CS 520 – Principles of Programming Languages

A Report on Tcl/Tk

Balaji Ramamurthy – balajir@cs.arizona.edu **Xuchen Wang – xuchenw@cs.arizona.edu**

TABLE of CONTENTS

1. INTRODUCTION

Tcl (pronounced as "tickle" or "tee-cee-ell") is a scripting language. Tcl was created by Professor John Ousterhout of the University of California, Berkeley. Tcl is believed to be a programming language that is easy to learn. It is a general purpose scripting language originally intended to be embedded in other applications as a configuration and extension language.

Tcl is a dynamic, string oriented language, which makes it an ideal candidate for developing web applications. It also has the ability to interact seamlessly with other software and hardware, which makes it a great choice in the area of testing and automation. Tcl also has many packages and extensions that enable to integrate with the majority of the standard databases available in the market. It can also seamlessly integrate with embedded databases using these packages and extensions. Due to the presence of such fantastic features as mentioned above, Tcl is the most popular choice for embedded development. The creation of the Tk GUI Toolkit paved way for Tcl to enter the domain of desktop GUI applications development. Tcl is the leader when it comes to rapid prototyping. It also has event driven programming features that make it a suitable candidate for network programming and network applications. There are extensions that have even added object oriented functionalities to Tcl.

The most important idea behind the design of Tcl was to keep the core small and clean, and to make it adaptable to various requirements using new extensions. Hence it is a general purpose programming language that is designed to be flexible. Essentially, TCL is really a combination of two beautiful features - a scripting language and an interpreter for that very language, that is designed to be easy to embed into the most popular applications. And to make things even more attractive, Tcl/Tk is free and open source. Moreover, it works really well under all platforms - Tcl is platform independent.

2. FEATURES of Tcl/Tk

So, what makes Tcl click? Well, the answer is quite simple. It is the vast array of powerful features and functionalities that Tcl provides in the simplest of ways. The characteristics and features of Tcl will be discussed in depth and detail in the subsections that follow.

2.1. The interpreted nature of Tcl

One of the main design issues in programming language design is the choice between compilation and interpretation. Interpreted languages usually make use of code representation known as intermediate code, which combines both compilation and interpretation. In this case, a compiler usually spits out a form of bytecode or threaded code. This is the intermediate code which is then interpreted and executed by an interpreter. On the positive side, interpreted languages have a lot of advantages and flexibility over their compiler counterparts. Some features are ridiculously easy to implement in interpreted languages than in compiled languages. And some other important features are again easier to implement in interpreted languages.

Some of the major functionalities that the interpreted nature of Tcl provides are:

a. Platform Independence - Due to the pretty standard nature of the generated intermediate code, the

intermediate code can be interpreted and executed on virtually any platform (Windows/UNIX/Linux/Mac).

b. Dynamic Typing - In case of dynamic typing, type checking is performed during run-time instead of compile-time. This kind of type checking is also known as late binding. There are certain limitations to static typing (also known as early binding) which makes solutions to certain type checking problems very rigid and restrictive. Typically, static checking is rigid and a bit slow. Instead, dynamic typing may allow compilers and interpreters to run more quickly. In dynamically-typed languages, source code modifications may result in fewer checks and less code to visit again and again. Hence dynamic typing is very advantageous.

c. Easy debugging - Since it is comparatively easier to fetch source code information, debugging is all the more easier.

2.2. Syntax

Tcl has extremely simple and easy syntax. This syntax is efficiently applied in a consistent way. A Tcl script is treated as a plain vanilla string, which is nothing but a sequence of commands separated by delimiting characters such as newlines or semicolons. A command is again treated as a string. It is a list of words separated by whitespace. The first word is the name of the command itself and the words that follow are passed to it as its arguments.

Everything in Tcl is implemented as a command. This includes the implementation of all the major parts of a programming language such as declarations, definitions and even control structures and other programming language structures. All data types can be treated and used as strings, including code. Hence Tcl is a string-based command language. It has only a few fundamental constructs and relatively little syntax. The basic mechanisms are all related to string substitutions and manipulations. This basic model is extended with just a few pieces of syntax for grouping, which allows multiple words in one argument , and substitution, which is used with programming variables and nested command calls. The grouping and substitutions are the only mechanisms employed by the Tcl intepreter before it runs a command. As we all know, a regular expression is a compact way of describing complex patterns in texts. Tcl also has a very good and simple mechanism to handle regular expressions. Commands in this case are also variadic in nature. This means that the commands can take in variable number of arguments. A command can always interpret its parameters the way it wishes to. The notation followed is prefix notation.

