Name of Student:		Class:
Enrollment No:		Batch:
Date of Experiment:	Date of Submission:	Submitted on:
Remarks by faculty:		Grade:
Signature of student:	Signature of Faculty:	

EXPERIMENT 7: A Case Study on TCL Language

1. Introduction

The Tool Command Language (TCL) was designed by John Ousterhout around the year 1988. There are several motivations behind the development of TCL. First, each tool had its unique languages to invoke commands. As a result, language for one particular tool cannot be used or extended to another tool. However, a general purpose programmable command language can amplify the power of a tool by allowing users to write programs in the command language to extend a tool's built-in facilities. Second, the number of interactive applications increased significantly (around 1988's) compared to the number of batch-oriented applications. Each new interactive application requires a new command language to be developed. But command languages developed tend to have insufficient power and clumsy syntax. To overcome these, John introduced the notion of "embeddable command language" and implemented it in TCL language.

2.1 Design goals of TCL

According to John Ousterhout, there are three design goals for the TCL language.

1. The language must be very simple and generic so that it can work easily with different applications and does not restrict features that applications can provide.
2. The language must be extensible so that an application can add its own features in TCL language. Moreover, the application-specific features should appear natural, as if they had been designed into the language from the start
3. The first and second goals provide the roadmap for syntactic and semantic notions of TCL languages, while the third goal is more related to machine portability and Performance.

2.2 TCL Syntax rules:

(i) **Commands**: A TCL script consists of one or more commands separated by either semi-colons or newlines, except when commands are enclosed with quotation and close brackets. We describe these exceptions later in this section.

(ii) **Evaluation**: A command is evaluated in two steps. First, a TCL interpreter breaks the command into words and performs substitutions of variables. The first word is considered as a command and the remaining words are passed as arguments of a command (or procedure). Commands might have their own interpreters for arguments.

(iii) **Words**: Words (or arguments) of a command are separated by white space, except newline and semicolon, which are command separators.

(iv) **Double quotes**: If the first character of a word is a double-quote ("), the word must be terminated by the next double-quote character. If semi-colons, close brackets, or white space characters (including newlines) appear between the quotes, they are treated as ordinary characters. Command substitution, variable substitution, and backslash substitution are performed on the characters between the quotes. The start and end quotes are not considered as part of a word.

(v) **Braces**: If the first character of a word is an open brace (i.e., {) then the word must be terminated by a matching close brace (i.e.,}). For each additional open brace located in a word, there must be an additional close brace. However, an open brace or close brace preceded by a backslash character is not counted as a matching close brace. There is no substitution on the characters between the braces except for backslash-newline substitutions. Moreover, semi-colons, close brackets, or white spaces do not have any special interpretation. The word will consist of exactly the characters between the outer braces, not including the braces themselves.

(vi) **Command substitution:** If a word resides in an open bracket (i.e., [) then TCL performs command substitution. TCL interpreter is invoked recursively to process characters following the open bracket as a TCL script until terminated by a close bracket (i.e.,]). 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 can be any number of command substitutions in a single word.

(vii) **Variable substitution**: If a word contains a dollar sign ($) followed by one of the forms described below, then TCL performs variable substitution. A variable substitution can be one of the forms: $name (a scalar variable), $name(index) (an element of an array named name at index), and ${name} (a scalar variable).

(viii) **Backslash substitution**: If backslashes (i.e. \) appear within a word, substitutions occur (e.g., \n for newline or 0xA). Only a specific set of backslash characters are substituted as described in [4].

(ix) **Comments**: TCL comments begin with a hash character (i.e., #).

(x) **Order of substitution**: Each character is processed exactly once by a TCL interpreter as part of words in a command. Substitutions occur from left to right. If command substitution occurs then the nested command is processed entirely by recursive call to the TCL interpreter. For example, command sequences set y [set x 0][incr x][incr x] will set the variable y to the value 012.

(xi) **Substitution and word boundaries**: Substitutions do not affect word boundaries of a command even if the value of a variable contains spaces.

2.3 Features of TCL built-in commands

TCL is a scripting language where most of the language features are implemented by set of built-in commands. These are often sufficient to do various programming tasks. While we describe these built-in commands, we also highlight language design goals that are obtained by commands.

