CISC 860: Course Project

A Case Study on TCL Language

By H. Shahriar

Abstract

The Tool Command Language (TCL) is a widely used command language in today's programming world. Although it was designed and developed around 1988, it is still evolving through ongoing addition of features every year by an active community of programmers and researcher. This project aims at understanding the original design goals of TCL language, study the current features of TCL, and relate the features with initial design goals. Moreover, an analysis has been performed to identify whether TCL features violate any of general language design principles proposed by Weinberg. Our study and analysis indicate that most TCL features came from other language features and several features affects Weinberg's language design principles. Still, TCL features and implementation can be considered a success in realizing the initial design goals.

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 [1, 2]. First, while working in a research lab at Berkley, Ousterhout observed that researcher built their own tools for circuit design task. 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. Although there were command languages around that time such as UNIX shell, they tend to be tied to specific programs. Moreover, application programmers do not have the time or inclination to implement a general purpose command languages. As a result, *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. The primary idea is that an interpreter will provide a set of relatively generic facilities such as variables, control structures, and procedures. On top of these features, an application can add its own features (or new commands).

Although the development of TCL started around the Spring of 1988, the first version of TCL came to public in 1990 through the USENIX winter conference [1]. The current version of TCL is 8.6. TCL comes with a nice GUI based library called TK. This project focuses on the TCL scripting language only. We address the following issues in details:

1. The design goals of TCL and the way TCL features meet these goals.
2. We discuss if any TCL feature is similar to other languages, any feature is entirely new and influence future language.

3. We analyze TCL language features with respect to language design principles proposed by Weinberg [3]. These include uniformity, compactness, locality, linearity, and tradition.

The paper is organized as the following. Section 2 describes the overall design goals of TCL, basic syntax of TCL, different features of TCL commands, relating design goals with TCL features (*i.e.*, syntax rules and commands). Section 3 relates TCL language features with other programming languages, and discusses if TCL brings any new features. In Section 4, we analyze several features of TCL based on Weinberg's language design principles that might affect TCL programming in a bug free manner. Finally, Section 5 draws conclusions and future work of this project.

2. Design goals, syntax rules, and features of TCL

This section is organized as following. Section 2.1 describes the design goals of TCL, followed by the basic syntax rules of TCL in Section 2.2. Section 2.3 provides a brief description of the different commands of TCL. Section 2.4 relates the features of TCL with original design goals.

2.1 Design goals of TCL

According to John Ousterhout, there are three design goals for the TCL language [1, 2]. First, 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. The reason behind this is because *the language is for commands* [1]**.** Most TCL programs will be short and executed once or perhaps a few times, and then discarded. *This goal also emphasizes that the language should have a simple syntax so that it is easy to type commands.*

Second, *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. In other words, *the language should permit a simple interface to applications implemented in other languages***.** It should be easy for those applications to invoke TCL interpreter and extend TCL built-in commands with application-specific commands. It enforces TCL to support data types that are accepted to other language (*e.g.*, string type is common in C). Moreover, this design goal emphasizes that TCL is programmable, have simple set of syntax rules, and contain general programming constructs such as variables, procedures, conditionals, and loops. Therefore, users can extend built-in command sets by writing TCL procedures.

Finally, since most of the interesting functionalities will come from applications, the primary purpose of the language is to *glue together the extensions*. Therefore, TCL must have good facilities for integration, and an efficient interpreter. The interpreting mechanism of TCL commands must be fast enough to be usable for events that occur very frequently, such as mouse motion. Moreover, the interpreter must not occupy much memory, and handle internal details while integrating TCL interpreter with another languages such as garbage collection.

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

TCL language has 11 simple rules that define the syntax and semantics [4]. These minimal set of rules comply with the first design goal. We briefly describe them below:
(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). Assigning data value to a variable is performed using the *set* command. Every variable stores value by default as string type (aligned with the second design goals to extend of TCL with other languages). For example, *set name "John"* command would assign the value *John* in the variable name. We will discuss more about list data type and scoping rules of variables later in this section.

