The following paper was originally published in the
Proceedings of the Fourth USENIX Tcl/Tk Workshop
Monterey, CA, July 10-13, 1996

Tksh: A Tcl Library for KornShell

Jeffrey Korn
Princeton University
Princeton, New Jersey

Tksh: A Tcl Library for KornShell

Jeffrey Korn
Princeton University
35 Olden Street
Princeton, NJ 08544
`jlk@cs.princeton.edu`

Abstract

This paper describes Tksh, an implementation of the Tcl C library written on top of the library for the new KornShell (ksh93). Tksh emulates the behavior of Tcl by using the API that is provided for extending ksh93, which is similar to the Tcl library in that it allows access to variables, functions and other state of the interpreter. This implementation requires no modification to ksh93, and allows Tcl libraries such as Tk to run on top of ksh93 unchanged, making it possible to use shell scripts in place of Tcl scripts. ksh93 is well suited for use with Tk because it is backward compatible with sh, making it both easy to learn and easy to extend existing scripts to provide a graphical user interface. Tksh is not yet another port of Tk to another language – it allows Tcl scripts to run without modification using the ksh93 internals. This makes it possible to combine Tcl and ksh93, which is useful for writing ksh93 scripts that use components that have been implemented in Tcl (such as Tk widgets).

1 Introduction

Tcl[9] is a general-purpose scripting language which can be reused for a variety of different tools. It is designed to be extensible by allowing operations to be performed through a C library interface, such as adding new commands and manipulating variables. Programs such as `expect`[8] and `Tcl-DP` have been developed this way. The most well known and one of the most useful extensions to Tcl is Tk, which allows graphical user interfaces to be developed quickly and easily.

One of the reasons that Tcl/Tk is widely used is that developing user interfaces with high-level scripting languages can often be easier and faster than using a low-level language such as C or C++. High-level languages are able to hide many bothersome details such as storage management from the user.

Tcl/Tk is not the only user interface scripting language available for UNIX systems. The Desktop KornShell (dtksh, formerly wksh), which is part of the Common Desktop Environment (CDE), is one alternative. The CDE is being (or will be) shipped as a standard part of most UNIX systems. dtksh allows users to develop Motif user interfaces using the new KornShell command and programming language (ksh93)[2]. Several reasons make KornShell a reasonable choice to be the underlying scripting language for CDE:

1. *Backward Compatibility* - A large number of users have programmed in either the Bourne Shell or KornShell. It is possible to write scripts without having to learn a new language. Furthermore, existing shell scripts can be easily adapted to provide a user interface.

2. *Conformance to Standards* - KornShell conforms to the IEEE P1003.2 and ISO 9945-2 shell standards[13]. It deals with the issues of internationalization such as error messages, collation order, and international character classes used in pattern matching.

3. *KornShell is a Powerful Language* - ksh93 has a full set of programming constructs such as functions, floating-point math, associative arrays, loops, and pattern matching. With such a rich set of features, having to spawn other processes is infrequent, making ksh93's performance good.

4. *Additional Features* - ksh93 has additional powerful features that are not typically found in other scripting languages. ksh93 is good for interactive use, with features such as command line editing and job control. It also has features that make it easy to work with processes and files, yet is not tied to UNIX and runs on other operating systems.

Despite the strengths that dtksh has as a scripting language, users often prefer to use Tk rather than the Motif API for developing user interfaces. It can be harder to develop interfaces using dtksh because the set of graphics primitives included with dtksh is similar to the interface to Motif and X Intrinsics, which were designed to be used with a low level language like C. Since Tk is not designed around these interfaces, it is both easier to use and portable to other windowing environments (such as Windows and MacOS). With the growing popularity of Tk, there have been efforts to port Tk to other programming languages so users of these languages can take advantage of Tk's features. There are versions of Tk for Perl (TkPerl[1]), Scheme (STk[4]), Python (Tkinter[3]), and ML (CamlTk[12]). Tksh has come into existence because of the growing demand for a toolkit with the strengths of ksh93 as a scripting language coupled with the strengths that Tk has for building user interfaces.

The approach taken here is similar to the approach taken in STk; Tksh is not a modification to Tk (as is TkPerl). Instead, it attacks the problem at a lower level. Since Tk is written using the set of Tcl internals, Tksh is a layer in between Tk and ksh93 that provides the interface of the Tcl C API on top of the internals of ksh93.

