rgb, "#%02X%02X%02X", red, green, blue);
10     puts (rgb);
```

**sprintf** deposits eight characters into memory, including a terminating **'\0'**:

```
#007FFF
```

The result of the following **format** command is the same as string written by the above **sprintf** function. The double quotes around the **#%02X%02X%02X** are unnecessary: even without them, the **#** would not be a comment delimiter because it is not at the start of a command. And **%** is not a special character in Tcl.

```
#!/usr/local/bin/tclsh

set red 0
set green 127
set blue 255

set rgb [format "#%02X%02X%02X" $red $green $blue]
puts $rgb

exit 0
```

```
#007FFF
```

**The set command with one argument**

With only one argument (the name of a variable), the **set** command does not change the value of the variable. It merely returns the value of the variable as its result. For example,

```
#!/usr/local/bin/tclsh

set a 10

puts $a
puts [set a]

exit 0
```

```
10
10
```

This lets you write a variable immediately followed by a character which could be part of a variable name:

```
#!/usr/local/bin/tclsh

set a 7
set aup 8

puts $aup            ;#unsuccessful attempt to print 7up
puts [set a]up
puts ${a}up          ;#simpler way to do the same thing

exit 0
```

```
8
7up
7up
```

The simpler way **${a}** can be used only with a scalar variable, not an array element. But the real use of the **set** command with one argument will be when running **tclsh** interactively.

**Why couldn't they leave the familiar shell punctuation marks alone?**

Why did they change the shell's single quotes and back quotes to Tcl curly braces and square brackets respectively? Let's consider the shell's back quotes as an example. To nest them, you need a backslash before each member of the inner pair:

```
#!/bin/sh

if [ ‘expr \‘wc -l < myfile\‘ / 60‘ -gt 100 ]
then
    echo At 60 lines per page, it would take more than 100 pages
    echo to print myfile.
fi

if [ ‘lpq -Pprinter1 | wc -l‘ -lt ‘lpq -Pprinter2 | wc -l‘ ]
then
    echo printer1 is less busy than printer2.
fi

exit 0
```

In the shell language, the above backslashes are necessary because the two back quotes of a pair are identical: there are no such things as a ‘‘left back quote’’ and a ‘‘right back quote’’. But if the two characters of the pair were different, we wouldn’t need the presence or absence of backslashes to tell which one pairs up with which one:

```
#!/usr/local/bin/tclsh

puts [pid]
puts [expr [pid] + 1]              ;#nested
puts "[pid] [pid]"                 ;#not nested

puts {Here are {curly braces} with a curly-braced string.}
exit 0
```

```
1000
1001
1000 1000
Here are {curly braces} with a curly-braced string.
```

Curly braces and square brackets are often heavily nested in Tcl, so it made sense to eliminate the need for the backslashes. Double quotes are not nested heavily in Tcl, so Ousterhout let them remain the same characters as in the shell language:

```
#!/usr/local/bin/tclsh

puts "Here are \"double quotes\" within a double-quoted string."
exit 0
```

```
Here are "double quotes" within a double-quoted string.
```

**The order in which everything happens**

As the Tcl interpreter **`tclsh`** reads each command, it does the three things:

(1) The interpreter divides each line into separate words (or *arguments*) wherever it sees white space. This is called *parsing*. Double quotes, curly braces, or square brackets will make two or more words count as a single word.

(2) The interpreter then performs three kinds of *substitutions* on each word that is not enclosed in curly braces:

(2a) The interpreter performs *variable substitution* when it changes the name of a variable (with a leading **$**) into the value of the variable. For example, the interpreter will cause the following **puts** command to receive the three-character string **100** rather than the two-character string **$x** as its argument:

```
set x 10
puts $x
```

(2b) The interpreter performs *backslash substitution* when it changes a backslash and the following character into a single (nonprinting) character. For example, the interpreter will cause the following **puts** command to receive a three-character string whose middle character is a newline, rather than the four-character string **a\nb** containing a backslash and lowercase **n**:

```
puts a\nb
```

(2c) The interpreter performs *command substitution* when it changes a command in **[**square brackets**]** into the result of the command. For example, the interpreter will cause the following **puts** command to receive a string such as **1000** rather than the five-character string **[pid]** as its argument:

```
puts [pid]
```

After all the substitutions (if any), the surrounding double quotes or curly braces (if any) are stripped off the word. Therefore all three of the following give exactly the same second argument to the **set** command:

```
set x A
set x "A"
set x {A}
```

(3) The interpreter then executes the command.

For example, in

```
set x {$a==3}
```

the curly braces prevent the interpreter from performing substitution on the **$a**: it will simply give the two arguments **x** and **$a==3** to the **set** command. The **set** command then copies the five characters **$a==3** into the variable **x**. Note that the **set** command is unaware of the curly braces around its argument: they're stripped away before it sees them.

**Normal vs. deferred substitution**

The following **while** command goes into an infinite loop, because it receives the argument **1<=10** from the interpreter. The interpreter changed the **$i** into a **1** before the **while** command gets to see its own arguments:

```
#!/usr/local/bin/tclsh

set i 1

while $i<=10 {puts $i; incr i}

exit 0
```

To prevent the interpreter from performing this substitution, enclose the first argument of the **while** command in curly braces. Now the command will receive the argument **$i<=10** from the interpreter.

The command will then perform an additional round of substitution on the first argument that it receives. This substitution is called *deferred substitution*. It happens *after* a command has begun to execute, and is performed (possibly many times) by the command. The normal kind of substitution happens *before* a command has begun to execute, and is performed (only once) by the interpreter.

Not every command performs deferred substitution on its arguments. Only three of the commands we have seen so far do this:

**while** *performs deferred substitution on its first argument, and maybe also on its second:* **while {$a==$b} {puts $c}**
**if** *performs deferred substitution on its first argument, and maybe also on its second:* **if {$a==$b} {exit $c}**
**expr** *performs deferred substitution on all its arguments:* **expr {$a + $b}**

It's too bad that the only way to tell if a command performs deferred substitution on any of its arguments is to look it up in the documentation.