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. Various operators are allowed inside brackets such as math library functions (*e.g., sqrt*(),*cosh*()), logical operators, and bitwise operators. For example, the command *set a [expr 3*10]* assigns the value of variable *a* to *30*. Operands are first interpreted as integer values if possible; if not then a floating-point number; otherwise a string value. TCL has all numeric format supported by ANSI C.

As string is a basic data type, four operators perform textual comparison which include *eq* (equality among two strings), *ne* (inequality among two strings), *in* (a string is in a list), and *ni* (a string is not in a list). TCL has a ternary operator of the form *x?y:z*. Here, *y* is evaluated, if *x* is true. Otherwise, *z* is evaluated.

Conditional statement

TCL has an *if* command that supports single and nested if else like statement. The general syntax is *if expr1?then? body1 elseif expr2 ?then? body2 elseif ... ?else? ?bodyN?*. If a test expression (*e.g.*, *expr1*) is evaluated to true (*i.e.*, any non zero numeric value, 'true', or 'yes'), the block corresponding to *if* will be executed. If a test expression

is evaluated to false, then the word after *body1* will be examined. If the next word is *elseif*, then the next test expression will be tested as a condition. If the next word is *else*, the final *body* will be evaluated as a command. The word *else* and *then* are optional. Braces are used to group the body argument. Table 2.1 shows an example of *if* statement (the left column), which results *1 is 1* in the console (the right column).

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

The syntax of *switch* command is *switch string pattern1 body1? pattern2 body2? ... ? patternN bodyN?* Here, *string* is the string to test, and *pattern1*, *pattern2*, etc. are the patterns that the *string* is compared to. If string matches a pattern then the associated code body is executed. The command is flexible enough to compare both strings and integers. If there is no default argument and no patterns are matched, the switch command will return an empty string.

Loop commands

TCL has two loop commands: *while* and *for*. The syntax for the while loop is *while test body*. 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 shows an example while loop (the left column) that prints the value of *x* from 0 to 4.

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"
	}

The general syntax of *for* loop is *for start test next body*, which is similar to C for loop. It takes four arguments; an initialization (*start*), a test (*test*), an increment (*next*), and the body of code (*body*) to evaluate on each pass through the loop. The right column of Table 2.2 shows an example for *loop* equivalent to the while loop of the left column. A *break* within body will break out of the while loop, and execution will continue with the next line of code after body. The *continue* statement within loop body will stop the execution of the code and re-evaluate the test. TCL has a special command *incr* for incrementing loop variable (*incr varName?increment?*). By default the command increases a variable value by one.

Addition of new commands

One of the design goals (the second one) of TCL is that it must be extensible. The *proc* command is a way to achieve this goal. The syntax for the proc command is *proc name args body*. When *proc* is evaluated, it creates a new command with name *name* that takes arguments *args*, and runs the code contained in *body*. An example of procedure creation (*sum*) and its invocation is shown in the left column of Table 2.3. The bottom row of the left column shows the output.

In procedure, argument values are passed without creating any aliases. Argument values can be set to default values (unlike Perl and similar to C++). For example, in the right column of Table 2.3, *arg2* has a default value 1. Arguments can also be specified as required (*e.g., arg1*). Moreover, a procedure can accept variable number of arguments by mentioning the last argument as *args*. In this case, arguments are considered as a string list.