Continuing the detailed description on the string nature of Tcl, a word is again a string. It can be a simple word. It can begin with a { and end with a matching } or it can begin with a " and end with a corresponding matching ". The parser does not evaluate words that have been braced. In quoted words, substitutions can occur before the actual command itself is called. A word or even a part of it can be an embedded script. The contents inside the [] type brackets are evaluated as a script first and then the current command is called. Thus syntactically, scripts and commands contain words and vice-versa! Arithmetic and logic expressions are not a part of the core Tcl programming language. They are a part of the language of the command known as expr.

Also, everything in Tcl can be redefined and overridden dynamically.

2.3. Scope rules in Tcl

In Tcl, the topmost level of scoping is called the global scope. This scope is outside of any procedure. Variables defined at the global scope have to be made accessible to a procedure by using the "global" command. Otherwise global variables are not accessible to the procedure. When referring to variables inside a procedure, the program will, by default, only recognize those variables that were defined in that procedure. The succinct way to put it is saying that the program will only recognize the local variables by default. Another scoping rule is that the local variables are not recognized outside their local scope, which is also called the local namespace.
However, by adding the namespace path separator '::' to the local variable without specifying a sub-namespace, the program knows that the variable is being declared and defined inside the global namespace.

The Tcl commands 'upvar' and 'uplevel' allow a procedure to modify the local variables of any procedure on the call stack. Now these are very powerful. Yet on the flip side of the coin, it is easy to use them to create code that is very hard to understand and maintain. The 'uplevel' command enables Tcl to evaluate a command in a different scope other than the current procedure.

To summarize, scoping in Tcl is highly flexible. The variable visibility is restricted to lexical (static) scope by default, but the 'uplevel' and 'upvar' commands allow procedures to interact with the enclosing functions' scopes.

2.4. Procedures

In Tcl, procedures are actually what other programming languages call procedures, subroutines or functions. Procedures always return a result. The result may also be an empty string "". For all practical purposes we can call Tcl procedures as functions. They have all the common features and attributes that functions have in any programming language. The reason for the name 'procedure' is quite intuitive. In Tcl, the 'proc' command is used to create a function. And so according to the naming conventions, these programming structures are better known as procedures in the world of Tcl.

The syntax for a procedure is as follows.

proc name {argument list} {body}

Eg. proc sum {a b} {expr {$a+$b}} (or) proc sum {a b} {return [expr {$a+$b}]}

Here, the name of the procedure created by the 'proc' command is "sum" and it takes in two arguments "a and b". It returns the sum using the 'expr' command on the two variables.

The return statement at the end of a procedure is actually quite redundant and may not be needed at all times. The procedure returns anyway upon reaching its end, returning its last result. The result in the case of our example is simply the sum computed by 'expr' upon the two variables.

2.5. Call by Value and Reference

Usually, the arguments to commands are passed by value. This is done by prefixing the variable name with '$' or as such constants are passed by value. This is similar to the call by value feature in the C

programming language. The called function has its own copy of the variable which it operates upon, and not the original variable itself. This effectively prevents illegal or unwanted manipulations on the original variable as the command will only get a copy of the value, and would not be able to change the value of the original variable.
However, in some situations the change in the original value is just what is wanted. In this case at the time of the procedure call, the name of the variable is specified (without $) and in the 'proc' we use 'upvar' to link the name to a local variable. Hence we have linked variables and the procedure can do the manipulations on the local variable version of the arguments received and still see the change in the original argument.

2.6. Conditionals:

There are basically two kinds of control structures in Tcl. Firstly there are the conditionals and then come the iterators. Finally, there are even error handling control structures and some other miscellaneous control structures. Basically, there are the following two types of conditionals in Tcl:

a. The 'if' statement looks like this:

```
if {condition1} {
    body1
} elseif {condition2} {
    body2
} else {
    body3
}
```

There can be any number of 'elseif' statements for a corresponding 'if'. Also, just like in C and other programming languages, we can also have nested ifs.

b. The 'switch' statement compares its first argument with patterns inside the 'switch' and executes the corresponding body of statements depending on which pattern matched.

```
    switch flags value {
    pattern1 body1
    pattern2 body2
    ...
}
```

If value matches pattern1, then body1 is executed.

2.7. Iterators:

Iterators are powerful control structures. The main kinds of iteration tools in Tcl are as follows:

a. The 'while' command in Tcl operates as follows:

```
while { condition } {
        body
}
```

The condition is checked for and till it is satisfied, the body of statements keep getting executed.

b. The 'for' looping command is good for traditional "for i from 1 to 10" kind of iteration.

for start test next body

'start' is where the loop variables might be initialized, 'test' being the loop testing condition, 'next' being the stepping values for the loop variable and finally 'body' is the actual body of statements to be executed on successful pass of the condition.

c. The 'foreach' command loops through the elements of a list, setting a loop variable to each element in the list.

```
foreach variable list {
    body
}
```

Here the 'variable' is tested against each item in the 'list'.

d. In Extended Tcl, there is one more iterator called 'loop', which is just a more refined and constrained form of 'for'.