```
#!/usr/local/bin/tclsh

set i 1

while {$i<=10} {puts $i; incr i}

exit 0
```

```
1
2
3
4
5
6
7
8
9
10
```

We could have right-justified the output numbers with

```
puts [format "%2d" $i]
```

Now that we have curly braces around the first argument of the **while** command, we can write white space for legibility there:

```
while {$i <= 10} {puts $i; incr i}
```

**Loop commands need more curly braces than if commands**

We don't care whether the following logical expression receives normal substitution by the interpreter or deferred substitution by the **if** command. Therefore curly braces are not required around the logical expression (provided that it contains no white space):

```
set i 1
if  $i<=10  {puts $i; incr i}       ;#if command receives the argument 1<=10
if {$i<=10} {puts $i; incr i}       ;#if command receives the argument $i<=10
```

But if the logical expression of a **while** command received normal substitution by the interpreter, the loop would loop forever. It must therefore receive deferred substitution by the **while** command itself, which is smart enough to perform deferred substitution over and over. Therefore the curly braces are required around the logical expression of w **while**, even if it contains no white space.

**It matters where you put the curly braces**

The **if** command can take two arguments:

```
if {$a == $b} {puts "equal"}
```

or four arguments:

```
if {$a == $b} {puts "equal"} else {puts "not equal"}
```

or seven arguments:

```
if {$a < $b} {puts "less than"} elseif {$a == $b} {puts "equal"} \
    else {puts "not equal"}
```

or ten arguments, etc.

| | |
|---|---|
| *C, C++, Java, JavaScript, awk, C shell* | **else if** |
| *C and C++ preprocessors* | **#elif** |
| *Bourne, Korn, and Bourne-Again shells* | **elif** |
| *Perl* | **elsif** |
| *Tcl* | **elseif** |
| *troff, nroff* | **.el .if** |
| *m4* | **.ifelse** |

There are three ways to show the interpreter that all the arguments belong to the same command:

(1) Write the entire command on one line:

```
if {$a == $b} {puts "equal"} else {puts "not equal"}
```

(2) Continue the command onto additional lines with backslashes:

```
if {$a == $b} \
    {puts "equal"} \
else \
    {puts "not equal"}
```

(3) A command that contains a **{**, **[**, or **"** automatically continues at least until the line that contains the corresponding **}**, **]**, or **"**:

```
if {$a == $b} {
    puts "equal"
} else {
    puts "not equal"
}
```

The following **if** command is wrong because it has only one argument:

```
if {$a == $b}
    {puts "equal"}
else
    {puts "not equal"}
```

The above **while** command would usually be written

```
#!/usr/local/bin/tclsh

set i 1

while {$i <= 10} {
    puts $i
    incr i
}

exit 0
```

**Perform substitution on the same argument twice**

For pedagogical purposes only, we can even apply both kinds of substitution to the same argument if we use **while**, **if**, **expr**, or **catch**, without curly braces. In the following **if** command, for example, the interpreter will perform substitution on the argument **$v2**, changing it to the three-character string **$v1**, and will then give this string **$v1** to the **expr** command. The **expr** command will then perform deferred substitution on the **$v1**, changing it to the two-character string **10**.

```
#!/usr/local/bin/tclsh

set v1 10
set v2 "\$v1"           ;#Put the three characters $v1 into v2.

puts $v2                ;#only the normal substitution
puts [expr $v2 + 1]     ;#normal followed by deferred substitution

exit 0
```

```
$v1
11
```

In real life, you will never put a dollar sign and the name of a variable into another variable. Therefore a single performance of substitution will suffice for the first argument of an **if** command. A second performance of substitution would be harmless but would have no effect.

**Curly braces vs. double quotes around the second argument of an if command**

If the second argument of an **if** command is a single word, it don't need curly braces or double quotes. But put curly braces in anyway for documentation:

```
if {$i <= 10} exit   ;#exit status 0 by default
if {$i <= 10} {exit}
puts stderr "hello"
```

If the second argument of an **if** command is more than a single word, it must be enclosed in double quotes or curly braces:

```
if {$i <= 10} "exit 0"     ;#double quotes legal here, but misleading
if {$i <= 10} {exit 0}     ;#clearer
```

Here's a case where you must use curly braces instead of double quotes to avoid a bug:

```
1 /* C fragment */
2 #include <stdio.h>
3
4     int i = 1;
5
```

```
6     if (++i <= 10) {
7         printf ("%d\n", i);
8     }
```

```
#!/usr/local/bin/tclsh

#The if command will perform deferred substitution on the $i
#in the puts:
set i 1
if {[incr i] <= 10} {puts $i}       ;#good

#The Tcl interpreter will perform substitution on the $i
#in the puts before the if command is executed:
set i 1
if {[incr i] <= 10} "puts $i"       ;#bug
```

```
    2
    1
```

**Execute another program, i.e., spawn a child**

A Tcl program can run another program, in any language, with the **exec** command:

```
#!/usr/local/bin/tclsh

exec rm myfile
exit 0
```

The result of the **exec** command is the standard output of the child:

```
#!/usr/local/bin/tclsh

set d [exec date]
puts "The current date is $d."