However, Tksh is significantly different than other Tk ports in several ways. Tksh allows unmodified Tcl scripts to be evaluated in the KornShell interpreter, providing compatibility with the large number of existing Tcl/Tk scripts. Thus, transition from Tcl/Tk to Tksh is easier since scripts do not need to be rewritten. The execution of a Tcl script shares state with the ksh93 interpreter, which means that ksh93 and Tcl scripts can be used together, sharing the same function and name space. For example, a Tcl script can specify a ksh function to be executed as a callback for a Tk event. Similarly, a Tksh script can specify a Tcl procedure to be executeded for a callback. It is therefore possible to use widgets written in Tcl (such as an open file dialog box) in a Tksh script.

Tksh can also be more convenient than Tcl/Tk for some tasks. For interactive scripting, features such as command line editing and job control are useful. Scripts that do a lot of work with processes and files are easy to construct with Tksh.

The rest of this paper is organized as follows: The next section introduces the fundamental concepts underlying Tcl and ksh93, emphasizing the differences and similarities in the languages. Following that, the use and implementation of Tksh is described. Examples of applications using Tksh will be given. We conclude with information on future directions of this work.

1.1 Tcl and ksh93

Tcl is generally recognized as an effective scripting language suitable for writing useful programs. Tcl is often compared to languages such as Perl or Python, but is rarely compared to programming with shell due to the misconception that shells are inherently inadequate for writing large programs. This notion has developed partly because most of the widely used shells lack important features that are found in languages such as Perl and Tcl. Another reason for this opinion is that shells are traditionally slow because so much time is spent in the `fork` and `exec` system calls.

ksh93, the most recent version of the KornShell, offers many new features absent in traditional shells. Functionality is similar to Tcl, and spawning external programs is infrequently necessary resulting in increased performance (for example, command substitution for built-in commands does not create a separate process). Therefore, it is worthwhile to compare ksh93 to Tcl. This section gives an overview of some of the differences and similarities between the two languages.

C Library Interface

Both Tcl and ksh93 are built on top of a C library that consists of a set of functions which manipulate the state of the interpreter. This library is useful for embedding and extending the language. The two libraries provide roughly the same set of features (such functions to manipulate variables, add commands, use hash tables and dynamic strings, etc.), but provide different interfaces.

Interpreters

Tcl has the notion of an interpreter, which is a structure that maintains the state of a Tcl script such as defined commands and variables, and the execution stack. Although only one interpreter is used in most Tcl applications, it is possible to use several interpreters in a single program. ksh93 does not have the same notion of multiple interpreters, but does have the capability to create separate name and function spaces.

Commands

In Tcl, commands can be added to an interpreter using the function `Tcl_CreateCommand`. Once a command has been created, scripts that call the command with the given name will result in a call to the given function. ksh93 allows built-in commands to be added to the interpreter in a similar manner. In both languages, the function for created commands takes an `argv` list along with a pointer for private data. On architectures that support dynamically linked shared libraries, built-ins can be added at runtime by using the `builtin` command in ksh93. Although Tcl 7.4 does not support dynamic loading of built-ins, Tcl 7.5 has this feature.

Traces

A trace is a function that is associated with an action performed on a particular Tcl variable. Traces can be set for three types of events: (1) when the value of the variable is read from, (2) when the value of the variable is written to, and (3) when the variable is unset (destroyed). ksh93 has an equivalent notion called *disciplines*. Discipline functions can be C functions or shell functions and can be stacked so that multiple discipline functions apply to a variable. At the shell level, functions whose names are in the format *variable_name.action* are discipline functions. The discipline actions `get`, `set`, and `unset` are permitted for all variables, and other discipline names can be defined using C code extensions.

Results

Commands in Tcl produce two results. First, there is an integer completion code that indicates success (with `TCL_OK`) or failure (with `TCL_ERROR`). Second, there is a field in the interpreter structure that points to a result string. The result string may either be the result of the successful command or the error message generated by an unsuccessful one. ksh93 is similar, but uses uses standard output and standard error to return strings. This model makes it possible for shell functions and built-ins to behave like external UNIX commands.

Control Structures

Tcl implements control structures such as `if`, `while`, `for` and `foreach` as regular commands. They are not handled specially by the language, as Tcl consists of very few rules for parsing commands. ksh93 has the same set of control structures, but they are part of the language.

Name Space

Tcl has a single flat name space for each interpreter. Multiple name spaces can introduced by creating multiple interpreters. ksh93 uses a hierarchical name space for variables with `.` as the separator. A compound assignment statement makes it easy to assign compound variables. For example,

```
point=(x=3 y=4)
```

assigns 3 to the variable `point.x` and 4 to the variable `point.y`. This method can be used to define data structures.

