Chapter 3

Shell Programming

Command Interpreter

- The **command interpreter** is the user interface of an operating system
  - May or may not be part of the OS kernel
- Kinds of command interpreters:
  - Mouse-based window and menu system
  - Command line interpreter (called a **shell** in Unix)
    - `sh` — the original UNIX shell — the “Bourne shell”
    - `ksh` — Korn shell
    - `zsh` — Z-shell
    - `bash` — the GNU shell — “Bourne-Again SHell”
    - `csh` — the C-shell; with CLE: `tcsh`

BASH

- The GNU shell: “Bourne-Again SHell”
- (In-terminal) read-eval-print-loop
- **Executing commands**, supplying arguments
- **Composing** commands
- ***Programming:*** writing **shell scripts**

  (**Note:** `sh` better than `csh` for programming)

Invoking Commands

- Traditional UNIX command invocation:

  <command> <options> <arguments>
- Space separation is essential
- Special characters (for the shell) can be escaped using backslash “\”
- Options start with “-”
- <command> can be:
  - an absolute path “/usr/bin/echo”
  - a relative path “../../bin/echo”, “./argv”
  - a command name (without “/”)

Commands

Commands can refer to:

- Executable files
  - Executable machine-code programs
  - Interpreted scripts

    (starting with “#!*interpreter*” or shell scripts)
- Shell built-in commands
- Shell aliases
- Shell functions

Command-Line Arguments in C

```
#include <stdio.h>                             // args.c
int main(int argc, char *argv[]) {
  for (int i = 0; i < argc; i++)
    printf( "argv[%d] = “%s”\n", i, argv[i]);
  return 0;
}
```

- *argv* contains the **whole** command line
- char * *argv*[] can be used as an array with *argc* elements:
  - *argv*[0] is the **command**, and
  - *argv*[1] … *argv*[*argc*–1] are the arguments
  - the number of arguments is (*argc*–1)

Command-Line Arguments in Haskell

```
#!/usr/bin/runhaskell
import System                           -- Args.hs

main = do
  getProgName >>= putStrLn ∘ ("ProgName: " ⧺ )
  args ← getArgs
  putStrLn $ unlines $ zipWith f [1 .. ] args
    where f i n = show i ⧺ ": " ⧺ show n
```

Command line parts are accessed separately:

- *System.getProgName* :: *IO String*
- *System.getArgs* :: *IO* [*String*]

Command-Line Arguments in Shell Scripts

```
#!/bin/sh
# This is Args.sh

echo "Progname: \‘\‘$0”"
for i in $*
do
  echo "\‘\‘$i”"
done
```

Command line parts are accessed in different ways:

- `$0` is the command name
- `$1`, …, `$9` are the first nine arguments
- `$*` is the whole argument list
- `shift` replaces `$*` with its tail

I/O Channels

- UNIX programs have at least the following I/O Channels (***file descriptors***) available to them:
  - **0:** *stdin*, the **standard input channel** — (keyboard)
  - **1:** *stdout*, the **standard output channel** — (terminal)
  - **2:** *stderr*, the **standard error channel** — (terminal)
- All file descriptors can be **redirected** in the shell
- Simple redirection “<” for *stdin* and “>”, “>>” for *stdout*
- Merging *stdout* and *stderr* into a single file:

  ```
  doSomeThing >alloutput 2>&1
  ```

  Short form: `doSomeThing &>alloutput`

Conditionals and “Shell-Truth”

```
#!/bin/sh
if true
then
  echo "Good Morning!"
else
  echo "Good Afternoon!"
fi
```

- **Conditional:** if ... ; *then* ... ; { *elif* ... ; *then* ... ;} [else ... ;] *fi*
- true does nothing and returns successfully
- false does nothing and returns without success
- Returning successfully ⇔ exit status **0**

Pipes

- `command1 | command2`
- Connects *stdout* of `command1` with *stdin* of `command2`
- Pipeline length is unlimited

  ```
      find . -type f | xargs grep pattern |
  awk '{print $2}' | sort -u | less
  ```

- Frequently line-oriented list processing
- “Lazy evaluation”: `command1` only continues writing if `command2` keeps reading

  ```
  find / -type f | head -n 5
  ```

Tests

- `test` *expression*
- `[` *expression* `]`

- propositional junctors
- unary and binary string predicates
- unary and binary integer predicates
- file system predicates: existence, permissions, file types

Composition and Conditional Execution

- **Sequence:** `command1 ; command2`
- **Conditional on success:** `command1 && command2`
- **Conditional on failure:** `command1 || command2`
- **Group command:** `{list; }`
- **Subshell:** `(list)`

Loops

- `for i in w1 w2 w3 … ; do list; done`
- `while list; do list; done`
- `until list; do list; done`
- `for (( expr1 ; expr2 ; expr3 )) ; do list ; done`

`man bash`

- **Options:** shell mode, and `set` options
- **Invocation:** shell modes, initialisation, `sh` emulation
- **Shell Grammar:** simple commands, pipelines, lists, compound commands, function definitions
- **Quoting:** special characters in different contexts
- **Parameters:**
  - positional parameters (arguments)
  - special parameters $[*@#?-$0_]
  - shell variables `$BASH*`, `$HOSTNAME`, `$PWD`, `$HOME`, `$PS1`
  - user-defined variables
  - array variables
- **Expansion:** ***“seven kinds”!***

`man bash` (ctd.)

- **Redirection:** *command args* < *infile* > *outfile* 2>> *errfile*
- **Aliases:** “The rules […] are somewhat confusing”.

  “For almost every purpose, aliases are superseded by shell functions.”
- **Functions:** local variables need to be declared **local**
- **Arithmetic evaluation, conditional expressions**
- **Simple command expansion, command execution environment**
- **Signals, job control**
- **Prompting, readline, history, history expansion**
- **Shell builtin commands**

Expansion

- **Brace expansion:** `a{d,c,b}e` $\rightarrow$ `ade ace abe`
- **Tilde expansion:** Home directories `~user` and `~`
- **Parameter and variable expansion:** `$varname`
  with default: `$varname:-default`
  various string manipulations possible
- **Command substitution:** `$(command)` or `‘command‘`
- **Arithmetic expansion:** `$((expression))`
- **Process substitution:** (obtaining FIFOs for processes)
- **Word splitting:** results of parameter expansion, command substitution, and arithmetic expansion are split into words (outside `"..."`)
- **Pathname expansion (globbing):** `dir*/file?.[coh]`
- **Quote Removal**

Bash Summary

- Complex language
- Context-sensitive lexing
- Complete imperative control structures
- Mostly dynamic binding (static binding with local)
- Iterated expansion mechanisms — functional flavour
- Concise syntax for command-line interaction
- Shell scripts **need documentation!**
- Shell scripts **need robustness!**
- Shell scripts **need security awareness!**