Data types, variables, and assignment of variables

There are two three basic data types namely numeric, string, and list. A variable needs not be declared explicitly like other programming languages such as C. However, naming convention of a variable is similar to ANSI C. Moreover, there are no static typing of variables (i.e., a variable can take integer, float, and string type values.

Expression, operators, and operands

A TCL expression consists of a combination of operands, operators, and parentheses. If white space characters are used to separate between operands, operators, and parentheses, then they are ignored by the expression command processor. The expr command evaluates a list of arguments located in square brackets (e.g., expr [2 + 2]), and returns the value.

Conditional statement

TCL has an if command that supports single and nested if else like statement.

Table 2.1: Example of *if* commands

Program	Output

set x 1
if {$x != 1} {
  puts "$x is != 1"
} *else* {
  puts "$x is 1"
}
	1 is 1

Loop commands

TCL has two loop commands: while and for. The command evaluates test as an expression. The code in body is executed, if test is true. After the code is executed, test is evaluated again as long as test is evaluated as true.

Table 2.2: A while (left column) and for loop (right column) in TCL

while loop	for loop

set x 0
while {$x < 5} {
  set x [expr {$x + 1}]
  puts "x is $x"
}
	for {*set* x 0} {$x<5} {*incr* $x} {
	  puts "x is $x"
	}

Integrating TCL with other languages

TCL can be extended or embedded. The extension means that new commands implemented in other languages (e.g., C) can be run with TCL interpreter just like it's a built-in command in TCL. Embedding implies that TCL script can be invoked run from other languages through suitable API library functions. We describe both of the approaches in brief below for C programming language.

TCL commands implemented in C: The C code that implements a TCL command is called a command procedure. The interface to a command procedure takes an array of 9 values as inputs in a main method that corresponds to the arguments in the TCL script command. The result of the command procedure becomes the result of the TCL command. There are two kinds of command procedures: string-based and object-based. We discuss the string-based interface. Strings are generalized into the Tcl_Obj type, which can be a string or another native representation like an integer, floating number. Conversions between strings and other types are done in a lazy fashion, and the saved conversions help your scripts run more efficiently. The string-based interface to a C command procedure is much like the interface to the main program.

Table 2.6: An example implementation of TCL command in C

```
#include <stdio.h>
#include <tcl.h>

int equal (ClientData cdata, Tcl_Interp *ipointer, int argc, Tcl_Obj *CONST argv[]) {
    if (strcmp (eq_argv[1],eq_argv[2]) == 0) {
      ipointer->result = "1";
    } else {
      ipointer->result = "0";
    }
    return TCL_OK;
}
main (int argc, char *argv[]) {
  Tcl_Interp *myinterp;
  int status;
  myinterp = Tcl_CreateInterp();
  Tcl_CreateCommand (myinterp,"same",equal, (ClientData) NULL, (Tcl_CmdDeleteProc *) NULL);
  status = Tcl_Eval (myinterp,argv[1]);
}
```

3. Comparing TCL features with other languages

Table 3.1: Relation between TCL features with other languages

Language	Year	Features
Assembler	1950	1. incr command.
FORTRAN	1954	1. Parenthesis as marker for array elements.
LISP	1958	1. list as a primary data structure. 2. High level operators for list manipulation. 3. Polish prefix notation, command is always first word. 4. uplevel (LISP macros)
TRAC (Text Reckoning and Compiling)	1960	1. Everything is a string and the environment is a string to string mapping.
Multics Command Language	1965	1. Separation of commands by semicolon or newline and separation of parameters by whitespaces. 2. # as comment marker. 3. Grouping of words with double quotation.
Algol68	1968	1. pass by name (upvar).
Awk	1970	1. regular expressions, [regexp], [regsub]. 2. Associative array from Awk.
ANSI C	1972	1. for and while loop. 2. fopen, fputs, fgets, and fclose (TCL has similar name except the leading f). 3. sprintf like formatting by the format string syntax in TCL. 4. Putting code blocks in braces.
UNIX tools	1978	1. expr command and dash as switch marker.
Bourne family of shells	1987	1. Expansion of variables with $.

