UNIT-5
TCL

TCL stands for “Tool Command Language” and is pronounced “tickle”; is a simple scripting language for controlling and extending applications. TCL is a radically simple open-source interpreted programming language that provides common facilities such as variables, procedures, and control structures as well as many useful features that are not found in any other major language. TCL runs on almost all modern operating systems such as Unix, Macintosh, and Windows (including Windows Mobile). While TCL is flexible enough to be used in almost any application imaginable, it does excel in a few key areas, including: automated interaction with external programs, embedding as a library into application programs, language design, and general scripting. TCL was created in 1988 by John Ousterhout and is distributed under a BSD style license (which allows you everything GPL does, plus closing your source code). The current stable version, in February 2008, is 8.5.1 (8.4.18 in the older 8.4 branch). The first major GUI extension that works with TCL is TK, a toolkit that aims to rapid GUI development. That is why TCL is now more commonly called TCL/TK. The language features far-reaching introspection, and the syntax, while simple2, is very different from the Fortran/Algol/C++/Java world. Although TCL is a string based language there are quite a few object-oriented extensions for it like Snit3, incr Tcl4, and XOTcl5 to name a few. TCL is embeddable: its interpreter is implemented as a library of C procedures that can easily be incorporated into applications, and each application can extend the core TCL features with additional commands specific to that application.

Tcl was originally developed as a reusable command language for experimental computer aided design (CAD) tools. The interpreter is implemented as a C library that could be linked into any application. It is very easy to add new functions to the TCL interpreter, so it is an ideal reusable "macro language" that can be integrated into many applications. However, TCL is a programming language in its own right, which can be roughly described as a cross-breed between

- LISP/Scheme (mainly for its tail-recursion capabilities)
- C (control structure keywords, expr syntax) and
- Unix shells (but with more powerful structuring).

TCL Structure

The TCL language has a tiny syntax - there is only a single command structure, and a set of rules to determine how to interpret the commands. Other languages have special syntaxes for control structures (if, while, repeat...) - not so in TCL. All such structures are implemented as commands. There is a runtime library of compiled ’C’ routines, and the ’level’ of the GUI interface is quite high.

Comments: If the first character of a command is #, it is a comment.

TCL commands: TCL commands are just words separated by spaces. Commands return strings, and arguments are just further words.

command argument command argument

Spaces are important

expr 5*3 has a single argument expr 5 * 3 has three arguments

TCL commands are separated by a new line, or a semicolon, and arrays are indexed by text

set a(a\ text\ index) 4

Syntax

Syntax is just the rules how a language is structured. A simple syntax of English could say(Ignoring punctuation for the moment) A text consists of one or more sentences A sentence consists of one or more words' Simple as this is, it also describes Tcl's syntax very well - if you say "script" for "text", and "command" for "sentence". There's also the difference that a Tcl word can again contain a script or a command. So if {$x < 0} {set x 0} is a command consisting of three words: if, a condition in braces, a command (also consisting of three words) in braces. Take this for example is a well-formed Tcl command: it calls Take (which must have been defined before) with the three arguments "this", "for", and "example". It is up to the command how it interprets its arguments, e.g. puts acos(-1) will write the string "acos(-1)" to the stdout channel, and return the empty string "", while expr acos(-1) will compute the arc cosine of -1 and return 3.14159265359 (an approximation of Pi), or string length acos(-1) will invoke the string command, which again dispatches to its length sub-command, which determines the length of the second argument and returns 8.A Tcl script is a string that is a sequence of commands, separated by newlines or semicolons. A command is a string that is a list of words, separated by blanks. The first word is the name of the command; the other wordsare passed to it as its arguments.
In Tcl, "everything is a command" - even what in other languages would be called declaration, definition, or control structure. A command can interpret its arguments in any way it wants - in particular, it can implement a different language, like expression. A word is a string that is a simple word, or one that begins with { and ends with the matching } (braces), or one that begins with " and ends with the matching ". Braced words are not evaluated by the parser. In quoted words, substitutions can occur before the command is called: $[A-Za-z0-9_]+ substitutes the value of the given variable. Or, if the variable name contains characters outside that regular expression, another layer of bracing helps the parser to get it right
puts "Guten Morgen, ${Schuler}!"
If the code would say $Schuler, this would be parsed as the value of variable $Sch, immediately followed by the constant string üler.(Part of) a word can be an embedded script: a string in [] brackets whose contents are evaluated as a script (see above) before the current command is called.In short: Scripts and commands contain words. Words can again contain scripts and commands. (This can lead to words more than a page long...)
Arithmetic and logic expressions are not part of the Tcl language itself, but the language of the expr command (also used in some arguments of the if, for, while commands) is basically equivalent to C's expressions, with infix operators and functions.

Rules of TCL

The following rules define the syntax and semantics of the Tcl language:
(1) Commands A Tcl script is a string containing one or more commands. Semi-colons and newlines are command separators unless quoted as described below. Close brackets are command terminators during command substitution (see below) unless quoted.
(2) Evaluation A command is evaluated in two steps. First, the Tcl interpreter breaks the command into words and performs substitutions as described below. These substitutions are performed in the same way for all commands. The first word is used to locate a command procedure to carry out the command, then all of the words of the command are passed to the command procedure. The command procedure is free to interpret each of its words in any way it likes, such as an integer, variable name, list, or Tcl script. Different commands interpret their words differently. (3) Words of a command are separated by white space (except for newlines, which are command separators).

(4) Double quotes If the first character of a word is double-quote (") then the word is terminated by the next double-quote character. If semi-colons, close brackets, or white space characters (including newlines) appear between the quotes then they are treated as ordinary characters and included in the word. Command substitution, variable substitution, and backslash substitution are performed on the characters between the quotes as described below. The double-quotes are not retained as part of the word.
(5) Braces If the first character of a word is an open brace ({) then the word is terminated by the matching close brace (}). Braces nest within the word: for each additional open brace there must be an additional close brace (however, if an open brace or close brace within the word is quoted with a backslash then it is not counted in locating the matching close brace). No substitutions are performed on the characters between the braces except for backslash-newline substitutions described below, nor do semi-colons, newlines, close brackets, or white space receive any special interpretation. The word will consist of exactly the characters between the outer braces, not including the braces themselves.
(6) Command substitution If a word contains an open bracket ([) then Tcl performs command substitution. To do this it invokes the Tcl interpreter recursively to process the characters following the open bracket as a Tcl script. The script may contain any number of commands and must be terminated by a close bracket (``]). The result of the script (i.e. the result of its last command) is substituted into the word in place of the brackets and all of the characters between them. There may be any number of command substitutions in a single word. Command substitution is not performed on words enclosed in braces.
(7) Variable substitution If a word contains a dollar-sign ($) then Tcl performs variable substitution: the dollar-sign and the following characters are replaced in the word by the value of a variable. Variable substitution may take any of the following forms:
$name
Tcl: the Tool Command language
Name is the name of a scalar variable; the name is a sequence of one or more characters that are a letter, digit, underscore, or namespace separators (two or more colons).
$name(index)
Name gives the name of an array variable and index gives the name of an element within that array. Name must contain only letters, digits, underscores, and namespace separators, and may be an empty string.