Variables

Both Tcl and ksh93 deal primarily with strings, and support associative arrays. Tcl allows a variable to indirectly reference another variable through the use of the `upvar` command. With ksh93, references can be defined by using a reference variable type. For example, defining `nameref foo=bar` will cause each subsequent operation on variable `foo` to be an operation on the variable `bar`. Reference variables make call-by-name reference possible without requiring the use of `eval`. ksh93 also has some additional variable types not present in Tcl. Floating point numbers are supported for performing arithmetic, so invoking commands such as `bc` and `awk` are no longer necessary. ksh93 also has indexed arrays in addition to associative arrays.

Expressions and Patterns

Both Tcl and ksh93 support C style expressions, including most arithmetic operators such as `<<` and `^`. With Tcl, the command `expr` is used to evaluate a string representing a C expression. In ksh93, C expressions may be used inside `((...))` notation. ksh93 also allows C style assignments such as `+=` in expressions.

Tcl uses an extended form of regular expressions for pattern matching. Although shells traditionally provide notation that is far more restrictive than regular expressions, ksh93 also uses an extended form of pattern matching that is very similar to Tcl. The pattern matcher in ksh93 includes a way to negate expressions, which is not currently possible with Tcl (for example, it is possible to invoke the command `ls !(*.o)` to list all files that don't end in `.o`).

Quoting

The quoting constructs in Tcl are designed to be easily nested. For example, `[...]` notation is used for command substitution and {...} is used for literal quoting. KornShell behaves similarly for command substitution by introducing the `$(...)` syntax to replace the `` `...` `` syntax from Bourne shell. ksh93 does not have a way to nest literal strings, but this is generally not necessary since nested evaluation is much less common in the shell. ksh93 also allows ANSI C character sequences to be expanded with the `$'...'` syntax.

Performance

Although it is hard to precisely measure language performance, ksh93 has been reported to be comparable to the speed of Perl[5]. Testing shows there is only a 20% CPU-time penalty from using ksh93 as opposed to a totally C based application[11]. Another study[14] quotes the speed of ksh as being in between the speed of Tcl and Perl.

2 Using Tksh

Tksh implements the library interface for Tcl version 7.4, and works with Tk versions 3.6 and 4.0 (support for Tcl version 7.5 and Tk version 4.1 is in development). Tksh can be either statically linked with ksh, or used as a shared library. As a shared library, Tcl functionality can be loaded by using the ksh93 `builtin` command to load the initialization command `tclinit` which initializes the Tcl library. Tk can be loaded similarly with the command `tkinit`.

Once Tk functionality is loaded, all of the the commands from Tk exist as shell builtins, and ksh93 can accept Tk commands with ksh syntax. For example, the "*Hello, world*" program can be written as follows:

```
pack $(message .b -text "Hello, world")
```

Scripts can be written that take advantage of features in ksh93. For example, the following command can be used in an interactive shell to create a button that always displays the current directory. When the button is clicked, the current directory will be set to the user's home directory:

```
pack $(button .b -textvar PWD -command cd)
```

When using Tksh, all Tcl commands are available and can be used by prepending `tcl_` to the corresponding Tcl command. For example, the Tcl command `info` has the name `tcl_info` in ksh93. The exact Tcl names are not used because many of the commands have conflicting names with ksh93 and UNIX commands. For example, Tcl contains commands such as `while` and `if`, which conflict with the corresponding keywords in ksh93. Although using Tcl commands in a ksh93 script is infrequently necessary, doing so can be more convenient for a programmer familiar with Tcl. The following example uses the Tcl command `trace` to create a window that always displays the contents of the current directory:

```
pack $(listbox .list -width 20)
tcl_trace var PWD w trace_pwd
function trace_pwd
{
        .list delete 0 end
        .list insert end *
}
```

A discipline could be used instead of a trace in the above example by eliminating the `tcl_trace` command and changing the name of `trace_pwd` to `PWD.set`.

Figure 1 on the next page has the listing of a small Tksh program. The script displays a window containing the contents of the current directory as well as a list of visited directories in an interactive shell. Double clicking on the name of a file will invoke the editor on the selection; double clicking on a directory will set the current directory to the selection. If the a command is typed into the shell that causes the current directory to change, the window is updated. Sample output is shown in Figure 2.

As the listing of the program indicates, code to set up a user interface with Tksh tends to look similar to Tcl scripts with different quoting rules. The parts that differ the most are in the implementation of the script's functionality.

Evaluation of Tcl Scripts

Tcl code can be embedded into a ksh93 script by using the `source` built-in command. By issuing the command `source` [*filename*], the file *filename* (or standard input if no file is specified) is parsed and interpreted as a Tcl script. The sourced Tcl script has access to and can change the current state of the ksh93 interpreter.

