Unix Shell Programming

Dr. Christoph Bauer

Sep 2018

Preface

- The first release of this course was written in 2004, focusing on the Korn Shell (ksh) in Sun Solaris 8. Back then Solaris 8 was still widely used in production environments, and the ksh was the default programming shell available on every host.
- We keep the focus on ksh. Most script examples will probably run successfully in bash, too, without modifications. Both ksh and bash are extensions of the oldest shell, the Bourne Shell (sh), sh scripts will run in ksh and bash. ksh-specific commands, options and variables will be pointed out.
- Commands, script code and terminal output are written in `Courier font`. Optional input is surrounded by `[ ]`. Placeholders for e.g. file names are written in *`italic font`*. When a new command is introduced the first time, it is written `coloured`.
- Language items which are considered less important or which are specific to the Bourne Shell (sh) and therefore "historical" are written in gray.

A simple shell script

```
#!/bin/ksh
# Hello world script
OUTPUT="Hello World"
echo $OUTPUT
exit 0
```

Line 1 tells the system which shell to use to run this script.

Line 2 is a comment.

Line 3 is a variable assignment (declaration and definition).

Line 4 displays the contents of the variable to the screen.

Line 5 explicitly tells the executing Korn shell to exit. By default, the shell exits automatically when the end of the script file is reached.

Running a script

- `./scriptname` or `scriptname`
  starts a subshell: same as parent shell except when specified (`#!`)
- `ksh scriptname`
  explicit start of a sub shell; ignores `#!`
- `. ./scriptname`
  processes script in current shell (without starting a sub shell; ignores `#!`)

Generating output

- `echo "Hello world"`
  - available with all shells (sh, ksh, csh: builtin)
  - special control characters within the message string (ksh):
    - \t tab character
    - \n additional newline
    - \a bell
    - \b backspace (moves backwards without deleting, for overtyping)
    - \c suppress newline at the end (e.g. for user prompts)
    - \r carriage return without line feed
- `print [options] "Hello world"`
  - ksh builtin with enhanced functionality; control chars see above; options:
    - -n suppress newline at the end (e.g. for user prompts)
    - -- do not interpret subsequent "-" chars as options (print literally)
    - -r turn off special meaning of \

Shell variables

- Variable names may contain alphanumeric characters and _.
- Set a[n environment] variable: `[export] VAR[=value]`
  For sh, here two lines are required: `VAR=value; export VAR`.
- Access to variable contents: `$VAR` or `${VAR}`
  `{ }` are name delimiters: `${MYVAR}S` ≠ `$MYVARS`
- Remove (unset) a variable: `unset VAR`
- Switch between env./local variable: `typeset ±x VAR`
- Display all variables: `set`
- Display all environment variables:
  ```
  typeset -x
  export
  env
  ```

Quoting characters

- `'  '` mask all special characters except `'`
- `"  "` mask all special characters except `"  \  ` $`

  `'` and `"` mask each other.

  Example: variable assignment

  ```
  MYTEXT="$OLDTEXT is invalid. Add a \$ sign."
  ```

- `\` masks a single char (including newline: mask end-of-line).
- `` `  ` `` command substitution (sh, ksh)

  Example: `` MYDATE=`date +%Y%m%d` ``
- `$(  )` command substitution (ksh only)

  Example: `MYDATE=$(date +%Y%m%d)`

Standard variables with meaning to ksh

- `CDPATH`: search path for `cd` command (default: not set)
- `COLUMNS`: width of edit window for shell and printing `select` lists
- `EDITOR`: which editor to use when a command invokes an editor (e.g., `crontab`); sets the shell's interactive mode (when `VISUAL` is not set)
- `ENV`: full path name of the file containing ksh-specific settings
- `FCEDIT`: default editor for fc command (command history)
- `FPATH`: search path for function definitions (later)
- `HISTFILE`: path name of the file that stores command history
- `HISTSIZE`: number of previously entered commands accessible to the current shell (default: 128)

Standard variables with meaning to ksh

- `HOME`: full path name of user's home directory, default argument to `cd`
- `IFS`: input field separator (later)
- `LINES`: determines column length for select lists (later)
- `PATH`: search path for executables (default: depends...)
- `SHELL`: full path name of user's login shell
- `TMOUT`: automatic logout when this no. of secs is exceeded without typing anything (default: not set)
- `VISUAL`: which editor to use when a command invokes an editor (e.g., `crontab`); sets the shell's interactive mode (same as `set -o vi|emacs|gmacs`), `VISUAL` takes precedence over `EDITOR`

Variables automatically set by ksh

- `ERRNO`: value of errno set by the most recent system call (for debugging)
- `LINENO`: line no of current line within script/function being executed
- `OLDPWD`: previous working directory set by the `cd` command
- `OPTARG`: last option argument processed by `getopts` (later)
- `OPTIND`: index of last option argument processed by `getopts` (later)
- `PPID`: process number of shell's parent process
- `PWD`: present working directory set by `cd` command
- `RANDOM`: a random integer between 0 and 32767
- `REPLY`: set by `select` or `read` statements (later)
- `SECONDS`: number of seconds since shell invocation

Miscellaneous variables

- `TERM`: the current terminal type (e.g. `vt100`, `xterm`, `dtterm`, ...)
- `DISPLAY`: the console system running the X server (graphics output)
- `LD_LIBRARY_PATH`: search path for shared libraries (`.so` files)
- `SSH_CLIENT` and `SSH_TTY` are set by ssh.
- `TZ`: the timezone (normally MET)
- `LOGNAME` and `USER` contain the current user name (not always set).
- `HOSTNAME` contains the current node name (not always set).

Standard shell variables

- `$$` is the PID of the current shell/shell script process.
- `$?` is the exit status of the last run command.

  A script can return a dedicated exit status just as any command:
  `exit` *`return_value`*
  (arbitrary integer value, will be mapped to integer in range 0 – 255; 0 means "success" by convention).

  `exit` immediately ends the script execution.

- `$!` is the PID of the most recently started background process (started with `&`; not effective with `bg` command).

Prompt variables (ksh)

- `PS1` is the standard prompt variable. Default `$`
  Same in sh, interesting only for interactive use
- `PS2` is used when newline is pressed but the command is not yet syntactically complete (e.g. missing `'` sign). Default `>`
  Same in sh, interesting only for interactive use
- `PS3` is used in `select` statements (prompt for selection). Default `#?`
- `PS4` precedes the command lines displayed in execution trace (xtrace) mode (`set -x`). Default `+`

Shell variables

- Automatic (conditional) variable assignment ("default values")

	Result if variable set, not null	Result if variable set, null	Result if variable not set
`${variable:-text}`	*variable*	*text*	*text*
`${variable-text}`	*variable*	*variable* (null)	*text*
`${variable:=text}`	*variable*	*text*	*text*
`${variable=text}`	*variable*	*variable* (null)	*text*
`${variable:?text}`	*variable*	(*text*/error, exit)	(*text*/error, exit)
`${variable?text}`	*variable*	*variable* (null)	(*text*/error, exit)
`${variable:+text}`	*text*	(null)	(null)
`${variable+text}`	*text*	*text*	(null)

(Interactive shells do not exit, scripts do.)

The = and := versions do not only return a result, they also modify *variable*.

Shell variable types

- sh: only constants and strings (default)

  Declaration of constants: `readonly var[=value]`

- ksh: constants, strings (default), integers and arrays

  Declaration of constants: `typeset -r var[=value]`

Integer variables (ksh)

`typeset -i` *`var1`*`[=`*`value1`*`]` *`var2`*`[=`*`value2`*`]` …

or

`integer` *`var1`*`[=`*`value1`*`]` *`var2`*`[=`*`value2`*`]` …

→ *`var1`*, *`var2`*, … may only contain integer values.

Default: base 10, change with, e.g., `typeset -i8` *`var`* (it is also possible then to make calculations in another base)

Arithmetic operations

- External `expr` command (the spaces are important):
  ```
  expr $a + $b
  c=`expr $a + $b`
  ```
- Arithmetic operations: `+ - \* / %`
- Comparison operations: `= \> \< \>= \<= !=`
  (result of `expr` is 1 if true)
  Note that masking is required for operation characters with meaning to the shell!
- The `expr` command is required for sh (no built-in arithmetics) and optional for ksh which has built-in arithmetics

Arithmetic operations (ksh)

- Recommended syntax: `((x=$a+$b)); echo $x`
  (`$` signs and spaces optional; no masking required for `* < >`)
- Alternative: `let` expression (equivalent to `(( ))`):
  ```
  let a=1; let b=2
  let c=a+b; echo $c
  ```
  (no spaces in the arithmetic expression; use masking for `* < >`)
- Arithmetic operations: `+ - * / %`
  The prefixed − operator changes the sign of a subexpression.
  var`+=10` means *var*`=`*var*`+10`; same for `-= *= /= %=`
- Comparison operations: `== > < >= <= !=`
- Evaluation: left to right, statements in `( )` first, `* / %` before `+ -`

Arithmetic operations (ksh)

Exit status and evaluation result:

- `((x=2>1))` — `x` is assigned the value 1 (true), but the exit status of the command is 0 ("success").
- `((x=1>2))` — `x` is assigned the value 0 (false), but the exit status of the command is 1 ("failure").
- `((1>2))` — the result is not stored or displayed, the exit status is 1 ("failure").

(Works similarly for `expr` statements.)

Shell metacharacters

- `*` matches any sequence of arbitrary characters, including none.
- `?` matches exactly one arbitrary character.
- `[chars]` exactly one of the characters between `[ ]`
- `[A-Z]` exactly one from this range of characters
- `[!chars]` exactly one character, but not one of the specified
- `\` masks the subsequent character (it is treated literally).
- Examples:

```
ls *.ksh
ls [A-Z][abc][0-9].txt
```

Shell metacharacters (ksh)

- `?(pat1|...|patn)` one of these patterns at most once
- `@(pat1|...|patn)` one of these patterns exactly once
- `+(pat1|...|patn)` one of these patterns at least once
- `*(pat1|...|patn)` one of these patterns arbitrarily often (incl. zero)
- `!(pat1|...|patn)` not one of these patterns
- Example:

  ```
  ls myscript.@(ksh|bak|old)
  ```

  would match and list the three files `myscript.ksh`, `myscript.bak` and `myscript.old`, if they existed.

Shell metacharacters (ksh)

In ksh tilde substitutions for directory names are available:

- `~` or `~/` my own home directory
- `~`*`user`* *`user`*'s home directory
- `~+` current working directory (same as `$PWD`)
- `~-` or `-` last working directory (before `cd`)

String variables (ksh)

- `typeset` command for string variables (ksh):
  - `typeset -u var` convert all chars to uppercase automatically
  - `typeset -l var` convert all chars to lowercase automatically
  - `typeset -Ln var` left-justified in a field of length *n*

    If *n* is not given, the value of first assignment determines the field length.
    The variable is filled on the right with blanks or truncated on the right.
    Leading zeroes are removed if the `-Z` flag is also set.
  - `typeset -Rn var` right-justified in a field of length *n*

    If *n* is not given, the value of first assignment determines the field length.
    The field is filled on the left with blanks or truncated on the left.
  - `typeset -Zn var` right-justified in a field of length *n*

    If the first non-blank character is a digit and the `-L` flag is not set, the field will be filled with leading zeroes. `-LZ`: strip leading zeroes

String operations (ksh)

- `${svar}` contents of a string variable (same as `$svar`)
- `${#svar}` length of a string
- `${svar%pat}` removes shortest string matching *pat* from the right.
- `${svar%%pat}` removes longest string matching *pat* from the right.
- `${svar#pat}` removes shortest string matching *pat* from the left.
- `${svar##pat}` removes longest string matching *pat* from the left.

pat is a pattern (a sequence of characters including shell wildcards such as `*` or `?`).

svar is the name of the string variable (without `$`).

Without wildcards, `%` and `%%` or `#` and `##` have the same effect.

Example:

```
FULL_PATH_NAME=/usr/bin/ls
COMMAND_ONLY=${FULL_PATH_NAME##*/}
COMMAND_WITH_BIN=${FULL_PATH_NAME#*/}
```

Arrays (ksh)

- `arr[index]=val` — assign a value to an array element (*index* = [0,] 1, 2, …)
- `${arr[index]}` — access an array element (the `{ }` are important here)
- `set -A arr v1 …` — assign a list of values to an array (index starts with 0)
- `${arr[*]}` — all array elements (values) in a list
- `${#arr[*]}` — number of defined array elements

The maximum number of array elements is at least 512, but depends on the shell version and implementation. In addition to simple number-based arrays, modern shells also know "associative arrays" (key-value pairs).

Input and output redirection

- By default, commands take input from stdin ("standard-in") and write output and error messages to stdout and stderr, respectively.
- stdin points to the console (keyboard/active pseudo terminal).
- stdout and stderr point to the console (screen/active pseudo terminal).
- User is not interested in output or error messages:

```
grep "search string" myfile >/dev/null
ls -al 2>/dev/null
```

- Pipes: stdout of first command is directed to stdin of second command

```
ls -al | grep passwd
```

Input and output redirection

- Input redirection (read from file instead of stdin): `<` or `0<`
- Output redirection (write to file instead of stdout): `>` or `1>`
  Existing files will be *overwritten* (except when `noclobber` option is on; in this case, `>|` can be used to force overwriting).
- Output redirection (append to file): `>>` or `1>>`
- Error redirection: `2>`
  Error redirection, append: `2>>`
  Error redirection to the same target as stdout: `2>&1`
- Examples

```
ls -lR >outfile1 2>errorfile1
grep "$STRING" file3 >results.txt 2>&1
find . -name "*ksh" 2>>errorfile2
```

Input and output redirection

Example: `output_and_error.ksh`

```
#!/bin/ksh

echo "This is regular output."
echo "This is regular output, too."
echo "This is an error message." >&2
```

The first and second line will be written to the stdout channel (1).

The third line will be written to the error channel (2).

Pipelines, lists and groups

- Pipeline or pipe: sequence of commands separated by `|`
- List: sequence of one or more pipelines separated by
  - `;` strictly sequential execution
  - `&` asynchronous execution (shell does not wait for pipelines to finish)
  - `&&` next list is only executed if preceding list returns 0
  - `||` next list is only executed if preceding list does not return 0
- `{ list }` simply execute list (used to group commands)

  Example: `{ cmd1; cmd2; cmd3; } > outfile`
- `( list )` execute list in a separate environment (used to group commands including environment modifications that do not affect the current environment)

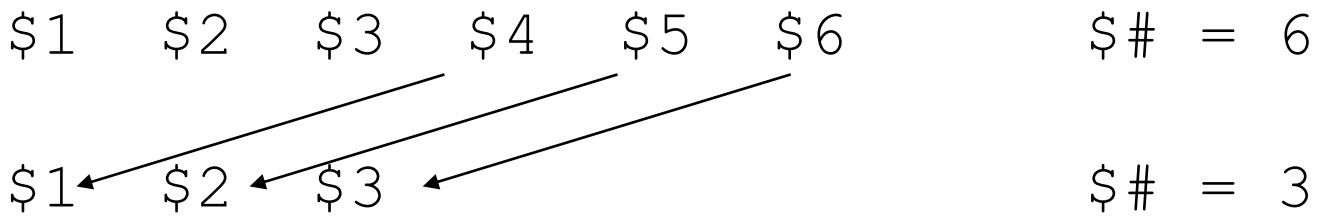
  Example: `( cd dir1; ls; )`

Command line arguments

- Access to command line arguments given to the script:
  - `$0` name of the script
  - `$1`, ..., `$9` first, ..., ninth argument to the script
  - `${10}`, ... tenth, ..., argument (ksh only; sh: use `shift` command)
  - `$#` number of arguments to the script (not incl. `$0`)
  - `$*` or `$@` all arguments (without `$0`) in a list on one line; all quoting characters are removed
  - `"$*"` all arguments (without `$0`) as one string
  - `"$@"` all arguments (without `$0`) as individual strings
  - `${#n}` length of value of argument *n* (ksh only)

Command line arguments

- `$@` and `$*` have the same meaning (list of all arguments with all quoting characters removed), but:
  `"$@"` is expanded to `"$1" "$2"` ... (`$#` separated strings, for processing in, e.g., `for` loops or `select` statements)
  `"$*"` is expanded to `"$1 $2 ..."` (one single string).
- `shift` command, e.g. `shift 3`:



  Beware that `shift` works destructively!

Command line arguments

Example: `command_line_arguments.ksh`

```
#!/bin/ksh

echo "The list of command line arguments is: $@"
echo "The first argument is $1"
shift 2
echo "The first argument is now $1"
```

Example output:

```
# ./command_line_arguments.ksh arg1 arg2 arg3
The list of command line arguments is: arg1 arg2 arg3
The first argument is arg1
The first argument is now arg3
```

Command line arguments

- Redefine command line arguments within the script using:

  `set [--] arg1 arg2 …`

  → `$1=arg1 $2=arg2 …`

  or alternatively:

  `set [--] $(ls)` from command substitution result

  `set [--] $var` from variable *var*

  It is recommended to use the `--` option always: otherwise, if the first argument to the command starts with – (dash), it will be misinterpreted as an option to `set`.

- Lexical ordering of arguments: `set -s`
- Delete all arguments: `set --`

`test` statement

`test` evaluates a condition and indicates the result of the evaluation by its exit status (0: condition true/"success", 1: false/"failure"). Examples:

- `test "$HOME" = "/home/cbauer"; echo $?`
- `a=10; test $a -eq 100; echo $?`
- `test -f /etc/passwd; echo $?`
- `[ -f /etc/passwd ]; echo $?`

Conditional expressions

- Example of an `if` statement

```
if grep bla blafile >/dev/null; then
  echo "String found"
else
  echo "String not found"
  # abort: return an error code
  exit 1
fi
```

Conditional expressions

- Example of an `if` statement

```
if test -f file1; then
  cp file1 file2
  echo "File found and copied"
else
  echo "File file1 not found"
  # abort: return an error code
  exit 1
fi
```

Conditional expressions

- Example of an `if` statement

```
if [[ -f file1 ]]; then
   cp file1 file2
   echo "File found and copied"
else
   echo "File file1 not found"
   # abort: return an error code
   exit 1
fi
```

Conditional expressions

```
if command1
then
  block of commands
elif command2
then
  block of commands
...
else
  block of commands
fi
```

Conditional expressions

- All on one line (hard to read except when statements are very short):
  `if ...; then ...; elif ...; then ...; else ...; fi`
- `if` and `elif` check the exit status of the associated command:
  "true" if 0, "false" if anything else than 0.
  Note: the associated command is actually performed!
- Exit status of `if` statement:
  - Exit status of last command executed (in `then` or `else` part)
  - If no command executed in `then` or `else` part: 0
- More examples in files `if.ksh`, `if_2.ksh`, `if_3.ksh`, `if_4.sh`

Conditional expressions

- Numerical comparison

sh and ksh	ksh only	true if
`[ $a -eq $b ]`	`(( a == b ))` `[[ a -eq b ]]`	a and b equal
`[ $a -ne $b ]`	`(( a != b ))` `[[ a -ne b ]]`	a and b not equal
`[ $a -gt $b ]`	`(( a > b ))` `[[ a -gt b ]]`	a greater than b
`[ $a -lt $b ]`	`(( a < b ))` `[[ a -lt b ]]`	a less than b
`[ $a -ge $b ]`	`(( a >= b ))` `[[ a -ge b ]]`	a greater than or equal b
`[ $a -le $b ]`	`(( a <= b ))` `[[ a -le b ]]`	a less than or equal b

sh, ksh: [ ]: is a command, equivalent to external `test` statement
ksh: (( )): is a command, equivalent to `let` statement (both ksh builtins)
ksh: [[ ]]: is a builtin enhancement of [ ] and `test`

Conditional expressions

- String comparison

sh and ksh	ksh only	true if
`[ "$a" = "$b" ]`	`[[ $a = $b ]]`	sh: a equals b ksh: a pattern-matches b
`[ "$a" != "$b" ]`	`[[ $a != $b ]]`	sh: a and b not equal ksh: s1 does not pattern-match s2
	`[[ $a > $b ]]`	a after b (lexicalic)
	`[[ $a < $b ]]`	a before b (lexicalic)
`[ -z "$a" ]`	`[[ -z $a ]]`	a is an empty string
`[ -n "$a" ]`	`[[ -n $a ]]`	a is not an empty string

ksh shows unexpected behaviour when using `[]` for string comparisons if `""` are omitted.

Conditional expressions

- File properties

sh and ksh	ksh only	true if
`[ -r file ]`	`[[ -r file ]]`	*file* readable
`[ -w file ]`	`[[ -w file ]]`	*file* writable
`[ -x file ]`	`[[ -x file ]]`	*file* executable
`[ -u file ]`	`[[ -u file ]]`	*file* has setuid bit set
`[ -g file ]`	`[[ -g file ]]`	*file* has setgid bit set
`[ -k file ]`	`[[ -k file ]]`	*file* has sticky bit set
`[ -s file ]`	`[[ -s file ]]`	*file* exists with size>0
`[ -f file ]`	`[[ -f file ]]`	*file* is a regular file
`[ -d file ]`	`[[ -d file ]]`	*file* is a directory
`[ -c file ]`	`[[ -c file ]]`	*file* is a character device
`[ -b file ]`	`[[ -b file ]]`	*file* is a block device
`[ -p file ]`	`[[ -p file ]]`	*file* is a named pipe

Conditional expressions

- File properties (ksh)

ksh only	true if
`[[ -O file ]]`	owner of *file* and EUID of process identical
`[[ -G file ]]`	owner of *file* and EGID of process identical
`[[ -S file ]]`	*file* is a socket
`[[ -L file ]]`	*file* is a symbolic link
`[[ -e file ]]`	*file* exists

Conditional expressions

- Logical operators

sh and ksh `[ ]`	ksh only `[[ ]]` and `(( ))`	meaning
`-a`	`&&`	AND
`-o`	`||`	OR
`!`	`!`	NOT

Logical expressions can be included in `( )` to force an order of evaluation.
Priority: `( )` then `!` then `-a` then `-o` (sh)
`( )` then `!` then `&&` then `||` (ksh)
Within `[ ]` brackets must be masked: `\(` and `\)`

Conditional expressions

- String variables in conditional expressions:
  `if [[ "$name" = "C. Bauer" ]] ...`
  → use " " so that the expression is correctly evaluated; *but*:
  `if [[ "$name" = C* ]] ...`
  → in " " shell wildcards lose their special meaning
- Use `$1`, `$2`,... to access command line arguments in conditional expressions

Conditional expressions

- `command_block_1 && { command_block_2; }`
  is the short form of
  ```
  if command_block_1
  then
      command_block_2
  fi
  ```
- `command_block_1 || { command_block_2; }`
  is the short form of
  ```
  if ! command_block_1
  then
      command_block_2
  fi
  ```

`case` statement

Example: `case.ksh`

```
#!/bin/ksh

TYPE=`ls -ld $1 | cut -c1`

case $TYPE in
  -)      echo "$1 is a regular file.";;
  d)      echo "$1 is a directory.";;
  c)      echo "$1 is a character device file.";;
  b)      echo "$1 is a block device file.";;
  p)      echo "$1 is a named pipe.";;
  l)      echo "$1 is a symbolic link.";;
  *)      echo "File type of $1 unknown.";;
esac
```

case statement

```
case value in
 pattern1)
   block of commands
 ;;
 pattern2)
   block of commands
 ;;
 ...
 *)
   block of commands for default action
 ;;
esac
```

`case` statement

- *`pattern`* is a sequence of characters, which may contain the known shell metacharacters (wildcards)
  - `* ? [ - ]` (sh, ksh)
  - `@( ),*( ),?( ),+( ),!( )` (ksh only)
- Example:

```
case $EXTENSION in
    *sh)          echo "The file is a shell script";;
    @(txt|dat))   echo "The file is a text or data file";;
    log)          echo "The file is a log file";;
esac
```

The `for` loop

Example: `for.ksh`

```
#!/bin/ksh

for VAR in string1 string2 string3 string4
do
  echo "String is $VAR"
done
```

Example output:

```
# ./for.ksh
String is string1
String is string2
String is string3
String is string4
```

The `for` loop

```
for var [in arg1 arg2 ...]
do
  block of statements
done
```

- Argument list empty: `for` loop automatically assigns `$1`, `$2`, ... to *var*
- Argument list:
  - explicit list of arguments, separated by spaces or tabs
  - contents of a variable (e.g. read with `read`)
  - the list of command line arguments: `$*` or `$@`
  - result of a command substitution: `$(cmd)` or `` `cmd` ``
  - list of files, produced, e.g., with `*` or `$(ls)` or `$(ls a*)` etc.

The `for` loop

- Examples of `for` loops

```
for VAR in *
do
  echo "File name is $VAR"
done

for ARG in $@
do
  echo "Argument is $ARG"
done
```

The `for` loop

Example: `for_3.ksh`

```
#!/bin/ksh

for VAR in $@; do
  echo "(1) Argument is $VAR"
done
for VAR in "$@"; do
  echo "(2) Argument is $VAR"
done
for VAR in "$*"; do
  echo "(3) Argument is $VAR"
done
```

The `for` loop

Example output:

```
# ./for_3.ksh arg1 arg2 "arg3 arg4"
(1) Argument is arg1
(1) Argument is arg2
(1) Argument is arg3
(1) Argument is arg4
(2) Argument is arg1
(2) Argument is arg2
(2) Argument is arg3 arg4
(3) Argument is arg1 arg2 arg3 arg4
```

The `while` loop

Example of a counting `while` loop: `while.ksh`

```
#!/bin/ksh

SECS=10
while (( SECS > 0 )); do
  echo "T minus $SECS seconds..."
  sleep 1
  (( SECS -= 1 ))
done
echo "LIFTOFF!"
```

The `while` loop

```
while condition
do
  block of statements
done
```

- *condition* can be any conditional check, e.g.:
  - `[ "$var" = "string" ]` (sh, ksh)
  - `[[ "$var" = "string" ]]` (ksh)
  - `[ $num -le 10 ]` (sh, ksh)
  - `(( num <= 10 ))` (ksh)
  - `true` infinite loop (`true` is always true)
  - `false` loop does not run at all (for debugging)

The `while` loop

- *`condition`* can be (cont.):
  - `(( num ))` shell continues to execute loop as long as *`num`* is not equal to 0 (ksh only)
  - `read var` loop with input: shell executes loop as long as `read` does not read EOF (CTRL + d)
- Loop to process command line arguments: e.g.

```
while (( $# )); do
  echo $1
  shift
done
```

The `until` loop

```
until condition
do
  block of statements
done
```

- *condition*: see `while` loop

`break` and `continue`

- `break` immediately exits the (innermost) loop within which it is executed and jumps to the first statement after the end of this loop; no further loop run is performed.
  `break` *n* breaks *n* levels.
- `continue` immediately proceeds with the next loop run of the (innermost) loop within which it is executed, without further processing of all commands in the loop after the `continue`.
  `continue` *n* resumes at the *n*th enclosing loop.

The `select` statement (ksh)

Example: `select.ksh`

```
#!/bin/ksh

PS3="Please make your choice (1-5): "

select CHOICE in Talisker Laphroaig Glenlivet Glenmorangie "Caol Ila"
do
  echo "Your choice is: $CHOICE."
done
```

The `select` statement (ksh)

Example: `select_2.ksh`

```
#!/bin/ksh

PS3="Please make your choice (1-3; 4 to exit): "

clear
select CHOICE in Talisker Laphroaig Glenlivet Exit
do
  echo "Your choice, numerical, is: $REPLY"
  case $CHOICE in
    "Talisker") echo "One of my favourite choices, the only one from Isle of Skye";;
    "Laphroaig") echo "Peaty and rich, an Islay pleasure";;
    "Glenlivet") echo "Good for starters, widely known and available";;
    "Exit") break;;
    *) echo "???";;
  esac
  REPLY=
done
```

The `select` statement (ksh)

```
select var in choice1 choice2 ...
do
  block of statements
done
```

- *`block of statements`* usually contains a `case` switch to process the choice.
- Selection prompt: variable `PS3` (default: `#?`)
- `select` acts as an infinite loop: interrupt by typing CTRL + d, a `break` statement within the statement block, or a dedicated menu item with a `break` statement
- Submenu: `select` within `select` (modify `PS3`)

Reading input

- `read var1 var2 ...`
  reads character sequences from stdin (or a pipe).
- Record separator: variable `IFS`
  (defaults to whitespace: space, tab, newline; however, newline is treated specially in `read` statements)
  Modify like, e.g.: `IFS=" . ! ? : "` (save old value with `OLDIFS=$IFS` before modification: newline cannot be entered into `IFS` within scripts!)
- Assignment of variables:
  - No. of tokens = no. of variables → each variable takes exactly one token
  - No. of tokens < no. of variables → last variables remain empty
  - No. of tokens > no. of variables → last variable takes the entire remaining string

Reading input

- `read` without arguments:
  - sh: error message
  - ksh: all user input automatically in variable `REPLY`
- User prompt:

  `read var1?"Prompt: " [var2 ...]`

  is the same as

  `echo "Prompt: \c"; read var1 [var2 ...]`
- A loop that reads lines of input from a file:

```
while read var1 var2 ...
do
  block of commands
done < filename
```

Reading input

Example: `while_read.ksh`

```
#!/bin/ksh

while read VAR1 VAR2 VAR3
do
  echo "VAR1 is: $VAR1"
  echo "VAR2 is: $VAR2"
  echo "VAR3 is: $VAR3"
done < INPUT
```

Input file `INPUT`:

```
eins zwei drei
eins zwei
eins zwei drei vier
```

Reading input

Example output:

```
# ./while_read.ksh
VAR1 is: eins
VAR2 is: zwei
VAR3 is: drei
VAR1 is: eins
VAR2 is: zwei
VAR3 is:
VAR1 is: eins
VAR2 is: zwei
VAR3 is: drei vier
```

Here document

- "Here document": command processes a bunch of input lines

```
sqlplus -s /nolog >/dev/null << END_OF_SQL
connect ${DBUSER}/${DBPW}@$ORACLE_SID
set colsep ';'
set linesize 1000
set pagesize 0
set feedback off
spool $SQLOUTFILE
select * from customer where customer_no='1234567';
spool off
exit
END_OF_SQL
```

The string indicating the end of the "here document" must be at the beginning of the line, without leading whitespace (a common error if it is not).

Useful commands in shell scripts

- `sort`: sort lines of input lexically, numerically, ... based on one or more sort keys extracted from the input lines
- `tr`: translate characters into other characters
- `wc`: count lines, words and characters in a file
- `tee`: "tee junction", write output to file before further processing it
- `cut`: cut out selected fields from each line of a file
- `uniq`: report or filter out repeated lines in a file
- `xargs`: construct command lines by taking arguments from stdin (e.g. produced as the output of another command such as `find`)
  Examples:
  ```
  ls | xargs cat
  ls | xargs -n 100 rm
  ```
- See man pages for further information.

Useful commands in shell scripts

- `grep`, `egrep`, `fgrep`: search for regular expressions (`grep`, `egrep`) or fixed strings (`fgrep`)
- `sed`: stream editor
- `[n]awk`: record-oriented textual data manipulation

Examples for applying these three will be shown on the next slides.

[e,f]grep

Commands: `grep`, `egrep`, `fgrep`

`[e,f]grep 'searchstring' file[s]`

- "grep" stands for "Global Regular Expression Parser" (or "Globally search for a Regular Expression and Print if found").
- `grep` is the regular form of `grep`.
- `egrep` means "expression grep" (not enhanced or extended grep).
  `egrep` may be very memory-consuming.
- `fgrep` means "fast grep."
  `fgrep` can only search for fixed character sequences, does not perform any regular expression pattern matching. Therefore it is faster than `grep` and `egrep`.
- Quoting (`'  '`) of the search string is recommended.
  The shell may interpret some of the RE metacharacters.

[e,f]grep

- Simple examples of using `grep`:

```
grep 'Bauer' /etc/passwd
grep '[Bb]auer' /etc/passwd
grep '[Bb]...r' /etc/passwd
grep '[Bb].*r' /etc/passwd
grep '^[Bb]...r$' /etc/passwd
grep '^[Bb]...r\$$' /etc/passwd
grep '[A-Z][a-z]..r' /etc/passwd
```

[e,f]grep

- Common options of [e,f]grep
  - -n show line numbers
  - -i ignore case of letters
  - -v show all lines that do *not* contain the search pattern
  - -l show only names of files that contain the search pattern
  - -c count the lines that contain the pattern

Regular expressions

- Combinations of ASCII characters, some of which have special meanings
- Special characters (metacharacters) common to all classes of RE:

Symbols	Meaning
`.`	exactly one arbitrary character
`*`	preceding character arbitrarily often (including none)
`^`	beginning of line
`$`	end of line
`\`	masks the one following character
`[ - ]`	exactly one from this list (range) of characters
`[^ - ]`	exactly one character, but not from this list (range)

Within `[ ]` the special characters `.` `*` `[` and `\` lose their meaning.

Regular expressions

- `grep`/`ed`/`sed` RE ("limited RE") only

Symbols	Meaning
`\(  \)`	save a pattern for later reuse
`\`*n*	insert *n*th saved pattern here (*n*=1 to 9)
`\{  \}`	repetition of preceding character or pattern: `\{`*m*`\}`: exactly *m* times, `\{`*m*`,\}` and `\{,`*m*`\}`: at least/most *m* times, `\{`*m*`,`*n*`\}`: from *m* to *n* times; *m*, *n*=0 to 255
`\<  \>`	beginning and end of word

Regular expressions

- `egrep`/`awk` RE ("full RE") only

Symbols	Meaning
( )	grouping of characters/expressions
|	separator between choices (e.g. in (|))
+	preceding character/pattern at least once
?	preceding character/pattern at most once

sed

`sed [opts] '[adr]act[arg]' file[s] [>output]`

- Purpose: Stream Editor (read input file, process/edit, and write to stdout)
- Common options:

`-n`	suppress default output
`-f file`	read `sed` commands from a file

- *adr*: addressing of lines (which lines to process):

`3,5`	lines 3 up to (and including) 5
`1,$`	all line numbers (entire file)
/RE/	all lines containing this regular expression
/RE/,$	all lines from first containing this RE up to end of file
/RE1/,/RE2/	all lines from first containing RE1 up to the first subsequent line containing RE2

sed

- *act*[*arg*] : Action to perform with the addressed lines:

`d`	delete lines (and print those that are not deleted; with `-n` no output at all)
`p`	print lines (here `-n` is required: default action is print all lines)
`s/`*RE*`/`*str*`/`	replace first occurrence of RE with *str*
`s/`*RE*`/`*str*`/`*n*	replace *n*th occurrence of RE with *str*
`s/`*RE*`/`*str*`/g`	replace all occurrences of RE with *str*
`s/`*RE*`/&`*str*`/g`	append *str* to all occurrences of RE
`s/.../.../...w` *file*	save the modified lines to file (only with `s` action)
`r` *file*	insert the file after the addressed line

It is possible to store sub-patterns in the first RE that are reused in *str*. E.g.:

```
sed 's/user\([12]\)/user_\1/g'
```

replace all `user1` by `user_1` and all `user2` by `user_2`

sed

- `sed` commands from a file:
  With the file

```
1,4d
s/^east/East/
```

  the command `sed -f file ...` has the same effect as:
  `sed -e '1,4d' -e 's/^east/East/' ...`

awk

`awk 'statement' inputfile`

- Purpose: record-by-record (line-by-line) textual data manipulation
- *statement*: an expression such as

pat{*act*}	perform action for all records matching the pattern
pat	print all lines matching the pattern
{*act*}	perform action for all records

  The pattern is a regular expression in / /.

`awk -f scriptfile inputfile`

- *scriptfile:* a list of awk commands

  pattern{*action*}
  pattern{*action*}
  ...

awk

Important application: formatted printing of selected columns from files

Example:

`awk '/root/{print $1,$5"\t"$6}' inputfile`

prints the first, fifth and sixth column of all lines containing "root".

- `$0` means the entire record (by default, a line of input).
- `$1`, `$2`, ... are the first, second, ... columns in the record (default separator whitespace: blanks and/or tabs).
- Comma translates to a single space in the output.
- `\t` is a tab character in the output, `\n` a newline, `\042` a " character and `\044` a `$` character.

awk

- `BEGIN` pattern (one or several): action(s) performed before the processing of the first input record; example:

  ```
  awk 'BEGIN{FS=" :"}; further statements'
  ```

  changes the field separator to one of space or `:` (or combinations of these).
- `END` pattern (one or several): action(s) performed after processing the last input record.

  ```
  awk 'BEGIN{count=0};/root/{print NR,$0; \
  count=count+1};END{print "Count:",count}' \
  /etc/passwd
  ```

awk

- User-defined variable names: everything but predefined variables and function names; references to contents without `$`.
- Predefined:

`NF`	number of fields in current record (line)
`NR`	number of input records read so far (from beginning of first input file)
`FS`	input field separator (default: space or tab)
`OFS`	output field separator (default: space)

- Variables are automatically preset to null strings (interpreted as 0 in arithmetic ops.) when `awk` first encounters them.
- `awk` provides a powerful programming language of its own with flow control structures, formatted printing, arithmetic operations etc. etc.

awk

Example awk script example.awk:

```
#!/bin/awk -f
# Call this script with one argument: the input file
BEGIN {
  print "Start processing the input file.";
  ncount=0; rcount=0; FS=":"
};
/nologin/ {
  ncount=ncount+1;
  print "Record",NR,": user name",$1,", UID",$3;
};
/root/ {
  rcount=rcount+1;
};
END {
  print "The number of nologin records is",ncount;
  print "The number of root records is",rcount;
};
```

awk

Example output:

```
# ./example.awk /etc/passwd
Start processing the input file.
Record 2 : user name bin , UID 1
Record 3 : user name daemon , UID 2
(…)
The number of nologin records is 84
The number of root records is 1
```

Input and output redirection: file descriptors

- A file descriptor is an integer number by which a process (shell) refers to a file opened for reading or writing.
- `exec n>file` — open file for output and assign file descriptor *n*
- `exec n<file` — open file for input and assign file descriptor *n*
- `exec n<>file` — open file for input and output, assign f.d. *n*
- `... >&n` — write (append) stdout to file descriptor *n*
- `read <&n var` — read from file descriptor *n* and store in *var*
- `exec n<&-` — close file descriptor *n*
- The maximum value for the file descriptor number *n* is 9.

Functions

sh syntax:

```
name()
{
  block of statements
}
```

ksh syntax:

```
function name
{
  block of statements
}
```

Functions

- Functions are called like this: `name arg1 arg2` ...
- No explicit declaration of an argument list
- Arguments (positional parameters `$1`, `$2`, ...) given to the function are only defined within the function ("locally") and must be distinguished from those of the shell (which remain "alive" until after the function processing).
- Functions defined within a script are only known there.
- Show all functions (ksh): `functions` or `typeset -f`
- Show all known function names (ksh): `typeset +f`
- Delete a function definition: `unset -f name`
- `return n` exits the function and returns the exit status `n` to the calling script.

Functions

- External functions (function autoload; ksh):

  1) Put each function into one file, file name = function name

  2) Put all the files (and only these files) into one directory

  3) Set permissions rwx for the owner on these files

  4) Define the environment variable `FPATH` holding the path name of this directory; `FPATH` may contain several directories.

Functions

- Function inheritance (ksh)
  - Functions defined within a script are only known within the script.
  - Function definitions can be stored in `.kshrc` (the file specified with `ENV`). They are read by shells subsequently started with `ksh` or `#!/bin/ksh`.
  - Functions can be exported using `typeset -f -x` *`name`* (display all exported functions with `typeset -f -x`). These functions are only inherited by sub shells which are started by invoking a script without `#!/bin/ksh` in the first line.
- Scope of variables (ksh)
  - Variables defined outside of functions: valid globally (in the function, too).
  - Variables defined within a function: valid globally (outside the function, too).
  - Variables defined with `typeset [`*`opt`*`]` *`var`* within a function: *local*.

Signals and traps

- Important signals (see `signal(3head)` for Solaris or `signal(7)` for Red Hat):

Name	No	Def. Act.	Description
SIGHUP	1	Exit	Hangup
SIGINT	2	Exit	Interrupt (CTRL+c)
SIGQUIT	3	Core	Quit (CTRL+\)
SIGFPE	8	Core	Arithmetic Exception
SIGKILL	9	Exit	Kill
SIGBUS	10	Core	Bus Error (Solaris)
SIGSEGV	11	Core	Segmentation Fault
SIGTERM	15	Exit	Terminate
SIGUSR1	16	Exit	User Signal 1
SIGUSR2	17	Exit	User Signal 2
SIGSTOP	23	Stop	Stop (signal; CTRL+s)
SIGTSTP	24	Stop	Stop (user; CTRL+z)
SIGCONT	25	Ignore	Continue (CTRL+q)

Signals and traps

- List all signals: `kill -l` (letter "ell")
- KILL and STOP cannot be blocked (system enforces this silently), their handlings cannot be altered.
- `trap 'action command' signal`
  - `'action command'` can extend over several lines.
  - `trap` without arguments: display current (non-default) settings
  - `trap - signal` (ksh) or `trap signal` (sh, ksh): restore default action
  - `trap '' signal`: tell shell to ignore this signal (action is null string).
  - `trap 'action' ERR`: catch all errors during shell execution, whenever a command returns an exit status ≠ 0 this action is performed (→ debugging).

`getopts` (ksh)

Processing of (single-character) options given to a script

First: a convention

- `-` *`option`* "set a flag"
- `+` *`option`* "remove/unset a flag" (ksh only)

Second: an example, options without arguments

```
while getopts xy OPT_CHAR; do
  case $OPT_CHAR in
    x) print "Option is -x";;
    y) print "Option is -y";;
    +x) print "Option is +x";;
    +y) print "Option is +y";;
  esac
done
```

`getopts` (ksh)

- `xy` is the list of valid options
- The `while` loop ends when all actually given options have been processed.
- `getopts :xy` → processing of invalid options switched on
  When `getopts` finds an invalid option, `OPT_CHAR` is set to `?` (in `case` statement: access with `\?`). The shell variable `OPTARG` contains the invalid option (ksh only).
- Option requires an argument: insert `:` character after option character:
  `while getopts x:y …`
  (`x` expects an argument). When running the shell script, the argument must immediately follow the option character (with or without spaces).
  The argument value is stored in the variable `OPTARG`.

`getopts` (ksh)

- Handle an option that was given without argument although it requires one: insert this `case` branch (ksh only):

```
…
   :) print "Option $OPTARG requires an argument."
 …
   ;;
…
```

If this is not defined (or in sh), the option will be ignored without any message.

`wait` (sh and ksh)

- Stop script execution until a child process has ended
- Synopsis: `wait [PID1 [ ... ] ]`
- `wait` returns the exit status of the awaited process.
- No argument: wait for all background processes to end
  (exit status of `wait` is always 0 then)
- More than one argument: exit status of `wait` will be exit status of last PID.

`wait` (sh and ksh)

Example (similar to `wait_2.ksh`):

```
#!/bin/ksh
/usr/openwin/bin/xterm -bg red &; PID1=$!
/usr/openwin/bin/xterm -bg green &; PID2=$!
if wait $PID1; then
  echo "1st proc, PID $PID1 exited successfully."
else
  echo "1st proc, PID $PID1 exited with failure."
fi
# Shell script execution will not reach this point until
# first xterm has exited.
if wait $PID2; then
  echo "2nd proc, PID $PID2 exited successfully."
else
  echo "2nd proc, PID $PID2 exited with failure."
fi
echo "All xterm processes have exited."
```

Parsing order (ksh)

1) Decomposition of the command line into tokens and recognition of key words (`if`, `else`, `fi`, `do`, `done` etc.) and special characters (`<`, `>`, `&`, `"`, `'`, `` ` `` etc.)
2) Replacement of aliases, if required
3) Identification of builtin commands (`alias`, `bg`, `cd`, `echo` etc.)
4) Identification and replacement of functions
5) Tilde replacement (as in `~user`)
6) Variable substitutions (e.g. `$HOME`)
7) Execution of command substitutions (`` ` ` `` or `$( )`)
8) Calculation of arithmetic expressions
9) Execution of file name substitutions (`*`, `?`, `@( )` etc.)
10) Replacement of quoting/masking characters
11) Localisation of the command (`$PATH`) and execution

Debugging a script

- Option to the shell, on first line; for example: `#!shell -x`
- Or within the script (effective only for a section of the script):
  - `set -x` (sh, ksh) or `set -o xtrace` (ksh only)

    ```
    ...                 ...
    set +x              set +o xtrace
    ```

    → reply all cmds. after metacharacter and variable substitution (same as `-x` option for the shell)

  - `set ±v` (sh, ksh) or `set ±o verbose` (ksh only)

    → reply all cmds. before metacharacter and variable substitution

    `xtrace` and `verbose` options may be combined.

  - `set ±f` (sh, ksh) or `set ±o noglob` (ksh only)

    → switch off shell wildcards for file names
- Use `#!/bin/ksh -p` on first line and set all variables explicitly.