Command substitutions, variable substitutions, and backslash substitutions are performed on the characters of index.
${name}
Name is the name of a scalar variable. It may contain any characters whatsoever except for close braces. There may be any number of variable substitutions in a single word. Variable substitution is not performed on words enclosed in braces.
(8) Backslash substitution If a backslash (\) appears within a word then backslash substitution occurs. In all cases but those described below the backslash is dropped and the following character is treated as an ordinary character and included in the word. This allows characters such as double quotes, close brackets, and dollar signs to be included in words without triggering special processing. The following table lists the backslash sequences that are handled specially, along with the value that replaces each sequence.
\a
Audible alert (bell) (0x7).
\b
Backspace (0x8).
\f
Form feed (0xc).
\n

Newline (0xa).
\r
Carriage-return (0xd).
\t
Tab (0x9).
\v
Vertical tab (0xb).
\<newline>whitespace
A single space character replaces the backslash, newline, and all spaces and tabs after the newline. This backslash sequence is unique in that it is replaced in a separate pre-pass before the command is actually parsed. This means that it will be replaced even when it occurs between braces, and the resulting space will be treated as a word separator if it isn't in braces or quotes.

Contents
Literal backslash (\), no special effect.
\ooo
The digits ooo (one, two, or three of them) give an eight-bit octal value for the Unicode character that will be inserted. The upper bits of the Unicode character will be 0.
\xhh
The hexadecimal digits hh give an eight-bit hexadecimal value for the Unicode character that will be inserted. Any number of hexadecimal digits may be present; however, all but the last two are ignored (the result is always a one-byte quantity). The upper bits of the Unicode character will be 0.
\uhhhh
The hexadecimal digits hhhh (one, two, three, or four of them) give a sixteen-bit hexadecimal value for the Unicode character that will be inserted. Backslash substitution is not performed on words enclosed in braces, except for backslash newline as described above.
(9) Comments If a hash character (#) appears at a point where Tcl is expecting the first character of the first word of a command, then the hash character and the characters that follow it, up through the next newline, are treated as a comment and ignored. The comment character only has significance when it appears at the beginning of a command.
(10) Order of substitution Each character is processed exactly once by the Tcl interpreter as part of creating the words of a command.
For example, if variable substitution occurs then no further substitutions are performed on the value of the variable; the value is inserted into the word verbatim. If command substitution occurs then the nested command is processed entirely by the recursive call to the Tcl interpreter; no substitutions are performed before making the recursive call and no additional substitutions are performed on the result of the nested script. Substitutions take place from left to right, and each substitution is evaluated completely before attempting to evaluate the next. Thus, a sequence like set y [set x 0][incr x][incr x] will always set the variable y to the value, 012.
(11) Substitution and word boundaries Substitutions do not affect the word boundaries of a command. For example, during variable substitution the entire value of the variable becomes part of a single word, even if the variable's value contains spaces.

Variables and Data in TCL

As noted above, by default, variables defined inside a procedure are "local" to that procedure. And, the argument variables of the procedure contain local "copies" of the argument data used to invoke the procedure.

These local variables cannot be seen elsewhere in the script, and they only exist while the procedure is being executed. In the "getAvg" procedure above, the local variables created in the procedure are "n" "r" and "avg". TCL provides two commands to change the scope of a variable inside a procedure, the "global" command and the "upvar" command. The "global" command is used to declare that one or more variables are not local to any procedure. The value of a global variable will persist until it is explicitly changed. So, a variable which is declared with the "global" command can be seen and changed from inside any procedure which also declares that variable with the "global" command. Variables which are defined outside of any procedure are automatically global by default. The TCL "global" command declares that references to a given variable should be global rather than local. However, the "global" command does not create or set the variable … this must be done by other means, most commonly by the TCL "set" command.

For example, here is an adjusted version of our averaging procedure which saves the input list length in the global variable "currentLength" so that other parts of the script can access this information after "getAvgN" is called:

```
proc getAvgN { rList } \
{
global currentLength
set currentLength [llength $rList]
if {!$currentLength} {return 0.0}
set avg 0.0
foreach r $rList \
{
set avg [expr $avg + $r]
}
set avg [expr $avg/double($currentLength)]
return $avg
}
```

Then, this adjusted version "getAvgN" could be used elsewhere as follows

```
global currentLength
set thisList "1.0 2.0 3.0"
set a [getAvgN $thisList]
puts "List: $thisList Length: $currentLength Avg: $a"
```

We can also use global variables as an alternative to procedure arguments. For example, we can make a version of our averaging application which assumes that the input list is stored in a global variable called "currentList"

```
proc getCurrentAvg { } \
{
global currentList currentLength
set currentLength [llength $rList]
if {!$currentLength} {return 0.0}
set avg 0.0
foreach r $currentList \
{
set avg [expr $avg + $r]
}
set avg [expr $avg/double($currentLength)]
return $avg
}
```

Then, this adjusted version "getCurrentAvg" could be used elsewhere as follows

```
global currentList currentLength
set currentList "1.0 2.0 3.0"
set a [getCurrentAvg]
puts "List: $currentList Len: $currentLength Avg: $a"
```

A procedure can use global variables for persistent storage of information, including the possibility to test whether the procedure has been called previously; this is useful for procedures that might need to perform a one-time initialization. In these cases, a procedure will use a global variable which is not set anywhere else. This means, the first time the procedure is called, the global variable will not yet exist (recall that the "global" statement declares that a variable will be accessed as a global variable, but it does not define or create the variable itself).