Table 2.3: Example of procedures having fixed and variable number of arguments

proc sum {arg1 arg2} {
  set x [*expr* {$arg1 + $arg2}];
  return $x
}
puts " Result = [sum 2 3]"
	proc sum {*required* arg1 {arg2 1} *args*} {
	  set x [*expr* {$arg1 + $arg2}];
	  return $x
	}
	puts "Result = [sum 10]"

Output: Result = 5	**Output:** Result = 11

Variable scope changes

By default a variable is local with respect to the block where it is defined (*e.g.*, in a loop body, procedure body, etc.). However, there are two commands to change variable's scope: *global* and *upvar*. The *global* command allows accessing and modifying a variable declared in main code from a procedure. The *upvar* command can be considered similar to call by reference in a procedure. It ties the name of a variable in the current scope to a variable in a different scope. The syntax for upvar is *upvar ?level? otherVar1 myVar1... ?otherVarN myVarN?* Here, the command causes *myVar1* to become a reference to *otherVar1*, and so on. The *otherVar* variable is declared to be at *level* relative to the current procedure. By default *level* is 1, which is the next level up (or the procedure that invoked the current procedure). If the level number is preceded by a # symbol, it references that many levels down from the global scope. If level is #0, the reference is to a variable at the global level.

Procedure scope changes

The *uplevel* command executes a script in a different stack frame. The syntax of the command is *uplevel* ?*level*? *arg1* ?*arg2* ...?, which concatenates all the arguments and evaluates in the variable context indicated by *level*. It was intended to create custom control structures in procedural calls. For example, suppose that procedure *x* is invoked from top-level, and it calls *y*, which calls *z*. Suppose that *z* invokes the *uplevel* command *uplevel 1 {set x 43; p}*. This results in setting 43 to *x* in the context of procedure *y* (that is one level up of current procedure) followed by invocation of procedure *p*. The *uplevel* command causes the invoking procedure to disappear from the procedure calling stack while the command is being executed.

Variable deletion

The **unset** command removes a variable definition. The syntax is *unset ?name1 ?name2?...*, which deletes variables *name1*, *name2*, etc.

List data structure

The ***list*** is a basic TCL data structure (similar to Awk, Perl, and Lisp language). A list is an ordered collection of elements. Several commands facilitate list manipulation such as accessing an element using index (*lindex*), obtaining length (*llength*), and adding elements (*linsert*).

String subcommands

Several built-in commands facilitate high level string manipulation in TCL. These include obtaining string length (*string length*), accessing an element through index (*string index*), extracting a substring between two indexes (*string range*), comparing two strings (*string compare*), identifying the first and last occurrence location of a substring (*string first* and *string last*), and matching pattern (*string match pattern*). Strings can be formatted using the *format* command, which is similar to *sprintf* function of ANSI C.

Regular expressions

There are also two commands for parsing regular expressions: *regexp* and *regsub*. The syntax for *regexp* is *regexp?switches?exp string?matchVar? ?subMatch1 ... subMatchN?*. Here, a string (*string*) that matches a regular expression (*exp*) is copied to *matchVar*. If *subMatchN* variables exist, then the parenthetical parts of the matching string are copied to the *subMatch* variables, working from left to right. The *Regsub* command not only matches an expression, but also substitutes the matched string with a new one. The syntax is *regsub?switches? exp string subSpec varName*, where an expression (*exp*) is searched in a *string*. If there is a match, then it is replaced with *subSpec*. The result is copied in *varName*.

Associative array

Like most scripting languages (Perl, Python, PHP), TCL supports associative arrays (also known as hash tables), where indexes of arrays cab be strings. There are several commands to access, create, and modify an array (through *array* command). Table 2.4 shows a list of commands with brief description.

Table 2.4: Several commands related to associative array

Command	Brief description
array exists arrayName	Returns 1 if *arrayName* is an array variable; otherwise, the command returns 0.
array names arrayName ?pattern	Returns a list of the indices for the associative array *arrayName*.
array size arrayName	Returns the number of elements in array *arrayName*.

File access

TCL has several commands to open, access, and modify files that are similar to C library function calls. The simplest methods to access a file are via *gets* and *puts*. For large amount of data to be read, it is possible to load an entire file in a string, and then parse the string with the *split* command.

Running external commands and catching error

Applications implemented in other languages can be executed by *open* and *exec* command. The *open* command runs a new program with an I/O connected to a file descriptor. If the first character in the file name argument is a pipe symbol (|), then open will treat the rest of the argument as a program name, and run that program with the standard input or output connected to a file descriptor. This pipe can be used to read output from other program or to write fresh input data to it.