All iterators can return from an iteration, break from an iteration or continue to the next iteration using 'return', 'break' and 'continue' commands.

2.8. Error and Exception handling in Tcl

Tcl is a truely interpretive language. And that means that you can get a syntax error if a user makes an illegal entry, even in the middle of a running program. Tcl's 'catch' command returns a true or false result depending on whether or not the embedded command succeeds, and if it fails it also returns the error message describing the problem, in a variable of your choice. Using the 'catch' command also enables us to carry out any necessary actions after the exception has been caught by 'catch'.

When not running interactively, the Tcl interpreter generates a stack trace upon receipt of an error. The stack trace indicates the nesting of the erroneous command within contriol structures (like while and if) and also within procedure bodys. If an error occurs deep within several levels of procedure invocations, the stack trace can be very long.

Hence Tcl possesses simple exception handling mechanisms using the exception codes returned by all command executions. All the commands defined by Tcl itself generate informative error messages on incorrect usage.

3. EMBEDDABILITY and EXTENSIBILITY

Tcl is a small language. Tcl is designed to be embedded in other applications. This can be done by embedding Tcl as a configuration and extension language. This makes a programmer's life simple with less need to learn more and more new languages. Tcl is a very complete and well designed language that can be embedded into almost any sort of application. The pressing reason is that it may not be possible to provide the same functionality purely in Tcl.

Tcl was also specifically designed to be all the more easy to extend by the addition of new primitives and extensions to a base language. These new primitives perform the same error handling and memory management activities as the original primitives. This has led to many useful extensions, such as database access. Extensions currently exist for Oracle, Sybase, Ingres, Postgresql, MySql and many others. Extensions also exist for SNMP, Motif, etc. The extensibility option allows Tcl to exploit the positives of other base programming languages. For instance a critical code may be allowed to be extended on to C using these primitives.

4. OBJECT ORIENTED?

Object Oriented features are not full fledged in native Tcl. Tcl did not originally support OOP. Tcl was largely a functional programming language. But recent versions of Tcl do support extensions which provide OOP features. Examples are the XOTcl extension to Tcl, incr Tcl, Snit, and STOOOP (Simple Tcl-Only Object-Oriented Programming).

Object oriented features are not full fledged on Tcl without proper OO extensions and this in fact also relates to Garbage Collection, which we will discuss shortly.

5. MEMORY MANAGEMENT and GARBAGE COLLECTION

Tcl has Automatic Memory Management. All Tcl data structures are dynamically allocated and fully variable in size. The programmer does not need to allocate memory or specify sizes. However, this is only true if Tcl is used as a functional language. If we introduce object oriented features into Tcl, memory management mechanism may not be necessary. However, garbage collection becomes absolutely vital if one wants to work with object oriented extensions into Tcl. Tcl has built-in garbage collection on a C-level. The main problem is that this existing approach assumes that every Tcl object has string representation. This works fine for lists, numerics and strings (since they are string based types). This does not work if you want to have handles to objects, since these handles do not have any string representation . In such cases garbage collection is not possible. Another way to view this problem is that we create only one kind of handles that are able to be garbage collected, and when garbage collection is needed in some other handle we use the trick to encapsulate this handle in a garbage collectable one. But sometimes it is just a long shot.

6. ADVANCED CONCEPTS

Tcl is an interpreted language. Also it is an on-the-fly byte-compiled language. Hence an 'interpreter' is a pivotal object. Every time Tcl is running, at least one 'interpreter' keeps running. This 'interpreter' accepts scripts and evaluates them. And "slave" interpreters can also be created to encapsulate data and processes. These slaves can again have their "sub-slaves" and so on. When an interpreter is deleted, all its global and namespaced variables are freed. An example follows to highlight this feature.

```
Eg. % interp create calc
        calc
        % calc eval {expr 4*6}
        24
        % interp delete calc
        % helper eval {expr 4*6}
        invalid command name "calc"
```

To list the slaves of an interpreter we can just issue this command:

```
% interp slaves
```

Moving to further advanced concepts. Ensembles (from Tcl 8.5 onwards) are commands that are composed out of sub-commands according to a standard pattern. Subcommands are in a dict structure called "-map", with alternating name and action. Ensembles are also a good base for implementing object oriented programming features. In that case then, the command is the name of the object, and the map contains its methods.

```
Eg. namespace ensemble create -name ensembletestcommand -map \
        {one {puts Hey} two {puts dude}}
```