The TCL command "info exists" will evaluate to true if the given variable exists. For example, suppose we wanted to make a version of our procedure "getAvg" which keeps an internal count of how many times it has been called. In this version, we use a global variable named "callCount_getAvg" to keep track of the number of times "getAvg" is called. Because this global variable will actually be used to store information for the specific use of the "getAvg" procedure, we need to choose a global variable name which will not be used for a similar purpose in some other procedure. The first time "getAvg" is called, the global variable does not yet exist, and must be set to zero.

```
proc getAvg { rList } \
{
global callCount_getAvg
if {![info exists callCount_getAvg]} \
{
set callCount_getAvg 0
}
incr callCount_getAvg
puts "getAvg has been called $callCount_getAvg times"
set n [llength $rList]
if {!$n}
{return 0.0}
set avg 0.0
foreach r $rList \
{
set avg [expr $avg + $r]
}
set avg [expr $avg/double($n)]
return $avg
}
```

A more flexible way to manipulate persistent data is to use global arrays rather than scalar variables. For example, instead of the procedure-specific scalar variable "callCount_getAvg" used above, we can use a general-purpose array "callCount()" which could be used to record the call counts of any number of procedures, by using the procedure name as the array index. Many nmrWish TCL scripts use global arrays in this fashion, to simplify the sharing ofmany data values between procedures. Here is a version of the "getAvg" procedure with the call count tallied in a global array location … note that an array is declared global simply by listing its name in a "global" command, exactly as for a scalar variable; no ( ) parenthesis or index values are used.

```
proc getAvg { rList } \
{
global callCount
if {![info exists callCount(getAvg)]} \
{
set callCount(getAvg) 0
}
incr callCount(getAvg)
puts "getAvg has been used $callCount(getAvg) times"
set n [llength $rList]
if {!$n} {return 0.0}
set avg 0.0
foreach r $rList \
{
set avg [expr $avg + $r]
}
set avg [expr $avg/double($n)]
return $avg
}
```

TCL Variable Scope and the upvar Command

We have already seen that TCL procedures can generate a return value as a way to pass information back to their caller. And, we have also seen that global variables can be used to share information between parts of a TCL script, and so these also serve as a mechanism for returning information to a caller. TCL includes the "upvar" command as a method for a given procedure to change the values of variables in the scope of its caller. This provides a way for a procedure to provide additional information to the caller, besides by using the procedure's return value.

In the "upvar" scheme, a procedure's caller provides the names of one or more of its own variables as arguments to the procedure. The procedure then uses the "upvar" command to map these variables from the caller onto variables in the procedure. For example, here the caller passes its variable name "count" as the first argument to procedure "getNAvg":

```
set count 0
set a [getNAvg count "1.0 2.0 3.0 4.0"]
```

Then, in this version of procedure "getNArg" the "upvar" command is used to map the first argument value "$nPtr" onto the procedure's variable called "n" … this means that whenever the procedure gets or changes the value of variable "n" it will actually be using the caller's variable "count".

```
proc getNAvg { nPtr rList } \
{
upvar $nPtr n
set n [llength $rList]
if {!$n} {return 0.0}
set avg 0.0
foreach r $rList \
{
set avg [expr $avg + $r]
}
set avg [expr $avg/double($n)]
return $avg
}
```

Control Flow: In Tcl language there are several commands that are used to alter the flow of a program. When a program is run, its commands are executed from the top of the source file to the bottom. One by one. This flow can be altered by specific commands. Commands can be executed multiple times. Some commands are conditional. They are executed only if a specific condition is met.

The if command

The if command has the following general form:

if expr1 ?then? body1 elseif expr2 ?then? body2 elseif ... ?else? ?bodyN?

The if command is used to check if an expression is true. If it is true, a body of command(s) is then executed. The body is enclosed by curly brackets.

The if command evaluates an expression. The expression must return a boolean value. In Tcl, 1, yes, true mean true and 0, no, false mean false.

```
!/usr/bin/tclsh
if yes {
puts "This message is always shown"
}
```

In the above example, the body enclosed by { } characters is always executed.

```
#!/usr/bin/tclsh
if true then {
puts "This message is always shown"
}
```

The then command is optional. We can use it if we think, it will make the code more clear. We can use the else command to create a simple branch. If the expression inside the square brackets following the if command evaluates to false, the command following the else command is automatically executed.

```
#!/usr/bin/tclsh
set sex female
if {$sex == "male"}
{
puts "It is a boy"
} else
{
puts "It is a girl"
}
```

We have a sex variable. It has "female" string. The Boolean expression evaluates to false and we get "It is a girl" in the console.

```
$ ./girlboy.tcl
It is a girl
```

We can create multiple branches using the elseif command. The elseif command tests for another condition, if and only if the previous condition was not met. Note that we can use multiple elseif commands in our tests.

```
#!/usr/bin/tclsh
# nums.tcl
puts -nonewline "Enter a number: "
flush stdout
set a [gets stdin]
if {$a < 0}
{
puts "the number is negative"
} elseif { $a == 0 }
```

```
{
puts "the numer is zero"
} else
{
puts "the number is positive"
}
```
In the above script we have a prompt to enter a value. We test the value if it is a negative number or positive or if it equals to zero. If the first expression evaluates to false, the second expression is evaluated. If the previous conditions were not met, then the body following the else commands would be executed.
```
$ ./nums.tcl
Enter a number: 2
the number is positive
$ ./nums.tcl
Enter a number: 0

the numer is zero
$ ./nums.tcl
Enter a number: -3
the number is negative
```
Running the example multiple times.

Switch command

The switch command matches its string argument against each of the pattern arguments in order. As soon as it finds a pattern that matches the string it evaluates the following body argument by passing it recursively to the Tcl interpreter and returns the result of that evaluation. If the last pattern argument is default then it matches anything. If no pattern argument matches string and no default is given, then the switch command returns an empty string.
```
#!/usr/bin/tclsh
# switch_cmd.tcl
puts -nonewline "Select a top level domain name:"
flush stdout
gets stdin domain
switch $domain
{
us { puts "United States" }
de { puts Germany }
sk { puts Slovakia }
hu { puts Hungary }
default { puts "unknown" }
}
```
In our script, we prompt for a domain name. There are several options. If the value equals for example to us the "United States" string is printed to the console. If the value does not match to any given value, the default body is executed and unknown is printed to the console.
```
$ ./switch_cmd.tcl
Select a top level domain name:sk
Slovakia
```

We have entered sk string to the console and the program responded with Slovakia.
While command: The while command is a control flow command that allows code to be executed repeatedly based on a given Boolean condition. The while command executes the commands inside the block enclosed by curly brackets. The commands are executed each time the expression is evaluated to true.

```
#!/usr/bin/tclsh
# whileloop.tcl
set i 0
set sum 0
while { $i < 10 }
{
incr i
incr sum $i
}
puts $sum
```

In the code example, we calculate the sum of values from a range of numbers. The while loop has three parts: initialization, testing, and updating. Each execution of the command is called a cycle.

```
set i 0
```

We initiate the i variable. It is used as a counter.

```
while { $i < 10 }
{
...
}
```