The *exec* command invokes a program as a sub-process (similar to invoke an application from shell). It supports output redirection. Note that if a command in *exec* call fails, it returns an error and the error output includes the last line describing the error. The *exec* considers any output to standard error an indication of external program failure. This assumption often results in wrong conclusion that error occurred. The workaround is to either write to standard error indicating that this not an error or guard against this using the *catch* command. The *catch* command helps to catch errors and write error handling code. Table 2.5 shows an example of using *exec* and *catch* commands, where error identified through *catch* does not represent a true error. A failure can be understood by inspecting the global *errorInfo* variable.

Table 2.5: A sample program showing exec and catch commands

```
if { [catch { exec ls *.tcl } msg] } {
   puts "Something seems to have gone wrong but we will ignore it"
}
```

Executing scripts with eval

TCL allows executing one or more lines of script through the *eval* command. For example, let us consider the command, *eval [list exec ls -l] [glob *.tcl]*. In this case, the *eval* command takes two arguments enclosed in brackets, which together comprise a script. The script is passed to interpreter. The *glob* command returns a list of files that are *tcl* type. Finally, *eval* returns all file names as a list structure.

Integrating TCL with other languages

TCL can be extended or embedded [5, 6, 7]. 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, although embedding and extension can be done with other languages such as C++, FORTRAN, and Java.

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

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. One can set and query TCL variables from C using the *Tcl_SetVar* and *Tcl_GetVar* procedures. The *Tcl_LinkVar* procedure creates an alias of a TCL variable with a C variable. *Tcl_Invoke* is used to invoke a TCL command without the parsing and substitution overhead of *Tcl_Eval*. The string-based interface to a C command procedure is much like the interface to the main program.

A command implemented in C needs to be registered with TCL interpreter using the command, *Tcl_CreateCommand (interp, data, "cmd", CmdProc, DeleteProc)*. Here, *interp* is an instance of TCL interpreter (*Tcl_Interp* *), *data* is client data pointer, *cmd* is a TCL command, and *CmdProc* is the implementation of *cmd* in C. The *DeleteProc* is called when the command is destroyed (*i.e.*, when the TCL interpreter is deleted). When *cmd* is invoked, TCL calls *CmdProc* as follows: *CmdProc (data, interp, argc, argv)*. The arguments from the TCL command are available as an array of strings (*argv* parameter) and *argc* parameter holds total number of arguments. Table 2.6 shows an example C program that implements a TCL procedure named *same*, where the actual C implementation is the function *equal*.

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]);
}
```

We note that extending TCL command set using native implementation language also servers the third design principles (*i.e.*, gluing together TCL extension with host application) as TCL shell (created by Tcl_CreateInterp function) handles the details of startup and shutdown, and it provides an interactive console.

Embedding TCL scripts in C: TCL script code can be embedded and evaluated inside a C application, which means that script code can be executed from C through standard API function Tcl_Eval. As before, an instance of interpreter has to be created. Table 2.7 shows an example C program that executes two lines of TCL script: *set a [expr 5*8]* and *puts $a*.

Table 2.7: An example of embedding TCL scripts in C

