**UNIVERSITY OF MALTA**
**FACULTY OF INFORMATION & COMMUNICATION TECHNOLOGY**
**DEPARTMENT OF COMPUTER SCIENCE**

# CPS1012 Assignment : `eggshell`

**Keith Bugeja, Joshua Ellul, Alessio Magro, Kevin Vella**

---

**Instructions:**

1. This is an **individual** assignment and carries **40**% of the final CPS1012 grade.
2. The report and all related files (including code) must be uploaded to the VLE by the indicated deadline. Before uploading, archive all files into a compressed format such as ZIP. It is your responsibility to ensure that the uploaded file and all contents are valid.
3. Everything you submit must be your original work. All source code will be run through plagiarism detection software. Please read carefully the plagiarism guidelines on the ICT website.
4. Reports (and code) that are difficult to follow due to poor writing-style, organisation or presentation will be penalised.
5. Always test function return values for errors; report errors to the standard error stream.
6. The Unix **man** command is your best friend, **Google search** your second best.
7. Each individual might be asked to present their implementation, at which time the program will be executed and the design explained. The presentation outcome may affect the final marking.

# Part I

## Introduction

A Unix shell is a command-line interpreter that provides users with the ability to control the operation of the computer through text commands. The goal of this assignment is to implement `eggshell`, a small and lightweight Linux shell, described in the second part of this document. This assignment will be evaluated on a number of criteria, including correctness, structure, style and documentation. Therefore, your code should do what it purports to do, be organised into functions and modules, be well-indented, well-commented and descriptive (no cryptic variable and function names), and include a design document describing your solution.

## Overview

There are two parts to this document; in the first part, Unix shell fundamentals are briefly explored, using examples from the Bourne-again shell (Bash) to illustrate how certain `eggshell` features are meant to be implemented. The second part describes `eggshell` in terms of its five cardinal components: shell variables, command-line interpreter, input and output redirection, piping and process management. Each component is broken down into its constituent features, which are sorted in order of (perceived) difficulty, where possible.

## Shell Fundamentals

In an interactive shell, a command prompt informs the user that the shell is ready to accept commands; the user then proceeds to type in commands, getting back output in return:

```
host@user:~$ command arg0 arg1 arg2 ...
output from command
some more output
and even more output...
# Command prompt : ready to accept user input
host@user:~$
```

Commonly, user input takes the form of a `command` followed by a sequence of arguments (`arg0 arg1` and so on). These commands are categorised into *internal / builtin* commands – commands the shell knows about and can execute without any external dependencies (e.g. `cd`) – or *external* commands, which the shell knows nothing about. In fact, external commands are merely binary program images that live somewhere on the file system and the shell can launch for the user. An external command may be specified through an absolute path (e.g. `/bin/ls`) or relative to some entry in the file system (e.g. `ls`). In the latter case, where ambiguity may arise, the shell employs a resolution strategy which iterates through a list of search paths in some predefined order, looking for the program binary every step along the way. These search paths are stored in one of a number of configuration variables the shell employs, called the `PATH`. These variables are alternatively known as environment variables. A typical path on a Unix-like system reads as follows:

```
$ echo $PATH
/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/opt/X11/bin
```

These variables may be modified by the user through an assignment statement:

```
$ PATH=/usr/bin:/bin:/home/luffy
$ echo $PATH
/usr/bin:/bin:/home/luffy
```

The assignment overwrites the value of a variable with the right hand side of the assignment; if the variable doesn't yet exist, it will be created.

```
$ echo $SOMEVAR

$ SOMEVAR=something
$ echo $SOMEVAR
something
```

When a program is launched, the shell or environment variables are made available to it. From within a C program, there are two ways by which these variables may be accessed: the `environ` external declaration

```
extern char **environ;
```

and the third argument of the `main` function.

```
int main(int argc, char **argc, char **env)
```

Additionally, programs launched by the shell have a number of pre-connected communication channels made available to them at start-up; these communication channels, or streams, exist to abstract I/O operations, irrespective of the actual device these channels are wired to. There are three such channels: *standard input (stdin)*, *standard output (stdout)*, and *standard error (stderr)*. Initially wired to the terminal, the shell provides redirection operators to override the default bindings of these streams: `<, <<<, >, >>`.

```
$ sort < myfile.txt # Redirect input from file
$ sort <<< 'one
  two
  three' # Use 'here strings' for input redirection
$ cpp myfile.c > myfile.i # Redirect output to file
$ webserver >> ws.log # Redirect and append output to file
```