The expression inside the curly brackets following the while command is the second phase, the testing. The commands in the body are executed, until the expression is evaluated to false.

```
incr i
```

The last, third phase of the while loop is the updating. The counter is incremented. Note that improper handling of the while loops may lead to endless cycles.
FOR command: When the number of cycles is know before the loop is initiated, we can use the for command. In this construct we declare a counter variable, which is automatically increased or decreased in value during each repetition of the loop.

```
#!/usr/bin/tclsh
for {set i 0} {$i < 10} {incr i}
{
puts $i
}
```

In this example, we print numbers 0..9 to the console.

```
for {set i 0} {$i < 10} {incr i}
{
puts $i
}
```

There are three phases. First, we initiate the counter i to zero. This phase is done only once. Next comes the condition. If the condition is met, the command inside the for block is executed. Then comes the third phase; the counter is increased. Now we repeat phases 2 and 3 until the condition is not met and the for loop is left. In our case, when the counter i is equal to 10, the for loop stops executing.

```
$ ./forloop.tcl
0
1
2
```

```
3
4
5

6
7
8
9
```

Here we see the output of the forloop.tcl script.

The foreach command: The foreach command simplifies traversing over collections of data. It has no explicit counter. It goes through a list element by element and the current value is copied to a variable defined in the construct.

```
#!/usr/bin/tclsh
set planets { Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune }
foreach planet $planets
{
puts $planet
}
```

In this example, we use the foreach command to go through a list of planets.

```
foreach planet $planets
{
puts $planet
}
```

The usage of the foreach command is straightforward. The planets is the list that we iterate through. The planet is the temporary variable that has the current value from the list. The for each command goes through all the planets and prints them to the console.

```
$ ./planets.tcl
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
```

Running the above Tcl script gives this output.

```
#!/usr/bin/tclsh
set actresses { Rachel Weiss Scarlett Johansson Jessica Alba \
Marion Cotillard Jennifer Connelly}
foreach {first second} $actresses
{
puts "$first $second"
}
```

In this script, we iterate througn pairs of values of a list.

```
foreach {first second} $actresses
{
puts "$first $second"
}
```

We pick two values from the list at each iteration.

```
$ ./actresses.tcl
Rachel Weiss
```

```
Scarlett Johansson
Jessica Alba
Marion Cotillard
Jennifer Connelly
```
This is the output of actresses tcl script
```
#!/usr/bin/tclsh
foreach i { one two three } item {car coins rocks}
{
puts "$i $item"
}
```

We can iterate over two lists in parallel.
```
$ ./parallel.tcl
one car
two coins
three rocks
```
This is the output of the parallel.tcl script.

The break and continue commands: The break command can be used to terminate a block defined by while, for, or switch commands.
```
#!/usr/bin/tclsh
while true
{
set r [expr 1 + round(rand()*30)]
puts -nonewline "$r "
if {$r == 22} { break }
}
puts ""
```
We define an endless while loop. We use the break command to get out of this loop. We choose a random value from 1 to 30 and print it. If the value equals to 22, we finish the endless while loop.
```
set r [expr 1 + round(rand()*30)]
```
Here we calculate a random number between 1..30. The rand() is a built-in Tcl procedure. It returns a random number from 0 to 0.99999. The rand()*30 returns a random number between 0 to 29.99999. The round() procedure rounds the final number.$ ./breakcommand.tcl 28 20 8 8 12 22 .We might get something like this.The continue command is used to skip a part of the loop and continue with the next iteration of the loop. It can be used in combination with for and while commands. In the following example, we will print a list of numbers that cannot be divided by 2 without a remainder.

```
#!/usr/bin/tclsh
set num 0
while { $num < 100 }
{
incr num
if {$num % 2 == 0} { continue }
puts "$num "
}
puts ""
```
We iterate through numbers 1..99 with the while loop.
```
if {$num % 2 == 0} { continue }
```

If the expression num % 2 returns 0, the number in question can be divided by 2. The continue command is executed and the rest of the cycle is skipped. In our case, the last command of the loop is skipped and the number is not printed to the console. The next iteration is started.

Data Structures

The list is the basic Tcl data structure. A list is simply an ordered collection of stuff; numbers, words, strings, or other lists. Even commands in Tcl are just lists in which the first list entry is the name of a proc, and subsequent members of the list are the arguments to the proc. Lists can be created in several way by setting a variable to be a list of values set lst {{item 1} {item 2} {item 3}} with the split command set lst [split "item 1.item 2.item 3" "."] with the list command. set lst [list "item 1" "item 2" "item 3"] An individual list member can be accessed with the index command. The brief description of these commands is

list **?arg1? ?arg2? ... ?argN?**

makes a list of the arguments

split **string ?splitChars?**

Splits the *string* into a list of items wherever the *splitChars* occur in the code. *SplitChars* defaults to being whitespace. Note that if there are two or more *splitChars* then each one will be used individually to split the string. In other words: split "1234567" "36" would return the following list: {12 45 7}.lindex **list index**

Returns the *index*'th item from the list.

Note: lists start from 0, not 1, so the first item is at index 0, the second item is at index 1, and so on.llength **list.**Returns the number of elements in a list.The items in list can be iterated through using the foreach command.foreach **varname list body** The foreach command will execute the *body* code one time for each list item in *list*. On each pass, *varname* will contain the value of the next *list* item.In reality, the above form of foreach is the simple form, but the command is quite powerful. It will allow you to take more than one variable at a time from the list: foreach {a b} $listofpairs { ... }. You can even take a variable at a time from multiple lists! For xample: foreach a $listOfA b $listOfB { ... }

Examples

```
set x "a b c"
puts "Item at index 2 of the list {$x} is: [lindex $x 2]\n"
set y [split 7/4/1776 "/"]
puts "We celebrate on the [lindex $y 1]'th day of the [lindex $y 0]'th month\n"
set z [list puts "arg 2 is $y" ]
puts "A command resembles: $z\n"
set i 0
foreach j $x
{
puts "$j is item number $i in list x"
incr i
}
```

Adding and deleting members of a list

The commands for adding and deleting list members are

concat *?arg1 arg2 ... argn?*

Concatenates the *args* into a single list. It also eliminates leading and trailing spaces in the args and adds a single separator space between args. The *args*to concat may be either individual elements, or lists. If an *arg* is already a list, the contents of that list is concatenatedwith the other *args*.

lappend *list Name ?arg1 arg2 ... argn?*Appends the *args* to the list *listName* treating each *arg* as a list element.