```
#include <stdio.h>
#include <tcl.h>
main (int argc, char *argv[]) {
        Tcl_Interp *myinterp;
        char *action = "set a [expr 5 * 8]; puts $a";
        int status;
        myinterp = Tcl_CreateInterp();
        status = Tcl_Eval (myinterp, action);
}
```

2.4 Relationship between TCL features and design goals

In this section, we relate TCL language syntaxes and commands with the initial design goals. There are three design goals that include simplicity and generality of TCL, extensibility, and capability of gluing together extension with TCL interpreter (already discussed in Section 2.1). A brief summary of the relationship between the three goals and features is shown in Table 2.8.

The first design goal includes two parts simplicity and generic language. The simplicity lies within minimum set of syntax rules (11 rules discussed in Section 2.2). TCL is also generic due to its rich set of built in commands. These include commands for structured programming (expression evaluation, loop, if, switch), data manipulation (list and associative array processing, regular expression, file operation), and the built in rich data structure (list, associative array). Like any generic language, TCL provides necessary mechanism to define procedure, accessing global variables, creating aliases, as well as creating arbitrary accessing of procedure no matter where it is located in the stack. All these features make it generic and simple.

The second design goal (easy to extend TCL) is achieved by three ways. First, the *proc* command generates new TCL commands written in TCL language. Second, rich set of API library functions in other language such as C makes it possible to implement TCL command in another language followed by running them from TCL interpreter (shown in Section 2.3). Third, TCL script code can be run in another implementation language. We notice that arguments passing between TCL and other languages are done with *Tcl_Obj* objects that can capture any primary data types not only for TCL but also for other languages. Therefore, TCL is successful in terms of extensibility as a design objective.

The third design goal is satisfied with several features of TCL interpreter. A TCL interpreter handles internal startup and shutdown procedure during command invocation. TCL interpreter also performs several hectic low level tasks such as garbage collection. Therefore, programmers can only write the functional code as required. Even when a new TCL command is executed from other languages, interpreter states can be accessed and

controlled from the implementation language. Thus TCL glues together the extension efficiently.

Table 2.8: Mapping between TCL design goals and language features

Design goals	Features
Simple and generic language	1. Fewer syntax rules. 2. Structured programming like loop and nested if condition. 3. Data type manipulation facilities through rich set of operators (*e.g.*, string comparison, list enumeration), and file handling. 4. Procedure (*proc*) declaration and aliasing (*upvar*). 5. Advanced control structure through *uplevel* command.
The language must be extensible	1. Extension of TCL built-in command through *proc* command. 2. Extension by having command implementation in other implementation language and register them as TCL commands. 3. Embedding TCL script code and run them with TCL interpreter in an application implemented in another language. 4. Data types passing between TCL and other languages are passed as a generic Tcl_Obj to wrap almost all native data types into TCL types and vice versa.
Gluing together extension	1. Handling of startup and shutdown of TCL shell (Tcl_CreateInterp function) as well as other low level tasks such as garbage collection. 2. Conversion of data type to its nearest possible types with expression evaluator. 3. Accessing and controlling of interpreter states with API functions.

3. Comparing TCL features with other languages

This section describes features of TCL that are similar to other programming or scripting languages. We also discuss if any feature is novel in TCL that has not appeared in other languages or influence any future programming language.

Table 3.1 shows a non-exhaustive summary of TCL features that are similar to other previous languages. Array indexing using parenthesis in TCL is similar to FORTRAN language. The list data structure and list processing features mainly follows the Lisp language. The *uplevel* command in TCL is similar to LIST macros. The basic notion that every data is by default a string is derived from the TRAC language. Several features such as command separation by semicolon or newline, using '#' character as comment marker, grouping of words using quotations have been taken from the Multics command language. The *upvar* feature is similar to pass by name in Algol68. The regular

expression, pattern matching, and associative array features are common to the Awk language.

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 $.