#Of course, if all you want to do is print the standard output
#of date, there is no need to deposit it into a variable:

puts "The current date is [exec date]."
exit 0
```

```
    The current date is Fri Jan 9 00:41:41 EST 2004.
    The current date is Fri Jan 9 00:41:41 EST 2004.
```

The **exec** command implements **|**, **<**, **>**, **>>**, the Bourne shell **2>**, and the C shell **>&**. But it does not invoke the shell—it calls the **execve**(2) Unix system call directly. You must therefore put the Tcl delimiters **""** or **{}**, not the shell delimiters **""** or **''**, around a child's command line argument containing white space or characters that are special to Tcl. For example, we can put either the **""** shown below or **{}** around the argument of the following **sed** because it only contains white space,

```
set d [exec date | sed "s/Dec 25/Christmas/"]
```

but we must put the **{}** shown below around the argument of this **sed** because it contains the Tcl special character **$** as well as white space:

```
set d [exec date | sed {s/1999$/Nineteen Ninety-Nine/}]
```

**Harvest the child's exit status**

The **catch** command takes two arguments. The first is a dangerous command, e.g., **exec**, enclosed in double quotes or curly braces to make it count as a single argument of **catch**. The second argument of **catch** is the name of the variable (without the leading **$**) that will receive the standard output of the child. The result of **catch** is 0 if the child succeeded (i.e., returned exit status 0), and 1 otherwise.

```
#!/usr/local/bin/tclsh

set success [catch {exec grep root /etc/passwd} output]

if {$success == 0} {
    puts "grep returned exit status 0."
} else {
    puts "grep did not return exit status 0."
}

puts "The output of grep is \"$output\"."
exit 0
```

```
grep returned exit status 0.
The output of grep is "root:*:0:1:root,,,:/:/bin/csh".
```

If we **grep** for a word that is not in the file **/etc/passwd**, **grep** will return an non-zero exit status to **exec**, causing **catch** to deposit an error message into its second argument:

```
grep did not return exit status 0.
The output of grep is "child process exited abnormally".
```

If **grep** writes to both its standard output and standard error output, they're both stored in the second argument of **catch**:

```
#!/usr/local/bin/tclsh

set success [catch {exec grep root /etc/passwd /etc/groups} output]

if {$success == 0} {
    puts "grep returned exit status 0."
} else {
    puts "grep did not return exit status 0."
}

puts "The output of grep is \"$output\"."
exit 0
```

```
grep did not return exit status 0.
The output of grep is "/etc/passwd:root:*:0:1:root,:/:/bin/csh
grep: can't open /etc/groups".
```

**For loops**

As in C, the **for** command is a more localized notation for a **while**:

```
#!/usr/local/bin/tclsh

for {set i 1} {$i <= 10} {incr i} {
    puts $i
}

exit 0
```

The following **for** command is illegal because it has only three arguments:

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

The following is wrong because the variable **$i** has not yet been initialized when the interpreter performing substutution on the second argument of the **for** command:

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

**Strings**

The first argument of the **string** command tells it what to do: the operations familiar to C programmers from **/usr/include/string.h**:

```
#!/usr/local/bin/tclsh

set users "moe larry curly"

puts "The string is \"$users\"."
puts "The string contains [string length $users] characters."
puts "The second character of the string is [string index $users 1]."

exit 0
```

```
The string is "moe larry curly".
The string contains 15 characters.
The second character of the string is o.
```

**Lists**

A *list* is a string (in Tcl, *every* value is a string) whose curly braces (if any) are nested.

```
moe larry curly
{moe larry curly}
{moe larry} curly
{}
{moe {larry {curly}} {}}
```

The most important example of a list is the fact that every Tcl command is a list:

```
puts "hello"
if {$a == $b} {exit 0}
```

The empty list (which is the same value as the empty string) may be written either as **""** or **{}**. It's conventional to write the empty list as **{}** and the empty string as **""**.

```
#!/usr/local/bin/tclsh

#Exactly the same value as in previous script.
set users {moe larry curly}

puts "The list is {$users}."
puts "The list contains [string length $users] characters."
puts "The list contains [llength $users] elements."
puts "The second element of the list is [lindex $users 1]."

exit 0
```

```
The list is {moe larry curly}.
The list contains 15 characters.
The list contains 3 elements.
The second element of the list is larry.
```

Every list is a string, but not every string is a list. In other words, not every string can be the argument of a list function:

```
#!/usr/local/bin/tclsh

#Need the double quotes to avoid error message.
set s "{moe larry {curly shemp}"
puts $s

puts [string length $s]   ;#perfectly legal as a string
puts [llength $s]         ;#but illegal as argument of a list function

exit 0
```

```
{moe larry {curly shemp}
24
unmatched open brace in list
```

**Command line arguments**

The command line arguments (but not the name of the program) are passed to the program in a list named **`argv`**:

```
#!/usr/local/bin/tclsh

puts "The name of this program is $argv0."
puts "There are [llength $argv] command line arguments:"
puts ""

puts "The first command line argument is [lindex $argv 0]."
puts "The second command line argument is [lindex $argv 1]."
puts "The third command line argument is [lindex $argv 2]."

puts "The entire list is $argv."
puts ""

puts "There are $argc command line arguments:"
foreach arg $argv {        ;#no $ when giving a value to a variable
    puts $arg
}

exit 0
```

When run with the three arguments **`moe marry curly`**, the output is

```
The name of this program is myprog.
There are 3 command line arguments:

The first command line argument is moe.
The second command line argument is larry.
The third command line argument is curly.
The entire list is moe larry curly.

There are 3 command line arguments:
moe
larry
curly
```

**Simulate a C array of structures**

Lists can be nested, and can therefore hold rows and columns of information or even trees.

Since there is only one command line argument, **`$remaining`** could have been initialized like this:

```
set remaining $argv
```

**`$n`** receives the value 4.

The initial value of **`$denominations`** could have been written on one line:

```
set denominations \
{{"quarters" "quarter" 25} {"dimes" "dime" 10} {"nickels" "nickel" 5} {"pennies" "penny" 1
```

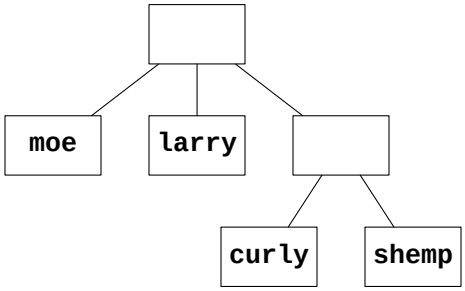
As in C, the value of **`==`** is 1 for true, 0 for false. Integer division truncates, and **`%`** is the remainder operator.

```
#!/usr/local/bin/tclsh
#This program takes one command line argument, giving a number of
#cents.  It then outputs the shortest possible list of coins with that
#total value.

if {$argc != 1} {
    puts stderr "$argv0: requires one command line argument"
    exit 1
}
set remaining [lindex $argv 0]

set denominations {
    {"quarters" "quarter"   25}
    {"dimes"    "dime"      10}
    {"nickels"  "nickel"     5}
    {"pennies"  "penny"      1}
}

set n [llength $denominations]

for {set i 0} {$i < $n} {incr i} {
    set denomination [lindex $denominations $i]
    set value [lindex $denomination 2]
    set amount [expr $remaining / $value]

    if {$amount > 0} {
        puts "$amount [lindex $denomination [expr $amount == 1]]"
    }
    set remaining [expr $remaining % $value]
}

exit 0
```

```
1$ myprog 82
3 quarters
1 nickel
2 pennies
```

**Build a tree**



```
#!/usr/local/bin/tclsh

set stooges {moe larry {curly shemp}}

puts "There are [llength $stooges] stooges."
puts "The third stooge had [llength [lindex $stooges 2]] incarnations."

exit 0
```

```
There are 3 stooges.
The third stooge had 2 incarnations.
```

**The list command**

The result of the **list** command is list of all its arguments. Since each of them may be a list, **list** can create nested lists:

```
#!/usr/local/bin/tclsh

set p1 "Madonna"
set p2 "Bill Clinton"
set p3 "Johann Sebastian Bach"
set p4 "George Herbert Walker Bush"

set people_string "$p1 $p2 $p3 $p4"
puts $people_string
puts [llength $people_string]

set people_list [list $p1 $p2 $p3 $p4]
puts $people_list
puts [llength $people_list]

exit 0
```

```
Madonna Bill Clinton Johann Sebastian Bach George Herbert Walker Bush
10
Madonna {Bill Clinton} {Johann Sebastian Bach} {George Herbert Walker Bush}
4
```

**Tcl special characters**

| | |
|---|---|
| **$ \ []** | *substitution* |
| **{} ""** | *grouping* |
| **;** *newline* | *command termination* |
| # | *comment* |

**Arrays**

Tcl has only one-dimensional arrays. Whitespace within the parentheses is significant, so don't put any there! When its first argument is the word **size**, the result of the **array** command is the number of elements in the array (in this case **4**). When its first argument is the word **names**, the result of the **array** command is a list of all the subscripts of the array.

```
#!/usr/local/bin/tclsh

set episode(1) "The Phantom Menace"
set episode(4) "A New Hope"
set episode(5) "The Empire Strikes Back"
set episode(6) "Return of the Jedi"

puts "The array has [array size episode] elements."
puts "The subscripts are [array names episode]."

for {set i 1} {$i <= 9} {incr i} {
    if {[info exists episode($i)]} {
        puts $episode($i)
    }
}

exit 0
```

```
The array has 4 elements.
The subscripts are 4 5 1 6.
The Phantom Menace
A New Hope
The Empire Strikes Back
Return of the Jedi
```

Every Tcl array is actually an *associative array:* one whose subscripts can be any string, not necessarily representing an integer. Don't use the **foreach** command if you're going to insert and delete elements as it loops through the array.

```
#!/usr/local/bin/tclsh

set uid(moe)   0                    ;#no double quotes around moe
set uid(larry) 1
set uid(curly) 2

puts "The subscripts are [array names uid]."

foreach loginname [array names uid] {
    puts "Loginname $loginname has the UID $uid($loginname)."
}

exit 0
```

```
The subscripts are moe larry curly.
Loginname moe has the UID 0.
Loginname larry has the UID 1.
Loginname curly has the UID 2.
```

To loop through the subscripts in alphabetical order, use the **lsort** command. Its argument and result are lists:

```
foreach loginname [lsort [array names uid]] {
```

As in Perl, the environment variables are stored in an associative array:

```
#!/usr/local/bin/tclsh

puts "The full pathname of my home directory is $env(HOME)".
puts "The IP address of my display is $env(DISPLAY)".

exit 0
```

To see the name and value of every environment variable (in alphabetical order),

```
#!/usr/local/bin/tclsh

foreach varname [lsort [array names env]] {
    puts "$varname $env($varname)"
}

exit 0
```

```
DISPLAY 192.168.5.21:0.0
EDITMODE emacs
EDITOR vi
EXINIT set showmode                    etc.
```

**Eval: the Tcl Heart of Darkness**

The **eval** command lets you store a command in a string for later execution. (The following section will explain why it won't work without the word **eval**.)

```
#!/usr/local/bin/tclsh

set saved "puts hello"
eval $saved

exit 0
```

```
hello
```

When we create a button or other widget in the language Tk, we often attach to it a string containing a command to be executed later when the button is pressed.

**Most commands are parsed only once.**

A **puts** command must have exactly one argument, not counting the optional file identifier (e.g., **stderr**). Is the following **puts** legal even though the variable contains three words?

```
set users {moe larry curly}
puts $users
```

This turns out to be legal because the interpreter splits a command into separate words once and for all before it performs any substitution. This splitting is called *parsing*. The interpreter does not go back and parse again after changing the **$users** into the three words **moe larry curly**.

Similarly, the following **set** is legal even though the result of the **exec** command is six separate words:

```
set d [exec date]
```

As before, the interpreter parses the command into words before it performs any substitution. Any section of a command enclosed in **"**double quotes**"**, **{**curly braces**}**, or **[**square brackets**]** counts as a single word. The interpreter does not go back and parse again after changing the **[exec date]** into the six words **Fri**

**`Jan 9 00:41:41 EST 2004`**.

**Parse a command more than once**

The **`glob`** command outputs a list of the names of some or all of the things in the current directory.

```
#!/usr/local/bin/tclsh

puts [glob *.o]
exit 0
```

```
prog1.o prog2.o prog3.o
```

Sometimes, however, the ‘‘single parse’’ rule makes a command more complicated. For example, let’s try to give the result of **`glob`** to the Unix **`rm`** command instead of to **`puts`**:

```
#!/usr/local/bin/tclsh

exec rm [glob *.o]
exit 0
```

```
rm: prog1.o prog2.o prog3.o nonexistent
```

We want to give three arguments to **`rm`**, but **`rm`** receives only one big argument containing embedded blanks (**`prog1.o prog2.o prog3.o`**) because a square-bracketed expression such as **`[glob *.o]`** counts as a single word. We have to use **`eval`** to make the interpreter parse the command again after running the command in the square brackets:

```
#!/usr/local/bin/tclsh

eval exec rm [glob *.o]
exit 0
```

Here’s another example where we need to make the interpreter parse after substitution as well as before. The **`unset`** command can remove more than one variable:

```
unset x y z
```

But the following **`unset`** receives only one big argument containing embedded blanks: **`x y z`**.

```
set vars {x y z}
unset $vars
```

We have to use **`eval`** to make the interpreter parse the command again after performing substitution on the **`$vars`**:

```
set vars {x y z}
eval unset $vars
```

One more example:

```
set loginnames {moe larry curly}
eval exec mail $loginnames < $env(HOME)/letter
```

Without the **`eval`**, you would get the message

```
moe larry curly... User unknown
```

**The horizontal vs. vertical problem: sometimes you need more than just eval**

```
#!/bin/sh
#Mail a letter to everyone logged in right now.

loginnames=‘who | awk ’{print $1}’‘
mail $loginnames < $HOME/letter

exit 0
```

The following example hangs because the first newline embedded in the string **$loginnames** terminates the **mail** command line, causing **mail** to wait for standard input from the keyboard:

```
set loginnames [exec who | awk {{print $1}}]
eval exec mail $loginnames < $env(HOME)/letter
```

Unlike a shellscript, a Tcl program has to change the newlines to blanks:

```
set loginnames [exec who | awk {{print $1}} | tr {\012} { }]
eval exec mail $loginnames < $env(HOME)/letter
```

You could also change the newlines to blanks purely in Tcl:

```
regsub -all "\n" [exec who | awk {{print $1}}] " " loginnames
eval exec mail $loginnames < $env(HOME)/letter
```

**Parse a saved command correctly**

I’d like to save a command in a string for later execution. The command should output the current value of **users**:

```
#!/usr/local/bin/tclsh

set users {moe larry curly}
set saved "puts $users"
puts "The saved command is: $saved"

eval $saved                ;#causes the error message shown below
exit 0
```

```
bad file identifier "moe"
The saved command is: puts moe larry curly
```

(The error message appears first because, as in C, **stdout** is buffered but **stderr** is not.) The second **set** saves the string **puts moe larry curly** into the variable **saved**. Unfortunately, the **eval** command then parses **$saved** into four separate words, causing the **puts** in the string to give us the error message.

Every Tcl command is a list. The correct way to create a list consisting of the one-word list **puts** and the three-word list in the variable **$users** is

```
#!/usr/local/bin/tclsh

set users {moe larry curly}
set saved [list puts $users]
puts "The saved command is: $saved"

eval $saved
exit 0
```

```
The saved command is: puts {moe larry curly}
moe larry curly
```

**File i/o**

The result of the **open** command is a file identifier, like **stdin**, **stdout**, or **stderr**. Save the result of the **open** command in a variable, just as you save the return value of the **fopen** function in C. The result of the **gets** command is the number of characters that were input, or −1 on end-of-file. The result of the **split** command is a list of substrings.

```
#!/usr/local/bin/tclsh

set passwd [open "/etc/passwd" r]

while {[gets $passwd line] >= 0} {
    set fields [split $line ":"]
    set loginname [lindex $fields 0]
    puts $loginname
}

close $passwd
exit 0
```

```
moe
larry
curly
```

To catch errors in opening a file, the above **set passwd** command should be changed to

```
if [catch {open "/etc/passwd" r} passwd] {
    #The variable $passwd now contains an error message.
    error "$argv0: $passwd"
}
```

which will write a message such as

```
myprog: couldn't open "/etc/passwd": Permission denied
```

to the standard error output and yield exit status 1.

The three commands in the body of the **while** loop may be combined to

```
puts [lindex [split $line ":"] 0]
```

**Using open files in an exec**

Use ordinary **<** and **>** when the following words are filenames:

```
#!/usr/local/bin/tclsh

set outfile "/tmp/[pid]"
exec grep root < /etc/passwd > $outfile
exec grep root < /etc/passwd > stdout  ;#create a file named "stdout"

exit 0
```

Use **@<** and **@>** when the following words are *file identifiers* , i.e., **stdin**, **stdout**, **stderr**, or the results of previous **open**'s:

```
#!/usr/local/bin/tclsh

set passwd [open "/etc/passwd" r]
set outfile [open "/tmp/[pid]" w]

exec grep root <@ $passwd >@ $outfile
exec grep root <@ $passwd >@ stdout    ;#stdout of tclsh

exit 0
```

We've already seen that if you use neither **>@** nor **>** the child's standard output becomes the result of the **exec** command:

```
set d [exec date]
```

**Pipe i/o**

The source of the input or the destination of the output can be a pipeline. The ''filename'' in either case must begin with the pipe character **|**, and the second argument of **open** tells if the pipe leads into the Tcl program (**r**) or out (**w**). Note that this is different from Perl, where an input ''filename'' ends with a pipe character and an output ''filename'' begins with a pipe character.

```
#!/usr/local/bin/tclsh

set passwd [open "| awk -F: {{print $1}} /etc/passwd | sort" r]

while {[gets $passwd loginname] >= 0} {
    puts $loginname
}

close $passwd
exit 0
```

**A user-defined procedure**

The following **proc** command creates a new command named **myproc**. You must execute the **proc** command before attempting to execute the new command.

**proc** takes three arguments. The first is the name of the new procedure; the second is a list of the names of the arguments of the new procedure; the third is the body of the new procedure. If the second argument of **proc** is the empty list, write it as **{}**.

Unless you declare them to be **global**, the variables in a procedure are local to it. Expect has slightly different scoping rules: see Libes, pp. 100, 241−245.

If the procedure does not execute a **return** command, or if the **return** command that is executed has no argument, the procedure will return the null string.

```
 1 #!/usr/local/bin/tclsh
 2
 3 proc myproc {} {
 4     global a
 5
 6     set a "moe"
 7     set b "larry"
 8
 9     return "curly"
10 }
11
12 set a 10
13 set b 20
14
15 set retval [myproc]
16
17 puts "\$a == $a"
18 puts "\$b == $b"
19 puts "The return value is $retval."
20
21 exit 0
```

```
$a == moe
$b == 20
The return value is curly.
```

**Arguments for user-defined procedures**

The first argument of the following **myproc** is passed by *value*, so **myproc** cannot change the value of the caller's variable **$a**. The second argument of **myproc** is passed by *reference*, so **myproc** can change the value of the caller's variable **$b**.

```
 1 #!/usr/local/bin/tclsh
 2
 3 proc myproc {x y} {
 4     upvar $y z
 5
 6     puts "The caller of this procedure thinks that the name of the"
 7     puts "variable is \"$y\", but this procedure thinks that the"
 8     puts "name of the variable is \"z\"."
 9
10     puts "This procedure received the arguments $x $z."
11
12     set x "moe"         ;#has no effect on $a
13     set z "larry"       ;#changes the value of $b
14 }
15
16 set a 10
17 set b 20
18
19 myproc $a b
20
21 puts "\$a == $a"
22 puts "\$b == $b"
23
```

```
24 exit 0
```

```
    The caller of this procedure thinks that the name of the
    variable is "b", but this procedure thinks that the
    name of the variable is "z".
    This procedure received the arguments 10 20.
    $a == 10
    $b == larry
```

The result of the command in line 19 is the null string, which is what we would have stored in the variable **retval** if line 19 said

```
19 set retval [myproc $a b]
```

**Pass an array as an argument**

You can't pass the contents of an array to a procedure: you have to pass the name of the array.