4. TCL features:

4.1 Uniformity

Lack of uniformity implies inconsistencies among syntaxes of a language. Weinberg defined it as "the same things should be done in the same way whenever they occur". We observe several features of TCL violate uniformity, which are mainly based on its syntax and command features.

3.2 Compactness

Compactness criterion is judged by different features of a language which allow expressing statements more concisely. We observe that TCL has several interesting features that go in favor of compactness. These can be found through the rich command sets.

4.3 Locality

The locality feature can be justified by features of the language which allows a programmer to find all parts of a code in the same place. Otherwise, one needs to go back and forth for finding any variable or function declaration. TCL has several features which allows better locality, e.g. there is no explicit variable declaration and typing. So, a variable can be set or reset.

4.4 Linearity

The linearity feature justifies how easily a program can be understood by reading it sequentially. It is well understood that branching, goto, etc. causes difficulty in understanding a program. TCL has no goto related command. However, the uplevel command allows arbitrary control structure among procedure calls. This command affects linearity considerably.

5. Conclusion and future work

This project studies the original design goals of a popular scripting language named TCL (Tool Command Language). We identify three design principles of TCL that include simple and generic language, ability to extend through other languages, and easy gluing together the extension with the language. These goals have been fulfilled by a small set of syntax rules, rich built-in set of command libraries that include structured programming notions, rich data structures and their manipulation.

Name of Student:		Class:
Enrolment No:		Batch:
Date of Experiment:	Date of Submission:	Submitted on:
Remarks by faculty:		Grade:
Signature of student:	Signature of Faculty:	

EXPERIMENT 8: To demonstrate the working of Stop N Wait ARQ

THEORY:-

Stop-and-wait ARQ also can be referred as **Alternating-Bit-Protocol** is a method used in telecommunications to send information between two connected devices. It ensures that information is not lost due to dropped packets and that packets are received in the correct order. It is the simplest kind of automatic repeat-request (ARQ) method. A stop-and-wait ARQ sender sends one frame at a time; it is a special case of the general sliding window protocol with both transmit and receive window sizes equal to 1. After sending each frame, the sender doesn't send any further frames until it receives an acknowledgement (ACK) signal. After receiving a good frame, the receiver sends an ACK. If the ACK does not reach the sender before a certain time, known as the timeout, the sender sends the same frame again.

The above behaviour is the simplest Stop-and-Wait implementation. However, in a real life implementation there are problems to be addressed.

Typically the transmitter adds a redundancy check number to the end of each frame. The receiver uses the redundancy check number to check for possible damage. If the receiver sees that the frame is good, it sends an ACK. If the receiver sees that the frame is damaged, the receiver discards it and does not send an ACK—pretending that the frame was completely lost, not merely damaged.

One problem is where the ACK sent by the receiver is damaged or lost. In this case, the sender doesn't receive the ACK, times out, and sends the frame again. Now the receiver has two copies of the same frame, and doesn't know if the second one is a duplicate frame or the next frame of the sequence carrying identical data.

Another problem is when the transmission medium has such a long latency that the sender's timeout runs out before the frame reaches the receiver. In this case the sender resends the same packet. Eventually the receiver gets two copies of the same frame, and sends an ACK for each one. The sender, waiting for a single ACK, receives two ACKs, which may cause problems if it assumes that the second ACK is for the next frame in the sequence.

To avoid these problems, the most common solution is to define a 1 bit *sequence number* in the header of the frame. This sequence number alternates (from 0 to 1) in subsequent frames. When the receiver sends an ACK, it includes the sequence number of the next packet it expects. This way, the receiver can detect duplicated frames by checking if the frame sequence numbers alternate. If two subsequent frames have the same sequence number, they are duplicates, and the second frame is discarded. Similarly, if two subsequent ACKs reference the same sequence number, they are acknowledging the same frame.