linsert *list Name index arg1 ?arg2 ... argn?*Returns a new list with the new list elements inserted just before the *index* th element of *listName*. Each element argument will become a separate element of the new list. If index is less than or equal to zero, then the new elements are inserted at

the beginning of the list. If index has the value *end* , or if it is greater than or equal to the number of elements in the list, then the new elements are appended to the list.
lreplace *list Name first last ?arg1 ... argn?*Returns a new list with N elements of *listName* replaced by the *args*. If *first* is less than or equal to 0, lreplace starts replacing from the first element of the list.If *first* is greater than the end of the list, or the word end, then lreplace behaves like lappend. If there are fewer *args* than the number of positions between *first* and *last*, then the positions for which there are no *args* are deleted.
lset *varName index newValue*
The lset command can be used to set elements of a list directly, instead of using lreplace. Lists in Tcl are the right data structure to use when you have an arbitrary number of things, and you'd like to access them according to their order in the list. In C, you would use an array. In Tcl, arrays are associated arrays - hash tables, as you'll see in the coming sections. If you want to have a collection of things, and refer to the Nth thing (give me the 10th element in this group of numbers), or go through them in order via foreach. Take a look at the example code, and pay special attention to the way that sets of characters are grouped into single list elements.

Example

```
set b [list a b {c d e} {f {g h}}]
puts "Treated as a list: $b\n"
set b [split "a b {c d e} {f {g h}}"]
puts "Transformed by split: $b\n"
set a [concat a b {c d e} {f {g h}}]
puts "Concated: $a\n"
lappend a {ij K lm} ;
# Note: {ij K lm} is a single element
puts "After lappending: $a\n"

set b [linsert $a 3 "1 2 3"] ;
# "1 2 3" is a single element
puts "After linsert at position 3: $b\n"
set b [lreplace $b 3 5 "AA" "BB"]
puts "After lreplacing 3 positions with 2 values at position 3: $b\n"
```

More list commands - lsearch, lsort, lrange

Lists can be searched with the lsearch command, sorted with the lsort command, and a range of list entries can be extracted with the lrange command.
lsearch ***list pattern***
Searches *list* for an entry that matches *pattern*, and returns the index for the first match, or a -1 if there is no match. By default, lsearch uses "glob" patterns for matching. See the section on globbing.
lsort ***list***
Sorts *list* and returns a new list in the sorted order. By default, it sorts the list into alphabetic order. Note that this command returns the sorted list as a result, instead of sorting the list in place. If you have a list in a variable, the way to sort it is like so: set lst [lsort $lst] lrange ***list first last***
Returns a list composed of the *first* through *last* entries in the list. If *first* is less than or equal to 0, it is treated as the first list element. If *last* is **end** or a value greater than the number of elements in the list, it is treated as the end. If *first* is greater than *last* then an empty list is returned.

Example

```
set list [list {Washington 1789} {Adams 1797} {Jefferson 1801} \
{Madison 1809} {Monroe 1817} {Adams 1825} ]
set x [lsearch $list Washington*]
set y [lsearch $list Madison*]
incr x
incr y -1 ;# Set range to be not-inclusive
```

```
set subsetlist [lrange $list $x $y]
puts "The following presidents served between Washington and Madison"
foreach item $subsetlist {
puts "Starting in [lindex $item 1]: President [lindex $item 0] "
}
set x [lsearch $list Madison*]
set srtlist [lsort $list]
set y [lsearch $srtlist Madison*]
puts "\n$x Presidents came before Madison chronologically"
puts "$y Presidents came before Madison alphabetically"
```

Input / Output

Tcl uses objects called channels to read and write data. The channels can be created using the open or socket command. There are three standard channels available to Tcl scripts without explicitly creating them. They are automatically opened by the OS for each new application. They are stdin, stdout and stderr. The standard input, stdin, is used by the scripts to read data. The standard output, stdout, is used by scripts to write data. The standard error, stderr, is used by scripts to write error messages.In the first example, we will work with the puts command. It has the following synopsis:

```
puts ?-nonewline? ?channelId? string
```

The channelId is the channel where we want to write text. The channelId is optional. If not specified, the default stdout is assumed.

```
#!/usr/bin/tclsh
puts "Message 1"
puts stdout "Message 2"
puts stderr "Message 3"
```

The puts command writes text to the channel.

```
puts "Message 1"
puts stdout "Message 2"

puts stderr "Message 3"
```

If we do not specify the channelId, we write to stdout by default. This line does the same thing as the previous one. We only have explicitly specified the channelId.

We write to the standard error channel. The error messages go to the terminal by default.

```
$ ./printing.tcl
Message 1
Message 2
Message 3
```

Example output.

The read command: The read command is used to read data from a channel. The optional argument specifies the number of characters to read. If omitted, the command reads all of the data from the channel up to the end.

```
#!/usr/bin/tclsh
set c [read stdin 1]
while {$c != "q"}
{
puts -nonewline "$c"
set c [read stdin 1]
}
```

The script reads a character from the standard input channel and then writes it to the standard output until it encounters the q character.

```
set c [read stdin 1]
```

We read one character from the standard input channel (stdin).

```
while {$c != "q"} {
```

We continue reading characters until the q is pressed.

The gets command

The gets command reads the next line from the channel, returns everything in the line up to (but not including) the end-of-line character.

```
#!/usr/bin/tclsh
puts -nonewline "Enter your name: "
flush stdout
set name [gets stdin]
puts "Hello $name"
```

The script asks for input from the user and then prints a message. The puts command is used to print messages to the terminal. The -no newline option suppresses the new line character. Tcl buffers output internally, so characters written with puts may not appear immediately on the output file or device. The flush command forces the output to appear immediately.

```
puts -no newline "Enter your name: "
flush stdout
set name [gets stdin]
```

The gets command reads a line from a channel.

```
$ ./hello.tcl
Enter your name: Jan
Hello Jan
```

Sample output of the script.

The pwd and cd commands

Tcl has pwd and cd commands, similar to shell commands. The pwd command returns the current working directory and the cd command is used to change the working directory.

```
#!/usr/bin/tclsh
set dir [pwd]
puts $dir
cd ..
set dir [pwd]
puts $dir
```

In this script, we will print the current working directory. Then we change the working directory and print the working directory again.

```
set dir [pwd]
```

The pwd command returns the current working directory.

```
cd ..
```

We change the working directory to the parent of the current directory. We use the cd command.

```
$ ./cwd.tcl
/home/janbodnar/prog/tcl/io
/home/janbodnar/prog/tcl
```

Sample output.