```
#!/usr/local/bin/tclsh

set loginname(0) "root"
set loginname(1) "operator"
set loginname(2) "daemon"

proc myproc {original_name} {
    upvar $original_name a

    puts "The caller of this procedure thinks that the name of the"
    puts "array is \"$original_name\", but this procedure thinks"
    puts "that the name of the array is \"a\"."

    puts [array size a]
    puts $a(0)
}

myproc loginname
exit 0
```

```
    The caller of this procedure thinks that the name of the
    array is "loginname", but this procedure thinks
    that the name of the array is "a".
    3
    root
```

**Catch an error**

If we changed the above command

```
    myproc loginname
```

to

```
    set x 10
    myproc x
```

the program would have died in the **array size** command. The standard error output from the **array size** command appears before the standard output from the **puts** commands:

```
"a" isn't an array
The caller of this procedure thinks that the name of the
array is "x", but this procedure thinks
that the name of the array is "a".
```

To die with an error message of our own choosing, put the sensitive **array size** command inside of a **catch**:

```
#!/usr/local/bin/tclsh

set loginname(0) "root"
set loginname(1) "operator"
set loginname(2) "daemon"

proc myproc {original_name} {
    upvar $original_name a

    if {[catch {array size a} n] != 0} {
        flush stdout
        puts stderr "proc myproc argument $original_name must be"
        puts stderr "the name of an array"
        exit 1
    }

    puts "The caller of this procedure thinks that the name of the"
    puts "array is \"$original_name\", but this procedure thinks"
    puts "that the name of the array is \"a\"."

    puts "The array contains $n elements."
    puts $a(0)
}

myproc loginname
set x 10
myproc x

puts "goodbye"
exit 0
```

```
The caller of this procedure thinks that the name of the
array is "loginname", but this procedure thinks
that the name of the array is "a".
The array contains 3 elements.
root
proc myproc argument x must be
the name of an array
```

**There's no hope of recovery if you drop dead**

The above **array size** command did not execute an **exit** command when it realized that it had received a scalar instead of an array. If it had done so, the program would have dropped dead instantly and it would be too late to **catch** the error. The command actually executed by **array size** is the less extreme **error** command, which can be caught by a **catch**.

Our procedure **myproc** should show the same courtesy to its callers:

```
#!/usr/local/bin/tclsh

set loginname(0) "root"
set loginname(1) "operator"
set loginname(2) "daemon"

proc myproc {original_name} {
    upvar $original_name a

    if {[catch {array size a} n] != 0} {
        flush stdout
        error "proc myproc argument $original_name must be\
the name of an array"
    }

    puts "The caller of this procedure thinks that the name of the"
    puts "array is \"$original_name\", but this procedure thinks"
    puts "that the name of the array is \"a\"."

    puts "The array contains $n elements."
    puts $a(0)
}

myproc loginname

set x 10
if {[catch {myproc x} retval] != 0} {
    puts "That's tough, but I'm not going to drop dead because of it."
    puts "\$retval == \"$retval\""
    puts ""
    puts "\$errorInfo == \"$errorInfo\""
}

puts "goodbye"
exit 0
```

```
    The caller of this procedure thinks that the name of the
    array is "loginname", but this procedure thinks
    that the name of the array is "a".
    The array contains 3 elements.
    root
    That's tough, but I'm not going to drop dead because of it.
    $retval == "proc myproc argument x must be the name of an array"

    $errorInfo == "proc myproc argument x must be the name of an array
        while executing
    "error "proc myproc argument $original_name must be the name of an array""
        invoked from within
    "if {[catch {array size a} n] != 0} {
        error "proc myproc argument $original_name must be the name of an array"
        }"
        (procedure "myproc" line 4)
        invoked from within
    "myproc x""
    goodbye
```

**source is just like #include**

```
#!/usr/local/bin/tclsh

source myfile.tcl             ;#assumed to be in the current directory
source /home1/m/mm64/myfile.tcl

exit 0
```

**Run tclsh interactively**

```
    1$ tclsh                  2$ is my Unix shell prompt
    % puts hello              % is the tclsh prompt
    hello
    % expr 1 + 2
    3
    % set tcl_interactive
    1
    % ls -l
```

*and now you see the output of the Unix program* **`ls -l`**

```
    % exit  or type control-d
    3$
```

Four things are different when you run the Tcl interpreter **`tclsh`** interactively:

(1) **`tclsh`** will begin by executing the commands in the file **`.tclshrc`** in your home directory.

(2) **`tclsh`** prints a prompt before each command. The default prompt is **`%`**, but you can write a **`puts`** command to display your own prompt:

```
    set tcl_prompt1 {puts -nonewline "hello "}
    set tcl_prompt1 {puts -nonewline "[history nextid] % "}
    set tcl_prompt2 {puts -nonewline "keep going: "}      ;#secondary prompt
```

(3) The value of the variable **`$tcl_interactive`** is 1, so you can write procedures that behave differently when invoked interactively.

(4) **tclsh** displays the result of each command, which had previously been thrown away when we were running non-interactively. For example, you can now say **expr 1 + 2** instead of **puts [expr 1 + 2]**. And you can say **set v** instead of **puts $v**.

**The unknown command (Ousterhout pp. 136−137)**

When you execute commands which are not recognized as Tcl,