Since TCL has been implemented in ANSI C, many features are similar to C. For example, string formatting (*sprintf* like function), for and while loop structure, and file processing commands work like C language. Moreover, loop and if-else structure body are written in braces in TCL (similar to C as well). Before TCL language was developed, UNIX already had a tool that evaluated expressions using *expr*, although in TCL the expr process is more powerful. Replacing a variable with its value through the use of '$' symbol is common in Bourne shell family. It can be inferred that almost all the features of TCL are brought from previous languages.

4. TCL features and Weinberg's language design principle

Weinberg [3] addressed several language design principles that affect programmers while learning a new language and correctly writing code. Sometimes, bugs remain silent, and programmers need considerable time to discover the problems and workaround solutions. These include uniformity, compactness, locality, linearity, tradition, and innovation. We describe the first five principles for TCL features in Section 4.1 to 4.5.

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.

(i) **Passing argument to procedures**: In TCL, as long as arguments are scalar *(i.e.*, numeric or string), their values are passed without any problem. However, if an argument is array data type, then an alias of the array **must be** created by using *upvar* command inside a procedure. Table 4.1 shows two examples of this non uniform argument passing to procedures. The program in the left column results in error. The right hand column shows a correct way to pass array in the proc print12.

Table 4.1: Non uniform rule for passing array to procedures

Code that does not pass array	Code that passes array

proc print12 {a} {
  puts "$a(1), $a(2)"
}
set array(1) "A"
set array(2) "B"
print12 $array
	proc print12 {array} {
	  upvar $array a
	  puts "$a(1), $a(2)"
	}
	set array(1) "A"
	set array(2) "B"
	Print12 array

Output: *can't read "array": variable is array*	Output: A, B

(ii) **Quoting hell**: The test expression following *if* can be enclosed within quotes or braces (the left column of Table 4.2) or not (the right column of Table 4.2). The expression is evaluated just like an *expr* command if inside quotes. However, programmers must be careful in this case. If expression is enclosed within braces, it will be evaluated within the *if* command, and if enclosed within quotes it will be first substituted before evaluated. The additional substitution might result in unwanted errors. For example, the program example shown in the right column of Table 4.2 will end up stopping just evaluating the condition expression due to additional substitution. In contrast, the left column will show the result "$x is 1".

Table 4.2: Example of if commands

If expression with braces	**If expression with quotes**

set y x
if {"$$y != 1"} {
  puts "$$y is != 1"
} else {
  puts "$$y is 1"
}
	set y {[exit]}
	if "$$y != 1" {
	  puts "$$y is != 1"
	} else {
	  puts "$$y is 1"
	}

Output: $x is 1	Output: Script stops!

(iii) **Catching error is not error sometimes**: The *catch* command does not always distinguish between actual error and a script in progress. Let us assume the following

script code: *exec f77 -o myprog myprog.f*. Here, two cases might appear: (i) the file *myprog.f* does not exist, or (ii) exist. The first and second rows of Table 4.3 show the same program running without and with the file *myff2.f*. By observing the corresponding outputs (the second column), it is clear that the return value of the *catch* command is 1 in both cases. The only distinguishing mechanism is by looking at the value of *errorCode*.

Table 4.3: Example error messages while using catch command

Program	**Output**
set rc [catch { exec f77 -c myff2.f } msg] set errc $errorCode; set erri $errorInfo puts "rc: $rc" puts "errc: $errc" puts "erri: $erri" puts "msg: $msg"	rc: 1 errc: CHILDSTATUS 7612 1 erri: myff2.f: Error: Cannot open file myff2.f while executing "exec f77 -c myff2.f " msg: myff2.f: Error: Cannot open file myff2.f
Same as above	**rc: 1** **errc: NONE** erri: myff.f: myff: while executing "exec f77 -c myff.f " msg: myff.f: myff:

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.

(i) Several high level operations on complex data structures are done just like scalar variables. For example, it is possible to convert a list structure (*dataList*) into associative array (named by *arrayName*) by using the command *array set arrayName dataList*. Each element of a list can be iterated easily by *foreach* command (*foreach varname list body*). The *lsort* and *lrange* commands sort lists and provide a subset of lists, respectively.

(ii) The *x?y:z* operator concisely express if-then-else statement. Moreover, the *incr* command increases a variables value by 1.

(iii) Forcing a variable to become float can be done by appending a dot at the end of an operand as opposed to use an explicit float function. For example: *set x 1; set j 2; expr $x/$j*. will result in 0.5. The alternative way is to use *expr { $x / double($j) }*.

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.