Most existing Tcl/Tk scripts can be evaluated using `source` with no modification. For example, the widget demonstration that comes with Tk works without having to make any changes. Because Tksh can interpret Tcl, it is able to work with the the large number of existing Tcl/Tk scripts. Thus, an application can be built with Tksh where the main functionality is written in ksh93, and widgets are used

```
function selectFile
{
        [[ -d "$1" ]] && cd "$1" > /dev/null
        [[ -f "$1" ]] && ${EDITOR-${VISUAL-emacs}} "$1"
}

function PWD.set        # Discipline called when directory changed
{
        nameref dir=.sh.value           # .sh.value is value being stored
        .f.ls.list delete 0 end
        .f.ls.list insert end .. *
        [[ ${VisitedDir["$dir"]} != "" ]] && return
        .f.dirs.list insert end "$dir"
        VisitedDir["$dir"]=1
}

typeset -A VisitedDir   # Associative array
set -o markdirs

pack $(frame .f)
pack $(frame .f.dirname -relief raised -bd 1) -side top -fill x
pack $(frame .f.ls) $(frame .f.dirs) -side left
label .f.dirname.label -text "Current directory: "
label .f.dirname.pwd -textvariable PWD
pack .f.dirname.label .f.dirname.pwd -side left

scrollbar .f.ls.scroll -command ".f.ls.list yview"
listbox .f.ls.list -yscroll ".f.ls.scroll set" -width 20 -setgrid 1
pack $(label .f.ls.label -text "Directory Contents") -side top
pack .f.ls.list .f.ls.scroll -side left -fill y -expand 1

scrollbar .f.dirs.scroll -command ".f.dirs.list yview"
listbox .f.dirs.list -yscroll ".f.dirs.scroll set" -width 20 -setgrid 1
pack $(label .f.dirs.label -text "Visited Directories") -side top
pack .f.dirs.list .f.dirs.scroll -side left -fill y -expand 1
bind .f.dirs.list "<Double-1>" 'cd $(selection get)'
bind .f.ls.list "<Double-1>" 'selectFile $(selection get)'

cd .
```

Figure 1: Listing of Sample Program

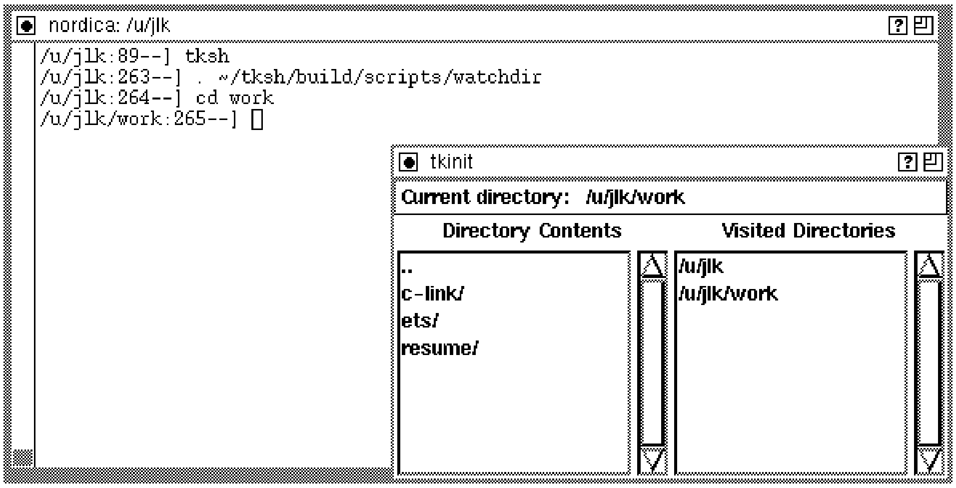

Figure 2: Screen Dump From Sample Program

that have already been written in Tcl. The transition from Tck/Tk to Tksh is therefore a smooth one. For example, if we were writing a Tksh script that required a dialog box to open file, we could take code directly from one of the many Tcl implementations rather than translate it or rewrite it from scratch.

Functions declared with `proc` will be parsed as Tcl source when they are subsequently called, even after `source` returns. For example, if we have the following script:

```
function foo
{
        X=37
        print "$(bar test)"
}

source <<'EOT'
proc bar {args} {
        global X
        set X [expr $X + 1]
        return "Proc bar: args: $args, X: $X"
}
EOT
```

A call to the function `foo` will result with the following output:

```
Proc bar: args: test, X: 38
```

Note that the Tcl procedure `bar` is able to use and modify the shell variable `X`. It is also possible for Tcl source to invoke ksh93 functions and built-ins.