The glob command

Tcl has a glob command which returns the names of the files that match a pattern.

```
#!/usr/bin/tclsh
set files [glob *.tcl]
foreach file $files
{
puts $file
}
```

The script prints all files with the .tcl extension to the console. The glob command returns a list of files that match the *.tcl pattern.

```
set files [glob *.tcl]
foreach file $files
{
puts $file
}
```

We go through the list of files and print each item of the list to the console.

```
$ ./globcmd.tcl
attributes.tcl

allfiles.tcl
printing.tcl
hello.tcl
read.tcl
files.tcl
globcmd.tcl
write2file.tcl
cwd.tcl
readfile.tcl
isfile.tcl
addnumbers.tcl
```

This is a sample output of the globcmd.tcl script.

Procedures

A procedure is a code block containing a series of commands. Procedures are called functions in many programming languages. It is a good programming practice for procedures to do only one specific task. Procedures bring modularity to programs. The proper use of procedures brings the following advantages

- Reducing duplication of code
- Decomposing complex problems into simpler pieces
- Improving clarity of the code
- Reuse of code
- Information hiding

There are two basic types of procedures: built-in procedures and user defined ones. The built-in procedures are part of the Tcl core language. For instance, the rand(), sin() and exp() are built-in procedures. The user defined procedures are procedures created with the proc keyword.The proc keyword is used to create new Tcl commands. The term procedures and commands are often used interchangeably. We start with a simple example.

```
#!/usr/bin/tclsh
proc tclver {}
{

set v [info tclversion]
puts "This is Tcl version $v"
}
tclver
```

In this script, we create a simple tclver procedure. The procedure prints the version of Tcl language. proc tclver {}
{
The new procedure is created with the proc command. The {} characters reveal that the procedure takes no arguments.

```
{
set v [info tclversion]
puts "This is Tcl version $v"
}
tclver
```
This is the body of the tclver procedure. It is executed when we execute the tclver command. The body of the command lies between the curly brackets.The procedure is called by specifying its name.
```
$ ./version.tcl
This is Tcl version 8.6
```
Sample output.

Procedure arguments: An argument is a value passed to the procedure. Procedures can take one or more arguments. If procedures work with data, we must pass the data to the procedures. In the following example, we have a procedure which takes one argument.
```
#!/usr/bin/tclsh
proc ftc {f}
{
return [expr $f * 9 / 5 + 32]
}

puts [ftc 100]
puts [ftc 0]
puts [ftc 30]
```
We create a ftc procedure which transforms Fahrenheit temperature to Celsius temperature. The procedure takes one parameter. Its name f will be used in the body of the procedure.
```
proc ftc {f} {
return [expr $f * 9 / 5 + 32]
puts [ftc 100]
```
We compute the value of the Celsius temperature. The return command returns the value to the caller. If the procedure does not execute an explicit return, then its return value is the value of the last command executed in the procedure's body.The ftc procedure is executed. It takes 100 as a parameter. It is the temperature in Fahrenheit. The returned value is used by the puts command, which prints it to the console. Output of the example.
```
$ ./fahrenheit.tcl
212
32
86
```
Next we will have a procedure which takes two arguments.
```
#!/usr/bin/tclsh
proc maximum {x y}
{
if {$x > $y}
{
return $x
}
else
{
return $y
}
}
set a 23
set b 32
set val [maximum $a $b]
puts "The max of $a, $b is $val"
```

The maximum procedure returns the maximum of two values. The method takes two arguments.

```
proc maximum {x y}
{
if {$x > $y}
{
return $x
}
else
{
return $y
}
```

Here we compute which number is greater. We define two variables which are to be compared.

```
set a 23
set b 32
set val [maximum $a $b]
```

We calculate the maximum of the two variables. This is the output of the maximum.tcl script.

```
$ ./maximum.tcl
The max of 23, 32 is 32
```

Variable number of arguments A procedure can take and process variable number of arguments. For this we use the special arguments and parameter.

```
#!/usr/bin/tclsh

proc sum {args} {
set s 0
foreach arg $args {
incr s $arg
}
return $s
}
puts [sum 1 2 3 4]
puts [sum 1 2]
puts [sum 4]
```

We define a sum procedure which adds up all its arguments. The sum procedure has a special args argument. It has a list of all values passed to the procedure.

```
proc sum {args}
{
foreach arg $args
{
incr s $arg
}
```

We go through the list and calculate the sum.

```
puts [sum 1 2 3 4]
puts [sum 1 2]
puts [sum 4]
```

We call the sum procedure three times. In the first case, it takes 4 arguments, in the second case 2, in the last case one. Output of the variable tcl script

```
$ ./variable.tcl
```

```
10
3

4
```
Implicit arguments
The arguments in Tcl procedures may have implicit values. An implicit value is used if no explicit value is provided.
```
#!/usr/bin/tclsh
proc power {a {b 2}}
{
if {$b == 2}
{
return [expr $a * $a]
}
set value 1
for {set i 0} {$i<$b} {incr i}
{
set value [expr $value * $a]
}
return $value
}
set v1 [power 5]
set v2 [power 5 4]
puts "5^2 is $v1"
puts "5^4 is $v2"
```
Here we create a power procedure. The procedure has one argument with an implicit value. We can call the procedure with one and two arguments.
```
proc power {a {b 2}} {
set v1 [power 5]
set v2 [power 5 4]
```

The second argument b, has an implicit value 2. If we provide only one argument, the power procedure then returns the value of a to the power 2.We call the power procedure with one and two arguments. The first line computes the value of 5 to the power 2. The second line value of 5 to the power 4. Output of the example.
```
$ ./implicit.tcl
5^2 is 25
5^4 is 625
```
Returning multiple values
The return command passes one value to the caller. There is often a need to return multiple values. In such cases, we can return a list.
```
#!/usr/bin/tclsh
proc tworandoms {}
{
set r1 [expr round(rand()*10)]
set r2 [expr round(rand()*10)]
return [list $r1 $r2]
}
puts [two randoms]
puts [two randoms]
puts [two randoms]
puts [two randoms]
```

We have a two randoms procedure. It returns two random integers between 1 and 10. A random integer is computed and set to the r1 variable.

```
set r1 [expr round(rand()*10)]
return [list $r1 $r2]
```

Two values are returned with the help of the list command. A sample output.

```
$ ./tworandoms.tcl

3 7
1 3
8 7
9 9
```

Recursion

Recursion, in mathematics and computer science, is a way of defining functions in which the function being defined is applied within its own definition. In other words, a recursive function calls itself to do its job. Recursion is a widely used approach to solve many programming tasks. Recursion is the fundamental approach in functional languages like Scheme, OCalm, or Clojure. Recursion calls have a limit in Tcl. There cannot be more than 1000 recursion calls. A typical example of recursion is the calculation of a factorial. Factorial n! is the product of all positive integers less than or equal to n.

```
#!/usr/bin/tclsh
proc factorial n
{
if {$n==0}
{
return 1
}
else
{
return [expr $n * [factorial [expr $n - 1]]]
}
}
# Stack limit between 800 and 1000 levels
puts [factorial 4]
puts [factorial 10]
puts [factorial 18]
```

In this code example, we calculate the factorial of three numbers.

```
return [expr $n * [factorial [expr $n - 1]]]
```

Inside the body of the factorial procedure, we call the factorial procedure with a modified argument. The procedure calls itself.

```
$ ./recursion.tcl
24
3628800
6402373705728000
```

These are the results. If we tried to compute the factorial of 100, we would receive "too many nested evaluations" error.