Furthermore, distinct streams from multiple programs can be wired together in a pipeline, such that the output of one process becomes input to the next, forming a chain of producers and consumers. This is accomplished through the pipe `|` operator:

```
$ echo $PATH | tr : '\n' | grep usr
/usr/bin
/usr/sbin
/usr/local/bin
```

A shell returns control to the user once a program has done executing; programs that block the shell while executing are said to be running in the foreground:

```
$ webserver
Web server running
...
Web server terminating
# Shell prompt; user given control after completion
$
```

The shell can be made to return control to the user immediately after launching a program, even though the latter will not have terminated. Such programs are said to be running in the background; a shell launches a program as a background process by appending the background `&` operator at the end of the command string:

```
$ webserver &
[1] 34003
Web server running
# User given control immediately after launch
$
...
Web server terminating
[1]+ Done
$
```

The shell provides mechanisms for suspending or interrupting processes running in the foreground. Typically such a process is suspended on receiving `SIGTSTP`, triggered via Ctrl + Z or interrupted when Ctrl + C are pressed and `SIGINT` is triggered. Suspended processes may be resumed either as foreground processes, using `fg`, or as background processes, using `bg`.

```
$ webserver
Web server running
# Ctrl + Z
[1]+  Stopped                 webserver
$ jobs
[1]+  Stopped                 webserver
$ bg 1        # Resume job 1, webserver, as a background process
[1]+ webserver &
$ jobs
[1]-  Running                 webserver &
```

For further information on Unix shells, the reader is referred to the Linux `man` pages.

## Deliverables

Upload your source code (C files together with any accompanying headers), including any unit tests and additional utilities, through VLE by the specified deadline. Include makefiles with your submission which can compile your system. Make sure that your code is properly commented and easily readable. Be careful with naming conventions and visual formatting. Every system call output should be validated for errors and appropriate error messages should be reported. Include a report describing:

- the design of your system, including any structure, behaviour and interaction diagrams which might help;
- any implemented mechanisms which you think deserve special mention;
- your approach to testing the system, listing test cases and results;
- a list of bugs and issues of your implementation.

Do **not** include any code listing in your report; only snippets are allowed.

# Part II

## `eggshell`

In your assignment, you are to implement `eggshell`, a simple command-line interpreter for the Linux operating system. There are five sections in this specification document: shell variables, commands (internal and external), output redirection, piping and process management.

## Shell Variables

A shell variable is a character string to which some value is assigned. The name of a shell variable can contain letters (a-z, A-Z), numbers (0-9) or the underscore character (_); conventionally, Unix shell variables are expressed in UPPERCASE.

(a) The shell is expected to manage the following variables:

(i) `PATH` - The search path used to launch external commands; subpaths are delimited by a colon, e.g. `/usr/bin:/bin:/usr/local/bin`.

(ii) `PROMPT` - The string presented to the user to show that the shell is ready to accept command input, e.g. `eggshell-1.0 >`.

(iii) `CWD` - The current working directory; all file operations are relative to this path, e.g. `/home/student/cps1012`.

(iv) `USER` - The username of the current user, e.g. `student`.

(v) `HOME` - The home directory of the current user, e.g. `/home/student`.

(vi) `SHELL` - The shell of the current user; should be the absolute path of the `eggshell` binary, e.g. `/home/student/cps1012/bin/eggshell`.

(vii) `TERMINAL` - The name of the terminal attached to the current `eggshell` session; for example: `/dev/pts/1`.

(viii) `EXITCODE` - The exit code returned by the last program run by the shell.

Note that variables unique to `eggshell`, such as `TERMINAL` or `CWD`, should be populated when the shell starts up.

(b) A user should be allowed to modify the content of shell variables or create new ones by using an assignment statement, `VARIABLE=VALUE`; for example:

```
$ USER=stevethreads
$ HOME=/home/steve
$ PATH=/usr/bin:/bin
```

(c) Shared variables appearing as command arguments or on the right-hand side of an assignment are prefixed with a $; the shell should recognise these occurrences and perform textual replacement with the variable value:

```
$ RANDOM_NAME=Billy
$ USER=$RANDOM_NAME
$ print USER
USER
$ print $USER
Billy
```

## Command-line Interpreter

The command-line interpreter in `eggshell` recognises a number of internal (or builtin) commands; if a command is not recognised as such, the shell treats the input as a call to an external command.