The above creates a command 'ensembletestcommand' that can be invoked as follows.

```
% ensembletestcommand one
Hey
% ensembletestcommand two
dude
```

Now we move on to describe Tcl namespaces in further detail. Namespaces are like containers. They can act as containers for procedures, non-local variables, and even other namespaces too. They form a hierarchical tree structure. The root of this namespace tree is located at the global namespace named "::". Their names are built with :: as separators. For instance, ::name1::name2 is a child of ::name1. And ::name1 is in fact a child of :: (the root). In a nutshell, a namespace is a separate area. As discussed before in scoping rules, a namespace is a scope. Procedures and variables are visible and private to that scope alone.

Also, there is an increasing interest to enable threads in Tcl builds. The underlying model is that every thread keeps running in an interpreter of its own. Hence we achieve thread encapsulation. Inter-thread communication must be done with explicit methods. There also exists methods to perform IPC between multiple Tcl applications. In Tcl, we can even pass the Tcl code in the form of messages between Tck

applications running on different machines across the Internet. Hence using Tcl, client-server protocols can be developed and maintained, which makes it possible to write very tightly-integrated applications.

Last but not the least, Tcl supports event-driven programming. This is a basic requirement for GUI programming! In Tcl, we have the ability to associate the Tcl code with any variable or array element. Whenever the variable is read or written, the entire associated code is executed automatically.

And finally the most important of the advanced concepts without which Tcl would not be what it is now. This important concept is nothing but that of packages and extensions. In Tcl, packages do the job of modularizing. They modularize software and other supporting libraries. Packages can be written in pure Tcl or as wrappers for extensions that come along with one or more compiled shared libraries or Tcl scripts. Some very popular extensions are XOTcl (advanced dynamic OO extension), TclOO (the canonical object-orientation extension from 8.5 onwards), Snit (OO extension, with support for "megawidgets" in Tk), BWidget (adds useful widgets to Tk), Img (adds support for additional image file formats to Tk), sqlite (a SQL database), tcllib (a collection of pure-Tcl packages), tcltcc (a built-in C compiler - see below) and Tk (the cross-platform GUI toolkit).

7. Tk - THE CROSS-PLATFORM GUI TOOLKIT

Tk is one of the most important additions to Tcl, so much so that the resultant product name Tcl/Tk has become so much more popular than Tcl. Tk is the cross-platform graphical user interface toolkit. Tk is used to develop desktop applications and user interfaces. Tk is the standard GUI not only for Tcl, but for many other dynamic languages. Tk produces rich, native applications. And moreover, these applications can run on any platform, given the cross-platform and platform-independent nature of Tk. Tk has already been successfully ported to run on most flavors of Linux, Apple Macintosh, Unix, and Windows.

Tk provides a number of widgets. These Tk widgets are commonly required to develop desktop applications such as buttons, menus, canvas, text, frames and labels. Tk was initially a bit archaic in nature. The user interfaces which could be designed were limited and had an old look and feel to them. However since the inception of Tcl/Tk 8, it offers the much wanted native look and feel. That is, menus and buttons are displayed in the exact manner in which the "native" software renders them for any given platform.

The extensibility feature again plays a huge role in enhancing Tk. Many extensions exist and are downloadable to provide external drag-and-drop features, non-rectangular windows and also native widgets. In the Tcl/Tk 8.5 release, a new theme engine has been developed. This is known as Tk Tile. It provides better native appearance, themes and native look and feel to the applications. The Tk Tile Widget Set is essentially a re-implementation of many core Tk widgets plus several new widgets. With Tile, Tk applications achieve an appearance, look and feel that is much closer to the native platform widgets. Tile also makes the development of new widgets very easy.

Besides being platform independent (Tk is also interpreted) and fully customizable (either during widget creation or later using the 'configure' command), Tk is also fully configurable. In fact option databases exist, which store options and make it extremely easy to configure the look of an application on a parametric basis. Thus saving and executing the option databases saves a lot of time and effort in designing user interfaces and applications.

8. SUMMARY and CONCLUSION

To summarize, Tcl is an interpreted scripting language that can do much more than scripting. Tcl's various features were discussed in good detail. Tcl can be a scripting language, a functional programming language, an object oriented language (through extensions) and even a user interface design and application development tool (using Tk). It is highly embeddable, extensible and an easily debugged interpreted programming language that makes it an ideal choice for rapid application deployment, be it web applications, network applications or desktop applications.

9. REFERENCES

http://www.tcl.tk - Homepage to Tcl/Tk.