Scope

A variable declared inside a procedure has a procedure scope. The *scope* of a name is the region of a program text within which it is possible to refer to the entity declared by the name without qualification of the name. A variable which is declared inside a procedure has a procedure scope; it is also called a local scope. The variable is then valid only in this particular procedure.

```
#!/usr/bin/tclsh
proc test {}
```

```
{
puts "inside procedure"
#puts "x is $x"
set x 4
puts "x is $x"
}
set x 1
puts "outside procedure"
puts "x is $x"
test
puts "outside procedure"
puts "x is $x"
```

In the preceding example, we have an x variable defined outside and inside of the test procedure. Inside the test procedure, we define an x variable. The variable has local scope, valid only inside this procedure.

```
set x 4
puts "x is $x"
set x 1
puts "outside procedure"
puts "x is $x"
```

We define an x variable outside the procedure. It has a global scope. The variables do not conflict because they have different scopes.

```
$ ./scope.tcl
outside procedure
x is 1
inside procedure
x is 4
outside procedure
x is 1
```

It is possible to change the global variable inside a procedure.

```
#!/usr/bin/tclsh
proc test {}
{
upvar x y
puts "inside procedure"
puts "y is $y"
set y 4
puts "y is $y"
}
set x 1
puts "outside procedure"
puts "x is $x"
test
puts "outside procedure"
puts "x is $x"
```

We define a global x variable. We change the variable inside the test procedure. We refer to the global x variable by the name y with the upvar command.

```
upvar x y
set y 4
```

We assign a value to the local y variable and also change the value of the global x variable.

```
$ ./scope2.tcl
outside procedure
x is 1
inside procedure
y is 1
y is 4
outside procedure
x is 4
```

From the output we can see the test procedure has changed the x variable. With the global command, we can refer to global variables from procedures.

```
#!/usr/bin/tclsh
proc test {}
{
global x
puts "inside test procedure x is $x"
proc nested {}
{
global x
puts "inside nested x is $x"
}
}
set x 1
test
nested

puts "outside x is $x"
```

In the above example, we have a test procedure and a nested procedure defined within the test procedure. We refer to the global x variable from both procedures.

```
global x
puts "inside test procedure x is $x"
```

With the global command, we refer to the global x variable, defined outside the test procedure.

```
proc nested {}
{
global x
puts "inside nested x is $x"
}
```

It is possible to create nested procedures. These are procedures defined inside other procedures. We refer to the global x variable with the global command.

```
test
nested
```

We call the test procedure and its nested procedure.

```
$ ./scope3.tcl
inside test procedure x is 1
inside nested x is 1
outside x is 1
```

Strings patterns

Files

The file command manipulates file names and attributes. It has plenty of options.

```
#!/usr/bin/tclsh
puts [file volumes]
[file mkdir new]
```

The script prints the system's mounted values and creates a new directory. The file volumes command returns the absolute paths to the volumes mounted on the system.

```
puts [file volumes]
[file mkdir new]
```

The file mkdir creates a directory called new.

```
$ ./voldir.tcl
/
$ ls -d */
doc/ new/ tmp/
```

On a Linux system, there is one mounted volume—the root directory. The ls command confirms the creation of the new directory.In the following code example, we are going to check if a file name is a regular file or a directory.

```
#!/usr/bin/tclsh
set files [glob *]
foreach fl $files
{
if {[file isfile $fl]}
{
puts "$fl is a file"
}
elseif
{ [file isdirectory $fl]}
{
puts "$fl is a directory"
}
}
```

We go through all file names in the current working directory and print whether it is a file or a directory.

Using the glob command we create a list of file and directory names of a current directory. We execute the body of the if command if the file name in question is a file.

```
set files [glob *]
if {[file isfile $fl]}
{
} elseif { [file isdirectory $fl]} {
```

The file is directory command determines, whether a file name is a directory. Note that on Unix, a directory is a special case of a file. The puts command can be used to write to files.

```
#!/usr/bin/tclsh
set fp [open days w]
set days {Monday Tuesday Wednesday Thursday Friday Saturday Sunday}
puts $fp $days
close $fp
```

In the script, we open a file for writing. We write days of a week to a file. We open a file named days for writing. The open command returns a channel id. This data is going to be written to the file. We used the channel id returned by the open command to write to the file.

```
set fp [open days w]
set days {Monday Tuesday Wednesday Thursday Friday Saturday Sunday}
```

```
puts $fp $days
close $fp
```

We close the opened channel.

```
$ ./write2file.tcl
$ cat days
Monday Tuesday Wednesday Thursday Friday Saturday Sunday
```

We run the script and check the contents of the days file. In the following script, we are going to read data from a file.

```
$ cat languages
Python
Tcl
Visual Basic
Perl
Java
C
C#
Ruby
Scheme
```

We have a simple file called languages in a directory.

```
#!/usr/bin/tclsh
set fp [open languages r]
set data [read $fp]
puts -nonewline $data
close $fp
```

We read data from the supplied file, read its contents and print the data to the terminal. We create a channel by opening the languages file in a read-only mode. If we do not provide a second parameter to the read command, it reads all data from the file until the end of the file.

```
set fp [open languages r]
set data [read $fp]
puts -nonewline $data
```

We print the data to the console.

```
$ ./readfile.tcl
Python
Tcl
Visual Basic
Perl

Java
C
C#
Ruby
Scheme
```

Sample run of the readfile.tcl script.

