An overview of Tcl programming

Tcl is a very simple programming language that is ideal for the assignments in this course. It is an easy language to learn, and this short pamphlet covers the language syntax and some basic commands.

Tcl is an acronym for "Tool Command Language," and is usually pronounced "tickle." It is commonly bundled with a graphical user interface library called Tk (for "toolkit.") The language is designed to be rudimentary and easily modified and extended. It's simple enough to learn that you're better off running through a list of coding examples than reading a book about it—this pamphlet is written with this approach in mind.

Your assignment is to read this document through, and to type in the code examples in the gray boxes. If you have any difficulties, contact the instructor. To learn more, you can consult the Tcl command manual pages at `http://www.tcl.tk/doc/`, or the excellent and comprehensive Tcl Wikibook at `http://en.wikibooks.org/wiki/Tcl_Programming/Print_version`.

Getting started

These directions are best followed on a cluster Macintosh, although they will work on any Unix-like object. This includes GNU/Linux machines, and maybe Windows PCs with Cygwin installed[1], although getting some of our code to run on Windows is not easy. **Step one is getting to a computer that can follow these directions.**

Step two: open a shell (command prompt). On a Mac, the Terminal can be found under Applications/Utilities. On a PC, launching Cygwin gives you a prompt. If you are a GNU/Linux user, you should know by know how to launch a command shell.

Step three: launch the interpreter. At your command prompt type the `tclsh` command, and run a few commands to make sure everything works. The `exit` command should quit the session; you can also type Control-D to tell the program that you have no more input, or in the worst case hit Control-C to kill the interpreter.

```
your-prompt$ tclsh
% puts Hello
Hello
% info tclversion
8.4
% exit
```

[1] Cygwin is a free download from cygwin.com, although it is apparently no longer installed in the clusters.

Step four: test the user interface. The following commands should create a window displaying the contents of the variable `foo`:

```
bash-3.2$ tclsh
% package require Tk
8.4
% pack [entry .e -textvar foo -font {Arial 48}]
% set foo "Hello World"
Hello World
```

Step five: create a script. Exit the interpreter, and use a text editor (Word is not a text editor) to create a file in your shell's current directory called `foo.tcl`. Fill `foo.tcl` with the following code[2]—make sure everything is verbatim, including the spaces:

```
#!/usr/bin/tclsh
package require Tk
pack [entry .e -textvar foo -font {Arial 48}]

proc every {ms script} {
     uplevel 1 $script
     after $ms [list every $ms $script]
}

every 1000 {
     set foo [clock format [clock seconds] -format %I:%M:%S]
}
```

You can find your shell's current directory by typing the `pwd` (print working directory) command. On the Macintosh, you can also type "`open .`" to open a folder where your home directory is. There are similar commands in GNU, like "gnome-open" in Ubuntu.

Step six: run your program. There are a few ways to do this, and you should try them all. First, we can launch the interpreter and feed it your program in various ways. In each case, hit Control-C to stop the program, or just close the window it creates:

```
bash-3.2$ tclsh foo.tcl
bash-3.2$ tclsh < foo.tcl
bash-3.2$ cat foo.tcl | tclsh
```

[2] This script, and several other examples, are either inspired by or directly ripped from the Tcl Wikibook.

In all of these commands, we run the interpreter and feed in the program as input. This is a bit ungainly, and we will usually prefer the second approach: make your program executable, and then launch it directly. Type:

```
bash-3.2$ chmod u+x foo.tcl
bash-3.2$ ./foo.tcl
```

The `chmod` command gives the user (u) permission to execute (x) the file. The first line in the file `(#!/usr/bin/tclsh)` is used by the shell to determine which language the program is in (or rather, which interpreter the shell should invoke.)

General Syntax

Tcl has a ridiculously simple syntax with only a handful of rules[3]. In general, a Tcl program is a list of commands—a "command" is analogous to a function in C or Java, but is written like a Unix or DOS shell command. That is, each command is a bunch of words, separated by space, with the first word denoting the command name and the remaining words denoting the arguments, which are all strings.

```
% puts Hello
Hello
% list 1 2 3
1 2 3
```

Note the conspicuous absence of punctuation: the arguments are not separated by commas, and are not enclosed in parentheses. The command does not end with a semicolon. The arguments are strings, but they are *not encased in quotation marks*.

You are *allowed* to end a statement with a semicolon if you want. This allows you to cram multiple commands on one line, or to append a comment to a line (in Tcl, a comment is any command line that starts with the octothorpe character #):

```
% puts Hello; puts World
Hello
World
% # this is a comment
% expr 2+2 ;# the expr command evaluates expressions
4
```

[3] 11, to be exact, although Tcl 8.5 has 12 rules. The entirety of the language's syntax can be summarized in less than one page of text. Because this is a tutorial, we will spread the syntax over four pages.

Quotation marks

You are also allowed to wrap your arguments in quotation marks, which is useful if they contain whitespace, or line breaks. Tcl recognizes two types of quotation marks: double quotes and curly braces.

```
% puts "Hello world"
Hello world
% puts {Hello world}
Hello world
% list "a" "b c"
a {b c}
```

I repeat for emphasis: in Tcl, *curly braces are quotation marks*. Not grouping symbols for source code, but quotation marks. For example, consider the following code:

```
% if {1<2} {
     puts "Excellent!"
  } else {
     puts "Bogus."
  }
```

This may seem like a run-of-the-mill if statement, spanning multiple lines and enclosing multiple commands. In reality, it is a *single command line*, with a single command (`if`) and five strings of text provided as arguments[4]. You could equivalently write:

```
% if "1<2" "\n puts Excellent!\n" "else" "\n puts Bogus.\n"
```

Note in particular that both statements are only one line: in both examples, the line breaks are newline characters *inside* the strings. Outside of the arguments, there are no line breaks—if there were line breaks on the command line, the command would end prematurely. Thus this code causes a syntax error:

```
% if {1<2}
  {
     puts "Excellent!"
  }
```

[4] Notice that since they are arguments in a command line, they must be separated from each other by at least one space. It is a common error to accidentally write “if{expression}” rather than “if {expression}.”

Now, you may ask: why do we have two different kinds of quotation marks? Is there a difference between the two? Yes, they are different in two crucial ways. First, curly brace quotes are taken literally, while double-quote quotes are a little bit interpreted, substituting in variables and special characters:

```
% set foo 5
5
% puts "$foo $foo\n$foo $foo"
5 5
5 5
% puts {$foo $foo\n$foo $foo}
$foo $foo\n$foo $foo
% list "$foo" $foo {$foo}
5 5 {$foo}
```

As that last command shows, unquoted strings are substituted just like double-quoted strings. Brace quotes are the only environment where substitution doesn't happen.

The second crucial difference is the way a quoted argument *ends*. If you start an argument with a double quote, it ends at the next double quote. If you start an argument with a left brace, it does *not* end at the next right brace: it ends at the right brace that *matches* the left brace. Tcl's quotation-braces form matching pairs, which can nest within one another, just like braces in C or other languages. To illustrate this, let's create a procedure called `howmany` that tells us how many arguments it is called with[5]; we'll then hand it a bunch of quoty-like expressions.

```
% proc howmany args {llength $args}
% howmany one two three
3
% howmany " one " two " three "
3
% howmany { one { two } three }
1
% howmany { two } three }
3
```

Because brace quotes are so literal, and because they can contain nested braces, they are ideal for passing code snippets to be evaluated by other commands. This makes Tcl code look a lot like C, Java or PHP code, but bear in mind that a Tcl command is nothing more than a single command line made out of strings.

---

[5] By default, a Tcl procedure returns the result of the last command in its body, in this case `llength`.

