ksh - An Extensible High Level Language

David G. Korn (dgk@research.att.com)

AT&T Bell Laboratories
Murray Hill, N. J. 07974

ksh is a high level interactive script language that is a superset of the UNIX system shell. ksh has better programming features and better performance. Versions of ksh are distributed with the UNIX system by many vendors; this has created a large and growing user community in many different companies and universities. Applications of up to 25,000 lines have been written in ksh and are in production use.

ksh–93 is the first major revision of ksh in five years. Many of the changes for ksh–93 were made in order to conform to the IEEE POSIX and ISO shell standards. In addition, ksh–93 has many new programming features that vastly extend the power of shell programming. It was revised to meet the needs of a new generation of tools and graphical interfaces. Much of the impetus for ksh–93 was wksh, which allows graphical user interfaces to be written in ksh. ksh–93 includes the functionality of awk, perl, and tcl. Because ksh–93 maintains backward compatibility with earlier versions of ksh, older ksh and Bourne shell scripts should continue to run with ksh–93.

1. INTRODUCTION

One of the earliest very high level languages for the UNIX* system was the shell. The shell, originally an interactive command language, has evolved over the years to become a complete programming language. One of the most popular versions of the shell, ksh, is being used as the primary programming language in many software development projects in many different companies and universities. The most recent version of ksh, ksh–93, incorporates the functionality of awk[1] while maintaining compatibility with earlier shells and with the IEEE POSIX and ISO shell standards.[2]

In this paper I give a history of the UNIX shells, focusing on ksh, primarily the new features in the 1993 version, ksh–93. I also describe the pros and cons of using ksh, rather than awk, perl[3], or tcl[4]. One of the new features of ksh–93 is that it is extensible by applications programmers; I suggest some possible extensions.

To illustrate the programming power of the new version of ksh, I include two shell programs written by Glenn Fowler of AT&T Bell Laboratories. I compare one of these programs with the same program written in perl.

2. HISTORY

The original UNIX system shell was a simple program written by Ken Thompson at Bell Laboratories, as the interface to the new UNIX operating system. It allowed the user to invoke single commands, or to connect commands together by having the output of one command pass through a special file called a pipe and become input for the next command. The Thompson shell was designed as a command interpreter, not a programming language. While one could put a sequence of commands in a file and

---

* UNIX is a registered trademark licensed exclusively through X/OPEN Corporation

run them, i.e., create a shell script, there was no support for traditional language facilities such as flow control, variables, and functions. When the need for some flow control surfaced, the commands `/bin/if` and `/bin/goto` were created as separate commands. The `/bin/if` command evaluated its first argument and, if true, executed the remainder of the line. The `/bin/goto` command read the script from its standard input, looked for the given label, and set the seek position at that location. When the shell returned from invoking `/bin/goto`, it read the next line from standard input from the location set by `/bin/goto`.

Unlike most earlier systems, the Thompson shell command language was a user-level program that did not have any special privileges. This meant that new shells could be created by any user, which led to a succession of improved shells. In the mid-1970s, John Mashey at Bell Laboratories extended the Thompson shell by adding commands so that it could be used as a primitive programming language. He made commands such as `if` and `goto` built-ins for improved performance, and also added shell variables.

At the same time, Steve Bourne at Bell Laboratories wrote a version of the shell which included programming language techniques. A rich set of structured flow control primitives was part of the language; the shell processed commands by building a parse tree and then evaluating the tree. Because of the rich flow control primitives, there was no need for a `goto` command. Bourne introduced the "here-document" whereby the contents of a file are inserted directly into the script. One of the often overlooked contributions of the Bourne shell is that it helped to eliminate the distinction between programs and shell scripts. Earlier versions of the shell read input from standard input, making it impossible to use shell scripts as part of a pipeline.

By the late 1970s, each of these shells had sizable followings within Bell Laboratories. The two shells were not compatible, leading to a division as to which should become the standard shell. Steve Bourne and John Mashey argued their respective cases at three successive UNIX user group meetings. Between meetings, each enhanced their shell to have the functionality available in the other. A committee was set up to choose a standard shell. It chose the Bourne shell as the standard.

At the time of these so-called ‘‘shell wars’’, I worked on a project at Bell Laboratories that needed a form entry system. We decided to build a form interpreter, rather than writing a separate program for each form. Instead of inventing a new script language, we built a form entry system by modifying the Bourne shell, adding built-in commands as necessary. The application was coded as shell scripts. We added a built-in to read form template description files and create shell variables, and a built-in to output shell variables through a form mask. We also added a built-in named `let` to do arithmetic using a small subset of the C language expression syntax. An array facility was added to handle columns of data on the screen. Shell functions were added to make it easier to write modular code, since our shell scripts tended to be larger than most shell scripts at that time. Since the Bourne shell was written in an Algol-like variant of C, we converted our version of it to a more standard K&R version of C. We removed the restriction that disallowed I/O redirection of built-in commands, and added `echo`, `pwd`, and `test` built-in commands for improved performance. Finally, we added a capability to run a command as a coprocess so that the command that processed the user-entered data and accessed the database could be written as a separate process.

At the same time, at the University of California at Berkeley, Bill Joy put together a new shell called the C shell. Like the Mashey shell, it was implemented as a command interpreter, not a programming language. While the C shell contained flow control constructs, shell variables, and an arithmetic facility, its primary contribution was a better command interface. It introduced the idea of a history list and an editing facility, so that users didn’t have to retype commands that they had entered incorrectly.

I created the first version of `ksh` soon after I moved to a research position at Bell Laboratories. Starting with the form scripting language, I removed some of the form-specific code, and added useful features from the C shell such as history, aliases, and job control.

In 1982, the UNIX System V shell was converted to K&R C, `echo` and `pwd` were made built-in commands, and the ability to define and use shell functions was added. Unfortunately, the System V syntax for function definitions was different from that of `ksh`. In order to maintain compatibility with the System V shell and preserve backward compatibility, I modified `ksh` to accept either syntax.

The popular inline editing features (`vi` and `emacs` mode) of `ksh` were created by software developers at Bell Laboratories; the `vi` line editing mode by Pat Sullivan, and the `emacs` line editing mode by Mike Veach. Each had independently modified the Bourne shell to add these features, and both were in organizations that wanted to use `ksh` only if `ksh` had their respective inline editor. Originally the idea of adding command line editing to `ksh` was rejected in the hope that line editing would move into the terminal driver. However, when it became clear that this was not likely to happen soon, both line editing modes were integrated into `ksh` and made optional so that they could be disabled on systems that provided editing as part of the terminal interface.

As more and more software developers at AT&T switched to `ksh`, it became the de facto standard shell at AT&T. As developers left AT&T to go elsewhere, the demand for `ksh` led AT&T to make `ksh` source code available to external customers via the UNIX System Toolchest, an electronic software distribution system. For a one-time fixed cost, any company could buy rights to distribute an unlimited number of `ksh` binaries. Most UNIX system providers have taken advantage of this and now ship `ksh` as part of their systems. The wider availability of `ksh` contributed significantly to its success.

As use of `ksh` grew, the need for more functionality became apparent. Like the original shell, `ksh` was first used primarily for setting up processes and handling I/O redirection. Newer uses required more string handling capabilities to reduce the number of process creations. The 1988 version of `ksh`, the one most widely distributed at the time this is written, extended the pattern matching capability of `ksh` to be comparable to that of the regular expression matching found in `sed` and `grep`.

In spite of its wide availability, `ksh` source is not in the public domain. This has led to the creation of `bash`, the ‘‘Bourne again shell’’, by the Free Software Foundation; and `pdksh`, a public domain version of `ksh`. Unfortunately, neither is compatible with `ksh`.

In 1992, the IEEE POSIX 1003.2 and ISO/IEC 9945-2 shell and utilities standards were ratified. These standards describe a shell language that was based on the UNIX System V shell and the 1988 version of `ksh`. The 1993 version of `ksh` is a version of `ksh` which is a superset of the POSIX and ISO/IEC shell standards. With few exceptions, it is backward compatible with the 1988 version of `ksh`.

The `awk` command was developed in the late 1970s by Al Aho, Brian Kernighan, and Peter Weinberger of Bell Laboratories as a report generation language. A second-generation `awk` developed in the early 1980s was a more general-purpose scripting language, but lacked some shell features. It became very common to combine the shell and `awk` to write script applications. For many applications, this had the disadvantage of being slow because of the time consumed in each invocation of `awk`. The `perl` language, developed by Larry Wall in the mid-1980s, is an attempt to combine the capabilities of the shell and `awk` into a single language. Because `perl` is freely available and performs better than combined shell and `awk`, `perl` has a large user community, primarily at universities.

The need for a reusable scripting language was recognized in the late 1980s by John Ousterhout at the University of California at Berkeley. He invented an extensible scripting language named `tcl`, an acronym for ‘‘tool control language’’. `tcl` is written as a library rather than a command. Thus, it can be embedded into any command to give it scripting capability. The `tcl` language has gained a sizable following, primarily because of an X Window* programming interface that is provided by an adjunct

---

* X Window is a trademark of Massachusetts Institute of Technology.

tk. With tk it is possible to write X Window applications as tcl scripts.

At about the same time that tk was developed, Steve Pendergrast at UNIX Systems Laboratories created wksh, a program that extends ksh for X Window programming in MOTIF* and OpenLook**. The extensions were added as a collection of built-in commands to create and manipulate widgets. Callback functions are written as shell functions. Another version of ksh for X Window programming similar to wksh, xksh, was developed by Moses Ling at AT&T and is used in several applications. wksh and xksh created new demands on ksh and were a major influence for the new features that have been added to ksh−93. A new windowing desktop shell, dtksh, based on ksh−93 and wksh, has been developed by Novell and will be part of the Common Operating System Environment, COSE.

3. REQUIREMENTS FOR A REUSABLE SCRIPTING LANGUAGE

The primary requirement for any language is that it enable you to easily specify what you want. Also, a scripting language should be able to run without requiring a separate compilation system. Since strings are a basic element of any general-purpose programming language, the language must handle arbitrary length strings automatically.

A general-purpose scripting language should be simple to learn. There is no easy way to measure how simple a language is to learn, but the time to learn a language can be reduced by making the language similar to one that users already know. For instance, arithmetic computations should use familiar notation and operators should have conventional precedences.

The language should be widely available and well documented in order to achieve wide usage. Programmers do not want to spend time learning a language that will not be available to them wherever they work. Also, since no single document is right for everyone, there should be several documents for different types of users.

In addition to performing arithmetic, a script language should have string and pattern matching capabilities. The details of memory management of variable sized objects should be handled by the language, not by the user. Many applications require the handling of aggregate objects. Even though very high level languages require fewer lines of code, real world applications are likely to be large; thus a good script language needs to have a method to write procedures or functions that have automatic variables and that can return arbitrary values.

Applications coded in the language should have performance comparable to that of the application if it were written in a lower level language. This means that the overhead for interpretation must be amortized by the useful work of the application. The lower the overhead for interpretation, the larger the class of applications for which the language will be useful. Some applications are short-lived and will be dominated by the time they take to start up.

For many applications, the language should interface simply with the operating system. It should be possible to open or create files and network connections, and to read and write data to these objects. It must be possible to extend the language in application-specific domains to achieve high reuse.

Finally, the language should make it easy to write portable applications. One should be able to write scripts that do not depend on the underlying operating system, file system, or locale.

---

* MOTIF is a registered trademark of Open Software Foundation, Inc.

** OpenLook is a registered trademark of UNIX System Laboratories, Inc.

4. HOW ksh IS USED

The most frequent use of ksh is as an interactive command interpreter. In this context, most users learn the basics of redirection and pipelines. The most important features for this use are command line editing and history interaction.

A second common use of ksh is for writing scripts that combine several commands into a single command. These scripts are usually placed into the user's private *bin* directory. These scripts are customized to the individual's needs and usually are not designed for reuse. However, it is not uncommon for some of these scripts to be useful to a broader community of users. In this case, the script can be moved into a public *bin* directory that is accessible by a larger group of users.

A third common use of ksh is as an embedded scripting language. In addition to library calls popen() and system(), tools such as make and cron have used the shell as their specification language. One weakness of the traditional C-to-shell interface is that the shell actions are carried out by a separate process so that no side effects are possible. With ksh–93 it is possible to have the shell interpreter run in the same process.

A fourth use of ksh is for writing administrative scripts. Since all UNIX systems are delivered with scripts written in the shell language, system administrators need to be able to read and write scripts to do their job. The most important features for these applications are the ability to generate and test files, and the ability to invoke pipelines.

A fifth use of ksh is for the generation of front ends. ksh provides a coprocess mechanism which makes it easy to run a process that is connected to a shell script via pipes. The script interacts with a user, and then generates commands to send to the coprocess to carry out most of the work. With wksh and dtksh, the front ends can be graphical.

A sixth use of ksh is for program generation. Scripts can be written that produce code for compiled languages such as C and C++, or for script languages such as ksh. For this use, the ability to handle arbitrary strings and patterns is essential. The ''here-document'' feature is well suited for program templates. The iffe command described elsewhere in this proceedings [5] uses the shell in this way.

The final use of ksh is for writing programs. In this context, ksh does virtually all the work without relying heavily on other utilities. This is the area in which shells have traditionally been weakest and the reason that languages such as perl and tcl were invented. The new version of ksh, ksh–93, eliminates this weakness.

5. NEW FEATURES IN ksh–93

ksh–93 is the first major revision of ksh in five years. It was revised to meet the needs of a new generation of tools and graphical interfaces. Much of the impetus for ksh–93 was wksh, which allows graphical user interfaces to be written in ksh as tk allows one to write graphical user interfaces in tcl. The intent was to provide most of the awk functionality as part of ksh, as does perl. Because ksh–93 maintains backward compatibility with earlier versions of ksh, older ksh and UNIX System V shell scripts should continue to run with ksh–93.

Many of the changes for ksh–93 were needed in order to conform to the POSIX shell standard. In addition, ksh–93 has many new programming features that vastly extend the power of shell programming.

In this section, several important new features introduced in ksh–93 are described.

5.1 Floating Point Arithmetic

Applications that required floating point arithmetic no longer have to invoke a command such as awk or bc. The comma operator, the **?:** operator, and the pre- and post-increment operators were added. Thus ksh–93 can evaluate virtually all ANSI C arithmetic expressions. An arithmetic for command, nearly identical to the for statement in C, was added. In addition, the functions from the math library were added.

5.2 Associative Arrays

Earlier versions of ksh had one-dimensional indexed arrays. The subscripts for an indexed array are arithmetic expressions which are evaluated to compute the subscript index. Associative arrays have the same syntax as indexed arrays, but the subscripts are strings; they are useful for creating associative tables. However, because the list of indices is not easily determined, a shell expansion character was added to give the list of indices for an array. Because associative arrays reuse the same syntax as indexed arrays, it is easy to modify scripts that use indexed arrays to use associative arrays.

5.3 Additional String Processing Capabilities

Shell patterns in ksh–93 are far more extensive than in the Bourne shell, having the full power of extended regular expressions found in awk, perl, and tcl. In addition, ksh–93 has new expansion operators for substring generation and pattern replacement. Substring operations can be applied to aggregate objects such as arrays.

5.4 Hierarchical Name Space for Shell Variables

One of the lessons learned from the UNIX system is that a hierarchical name space is better than a flat name space. With ksh–93, the separator for levels of the hierarchy is **.** (dot). It is possible to create compound data elements (data structures) in ksh–93. Name references were added to make it easier to write shell functions that take the name of a shell variable as an argument, rather than its value. With earlier versions of ksh, it was frequently necessary to use eval inside a function that took the name of a variable as an argument.

Shell variables in ksh–93 have also been generalized so that they can behave as active objects rather than as simple storage cells. This has been done by allowing a set of discipline functions to be associated with each variable. A discipline function is defined like any other function, except that the name for a discipline function is formed by using the variable name, followed by a **.** (dot), followed by the discipline name. Each variable can have discipline functions defined that are invoked when the variable is referenced or assigned a new value. This allows variables to be active rather than passive. For example, by defining the discipline function

```
function date.get
{
    date=$(date)
}
```

the value of $date will be the output of the date program. At the C library level, variables can be created that allow for any number of discipline functions to be stacked and associated with a variable. These functions can be invoked in an application-specific way. For example, an X Window extension can associate each widget with a shell variable, and the user can write callback functions as discipline functions.

5.5 Formatted Output

One of the most annoying aspects of shell programming is that the behavior of the echo command differs on various systems. The lack of agreement of the behavior of echo in the POSIX standard led to the requirement for printf. In ksh–93, printf is a built-in command that conforms to the ANSI C standard definition and has a few extensions. The two most important extensions are the %P format conversion which treats the argument to be converted as a regular expression, and converts it to a shell pattern; and the %q format conversion which prints the argument quoted so that it can be input to ksh as a literal string. These two simple extensions make it much easier to correctly write scripts that generate scripts.

5.6 Runtime Built-in Commands

With ksh−93, a user can add built-in commands at runtime on systems that support dynamic linking of code. Built-in commands have much less overhead to invoke, and unless they produce side effects, they are indistinguishable from commands that are not built in. Each built-in command uses the same argument signature as main(), and returns a value which is its exit status. Complete applications can be written in ksh−93 by writing a library that gets loaded into ksh−93 for the application-specific portion. The C language shell interface allows the user to access the shell variable name space and to insert C language discipline functions for variables.

To give an example of the performance improvement that arises by having a command built in, the script

```
for ((i=0; i < 1000; i++))
do   cat
done < /dev/null
```

takes approximately 20 seconds to run on a Silicon Graphics workstation when cat is not a built-in command, and takes .2 seconds to run when cat is a built-in.

The ability to extend the shell at runtime makes many new uses of ksh possible. Here are examples of some possible extensions:

5.6.1 Writing Servers One possible extension would make it simple to write servers in ksh. The addition of /dev/tcp... and /dev/udp... with redirection, already allows clients to be written as ksh scripts. A built-in command could be added to advertise the service and to associate it with a variable. Callback functions that handle message events could be written as discipline functions for this variable. A second built-in command would then process events received by the server and invoke the appropriate discipline function.

5.6.2 Persistence A built-in command can be added which declares that a portion of the variable name space of a script be persistent. The built-in would take a second argument that maps this store onto the file system. Each assignment to a variable under this part of the variable name space would also cause the file system to be updated.

5.6.3 Object-oriented Database Manipulation Built-in commands can be added to read and write objects stored in an object-oriented database. The objects can be represented as shell variables, and the methods as shell discipline functions.

5.7 Support for Internationalization

The earlier version of ksh was eight-bit transparent and had a compile option to handle multibyte character sets. The behavior of ksh−93 is determined by the locale. In addition to the earlier support for internationalization, ksh−93 handles:

- Character classes for pattern matching. In ksh−93 one can specify matching for all alphabetic characters in a locale-independent way.
- Character equivalence classes for pattern matching. In ksh−93 one can specify matching for all characters that have the same primary collation weight as defined by the locale definition file.
- Collation. The locale you are in determines the order in which files and strings are sorted.
- String translation. One can designate strings to be looked up in a locale-specific dictionary at run time by preceding the string with a **$**; for example, $"hello world".

- Decimal point. The character that represents the decimal point for floating point numbers is determined by the current locale.

5.8 Usability As a Library

`ksh−93` has been rewritten so that it can be used as a library and called from within a C program. This makes it possible to add `ksh`-compatible scripting capabilities to any command, similar to the way one can with `tcl`.

6. ADVANTAGES OF USING `ksh`

6.1 Advantages of `ksh` Compatibility with Bourne Shell

Compatibility with the Bourne shell reduces the learning curve and makes it possible to reuse the many thousands of existing scripts. Because of the large number of Bourne shell users, there is a large community who already know how to use `ksh`.

Because `ksh` is compatible with the Bourne shell, there is no limit to the length of variable names, the length of strings, or the number of items in a list. File manipulation and pipeline creation are simple, and ‘‘here-documents’’ allow script applications to be packaged into a single file.

Compatibility with the Bourne shell makes `ksh` a better interactive language than most other high level scripting languages. Having the same language for interactive use as for programming has several advantages. First of all, it reduces the learning curve since everything learned for interactive use can be used in programs and vice versa. Secondly, interactive debugging is simpler since it uses the same language as do the programs.

6.2 Advantages of `ksh` Over the Bourne Shell and POSIX Shell

Using `ksh` rather than the Bourne shell has many additional advantages other than improved performance. The inline editing feature makes `ksh` friendlier to interact with. Since the inline editing feature can be enabled by scripts that are read from the terminal, interactively debugging `ksh` scripts is easier.

The Bourne shell is notorious for its lack of arithmetic facility. The `expr` command is both slow and awkward. The string processing capabilities are inferior to languages that are based on regular expressions. `ksh` uses ANSI C style arithmetic, and a pattern matching notation that is equivalent to regular expressions.

`ksh` has a better function mechanism than the Bourne shell or POSIX shell. Large applications are difficult to write with the Bourne shell or POSIX shell because of an inadequate function facility. Bourne shell and POSIX shell functions do not allow local variables. In addition to allowing local variables, `ksh` allows functions to be linked into an application when they are first referenced, making it possible to write reusable function libraries. `ksh` is also more suitable for larger applications because it has better debugging facilities.

One advantage of using `ksh` is that it has been around for several years and has a large user community. The result is that `ksh` is well documented. There are several books on `ksh`.[6] [7] [8] [9]

6.3 Disadvantages of Compatibility with Bourne Shell

There are some drawbacks to using a Bourne shell compatible shell as a programming language, many of which have been rectified in `ksh`. One drawback, the unfortunate choices in the quoting rules in the Bourne shell, makes it more difficult to write scripts that correctly handle strings with special characters in them. `tcl` is better in this respect since it uses a different character to begin a quoted string than to end the string, which makes nested quoting much easier. The use in `ksh` of the `$(...)` syntax in place of `‘‘` is a major improvement that allows easy nesting of command substitution. Another Bourne shell mistake that remains is that ANSI C sequences are not expanded inside double quoted strings.

This makes it hard to enter non-printable characters in a script without sacrificing readability. It also leads to the different behaviors of the `echo` command on different systems. With `ksh−93`, any single quoted string preceded by **$** is processed using the ANSI C string rules.

A second problem with using a Bourne shell compatible language is that field splitting and file name generation are done on every command word. In purely string processing applications, this is not the desired default, thus these operations are better left to functions as with `perl` and `tcl`. With `ksh` it is possible to disable field splitting and/or file name generation on a per function basis, which makes it possible to eliminate this common source of errors.

A third drawback is that scripts depend on all the programs they invoke, and these programs may not behave the same on all systems. In addition, there are variations in the versions of the shell that exist on different systems. To overcome this, `ksh` has more built-in capabilities so that scripts can be less dependent on system commands.

A fourth drawback is performance. On many UNIX systems, the time to invoke a non built-in command is about 100 times more than the time to run a built-in command. This means that, to achieve good performance, it is necessary to minimize the number of processes that a script creates. To overcome this problem, `ksh` has much more built-in functionality so that more operations can be performed without creating a separate process.

7. A PROGRAMMING EXAMPLE

I have chosen a real world example to illustrate the programming power of `ksh`. This example was chosen because the programs are small enough to be included in their entirety and yet are in production use. One of the programs has also been coded in `perl` so that direct comparisons can be made. The `perl` code is in the **APPENDIX**. However, understanding the programs requires some knowledge of the `make` program or a similar software configuration program.

The programs exhibited below were written by Glenn Fowler of AT&T Bell Laboratories to deal with the porting of makefiles to different systems. As is commonly known, the standard UNIX system `make` program is inadequate for large project development. As a result, several other `make` programs have been written and used in software development. At AT&T, the most frequently used make tool is `nmake`[10], written by Glenn Fowler. Typical `nmake` makefiles are about an order of magnitude smaller than old `make` makefiles, they keep track of more information, and they are portable across virtually all UNIX systems. However, to build software on machines that do not have `nmake`, it is necessary to translate the `nmake` makefile to a format that can run on the given machine.

An intermediate `make` language called MAM, which stands for *Make Abstract Machine*, has been developed by Glenn Fowler. Using an intermediate form to express the information in a makefile reduces the number of such translation programs necessary to migrate from any `make` variant to another. A make abstract file, called a *mamfile,* encompasses the static and dynamic nature of `make`. It has a simple instruction set that has been used to abstract both old `make` (System V, BSD, GNU) and `nmake`. `make` abstractions form the basis for makefile conversion tools, makefile porting, regression testing, and makefile analysis. Tools written to process mamfiles can be used with any version of `make`.

Generating a mamfile from most versions of `make` requires a relatively straightforward source code modification. The code modification for GNU `make` has been forwarded to the Free Software Foundation.

7.1 The MAM Language

To understand the `ksh−93` programs below, I give a brief description of the MAM language. The MAM language provides a simple, concise notation for describing `make` entities (variables, rules, actions) and relationships (dependencies).

The mamfile syntax is akin to assembly. It consists of a sequence of instructions, one per line, read from top to bottom. The rule and variable definitions instructions are as follows:

`make` *rule* [*attribute* ...]

`done` *rule* [*attribute* ...]

> A `make/done` pair defines the target rule named *rule*. Nested `make/done` pairs define the dependencies: the enclosing *rule* is the parent, and the enclosed *rules* are the prerequisites. The optional *attributes* classify the rule or dependency relationship (older `make` programs may not emit any attributes) such as `archive`, `implicit`, and `virtual`.

`prev` *rule* [*attribute* ...]

> Used to reference rules that have already been defined by `make/done`.

`exec` *rule* [*action-line*]

> Appends *action-line* to the action (or recipe) for *rule*. The *rule* name – (minus) names the rule in the current `make/done` nesting. `exec` is not emitted until all prerequisite rules for *rule* have been defined.

`setv` *variable* [*value*]

> Assigns *value* to *variable*. A `setv` is required for each variable referenced in the mamfile, even if its value is null, and it must occur before the first reference of the variable. This allows an analysis program to determine all variables used in the makefile and the proper assignment order.

7.2 MAM Example

The makefile in Exhibit 1 illustrates MAM. The corresponding mamfile is listed in Exhibit 2, annotated with line numbers for easy reference. The first column contains mamfile line numbers and the second column contains line numbers from the makefile in Exhibit 1.

Exhibit 1. Sample makefile

```
1    DEBUG = –g –DDEBUG=1
2    CCFLAGS = $(DEBUG)
3    cmd : cmd.o lib.o
4        $(CC) $(CCFLAGS) –o cmd cmd.o lib.o
5    cmd.o : cmd.h lib.h
6    lib.o : lib.h
```

The `info mam` instruction (line 01) identifies the file as a mamfile, lists the MAM version, and also identifies the program that generated the mamfile. Variables are defined using `setv` (lines 02-04). Lines 02 and 03 come directly from the makefile and line 04 is inferred from the predefined rules. `exec` defines the rule actions (lines 13, 19, 21), but is not emitted until after all prerequisite rules have been defined. The rule name – (minus) is shorthand for the rule name in the current `make/done` nesting (lines 13, 19, 21).

Makefile variable references are converted to shell syntax in the mamfile (lines 03, 13, 19, 21). This provides a common target language for actions, and also makes it easy to analyze makefiles using the shell.

7.3 Programs to Process Mamfiles

The first program named `mamold`, listed in Exhibit 3 below, is a program that converts a `mamfile` to a makefile that can be processed by old versions of `make`. This program attests to both the power of `ksh–93` and the simplicity of MAM. A `perl` version of this program is included in the **APPENDIX**.

Exhibit 2. Resulting mamfile

```
01   -   info mam 01/01/94 oldmake
02   1   setv DEBUG -g -DDEBUG=1
03   2   setv CCFLAGS $DEBUG
04   -   setv CC cc
05   3   make cmd
06   3   make cmd.o
07   -   make cmd.c
08   -   done cmd.c
09   5   make cmd.h
10   5   done cmd.h
11   5   make lib.h
12   5   done lib.h
13   -   exec - $CC $CCFLAGS -c cmd.c
14   3   done cmd.o
15   3   make lib.o
16   -   make lib.c
17   -   done lib.c
18   6   prev lib.h
19   -   exec - $CC $CCFLAGS -c lib.c
20   6   done lib.o
21   4   exec - $CC $CCFLAGS -o cmd cmd.o lib.o
22   2   done cmd
```

The original `mamold` program was written as an 800-line C program about five years ago. Since it generates old `make` makefiles, `mamold` must differentiate between explicit (`prereqs[`*rule*`]`) and implicit (`implicit[`*rule*`]`) prerequisites for a given *rule* (lines 22-24). `setv` (lines 18-19) converts special *mam* variables in to shell syntax and prints the name and value. The assertions are emitted in mamfile order (lines 27, 39-44), and the `closure` function emits the transitive closure of the explicit and implicit prerequisites.

This example uses several of the features of `ksh–93` that were not in earlier versions of `ksh`. Line 26 uses the pre-increment operator (the post-decrement operator is used on line 34), which is also new in `ksh–93`. Line 18 uses the new string global replacement parameter expansion operator to change *mam* variables from shell syntax to `make` syntax.

Line 12 defines associate array variables `prereqs`, `implicit`, and `action` that are used elsewhere in the script. Lines 31, 32 and 42 use the new ANSI C string notation to represent newlines and tabs in a readable fashion. The remainder of the script is compatible with earlier versions of `ksh`, but uses several features that are not in the Bourne shell or the POSIX shell. In particular it uses the `[[...]]` compound command for pattern matching and the `((...))` command for arithmetic.

The performance of the `mamold` shell script has been measured against both the original C version and the `perl` script. The `nmake` makefile for a library was converted to a mamfile. The 136-line `nmake` makefile produced a 2912-line mamfile. The mamfile was converted to a 1510-line old `make` makefile using each of the `mamold` programs. The results for user+sys times in seconds on an unloaded SPARC 2 are presented in Exhibit 4.

In this example, the C version was more than an order of magnitude longer and more than an order of magnitude faster than the `ksh` or `perl` version. The `ksh` version is about half the size and about 5%

Exhibit 3. `mamold`: MAM to old `make` makefile converter

```
set –o noglob  #disable file expansioin
# generate implicit+explicit in list
function closure {
      typeset i j
      for i
      do    [[ " $list " == *" $i "* ]] && continue
            list="$list $i"
            for j in ${implicit[$i]}
            do closure $j; done
      done
}
typeset –A prereqs implicit action
typeset –i level=0
typeset rule list order target
print "# # makefile generated by mamold.sh # #"
while read –r op arg val
do    case $op in
      setv) [[ $val == *’${mam_’* ]] && val=${val//’${mam_’/’$${mam_’}
            print –r –– $arg = $val
            ;;
      make|prev) rule=${target[level]}
            [[ " $val " == *" implicit "* ]] &&
            implicit[$rule]="${implicit[$rule]} $arg" ||
            prereqs[$rule]="${prereqs[$rule]} $arg"
            [[ $op == prev ]] && continue
            target[++level]=$arg
            [[ " $order " != *" $arg "* ]] && order="$order $arg"
            ;;
      exec) [[ $arg == – ]] && arg=${target[level]}
            [[ ${action[$arg]} ]] &&
            action[$arg]=${action[$arg]}’$(newline)\’$’\n’$’\t’$val ||
            action[$arg]=$’\t’$val
            ;;
      done) ((level––))
            ;;
      esac
done
# dump the assertions
for rule in $order
do    [[ ! ${prereqs[$rule]} && ! ${action[$rule]} ]] && continue
      list=
      closure ${prereqs[$rule]} && print –r –– $’\n’"$rule :$list"
      [[ ${action[$rule]} ]] && print –r –– "${action[$rule]}"
done
```

slower than the `perl` version.

A second example is the program `mamexec`, listed in Exhibit 5. A program about four times the size of this has been written in standard Version 7 UNIX Bourne shell as well. The `mamexec` program executes a `mamfile` the way `make` executes a `makefile`. Using mamfiles and `mamexec` provides a

Exhibit 4. comparative `mamold` times (seconds)

	user	sys	user+sys
C	0.48	0.35	0.83
ksh	18.28	0.43	18.71
perl	16.83	0.74	17.57

way to port applications to environments that have the shell, but do not have `nmake`.

The `mamexec` source is straightforward. Lines 01-13 initialize the local variables (lines 01-02), associative arrays (line 03), and options (lines 04-13). Lines 15-16 compare the current state with the previous state (if it exists). `same[`*rule*`]` is `1` if *rule's* current modification time is the same as the statefile modification time (`ls` determines the time, `comm` and `sort` determine entries that have not changed). A rule is out-of-date if `same[rule]` is null. The main loop (lines 17-38) collects prerequisites and actions, and executes actions for out-of-date rules (line 33). Only built-in commands are executed in the main loop. `setv` assigns variables that have no previous value, unless the `setv` occurs inside a `make/done` nesting (line 20). Lines 39-40 update the statefile before the script exits. The state consists of two files; `mamfile.ml` contains the list of all rules, and `mamfile.ms` contains the state time and name for all rules, where *mamfile* is the name of the mamfile.

This program illustrates some other new `ksh–93` features. The `getopts` built-in command on line 04 is an extension of the earlier `getopt` version. It allows option strings inside **[** and **]** to be inserted as part of the `getopts` option list. These strings are ignored when searching for options, but are used to automatically generate usage messages. It also uses associative arrays and the ANSI C string syntax. Line 43 uses the name parameter expansion operator **!** to generate the list of subscripts for the associative array `list`. This program illustrates how natural it is to integrate traditional shell constructs such as pipelines and command substitution with string processing code.

Exhibit 6 is a table of user+sys time in seconds on an unloaded SPARC 2 for various `make −n` commands running on a makefile (or mamfile) with 97 up-to-date rules and 223 action lines. Since `mamexec` is a `ksh` script and the other `make` programs are compiled programs, one might assume that `mamexec` would perform poorly, but this is not the case. There are a few reasons for this: the mamfile parse is simple compared to makefiles, and all rule bindings are precomputed in the mamfile (i.e., metarules and file path searches have already been applied by the mamfile generator). `nmake` is slightly slower in this example because it computes the implicit prerequisites that have already been asserted in the mamfile and old `make` makefiles, and the startup cost for this outweighs the small makefile size.

8. CONCLUSION

ksh has proven to be a good choice as a scripting language. It has the capabilities of perl and tcl, yet it is backward compatible with the Bourne shell. Applications of up to 25,000 lines have been written in ksh and are in production use.

As stated earlier, a language needs to be widely available and well documented to achieve wide usage. The previous version of ksh is shipped with most UNIX systems, and ksh−93 will be on COSE systems. ksh−93 is currently being licensed by AT&T under the name pksh. Plans are underway to make ksh−93 source code available through the UNIX system Toolchest.

Based on a few simple tests, the performance of ksh is comparable to that of perl. While the performance for the mamold in this paper was about 5% worse with ksh than with perl, another applications written in both perl and ksh−93, have shown the reverse.* Limited testing with tcl has shown similar results.

Finally, because ksh is extensible, new reusable components are likely to be implemented as ksh libraries. Extensions for graphical user interface programming provided by dtksh are likely to be widely available. Additionally, it will be possible to use ksh with tk for graphical user interface programming.

* The application was a purify to gprof converter

Exhibit 5. `mamexec: make` with state

```
typeset beg="(set –ex;" end=") </dev/null" exec=eval mamfile=Mamfile
typeset –i accept=0 force=0 level=0
typeset –A list same
while getopts "AFf:[mamfile]n" opt
do    case ${opt#${opt%?}} in
      A) accept=1 ;;
      F) force=1 ;;
      f) mamfile=$OPTARG ;;
      n) exec=print beg= end= ;;
      esac
done
[[ $opt == "?" ]] && exit 1
ml=$mamfile.ml ms=$mamfile.ms
[[ $force == 0 && –f $ml && –f $ms ]] &&
for i in $(ls –ld $(<$ml) 2>/dev/null|sort|comm –12 $ms –|sed ’s/.* //’)
do same[$i]=1; done
while read –r op arg data
do    case $op in
      setv) eval val=’$’$arg
            [[ $level != 0 || $val ]] && eval $arg=’$data </dev/null’
            ;;
      make) level=level+1
            eval arg=$arg
            [[ " $data " == *" metarule "* ]] && same[$arg]=1 && continue
            name[level]=$arg cmds[level]= list[$arg]=1
            ;;
      prev) eval arg=$arg
            [[ ! ${same[$arg]} ]] && same[${name[level]}]=
            ;;
      exec) [[ ! ${cmds[level]} ]] && cmds[level]=$data ||
            cmds[level]=${cmds[level]}$’\n’$data
            ;;
      done) eval arg=$arg
            [[ ! ${same[$arg]} ]] &&
            {  [[ ${cmds[level]} ]] &&
               { $exec "$beg${cmds[level]}$end" || exit; }
               same[${name[level–1]}]=; }
            level=level–1
            ;;
      esac
done <$mamfile
[[ $accept == 1 || $exec == eval ]] &&
{ print ${!list[@]} > $ml; ls –ld ${!list[@]} 2>/dev/null|sort > $ms; }
```

Exhibit 6. `comparative make −n times (seconds)`

	user	sys	user+sys
gnumake	0.35	0.30	0.65
oldmake	0.43	1.08	1.51
mamexec	1.23	0.40	1.65
nmake	0.85	0.95	1.80

REFERENCES

1. Al Aho, Brian Kernighan, and Peter Weinberger, *The AWK Programming Language*, Addison Wesley, 1988.
2. *POSIX − Part 2: Shell and Utilities,* IEEE Std 1003.2-1992, ISO/IEC 9945-2, IEEE, 1993.
3. Larry Wall and Randal Schwartz, *Programming perl*, O'Reilly & Associates, 1990.
4. John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison Wesley, 1994, as a very high level scripting language.
5. Glenn S. Fowler, David G. Korn, John J. Snyder, and Kiem-Phong Vo, *Feature Based Portability*, Proceedings of the USENIX Symposium on Very High Level Languages, 1994.
6. Morris Bolsky and David Korn, *The KornShell Command and Programming Language*, Prentice Hall, 1989.
7. Morris Bolsky and David Korn, *The New KornShell Command and Programming Language*, Prentice Hall, 1995.
8. Anatole Olczak, *The Korn Shell - User & Programming Manual*, Addison Wesley, 1991.
9. Bill Rosenblatt, *Learning the Korn Shell*, O'Reilly & Associates, Inc., 1993.
10. G. S. Fowler, *The Fourth Generation Make*, USENIX Portland 1985 Summer Conference Proceedings, pp. 159-174, 1985.

BIOGRAPHY

David Korn received a B.S. in Mathematics in 1965 from Rensselaer Polytechnic Institute and a Ph.D. in Mathematics from the Courant Institute at New York University in 1969 where he worked as a research scientist in the field of transonic aerodynamics until joining Bell Laboratories in September 1976. He was a visiting Professor of computer science at New York University for the 1980-81 academic year and worked on the ULTRA-computer project (a project to design a massively parallel super-computer).

He is currently a supervisor of research at Murray Hill, New Jersey. His primary assignment is to explore new directions in software development techniques that improve programming productivity. His best know effort in this area is the Korn shell, `ksh`, which is a Bourne compatible UNIX shell with many features added. The language is described in a book which he co-authored with Morris Bolsky. In 1987, he received a Bell Labs fellow award.

APPENDIX

```
#!/usr/local/bin/perl

# convert MAM to oldmake makefile

print "# # makefile generated by mamold.pl # #\n";
$level = 0;
while (<>) {
	chop;
	($op, $arg, @val) = split;
	if ($op eq "setv") {
		if ("@val" =~ /\$\{mam_/) {
			local($buf) = "@val";
			$buf =~ s/\$\{mam_/\$\$\{mam_/g;
			@val = $buf;
		}
		print "$arg = @val\n";
	}
	elsif ($op eq "make" || $op eq "prev") {
		$rule = $target[$level];
		if ("@val" =~ /\bimplicit\b/) {
			$implicit{$rule} .= " $arg";
		}
		else {
			$prereqs{$rule} .= " $arg";
		}
		if ($op eq "make") {
			$target[++$level] = $arg;
			order: {
				foreach $i (@order) {
					if ($i eq $arg) {
						last order;
					}
				}
				$order[$#order+1] = $arg;
			}
		}
	}
	elsif ($op eq "exec") {
		if ($arg eq "-") {
			$arg = $target[$level];
			if ($action{$arg}) {
				$action{$arg} .= "\$(newline)\\\n\t@val";
			}
			else {
				$action{$arg} = "\t@val";
			}
		}
	}
	elsif ($op eq "done") {
```

```
                $level = $level – 1;
        }
}

# dump the assertions

for ($i = 0; $i <= $#order; $i++) {
        $rule = $order[$i];
        if ($prereqs{$rule} || $action{$rule}) {
                @list = ();
                %mark = ();
                &closure(split(/ /,$prereqs{$rule}));
                print "\n$rule :@list\n";
                if ($action{$rule}) {
                        print "$action{$rule}\n";
                }
        }
}

# generate explicit+implicit into @list

sub closure {
        local($i);
        if ($#_ >= 0) {
                foreach $i (@_) {
                        if ($mark{$i} != 1) {
                                $mark{$i} = 1;
                                $list[$#list+1] = $i;
                        }
                        &closure(split(/ /,$implicit{$i}));
                }
        }
}
```