```
putz hello
ls -l
```

the Tcl interpreter automatically executes the commands

```
unknown putz hello
unknown ls -l
```

The procedure **unknown** is already written for you. It's in the file **init.tcl** in the directory whose name is the result of the Tcl command

```
info library
```

**Write your own version of the unknown command**

Instead of

```
set i [expr 1 + 2]
```

I wish I could say

```
set i [1 + 2]  ;#illegal because 1 is not the name of a Tcl command
```

And instead of

```
puts [exec ls -l]
```

I wish I could say

```
ls -l          ;#illegal because ls is not the name of a Tcl command
```

The following procedure can take any number of arguments, which will be stored in the list **$args**. **args** is a special word when used as the only (or last) word of the second argument of a **proc** command (Ousterhout, p. 83). Use **lindex** to extract the individual words from **args**. For example, the result of **lindex $args 0** will be the misspelled command name.

The **string compare** in the following procedure lets us say

```
!!
```

to repeat the last command when using the Tcl interpreter interactively.

**>&@ stdout** directs the standard output and the standard error output of the Unix program **ls -l** to the same destination as the standard output of the Tcl script. Without the **>&@ stdout**, the standard output of the Unix program **ls -l** would become the result of the Tcl command **ls -l**, and (in a script) you'd have to say

```
puts [ls -l]
```

to see it.

If the arguments passed to the **unknown** procedure contain strings but no Tcl variables (e.g., **ls -l**), then you don't need the **uplevel** in the **return uplevel exec**. But if the arguments do contain variables (and they will!) we need the **uplevel** to tell the **unknown** procedure that these variables belong to the caller of the **unknown** procedure, not to the **unknown** procedure itself.

If the arguments passed to the **unknown** procedure contain constants but no variables (e.g., **1 + 2**), then you don't need the **uplevel** in the **if catch uplevel expr**. But if the arguments do contain

variables (and they will!) we need the **uplevel** to tell the **unknown** procedure that these variables belong to the caller of the **unknown** procedure, not to the **unknown** procedure itself.

```
#Just a procedure definition--not a complete program.

proc unknown {args} {
    global tcl_interactive

    if {$tcl_interactive && [string compare [lindex $args 0] "!!"] == 0} {
        return [uplevel history redo]
    }

    #Is it an expression?
    if {[catch {uplevel expr $args} result] == 0} {
        return $result
    }

    #Is it an invocation of a Unix program?
    if {[catch {exec which [lindex $args 0]} pathname] == 0} {
        return [uplevel exec $args <@ stdin >&@ stdout]
    }

    error "unknown command $args"
}
```

**Create a Tcl library: Ousterhout, pp. 137−138**

Each library is stored in a separate directory, and the list **$auto_path** lists these directories. The default **unknown** procedure automatically searches through the libraries in the order listed in **$auto_path**. **unknown** looks at the **tclIndex** file in each directory to find the names of the procedures in the **.tcl** files in that directory. You create these **tclIndex** files with the **auto_mkindex** command.

```
#!/usr/local/bin/tclsh

puts $auto_path
exit 0
```

```
/usr/local/lib/tcl
```

To create your own library, write **proc** and the name of the procedure side-by-side at the at the start of a line in the **.tcl** file(s) of your library:

```
#This file is /home1/m/mm64/lib/file1.tcl

#myproc0 is legal Tcl, but wrong in a Tcl library--
#for pedagogical purposes only.
proc \
myproc0 {} {
}

proc myproc1 {} {
    puts "This is myproc1."
}

proc myproc2 {} {
    puts "This is myproc2."
}
```

```
#This file is /home1/m/mm64/lib/file2.tcl

proc myproc3 {} {
    puts "This is myproc3."
}
```

After creating the above files, run the following program. ***** is not a special character in Tcl; it is special only to the **auto_mkindex** and **glob** commands.

```
#!/usr/local/bin/tclsh

auto_mkindex /home1/m/mm64/lib *.tcl
exit 0
```

It will create the following file **/home1/m/mm64/lib/tclIndex**:

```
# Tcl autoload index file, version 2.0
# This file is generated by the "auto_mkindex" command
# and sourced to set up indexing information for one or
# more commands.  Typically each line is a command that
# sets an element in the auto_index array, where the
# element name is the name of a command and the value is
# a script that loads the command.

set auto_index(myproc1) "source $dir/file1.tcl"
set auto_index(myproc2) "source $dir/file1.tcl"
set auto_index(myproc3) "source $dir/file2.tcl"
```

Finally, at the start of your Tcl program, say

```
#!/usr/local/bin/tclsh

set auto_path [linsert $auto_path 0 /home1/m/mm64/lib]
puts $auto_path                     ;#to verify that linsert worked

#Rest of program:
myproc2
exit 0
```

```
/home/m/mm64/lib /usr/local/lib/tcl
This is myproc2.
```

This is more efficient than writing a separate **source** command at the start of your script for each **.tcl** file in your libraries. You can even **set** your **auto_path** in your **init.tcl** file.

Change the **linsert** command to

```
lappend $autopath /home1/m/mm64/lib
```

if you don't want the procedures in your libraries to override the ones in the existing libraries.
□