Mixing Tcl and ksh93

Some Tk commands take arguments that are strings to be interpreted under certain conditions. For example, the `button` command allows a string to be specified (with the `-command` option) that will be executed when the button is pressed. A problem arises with this situation because Tksh needs to decide whether the string should be parsed as a Tcl command or a shell command. Normally, Tksh will use the current mode of the parser, but this is not always desirable. Therefore, Tksh has introduced a special notation to specify the language of the command string.

If a command string has as its first line `#!ksh`, ksh will be used, and if the first line is `#!tcl`, Tcl will be used (otherwise the current parser is used). Tksh augments the standard `bind` command in Tk (which binds a command string to a window event) with a wrapper function that automatically places the appropriate line to the beginning of the specified string (if one hasn't already been specified).

Lists

Tcl uses lists to deal with a collection of strings, whereas ksh usually uses arrays. Thus, Tcl has a built-in set of functions for dealing with lists and ksh does not. However, ksh does have the facilities that make it possible to provide lists. A list can be represented as a string if the elements of the list are quoted in such a way that splitting the list in ksh (by processing the arguments) yields the elements of the list. For example, the Tcl list `{a {b c} d}` corresponds to `'a "b c" d'` in ksh. If we were to invoke the command `eval set -- 'a "b c" d'` , `$1` would be `a`, `$2` would be `b c`, and `$3` would be `d`.

Tksh provides two different ways to deal with lists, which can be specified with the built-in command `listmode`. The first way is to use ksh style lists when parsing ksh, and Tcl style lists when parsing Tcl (this is the default). The second way is to always use Tcl style lists. This mode is useful because some Tcl applications don't use the appropriate Tcl API to manipulate lists, and assume the syntax of the list instead (for example, a function would return "`{a {b c}}`" directly instead of using the C function `Tcl_Merge` to create the list from the individual elements.

A Tcl style list can be used in a shell script by using the built-in command `setlist`. Depending on the options, the `setlist` command will convert a Tcl style list into either a ksh style list, an array, or the positional parameters ($1, $2, etc.).

Tcl Result Strings

In Tcl, commands return both a string and a completion code. In ksh, commands return only a completion code. The means by which strings are returned in ksh is by printing them to standard output. Thus, Tksh prints the contents of the result string to standard output upon successful completion of a Tcl command, and prints the result string to standard error otherwise. This allows command substitution in ksh behave like it does in Tcl. Since Tcl does not normally print the result of a command, Tksh only prints the result of a Tcl command if it is called inside command substitution.

One important difference between command substitution in Tcl versus ksh has to do with side effects. Command substitution in ksh93 behaves as if the command was executed in a separate process, which means that anything done inside substitution cannot change the state of the interpreter (the only exception to this is that built-in commands can be added). However, this is not the case with Tcl. Although not generally a problem, this occasionally leads to some unexpected behavior when substituting a Tcl command within a ksh command. Future versions of ksh93 are expected to have an option to allow side effects in command substitution.

3 Tksh Architecture

Tksh is written in C and consists of around 5,000 lines of source code along with a few modules that have been taken directly from the source of Tcl. Most of the code is straightforward and maps the semantics of a Tcl library function to the corresponding ksh93 call (if such a function exists). In several cases, there is either no corresponding function in the API, or the semantics of the similar functions differ in a nontrivial way. This requires writing code to emulate the correct semantics. Examples of this are described throughout this section.

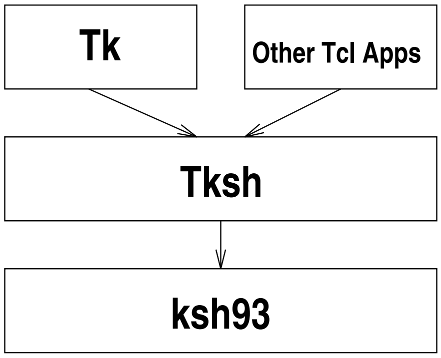

Figure 3: Tksh Architecture

The parts of code that have been taken from Tcl without requiring modification include the hash table and dynamic string libraries, the Tcl parser, the implementation of most of the builtin commands (such as `proc` and `set`), and a few miscellaneous routines. While ksh93 has a hash table and string library, it is easier to use Tcl's routines for the moment because semantics of the libraries differ.

Variables and Traces

Traces for variables are implemented on top of the discipline mechanism of ksh93. Although Tcl traces are similar to disciplines, there are a number of subtle differences in the mechanisms. For instance, when a Tcl trace is invoked on a variable, all other traces for that variable are turned off for the duration of the trace. ksh93, on the other hand, allows other disciplines to execute as long as they are further down on the discipline stack. Although it would have been possible to implement a separate tracing facility that was called from the `Tcl_GetVar` and `Tcl_SetVar` functions, using ksh93 disciplines is advantageous for several reasons reasons. If a variable is read from, written to or unset from a shell script, `Tcl_GetVar` and `Tcl_SetVar` are never called (and thus the traces wouldn't be). However, if the traces are installed as disciplines, they will be called in this case. Also, this implementation allows traces and disciplines to be interleaved on the same variable – a trace can be followed by a discipline which is followed by another trace.

Evaluation

In order to interpret Tcl scripts in ksh, an internal flag has been added to Tksh that indicates which language is currently being interpreted. If we are interpreting ksh code, the `Tcl_Eval` and `Tcl_EvalFile` routines effectively hand off the string to be evaluated to the ksh93 equivalent, `sh_eval`. If we are interpreting Tcl code, the Tcl parsing routines are called instead, which generate an `argv` list to evaluated. Before the `argv` list can then be handed off to `sh_eval`, command names are mapped appropriately – Tcl command names such as `set` and `proc` are mapped to names that won't conflict with ksh93 (`tcl_set` and `tcl_proc`).

The Tksh built-in command `source` sets the aforementioned flag to indicate Tcl code is to be evaluated, and calls `Tcl_EvalFile` with the argument. When `source` has completed, the flag is restored to its original state.

Another modification has been made to the way the Tcl command `proc` behaves. Normally, the implementation of the `proc` command creates a Tcl command with the given name that corresponds to a C function named `InterpProc`. When `InterpProc` is called, it passes a field of its private data, which contains the string of the function body, to `Tcl_Eval`. In Tksh, `InterpProc` first sets the flag indicating Tcl code is to be evaluated before calling `Tcl_Eval` on the function body. This way, a Tcl function can be called at any time (in ksh or Tcl code), not just within the `source` command.

Dynamic Scope

One important difference between ksh93 and Tcl is that Tcl uses dynamic scoping, while ksh93 uses static scoping (Tcl uses dynamic scoping in order to access an array defined in an enclosing function; ksh93 uses reference variables to accomplish this). The Tcl commands `upvar` and `uplevel` make use of such dynamic scoping, and have been rewritten in Tksh. Although the stack of enclosing scopes is inaccessible at the scripting level of ksh93, they are accessible through the C language API. Tksh uses these functions to achieve the correct behavior.

Files and Processes

Tcl has a set of commands that deal with files and processes. This part of Tcl has been rewritten in Tksh to make use of ksh93's handing of processes and files. In Tcl, a function called `Tcl_CreatePipeline` exists to start a set of processes and communicate with them (this library function is used by Tcl's `exec` command). Tksh implements this function by putting together an equivalent ksh command string from the arguments and evaluating it in the shell. For example, the Tcl command:

```
exec cat << "Hello" | tee foo >@stderr
```

would be translated to the following shell command:

```
cat <<HUP | tee foo 1>&2
Hello
HUP
```

In shell, only 10 files can be referred to at a time (by specifying a number from 0 to 9 in front of a redirection symbol). In Tcl, the number of files that can be specified at once is bound by the process limit and thus it is possible to specify more files at once in Tcl scripts. Tksh gets around this problem by mapping as many Tcl files to descriptors 0 through 9 as possible. If a file is not mapped, shell descriptors are moved around to map the file before the command is invoked, and restored afterwards. This implementation has the advantage that it allows files to be shared by ksh93 scripts and Tcl scripts, since ksh93's underlying file facilities are used for both.

Events

In Tcl/Tk, execution is driven by an event loop. Tk has a mechanism which is used to add events which will invoke a given procedure when the event occurs. ksh93, on the other hand, has it's own event mechanism that is part of the shell (the newest releases of Tcl/Tk have moved the event loop from Tk into Tcl).

The library for ksh93 contains a routine called `sh_waitnotify` which is used to specify a function to be called when the shell is waiting for either input or a child process to complete. The notify function should return when input is ready or when a signal is received (it must also return within timeout, which is specified). Tksh uses the `sh_waitnotify` mechanism to run the Tk event loop. It registers an event for the file descriptor that the shell is waiting on, as well as an event to occur after the timeout period. The function `Tk_DoOneEvent` is repeatedly called until such an event occurs. Signal handlers are also placed to trigger an event when a relevant signal occurs.

The event mechanism in ksh93 has a major advantage over Tcl's. If a command is executed, ksh93 will process events while waiting for the command to complete. Tcl, on the other hand, will freeze until the child process terminates. ksh93 is also able to parse its input as it is entered with this mechanism since it owns the event loop.

Parsing of Tcl Code

The code for Tcl is structured so well that it was possible to place Tcl's parser directly into Tksh without modification. The Tcl parser goes through a string and converts it into an argument list, performing any necessary variable or command substitutions on the way. Variable substitutions are done by using the library function `Tcl_GetVar`, which (as mentioned previously) has been rewritten to use the ksh93 API. Command substitutions are done by calling `Tcl_Eval` which recursively calls the parser.

The function `Tcl_Eval` calls the parser to generate the argument list, and then looks up the procedure to be invoked in the ksh93's built-in command table. If the command exists, it is executed with the argument list.

Patterns and Lists

The ksh93 API has a routine which converts a regular expression into a shell pattern. Shell patterns are more powerful than regular expressions, and the translation from regular expressions is fairly simple. The Tcl regular expression functions are implemented by first translating regular expressions into shell patterns, and then calling the appropriate ksh93 library function.

A string is converted into a ksh93-style list element using the function `sh_fmtq`, which returns a shell-quoted version of its input string. A list is thus created by concatenating quoted versions of each element.

Tcl Commands

Most Tcl commands that deal with internals of the interpreter are implemented using the API of the Tcl C library. As a result, such commands work with Tksh without modification. For example, the implementation of the `set` command is shown in Figure 4. The command makes use of `Tcl_SetVar`, `Tcl_GetVar`, and `Tcl_AppendResult`. Since each of these functions are implemented as part of Tksh, there is no need to make modifications to the `set` command itself. There are, however, a handful of Tcl commands that use functions which are not part of the exported interface. Commands such as `info` and `rename` are examples, and have been reimplemented for use with Tksh.

```
int Tcl_SetCmd(dummy, interp, argc, argv)
    ClientData dummy;
    register Tcl_Interp *interp;
    int argc;
    char **argv;
{
    if (argc == 2) {
        char *value;
        value = Tcl_GetVar(interp,
                argv[1], TCL_LEAVE_ERR_MSG);
        if (value == NULL) {
            return TCL_ERROR;
        }
        interp->result = value;
        return TCL_OK;
    } else if (argc == 3) {
        char *result;
        result = Tcl_SetVar(interp, argv[1],
                argv[2], TCL_LEAVE_ERR_MSG);
        if (result == NULL) {
            return TCL_ERROR;
        }
        interp->result = result;
        return TCL_OK;
    } else {
        Tcl_AppendResult(interp,
                "wrong # args: should be \"",
                argv[0]," varName ?newValue?\"",
                (char *) NULL);
        return TCL_ERROR;
    }
}
```

Figure 4: Listing of `Tcl_SetCmd`

4 Current Work

Tksh is currently being used as a foundation to write a graphical debugger for C called TkCdb. The debugger runs by spawning an execution of a line oriented debugger such as `gdb` or `dbx` and communicating with the process through a pipe. The debugger uses external programs, such as a graph drawing program (called `dotty`[7]) to display data structures. Shell scripts can be specified to be executed at breakpoints as well.

Tksh is used as the debugging language because of its strengths as an interactive command language. Debuggers are interactive programs because commands are usually entered one at a time to a prompt rather than being placed into a batch script. TkCdb takes advantage of the interactive facilities of ksh93 such as command line editing and job control. Using the command line interface to the debugger feels like being in the shell because it actually *is* a shell.

Although TkCdb has a rich set of graphical debugging features, it is less than 1,500 lines of Tksh. The simplicity, which can be attributed to using a high-level language and a powerful graphical toolkit, makes the debugger easy to understand, modify and extend. Future plans are to modify the C debugger to create a Tcl and ksh debugger.

Another project that is under development which uses Tksh is a file browser, similar to Window's Program Manager and the Finder in MacOS. The file browser is unique in that it is designed to work with an interactive shell; commands can be typed into the ksh command prompt that affect the browser, and actions triggered through the browser interface can affect the shell. For instance, the user can scroll through a list of the command history and double click on an entry to repeat that command.

5 Future Work

Support for Tcl 7.5

A significant number of changes have been made in the newest release of Tcl, version 7.5. These new features are currently being integrated into Tksh. One of the features introduced in Tcl 7.5 is a new I/O system, which is needed to support versions of Tcl on MacOS and Windows. The I/O system will be implemented for Tksh on top of the I/O system that is part of ksh93, called *sfio*[6]. sfio is a portable I/O library that is backward compatible with the C stdio library. It has functionality similar to the I/O system in Tcl 7.5, including the ability to expand the system for new file types and the ability to do nonblocking I/O. The fact that sfio is backward compatible with stdio is a major advantage.

Tcl 7.5 also provides facilities for loading dynamic libraries. ksh93 has this as well, so the proper interface should not be difficult to construct. Another major Tcl 7.5 feature is the ability to use multiple interpreters. This feature is also being developed in Tksh.

Once the support for Tcl 7.5 is complete, Tksh will be tested on non-UNIX platforms, such as Windows 95. Since Tksh has no system dependencies (the only dependencies are on the ksh93 API), it should work on any system that runs both Tk and ksh93. This includes most UNIX compatible systems, Windows NT and Windows 95.

Performance

Performance has not been measured yet with any precision. At this stage, focus is more on functionality than speed. However, when running the set of included Tk demos on top of Tksh, performance difference is not perceptible. The same is true when running Tcl scripts. One area in which performance could make a difference is in the `Tcl_GetVar` and `Tcl_SetVar` functions. Other aspects, such as the parsing of Tcl strings, will not be affected because the actual Tcl code is being used. Future work will include finding parts of Tksh in which performance can be improved.

Interface to Tk

Tksh currently provides the same interface to Tk that Tcl does. However, the use of disciplines in ksh93 makes it possible to provide a more object-oriented interface as well. For instance, suppose we wish to associate a function with a button that will be executed when the button is pressed. Currently, this might be implemented as follows:

```
button .b -text "Click here" -command foo

function foo
{
    ...
}
```

Using disciplines, we could do something like the following:

```
button .b
.b[text]="Click here"
function .b.mouseup
{
    ...
}
```

6 Summary

KornShell has many features that make it a suitable choice for writing scripts. It has functionality similar to both Tcl and Perl, good performance, and syntax that is less complex than Perl yet requires fewer levels of nested evaluation than Tcl does. It is also compatible with the Bourne Shell and conforms to

POSIX and international standards. The creation of an interface to the internals of ksh93 which is equivalent to the interface to Tcl internals allows ksh93 to be used with existing Tcl applications. The most notable of these applications is Tk. In addition, since Tksh can interpret Tcl scripts, existing Tcl/Tk scripts can be used with Tksh.

7 Availability

Tcl and Tk are available via anonymous ftp at `ftp.smli.com` in the directory `/pub/tcl`. Source to ksh93 is freely available for academic purposes (from the author), and binaries are free for non-commercial use. Binaries can be obtained through the World Wide Web at `http://www.research.att.com/orgs/ssr/book/reuse`. For information on obtaining a copy of Tksh, see the web page `http://www.cs.princeton.edu/~jlk/tksh`.

References

[1] M. Beattie, "TkPerl - A port of the Tk toolkit to Perl5", *Very High Level Languages Proceedings*, USENIX, 1994.

[2] M. Bolsky, D. Korn, *The New KornShell Command and Programming Language*, Prentice Hall, 1995.

[3] M. Conway, *A Tkinter Life Preserver*, `ftp://ftp.python.org/pub/python/doc/tkinter-doc.tar.gz`, 1994.

[4] E. Gallesio, "Embedding a Scheme Interpreter in the Tk Toolkit", *Proceedings of the Tcl/Tk 1993 Workshop*, USENIX, 1993.

[5] D. Korn, "ksh: An Extensible High Level Language", *Very High Level Languages Proceedings*, USENIX, 1994.

[6] D. Korn, P. Vo, "Sfio: Safe/Fast String/File IO", *Proceedings of Summer USENIX Conference*, USENIX, 1991.

[7] B. Krishnamurthy, *Practical Reliable UNIX Software*, Wiley, 1995.

[8] D. Libes, *Exploring Expect*, O'Reilly, 1994.

[9] J. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

[10] J. S. Pendergrast, *Desktop KornShell Graphical Programming*, Addison-Wesley, 1995.

[11] J. S. Pendergrast, *Answers to FAQ Questions*, `http://landru.unx.com/~pend/dtksh.html`, 1996.

[12] F. Pessaux, F. Rouaix, *The CamlTk Interface*, `ftp://ftp.inria.fr/lang/caml-light/camltk.dvi.tar.gz`, 1995.

[13] *POSIX - Part 2: Shell and Utilities*, IEEE Standard 1003.2, ISO/IEC 9945-2, IEEE, 1993.

[14] S. Raney, "A comparison of Tcl/Tk, the Desktop KornShell and MetaCard", `http://www.metacard.com`, 1996.

[15] B. Welch, *Practical Programming in Tcl and Tk*, Prentice Hall, 1995.