(i) There is no explicit variable declaration and typing. So, a variable can be set or reset when required and it exists with its current local scope such as loop, if, and proc.

(ii) There is no need to have a forward declaration of procedures in TCL. It can reside anywhere in the source file.

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.

Multiple substitution rules related to brackets and quotes hamper linearity, especially when an expression is written in double quote. For example, the left column of Table 4.4 results in a syntax error. Here, the interpreter assumes that the first word is a command in *if,* and it calls that command, passing other words as arguments. The *if* command treats its first argument as an expression. The expression has seven arguments separated by spaces. The *expr* command will try to perform its own set of substitutions. This causes an error, because the string contains unbalanced braces and brackets. This is the effect of multiple substitutions that result annoying syntax errors and often makes it difficult to understand program.

Table 4.4: Example TCL programs having syntax error and no error

Incorrect version	**Correct version**
set myString "This is a string with \[special characters\}" if $myString=="" {puts "empty string"} …	set myString "This is a string with \[special characters\}" if {$myString==""} {puts "empty string"} …

4.5 Tradition

Tradition identifies whether any language syntax or design principles are considered usual with respect to other languages. Several features are brought from contemporary or past language as a tradition. For example, TCL accepts white space character as argument separator. Moreover, it does not differentiate between single and multiple space characters. Statements are separated by new line characters or semicolon. We have discussed features similar to previous language in Section 3. Here, we discuss some more features that are not related to traditional languages.

(i) **Closing braces in if and loop command:** When braces are used for grouping, the newline is not treated as the end of a TCL command. However, the opening brace **must be** on the line with the ***for*** command. Otherwise, the TCL interpreter treats the close of the next brace as the end of the command, and it results an error. This is different than other languages like C or Perl, where it doesn't matter where you place your braces.

(ii) White space is often significant: Extra space characters matters in most of the commands as they are used for separating arguments. For example: *while{$x<5}{* will result in syntax errors as the entire string is considered as one word. In contrast, in C, C++, or Java, it does not matter whether one places spaces or not after *while* word. Thus TCL parser is more sensitive to white space than, say, the C compiler. With no white spaces between the braces and keywords, TCL treats the entire character string as a potential command.

(iii) **Indexing in list**: In list data structure, indexing starts from 0, not from 1 (Similar to C).

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. We also notice that TCL command features are blend of other previous languages. Moreover, extending command sets in TCL through other languages are flexible and easy; many lower level tasks such as garbage collection are not dealt by programmer. These combined features make TCL a powerful, generic purpose, and extendable command language.

We analyze TCL language syntaxes and related command features against language design principles (proposed by Weinberg) such as uniformity, compactness, locality, linearity, and tradition. While TCL has some nice appealing features, some of them violate these design principles. These are primarily due to multiple parsing of words, multiples substitutions, incorrect use of quotations and braces, and conservative assumption on error catching. Instead of these limitations, we can infer that the original goal of having an *embeddable command language* has been achieved by its design and related features.

The future work of this project is to study more features of TCL to identify if they strengthen or weaker original design principles. The work is not an exhaustive analysis TCL as the language itself is evolving in terms of novel features and commands every year. It is also important to analyze novel features of TCL against general language design principles. Furthermore, it will be interesting to explore whether inherent features and design of TCL make it as a milestone in programming languages.

6. References

[1] John K. Ousterhout, “Tcl: An embeddable Command Language”, *Proceedings of Winter USENIX Conference*, 1990, pp. 133-146.

[2] John Ousterhout, “History of TCL”, May 2008, Accessed from www.tcl.tk/about/history.html,

[3] Gerald Weinberg, *The Psychology of Computer Programming*, Dorset House Publishing, 1998.

[4] TCL Built-In Command, Accessed from http://www.tcl.tk/man/tcl8.4/TclCmd

[5] TCL Command Writing, Accessed from http://psg.com/~joem/CmdWrite.html

[6] Writing TCL-based Applications in C, Accessed from http://wiki.tcl.tk/2265

[7] C Programming and TCL, Accessed from www.beedub.com/book/3rd/Cprogint.pdf