Stop-and-wait ARQ is inefficient compared to other ARQs, because the time between packets, if the ACK and the data are received successfully, is twice the transit time (assuming the turnaround time can be zero). The throughput on the channel is a fraction of what it could be. To solve this problem, one can send more than one packet at a time with a larger sequence number and use one ACK for a set. This is what is done in Go-Back-N ARQ and the Selective Repeat ARQ.

```
File  Edit  View  Search  Terminal  Help
    invoked from within
"finish"
drh@drh-OptiPlex-3010:~/Desktop$ ls
A1-stop-n-wait.nam  A1-stop-n-wait.tr      ns-allinone-2.35
A1-stop-n-wait.tcl  A3-sliding-window.tcl
drh@drh-OptiPlex-3010:~/Desktop$ nam A1-stop-n-wait.nam

Command 'nam' not found, but can be installed with:

sudo apt install nam

drh@drh-OptiPlex-3010:~/Desktop$ apt install nam
E: Could not open lock file /var/lib/dpkg/lock-frontend - open (13: Permission denied)
E: Unable to acquire the dpkg frontend lock (/var/lib/dpkg/lock-frontend), are you root?
drh@drh-OptiPlex-3010:~/Desktop$ sudo apt install nam
[sudo] password for drh:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following package was automatically installed and is no longer required:
  linux-modules-4.18.0-15-generic
Use 'sudo apt autoremove' to remove it.
The following NEW packages will be installed:
  nam
0 upgraded, 1 newly installed, 0 to remove and 120 not upgraded.
Need to get 191 kB of archives.
After this operation, 683 kB of additional disk space will be used.
Get:1 http://in.archive.ubuntu.com/ubuntu bionic/universe amd64 nam amd64 1.15-4 [191 kB]
Fetched 191 kB in 1s (166 kB/s)
Selecting previously unselected package nam.
(Reading database ... 168389 files and directories currently installed.)
Preparing to unpack .../archives/nam_1.15-4_amd64.deb ...
Unpacking nam (1.15-4) ...
Setting up nam (1.15-4) ...
Processing triggers for man-db (2.8.3-2ubuntu0.1) ...
drh@drh-OptiPlex-3010:~/Desktop$ nam A1-stop-n-wait.nam
```

```
File  Edit  View  Search  Terminal  Help
drh@drh-OptiPlex-3010:~$ ns
 When configured, ns found the right version of tclsh in /usr/bin/tclsh8.6
but it doesn't seem to be there anymore, so ns will fall back on running the first tclsh in your path. The wrong version of tclsh may break th
e test suites. Reconfigure and rebuild ns if this is a problem.
% ^Z
[1]+  Stopped                 ns
drh@drh-OptiPlex-3010:~$ ls
abc.tcl  Documents  examples.desktop  Pictures  Templates
Desktop  Downloads  Music            Public    Videos
drh@drh-OptiPlex-3010:~$ cd Desktop
drh@drh-OptiPlex-3010:~/Desktop$ ls
ns-allinone-2.35
drh@drh-OptiPlex-3010:~/Desktop$ ls
A1-stop-n-wait.tcl  A3-sliding-window.tcl  ns-allinone-2.35
drh@drh-OptiPlex-3010:~/Desktop$ ns A1-stop-n-wait.tcl
 When configured, ns found the right version of tclsh in /usr/bin/tclsh8.6
but it doesn't seem to be there anymore, so ns will fall back on running the first tclsh in your path. The wrong version of tclsh may break th
e test suites. Reconfigure and rebuild ns if this is a problem.
filtering...
ns: finish: couldn't execute "tclsh": no such file or directory
    while executing
"exec tclsh ../bin/namfilter.tcl A1-stop-n-wait.nam"
    (procedure "finish" line 7)
    invoked from within
"finish"
drh@drh-OptiPlex-3010:~/Desktop$ ls
A1-stop-n-wait.nam  A1-stop-n-wait.tr      ns-allinone-2.35
A1-stop-n-wait.tcl  A3-sliding-window.tcl
drh@drh-OptiPlex-3010:~/Desktop$ nam A1-stop-n-wait.nam

Command 'nam' not found, but can be installed with:

sudo apt install nam

drh@drh-OptiPlex-3010:~/Desktop$ apt install nam
E: Could not open lock file /var/lib/dpkg/lock-frontend - open (13: Permission denied)
E: Unable to acquire the dpkg frontend lock (/var/lib/dpkg/lock-frontend), are you root?
drh@drh-OptiPlex-3010:~/Desktop$ sudo apt install nam
```

Output: