ksh
An Extensible High Level Language

David Korn
AT&T Laboratories
dgk@research.att.com
OUTLINE

- Motivation
- History
- How ksh is used
- Tokenizing Rules
- Variables
- Patterns
- Command line processing
- Compound commands
- Functions
- Interactive use
THE PROBLEM

- Interactive programs need command language
- Applications need interpretive language
- Traditionally done for each application
- Each language has its own syntax
- Languages are expensive to learn
- Languages don’t interface to rest of system
SCRIPT LANGUAGE REQUIREMENTS

- Interpretive
- Arbitrary length string handling
- String splitting/extraction
- String formatting
- Pattern matching
- Arithmetic evaluation
- Control flow
- Functions with local variables
- Arrays/Lists/Structures
ALTERNATIVE LANGUAGES

- Perl
- Tcl
- Python
- Icon
- JavaScript
- Rexx
- Visual Basic
- HyperCard
WHY ksh?

- Upward compatible with Bourne Shell
- Interfaces well to UNIX system
- Allows tool building approach
- Has many existing users
- Is well documented
- Defined by international standard
- Extensible by different applications
- Portable scripts
- Reasonable performance
HOW ksh IS USED

- Command language
- Software administration
- Software configuration
- Software testing
- Program front ends
- Program generation
- Cgi-bin scripting
- General purpose programming
- Embedded systems
SHELL SCRIPTS

A shell script is a file containing shell commands.

You need to make executable – `chmod +x script`

The script can be invoked as:

- `ksh name [ arg ... ]`
- `ksh < name [ arg ... ]`
- `name [ arg ... ]`
- `#! path_of_ksh`

If not invoked from ksh, might require
HOW KSH WORKS

ksh first reads a command as follows:

1. read one or more lines as needed
2. separate into tokens using space/tabs
3. form commands base on token types

ksh then evaluates the command forming simple commands and does the following:

1. Expand word tokens
2. Split words into fields
3. Compute arguments
4. Locate command
SPECIAL CHARACTERS

The shell process the following characters specially unless quoted:

| & \ ( ) < > ; " ' $ ` space tab new-line

The following are special whenever patterns are processed:

* ? [ ] ( )

The following are special at the beginning of a word:

# ~

The following are special when processing assignments:

= [ ]
SHELL COMMENTS

Comments begin with an unquoted #.
Comments end at the end of the line.
Comments can begin whenever a token begins.

Examples:
# This is a comment
# and so is this.
grep foo bar # this is a comment
grep foo bar # this is not a comment
TOKEN TYPES

The shell uses spaces and tabs to split the line or lines into the following types of tokens:

• Control operators
• Redirection operators
• Reserved words
• Assignment tokens
• Word tokens
OPERATOR TOKENS

Operator tokens are recognized everywhere unless quoted. Spaces are optional before and after operator tokens.

I/O Redirection Operators:

>  >>  >&  >|  <  <<  <<-  <&  <>

Each I/O operator can be immediately preceded by a single digit.

Control Operators:

|  &  ;  ( )  ||  &&  ;;  (( ))  |&  ;&
REDIRECTION

n< file  Open file descriptor \textit{n} for reading.

n> file  Open or create as descriptor \textit{n} for writing.

n>> file Open or create as descriptor \textit{n} for appending.

n<< file Open or create as descriptor \textit{n} for read/write.

n>| file Force create as descriptor \textit{n} for writing.

n>& fildes Duplicate \textit{fildes} onto \textit{n}.

n<< word Create here-document on descriptor \textit{n}. Document ends on line containing only \textit{word}.
The following are reserved words:
{ } case do done elif else esac fi for function if in select then time until while [[ ]] 

Reserved words are special when they appear:
- As the first word on a line.
- After the operators ; | || & && |& ( )
- As the first words after reserved word, except after case, for, in, select, and [[ .
- As the second word after case, for, and select.

Example:
print do done
ASSIGNMENT TOKENS

An assignment token is one of:

- `varname=word`
- `varname+=word`
- `varname[subscript]=word`
- `varname[subscript]+=word`

No spaces or tables are allowed before `=` or `+=`.

The `+=` form is for appending to a variable `varname`.

The `[]` act to quote special characters.

`subscript` can be string or arithmetic
SHELL QUOTING

Quoting causes characters to loose special meaning.

\ Unless quoted, \ causes next character to be quoted. In front of new-line causes lines to be joined.

'...' Literal quotes, cannot contain '.

$'...' Ansi-C quotes. All C \ escape sequences recognized plus a few extensions.

"..." Removes special meaning of all characters except, $, ", \, and '. The \ is only special before one of these characters and new-line. Double quoted
SIMPLE COMMAND

A simple command consists of three types of tokens:

1. Assignments (must come first)
2. Command Words tokens
3. Redirections - Redirection op + word op

The first token must not be a reserved word.

Command terminated by new-line or ;

Use ; to put multiple simple commands on a line.

Examples:

foo=bar  z=$(date)  print $HOME

ALIASES

An alias is a shorthand or macros that can be used wherever a command name is valid. When the shell is tokenizing its input, and it reads a word token not containing /, in a place that a command name belongs, it checks for an alias has been defined for this name and replaces this word with the alias value and resumes tokenization.

- ksh aliases are not like csh aliases. They do not allow arguments.
- The alias built-in command is used to define and list alias.
- The unalias built-in command unsets an alias
• Aliases are not recommended for scripts.
TILDE SUBSTITUTION

Words beginning with ~ are treated specially.

The ~ ends at first /.

The following substitutions are made:

- ~ is replaced by value of $HOME
- ~- is replaced by value of $OLDPWD
- ^+ is replaced by value of $PWD
- ~name is replaced home directory for user name

Otherwise, the word is left alone.

Examples:
print ~dgk
VARIABLE NAMES

ident := ([[.alpha.]][[.alnum.]])*  

varname := [.]?ident ([.]ident)?

Variables are case sensitive

Examples:
x  z1  _  ___ _1_2_3  .sh.version  x.y This_is_a_very_very_very_very_very_long_name

Not variables:
3 x..y  x-y 3z ’a b’

Variables store values
VARIABLE ATTRIBUTES

By default attributes hold strings of unlimited length

Attributes are set with typeset

- readonly (-r) – cannot be changed or unset
- export (-x) – value will be exported to commands
- upper (-u) – letters will be converted to uppercase
- lower (-l) – letters will be converted to lowercase
- ljust (-L width) – left justify to given width
- rjust (-R width) – left justify to given width
- zfill (-Z width) – justify, fill with leading zeros
- integer (-i [base]) – value stored as integer
• float (-E \textit{prec}) – value stored as C double
• nameref (-n) – a name reference
NAME REFERENCES

A name reference is a type of variable that refers to another variable. The name cannot contain a ..

Use typeset -n name=vname. You can use preset alias nameref instead of typeset -n.

Whenever a variable is referenced that does not begin with ., and the first component names a reference variable, the referenced variable is substituted.

Example:

page.point.x=400
nameref p=page.point
print ${p.x}
PARAMETER EXPANSION

A parameter is one of the following:

- A variable.
- A positional parameter, numbered starting at 1.
- A special parameter, $ ? ! – * @

Value of parameter – ${param}

Parameter expansion can be part of a word.

Parameter expansion even in double quotes.

The {} can be omitted for simple variables, special parameters, and single digit positional parameters.
POSITIONAL PARAMETERS

Positional parameters are set by one of the following:

• The arguments to a shell script.
• The arguments to a shell function.
• Argument to the set built-in command.
• The shift built-in command.

Parameter 0 ($0) is set to the name of the shell, the shell script, or the shell function.

Examples:
runcmd foo '' bar
set -- foo '' bar
shift
SPECIAL PARAMETERS

Similar to variables that are set by the shell.

$ The pid of the current shell.

! The pid of the most recent background process.

? The exit status of the last command.

- The invocation arguments.

@ The positional parameters.

* The positional parameters.
OTHER PARAMETER EXPANSION

\$\{#param\} - Length of param

\$\{param\-word\} - Default to word

\$\{param\#pattern\} - Left strip min pattern

\$\{param\##pattern\} - Left strip max pattern

\$\{param\%pattern\} - Right strip min pattern

\$\{param\%%pattern\} - Right strip max pattern

\$\{param\//pattern\//str\} - Replace pattern with str

\$\{param\////pattern\//str\} - Replace all
VARIABLES SET BY ksh

**PWD**  The current working directory.

**PPID**  The parent pid of the current shell.

**LINENO**  The current line number with script or function

**RANDOM**  A random number between 0 and 32767.

**SECONDS**  Time since script started.

_  Last argument of previous command.

**REPLY**  Set by read and select.

**OPTARG**  Option argument set by getopts.
PROMPT VARIABLES

Prompts are written to standard error. They are controlled by the values of the following shell variables:

PS1 Issued by interactive shells before reading each command.

PS2 Issued when a new-line is encountered while reading a token or after a token that does not complete a command.

PS3 Issued when running the select compound command.

PS4 Issued before each command trace is written when the xtrace (-x) option is on.
COMMAND SUBSTITUTION

Syntax: $( command )

This is replaced by standard output from command with trailing new-lines removed.

Can be part of a word - +++$(id -un)+++ $(< file) is replaced by file contents.

Expands even inside double quotes. Quotes inside $(() are not affected by outer quotes.

Example:
print "$(tty | sed "s;.*;/;;")"

Replaces obsolete `command` syntax.
ARITHMETIC SUBSTITUTION

Syntax: `$( ( expr ) )`

This is replaced by the value of the arithmetic expression `expr`.

Can be part of a word.

Expands even inside double quotes.

All substitutions are performed on `expr` before being evaluated.

Example:

```bash
x=5 y=3; print $( ( x* $y ) )
```

```
15
```
WORD SPLITTING

After parameter expansion, command substitution, and arithmetic expansion, the characters that are generated as a result of these expansions that are not inside double quotes, are checked to see if they contain split characters.

The split characters are defined by the value of the IFS variable.

IFS=’’ disables word splitting.

Example:
IFS=x v=exit
print exit $v "$v"

exit e it exit
PATHNAME EXPANSION

After word splitting, each field that contains pattern characters is replaced by the pathnames that match.

Quoting prevents pathname expansion.

set -o noglob (-f) disables pathname expansion.

A leading . in each filename must match explicitly.

Unmatched pathnames are left alone.

Example:
x=*.*
print  $x
set -o noglob
print  $x
==============
foo.c bar.c
EXIT STATUS

Every command and script has an exit status.

A value of 0 indicates success.

Normal exit values are from 0–255.

Termination by signal is 256+signal number.

Exit status of script is exit status of last command.

The command exit \( n \) causes script to exit with the value \( n \).
SEARCH RULES

- Special built-ins
- Functions
  - command bypasses search for functions
- Built-ins not associated with PATH
- Path search
- Built-ins associated with PATH
- Functions loaded with FPATH
- Executable images
BUILT-IN COMMANDS

Built-in commands do not create a separate process. Commands are built-in for the following reasons:

• They are intrinsic to the language – `exit`
• They produce side effects on the process – `cd`
• They perform much better (100x) – `print`

The `builtin` command lists commands that are built-in.

Extension libraries allow other commands to be built-in.
SPECIAL BUILTINS

- scoping for name=value command ...
- effect of errors
- Cannot be functions

LIST OF POSIX SPECIAL BUILTINS

:         exec            set
.         export          shift
break     exit            trap
continue  readonly        unset
eval      return

ADDITIONAL ksh SPECIAL BUILTINS

alias   login       typeset
disown  newgrp      unalias
eval SPECIAL BUILTIN

eval arg ...

Causes all the tokenizing and expansions to be performed again.

read -r 'line?$ '
eval "$line"

$ date | wc -w
  6
trap SPECIAL BUILTIN

```
trap action condition ...
```

Defines action to take when specified condition arises.

*action* can be any script. Invoked as `eval "action"`

*condition* can be signal name or number or one of the following:
EXIT ERR DEBUG KEYBD

EXIT trap runs when function or script exits.

Example:

```
trap 'rm -f $tmpfiles' EXIT
```
getopts BUILTIN

getopts *optstring* name [args ...]

Processes command line options specified in *optstring*

Generates man page and help options, **--man**.

Typically used in while loop to process all options.

Options end with **--** or non-option.

Shell built-ins use same method to generate option/help parser.

Uses OPTIND and OPTARG variables.

See getopts **--man** for details.
ARITHMETIC

• Double precision floating
• Constants – 10, 64#3aA, 1e5
• Variables – abc, $abc
• Arithmetic operators
  – All C language operators
  – C language precedence
  – Math library functions
• Statement – (( expression ))
• Expansion – $(( expression ))
CONDITIONAL EXPRESSIONS

[[ expression ]]

Space required after [[ and before ]].

Expression consists of unary and binary primitives.

Operators cannot be result of expansion.

Use parenthesis () for grouping.

use ! to negate a primary or group.
UNARY OPERATORS

-e file True if file exists.
-r file True if file is readable.
-w file True if file is writable.
-x file True if file is executable.
-d file True if file is a directory.
-L file True if file is a symbolic link.
-s file True if file has size greater than 0.
-t fildes True if file descriptor is a tty.
-n string True if string is not null.
-z string True if string is null.
BINARY OPERATORS

string == pattern True if string matches pattern.

string != pattern True if string does not match pattern.

string1 < string2 True if string1 comes before string2

string1 > string2 True if string1 comes after string2

file1 -nt file2 True if file1 is newer than file2

file1 -ot file2 True if file1 is newer than file2

file1 -ef file2 True if file1 is same as file2

expr1 && expr2 True if both expr1 and expr2 are true
PATTERNS vs. ERE's

?  
*  
[...]
[!]...]
?(...)  
*(...)  
+(...)  
@(...)  
{n}(...)  
{m,n}(...) 
!(...)  
\d  
a|b  
a&b

.  
.*  
[...]  
[^...]
(...)?
(...)*
(...)+
(...)
(...){n}
(...){m,n}
PATTERNS USE IN KSH

- Pathname expansion.
- `[[ string == pattern ]].`
- In `case` statements.
- In parameter expansions.
ARRAYS

- Indexed arrays – `name [expression]`
- Associative arrays – `name [string]`
- Declaration – `typeset -A name`
- Assignment – `name [subscript] = value`
- Additional indexed array assignments
  - `set -A name ...`
  - `read -A name`
- Reference – `$ {name [subscript] }`
- Subscript name – `$ { !name [subscript] }`
- Subscripts – `$ { !name [ * ] }`, `$ { !name [ @ ] }`
HERE DOCUMENTS

cat > file <<- EOF
  <HTML>
  <HEAD>
    <TITLE>Generated by here-doc</TITLE>
  </HEAD>
  <BODY>
    <CENTER>My current directory is $PWD</CENTER>
  </BODY>
  </HTML>
EOF
READING LINES

read is used to read a line from a file and to store the result into shell variables.

- Use `read -r` to prevent special \ processing.
- Uses `IFS` variable to split into words.
- If no variable specified, `read` uses `REPLY`.
- Use `-un` to read from file descriptor $n$.
- Use `-A var` to split line in array variable.
- Use `-t timeout` to specify a timeout in milliseconds.

Example:
```
read -r -A word
print -r -- "${word[2]} ${word[4]}"
```
FORMATTED OUTPUT

- `printf` or `print` with `-f` `format` option
- Supports full ANSI C conversions
- Extensions
  - `%b`—ANSI escape sequences.
  - `%q`—quote argument for re-input.
  - `\E`—Expands to 033.
  - `%P`—convert ERE to shell pattern.
  - `%H`—convert using HTML conventions
  - `%T`—date coverions using `date` formats
  - `%d`—base may follow precision, `%..32d`
COMPOUND COMMANDS

ksh allows the following types of compound commands:

- Arithmetic commands – `((expr))`
- Conditional commands – `[[expr]]`
- Subshell command
- Pipelines
- Coprocess
- Lists
- for, while, until
- if, case, select
- Function definition
SUBSHELL

A subshell a copy of the current shell.

A subshell command is a compound command that is contained in parenthesis, ( command ).

Changes made in the subshell do not affect the shell that runs this subshell.

The command run by parameter substitution runs in a subshell.

All parts of a pipeline, except the last run in a subshell.

Example:

```
x=10
( x=5 print $x)
print $x
=========
5
10```
PIPE COMMANDS

`command1 | command2` Create a pipe, run `command1` with standard output connected to the pipe and `command2` with standard input connected to the pipe.

All but last command is executed in a subshell.

By default, exit status of a pipe is exit status of last command. Use `set -o pipefail` to cause pipe to fail if any command fails.

A `!` in front of a pipeline negates exit status.

The `time` reserved word in front of a pipe causes timing results to be output when pipe completes.
LIST COMMANDS

ksh supports the following list commands:

`command &` Run `command` in the background. Use `wait $!` to wait for completion.

`command1 ; command2` Run `command1` then `command2`

`command1 && command2` Run `command2` iff `exit status of command1` is 0.

`command1 || command2` Run `command2` iff `exit status of command1` is non-zero.

Note, that each `command` can be a pipeline.
if COMMAND

if command1
then command2
[elif command3
then command4]
...
[else command5]
fi

If exit status of command1 0, then run command2. Otherwise, try each elif command until one has exit status 0. Otherwise, if else is defined run command5. standard input connected to the pipe.

Example:
if [[ -d $dir ]] && (( x < 5))
then print -r "$dir is a directory"
elif [[ -f $dir ]]
then print -r "$dir is a file"
else print -r "$dir type is unknown"
fi
case COMMAND

case string in
  pattern1) command1;;
  ...
  pattern-n) command-n;;
esac

Locates the first pattern that matches the string and then executes the corresponding command.

All expansion other than word splitting and pathname expansion are done to get string.

Example:
case $line in
  #*) ;;
  *) print -r -- "$line";;
esac
while and until

while command1
do command2
done

Runs command1 and if the exit status is 0, then command2 is run. This is repeated until command1 fails.

The break built-in causes the loop to terminate. The continue built-in causes the next iteration to start.

until command1 is equivalent to
while !command1.

Example:
while read -r line
do if [[ $line == END ]]
then break
fi
    process "$line"
done
for COMMAND

for var [ in items ]
do command
done

For each word in the items list, the variable var is assigned the value of this word, and then command is run.

If in items is omitted, the positional parameters are used.

The break built-in causes the loop to terminate. The continue built-in causes the next iteration to start.

Example:
for i in *.[ch]
do diff "$i" "$dir/$i"
done > outfile
ARITHMETIC for

for ( (init; condition; incr) )
do command
done

init, condition, and incr are arithmetic expressions. This for works like the C language for.

The break built-in causes the loop to terminate. The continue built-in causes the next iteration to start.

Example:
for ((i=1,sum=0; i < 20; i++))
do (( sum += i))
    print $i: $sum
done
Functions are similar to scripts and other commands except that they can produce side effects in the callers script. The positional parameters are saved and restored when invoking a function. By default, variables are shared between caller and callee. However, `typeset` within functions defined with `function` syntax defines local variables.

Functions return an exit status like other commands. Use command substitution to capture their output.

Example:

```bash
function average
{
    float i sum=0
    for i; do sum+=i; done
    print $sum
}
average 35 65 99 65
```
COMPOUND OBJECTS

- Index array assignment \( x=(a \ b \ c) \)
- Associative array assignment \( w=([\text{the}]=5 \ [a]=4) \)
- Struct assignment \( p=(x=1 \ y=2) \)

Example:

```bash
function dotprod # a b {
    nameref a=$1 b=$2
    float sum
    integer n=${#a[@]} m=${#b[@]}
    (( n = m<n?m:n ))
    for((sum=0,m=0; m < n; m++)) do (( sum += (a[m] * b[m]) )) done
    print -r -- ${sum}
}
float x= ( 1 2 3 4 5 )
float y= ( 5 4 3 2 1 )
predict -r dotprod=$(dotprod x y)
```
ACTIVE VARIABLES

• Per variable traps for lookup and assignment
  • function name.get{...}
  • function name.set{...}
  • function name.unset{...}

• Additional discipline functions can be defined with the C language interface.
INTERACTIVE SHELLS

The shell reads lines from a file or from standard input and processes them. When the shell reads from standard input and both standard input and standard output are terminal devices, then it is interactive. The following differences occur: in this case

- The shell prompts when it is waiting for input.
- The command lines can be edited as they are entered using either vi or emacs commands.
- The shell keeps a history file of commands that you previously entered so that they can be edited and reentered.
- The interrupt and kill signal does not terminate the shell.
- Pipelines are run as separate jobs that can be stopped and started. This is called job control.
JOB CONTROL

Each pipeline is run as a job either in the foreground or in the background by appending &. Jobs are given numbers, and \%n refers to job n. The following operations apply to jobs:

- ^Z stops the foreground job.
- A background job will stop when it needs input.
- jobs lists the current jobs know to this shell.
- bg restarts a job in the background.
- fg restarts a job in the foreground.
- disown prevents HUP signal when shell exits.
- kill can use the job number to kill the job.
KEY BINDING

• New KEYBD trap.

• New shell variables.
  – .sh.edchar – contains input char
  – .sh.edcol – cursor column
  – .sh.edmode – ESC in vi-insert
  – .sh.edtext – current input buffer

• .sh.edchar assignment changes binding

```
typeset -A KEYTAB
trap 'eval "${KEYTAB[${.sh.edchar}]}}"' KEYBD
function keybind # key seq
{
    KEYTAB[$1]="sh.edchar=${.sh.edmode}$2"
}
```