The eof command checks for end-of-line of a supplied channel.

```
#!/usr/bin/tclsh
set fp [open languages]
while {![eof $fp]}
{
puts [gets $fp]
}
close $fp
```

We use the eof command to read the contents of a file.

```
while {![eof $fp]}
{
puts [gets $fp]
}
```

The loop continues until the eof returns true if it encounters the end of a file. Inside the body, we use the gets command to read a line from the file.

```
$ ./readfile2.tcl
Python
Tcl
Visual Basic
Perl
Java
C
Ruby
Scheme
```

Sample run of the readfile2.tcl script.The next script performs some additional file operations.

```
#!/usr/bin/tclsh
set fp [open newfile w]
puts $fp "this is new file"
flush $fp
file copy newfile newfile2
file delete newfile
close $fp
```

We open a file and write some text to it. The file is copied. The original file is then deleted.
The file copy command copies a file.

```
file copy newfile newfile2
file delete newfile
```

The original file is deleted with the file delete command. In the final example, we will work with file attributes.

```
#!/usr/bin/tclsh
set files [glob *]
set mx 0
foreach fl $files
{
set len [string length $fl]
if { $len > $mx}
{
set mx $len
}
}set fstr "%-$mx\s %-s" C#

puts [format $fstr Name Size]
set fstr "%-$mx\s %d bytes"
foreach fl $files
{
set size [file size $fl]
puts [format $fstr $fl $size]
}
```

Tk

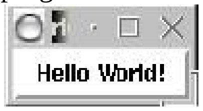
Wish - the windowing shell, is a simple scripting interface to the Tcl/Tk language. The language Tcl (Tool Command Language) is an interpreted scripting language, with useful inter-application communication methOds, and is pronounced ’tickle’. Tk originally was an X-window toolkit implemented as extensions to ’tcl’. However, now it is available native on all platforms.The program xspin is an example of a portable program in which the entire user interface is written in wish. The program also runs on PCs using NT or Win95, and as well on Macintoshes.

A first use of wish could be the following

```
manu> wish
wish> button .quit -text "Hello World!" -command {exit}
.quit
wish> pack .quit wish>
```

You can encapsulate this in a script:

If you create this as a file, and make it executable, you should be able to run this simple graphical program.



Structure of Tcl/Tk

The Tcl language has a tiny syntax - there is only a single command structure, and a set of rules to determine how to interpret the commands. Other languages have special syntaxes for control structures (if, while, repeat...) - not so in Tcl. All such structures are implemented as commands. There is a runtime library of compiled ’C’ routines, and the ’level’ of the GUI interface is quite high.

Comments: If the first character of a command is #, it is a comment

Tcl commands: Tcl commands are just words separated by spaces. Commands return strings, and arguments are just further words.

command argument argument command argument

Spaces are important

expr 5*3 has a single argument expr 5 * 3 has three arguments

Tcl commands are separated by a new line, or a semicolon, and arrays are indexed by text

set a(a\ text\ index) 4

Tcl/Tk quoting rules

The "quoting" rules come in to play when the " or { character are first in the word. ".." disables a few of the special characters - for example space, tab, newline and semicolon, and {..} disables everything except \{, \} and \nl. This facility is particularly useful for the control structures - they end up looking very like ’C’

```
While
{a==10}
{ set b [tst a]
}
```

Tcl/Tk substitution rules

Variable substitution: The dollar sign performs the variable value substitution. Tcl variables are strings.

set a 12b a will be "12b" set b 12$a b will be "1212b"

Command substitution: The []’s are replaced by the value returned by executing the Tcl command ’doit’.

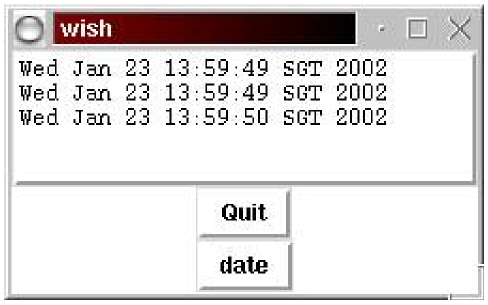
set a [doit param1 param2]

Backslash substitution:

set a a\ string\ with\ spaces\ \ and\ a\ new\ line

split <string> ?splitcharacters? concat <list> <list>
lindex <list> <index>
... + lots more
Control structures
if {test} {thenpart} {elsepart}1while {test} {body} for {init} {test} {incr} {body}
continue

% canvas <name> - optional parameter pairs ...
% button <name> - optional parameter pairs ...
% frame <name> - optional parameter pairs ...
% ... and so on
When you create a widget ".b", a new command ".b" is created, which you can use to further communicate with it. The geometry managers in Tk assemble the widgets
% pack <name> .... where ....
Tcl/Tk example software
Here is a very small Tcl/Tk application, which displays the date in a scrollable window:



```
Wed Jan 23 13:59:49 SGT 2002
Wed Jan 23 13:59:49 SGT 2002
Wed Jan 23 13:59:50 SGT 2002
```

C/Tk
In the following example, a Tcl/Tk program is integrated with a C program, giving a very small codesize GUI application, that can be compiled on any platform - Windows, UNIX or even the Macintosh platform without changes.

This section includes some extra material related to the use of Tcl/Tk for developing GUI applications. In particular - constructing menu items, using the Tk Canvas and structured data items. There are pointers to some supplied reference material. Note the following points related to trying out Tcl/Tk:If you are using cygwin-b20, the wish interpreter is called cygwish80.exe. This file is found in the directory /cygnus/cygwin-b20/H-i586-cygwin32/cygwish80.exe. Make a copy of this file in the same directory, and call it wish8.0.exe for compatibility with UNIX Tcl/Tk scripts.In the first line of your tcl files, you should put #!wish8.0If you download the file ~cs3283/ftp/demos.tar and extract it into /cygnus, you will have a series of Tcl/Tk widget examples in /cygnus/Demos. Change into the directory
/cygnus/Demos, and type ./widget.
There is a Tcl/Tk tutor, and many learn-to-program-Tcl/Tk documents available at many sites on the Internet - if you continue to have trouble, you may wish to try them.
There is no substitute for just trying to program - set yourself a small goal, and discover how to do it in Tcl/Tk.
Tcl/Tk menus

The menu strategy is fairly simple
1. Make up a frame for the menu

2. Add in the top level menu items

3. For each top level item, add in the drop-menu items

4. For each nested item, add in any cascaded menus.

5. Remember to pack it...

As an example, the following code creates a fairly conventional application with menus, a help dialog, and cascaded menu items.

Tk canvas

The Tk canvas widget allows you to draw items on a pane of the application. Items may be tagged when created, and then these tagged items may be bound to events, which may be used to manipulate the items at a later stage.This process is described in detail in Robert Biddle's "Using the Tk Canvas Facility", a copy of which is found at ~cs3283/ftp/CS-TR-94-5.pdf.Note also the use of dynamically created variable names (node$nodes).