Substitution

As we saw in the last section, Tcl makes variable and other substitutions in command line arguments, and in double-quoted strings. In fact, Tcl substitutes the whole dang command line, outside of bracey quotes. You can, for example, write this[6]:

```
% set foo pu; set bar ts
ts
% $foo$bar "I am executing the $foo$bar command"
I am executing the puts command
```

This highlights an important fact about how and when substitution happens. **Tcl takes each command line, scans over it in a single pass making all substitutions, and only then executes the resulting text.** Thus the first `$foo$bar` is substituted for `puts` before the first word on the command line is looked up and executed.

Tcl has three types of substitution, two of which you've already seen. First, Tcl performs variable substitution: any expression of the form `$name` is replaced with the contents of a variable of that name. Second, Tcl performs backslash substitution for special characters. These are mostly the same as in C: \n for a newline, \t for a tab, \\ for a backslash, \$ and \" to use those characters without invoking substitution. Tcl also recognizes hex and octal constants, such as \x41 for an 'A', and is fully Unicode compliant[7].

The third type of substitution gives Tcl a huge amount of power. Any expression inside square brackets is treated as another Tcl script. This is evaluated by the interpreter during the substitution step, and the script's return value is substituted in place of the bracketed expression. Note that brackets can nest, and all evaluation is left to right:

```
% puts AB[format %c 67]DE
ABCDE
% puts [clock format [clock seconds]]
Sat Jan 08 13:33:37 EST 2011
% puts [set foo 1][incr foo][incr foo]
123
```

---

[6] This example comes from Salvatore Sanfillipo, at http://antirez.com/articoli/tclmisunderstood.html

[7] One special backslash code is the backslash at the end of a line. This causes the interpreter to replace the backslash, the line break, and subsequent whitespace with a single space character, joining the line with the next one. This is unique in that it happens inside bracey quotes as well as regular ones.

Control Structures

As of the last section, we've just summarized about 99% of Tcl's syntax. But wait, what about loops and if-then structures? *The syntax doesn't have any!*

Most languages consist of two parts: the official language syntax as understood by the compiler/interpreter, and a standard library of functions that have become inseparable from the language itself. In C, for example, `if()` is part of the language syntax, while `printf()` is a standard library function.

The Tcl philosophy is to "outsource" virtually all language functionality to a standard library of 78 commands. This includes even such basic features as control structures, variable assignment (the `set` command) and arithmetic (the `expr` command.)

Branching

The `if` command accepts a variable number of arguments. The first argument is an arithmetic expression; the second argument[8] is a string that is evaluated as a script if the expression is true (nonzero). Further arguments may be attached using the `else` and `elseif` keywords.

There is also a `switch` statement, but it has so many features that we won't cover it here. Consult the Wikibook and the Tcl man pages for more information.

Loops

The Tcl library has a `while` command of the form `while expression script`. There is also a useful looping command called `foreach`, that loops over a list:

```
% foreach a {one two three} {puts $a}
one
two
three
% foreach {x y} {10 20 30 40} {puts "<$x $y>"}
<10 20>
<30 40>
% foreach x {a b c} y {1 2 3} {puts $x$y}
a1
b2
c3
```

[8] If you're not into the whole brevity thing, you may put a "`then`" between the expression and the script.

There's one warning you should bear in mind before you try writing a loop: any variables outside of bracey-quotes are substituted *before the loop command is called*, and are not evaluated by the loop command. Compare these two commands, for example, and try to explain the difference in their behavior:

```
% set foo 0;  while {$foo<100} {incr foo}  ;# halts
% set foo 0;  while "$foo<100" {incr foo}  ;# loops forever
```

Make your own

Because control structures are provided by commands instead of hard-wired into the syntax, you can create your own control structures, or redefine existing ones by defining new procedures. Let's steal a feature from BASIC and create an `on` command, which jumps to one of several pieces of code based on an index number:

```
% proc on args {
     set pos [uplevel 1 expr [lindex $args 0]]
     uplevel 1 [lindex $args $pos]
}
% on 1+1 {puts one} {puts two} {puts three}
two
```

The `uplevel` command is beyond the scope of these notes; it causes a script to be executed within the variable scope of whoever calls it. This makes a procedure behave less like a C function and more like an `if` or `while` statement inside a C function.