(a) Implement command-line input and tokenisation; the `linenoise` utility is suggested as a replacement for `readline`. A skeleton implementation is provided below:

```c
#define MAX_ARGS 255

char *line,
     *token = NULL,
     *args[MAX_ARGS];

int tokenIndex;

while ((line = linenoise("prompt> ")) != NULL)
{
        token = strtok(line, " ");

        for (tokenIndex = 0;
             token != NULL && tokenIndex < MAX_ARGS - 1;
             tokenIndex++) {
                args[tokenIndex] = token;
                token = strtok(NULL, " ");
        }

        // set last token to NULL
        args[tokenIndex] = NULL;

        while(tokenIndex --> 0) {
                printf("Arg %d = [%s]\n", tokenIndex, args[tokenIndex]);
        }

        // Free allocated memory
        linenoiseFree(line);
}
```

### Internal Commands

This section describes the `eggshell` internal commands. Your implementation should trap and handle errors; when a command fails, the user should be made aware through an appropriate error message.

(a) Implement the `exit` command, to terminate all running processes spawned by the `eggshell` instance and quit the program.

(b) Implement the `print` internal command, to echo text expressions or variable values to standard output.

(i) Echo text to standard output:

```
$ print hello, this is a test!
hello, this is a test!
```

(ii) Print the value of a shell variable:

```
$ print $PATH
\usr\bin:\bin:\home\student
$ print $USER
student
```

(iii) Print combined text and shell variable values (except in the case of string literals):

```
$ print "hello $USER, welcome to eggshell!"
hello $USER, welcome to eggshell!
$ print hello $USER, welcome to eggshell!
hello student, welcome to eggshell!
```

(c) Implement the `chdir` command to change the current working directory (see Shell Variables, `CWD`).

```
$ print $CWD
/home/student/cps1012
$ chdir ..
$ print $CWD
/home/student
$ chdir /home/student/cps1012/bin
$ print $CWD
/home/student/cps1012/bin
```

(d) Implement the `all` command to print all shell variables, in key-value pairs, to standard output, one pair per line:

```
$ all
$ PATH=/usr/bin:/bin:/home/student
$ PROMPT=eggshell-1.0>
$ SHELL=/home/student/cps1012/bin/eggshell
$ USER=student
$ HOME=/home/student
$ CWD=/home/student/cps1012
$ TERMINAL=/dev/pts/1
```

(e) Implement the `source` command to equip `eggshell` with scripting functionality. The command takes a single argument, specifying the name of the script file. It then proceeds to open

this file and execute commands from it, one line at a time, until the end-of-file is encountered. Each executed command should behave exactly in the same way as if it were typed in.

**Hint:** You should abstract command input from parsing and execution, to make the implementation of the `source` command more straightforward - a change of input modality.

### External Commands

The shell assumes commands not included in the list above to be external commands, and will search for a matching program binary to launch it as a separate process.

(a) Use the *fork-plus-exec* pattern to execute unrecognised internal commands as external commands.

(b) Use the system search path (`PATH`) to look for command binaries.

(c) Verify that programs launched from `eggshell` contain the `eggshell`-specific shell variable definitions (e.g. `TERMINAL` and `CWD`).

## Input and Output Redirection

A very powerful feature of Unix shells is the ability to redirect output and input to and from files. Likewise, `eggshell` supports a number of redirection operators:

(a) Implement output redirection operators `>` and `>>`:

`command` `>` `file` - Redirect the output from `command` into `file`.

`command` `>>` `file` - Redirect the output from `command` into `file`, appending to its current contents.

(b) Implement input redirection operators `<` and `<<<`:

`command` `<` `file` - Use the contents of `file` as input to `command`.

`command` `<<<` `text` - Use `text` as a *here string* input to `command`.

**Note** that `command` refers to an arbitrary internal or external command, with zero or more arguments. Likewise, `file` refers to an arbitrary file.

## Piping

In Unix-like computer operating systems, a command pipeline is a sequence of processes chained together by their input and output streams, so that the output of one process (stdout) feeds directly into the input (stdin) of the next, and so forth.

(a) Implement the pipe `|` operator:

`command1` `|` `command2` - The output of `command1` feeds the input of `command2`.

(b) Make sure your implementation works for an arbitrary number of commands:

`command1` `|` `command2` `|` . . . `|` `commandN`

## Process Management

The ability to interrupt or suspend a process is very important; typically, shells reserve two keyboard shortcuts, CTRL+C and CTRL+Z, to signal running processes with an interruption or a suspension signal respectively.

(a) Provide functionality for trapping interrupt (`SIGINT`) and suspend (`SIGTSTP`) signals; forward these signals to the process currently running in the foreground.

(b) Allow resumption of suspended processes as either background or foreground processes, using `bg` or `fg` respectively.

(c) Add the ability to run background processes to the program launcher; background processes are denoted by the use of the `&` operator at the end of a command string.

---



*end of paper*