Datatypes

Tcl doesn't really have datatypes like `int` and `double`; in Tcl, *everything is a string*. Even things that look like numbers are in fact strings: the `expr` command inputs a string that describes an arithmetic expression, and outputs a string that denotes its numerical value. Your brain sees numbers, but think "string."

```
% expr 4*atan(1.0)
3.14159265359
% string length [expr 4*atan(1.0)]
13
```

The standard library of commands also support two data structures, lists and arrays.

Lists

In Tcl, a list is a string composed of arguments separated by whitespace[9]. Tcl has a set of commands that can perform list operations on strings. These commands tend to start with the letter l, e.g. `llength, lrange, lsort, lindex`:

```
% set L "one two three"
one two three
% lindex $L 1
two
% lreplace $L 0 1 four
four three
```

Again, note that a list is really just a string, with a library of commands to manipulate them. However, when a string is used as a list, the interpreter will internally represent it using an efficient data structure so that subsequent list processing is not slow.

Arrays

To first order, Tcl implements arrays just by letting you use a pair of parentheses in your variable name. Anything or nothing can go in the parentheses. Thus you can view `$arr(5)` and `$arr(six)` and `$arr()` as three array elements of the array `arr`, or as three distinct variables that happen to have similar names.

Tcl has an `array` command, which lets you manage a group of similarly named variables as a single array. Like many Tcl commands, this one has subcommands: you can type "`array get`" to dump the contents of an array as a list, or "`array set`" to bulk-assign a list of elements to an array. The subcommand "`array names`" will return a list of the indexes used, but not the values.

```
% array set arr {0 one two 3 "" five}
% puts "$arr(0) $arr(two) $arr()"
one 3 five
% foreach x [array names arr] {puts "arr($x) is $arr($x)"}
arr() is five
arr(0) is one
arr(two) is 3
```

[9] In particular, a Tcl command is a list.

A note on pointers

Does Tcl have pointers? In a way, yes. In Tcl everything is a string, and everything is stored and referenced by (string) variable names. Thus you can store a pointer to a piece of data by storing its variable name in another variable.

If you have a variable named `foo`, you can access its value it either by writing `$foo`, or the command `[set foo]`: the command `set variable value` will set a variable and return the value, while `set variable` without a third argument will simply return the variable's contents, unaltered. You can use this for multiple levels of indirection:

```
% set foo 5
5
% set pointer foo
foo
% set pointer2pointer pointer
pointer
% puts [set [set [set pointer2pointer]]]
5
```

What you still need to know

This document should give you enough Tcl literacy that you can work through the coding examples we will use in class, as long as you have the Tcl command reference close at hand. As we introduce code we will also go over the common library commands that are used.

To complete the coding assignments you will need to learn a bunch of the standard library commands. Some important commands you will see in the near future:

- **I/O commands** `puts`, `gets`, `read`, `write`; file commands `open`, `close`, `fconfigure` and `file`; and the socket command `socket`.
- **String processing** the `string` command, the `split` and `join` commands, and the extremely powerful `regexp` command for applying regular expressions.
- **Defining procedures** the `proc` command has more features than we let on. There are also the commands `uplevel` and `upvar`, `global`, `break` and `return`.
- **Tk, the graphical toolkit** Tk is a package of commands for creating windows, views, buttons and handling UI events. It is not intended to be pretty or slick, but to give us a user interface with little code, whenever we need one.
- **Event programming** `fileevent`, `after` and `bind` are examples of commands that schedule a procedure to be called when an external event occurs, like a mouse click or data arriving on a socket. This will make our networking assignments very easy to write.