University of Canberra
Faculty of Information Sciences and Engineering

Computer Engineering 1 — 1997
Part 1 : Introductory Programming

Assignment 3
Due: Friday, 23rd May 1997

Unix `grep` Utility

Aims

The aim of this exercise is to develop a Haskell program that implements a restricted variant of the Unix `grep` (Global Regular Expression Print) utility.

To find out more about `grep` look at the online manual (`man grep`) and try some experiments. To find out more about *regular expressions* look at the online manual for `ed`.

`grep` prints all the lines in a specified file that contain a string matching a certain *pattern*. The pattern is specified as a *regular expression*. The class of regular expressions that your program should handle is based on a small subset of those available in Unix.

Requirements

You are to design, develop and test a Haskell program that reads in a string representing a regular expression and the name of a file, and prints out a list of all the lines in the file that contain a string that matches the regular expression pattern. Your main function should be:

```
> grep :: String -> FilePath -> IO()
```

where the first argument is a string representing a regular expression and the second argument is a file name. The function definition should be similar to the spelling checker in the week 7 practical exercise:

```
> grep regexp f
>   = do{contents <- readFile f;
>        putStr (..process.. contents)}
```

Regular Expressions

Regular expressions are made up of ordinary characters and *meta-characters*. The meta-characters have a special meaning in the language of regular expressions. The regular expressions that your program should handle are sequences made up from any of the following six elements:

- Single characters. For example, `grep "hello" "testfile"` will report all the lines in the file called `testfile` that somewhere contain the string `hello`.
- The meta-character `*` indicates zero or more of the preceding character. For example, the regular expression `ab*` is matched by the strings: `a`, `ab`, `abb`, `abbb` ...

- On occasions, we want to include the meta-characters as normal characters in a regular expression. The meta-character `!` indicates that whatever character follows it should be treated as a normal character. For example, the regular expression `a!*!!` matches the string `a*!`. (The usual escape character in `grep` is `\` but that would conflict with Haskell's own special-character escape in strings.)

- The meta-character `~` indicates that the pattern matches any character other than the one following the `~`, or an end of line. Also, in one sense `~` acts as an escape. For example, `~*` matches any single character other than `*`. Unless preceded by an escape, `~` cannot legally appear as the last character in a regular expression.

- The meta-character `^` acts as a start-of-line anchor when it appears as the first character in a regular expression. For example, `grep "^hello" "testfile"` will only report those lines that contain `"hello"` at the beginning of the line. At any other point it is treated as an ordinary character.

- The meta-character `$` similarly acts as an end-of-line anchor when it appears as the last character in a regular expression. At any other point it is treated as an ordinary character.

Advice

This is the first program that you are expected to design and develop from the ground up. Carefully analyse the problem and make sure you understand what is required. You should generate a complete and thorough design *before* you begin writing Haskell code. You should discuss any difficulties with your tutor, the lecturer, or a Student Resource Centre tutor during the design phase. Remember to follow the program development steps discussed in lectures.

As a starting point, I suggest that you identify *regular expressions* as one of the abstract types and design an appropriate algebraic data definition in Haskell to represent regular expressions.

At the level of detailed design you should begin with a simplification of the problem. For example, in the first instance you should not include the meta-characters in the regular expressions. Also, consider searching for patterns in a single string rather than a multi-line file. In other words, define a function like `grepLine string1 string2` to return `True` if `string1` is a *substring* of `string2`.

You can extend this basic solution to include the meta-characters, one at a time, testing each extension, until the full range of regular expressions is handled. Finally, extend your solution to multi-line files. The spelling checker from the week 7 practical exercise may be a useful guide for this step.

Submission of Assignments

The standard arrangement for submission of this assignment is described below. Alternative arrangements may be discussed with your tutor.

By the due date, you should put listings of your script(s) and listings of some test demonstrations, together with a floppy disk containing your script in an envelope and place it in your tutor's mail box. Make sure your floppy disk is clear of any material that is not relevant to this assignment.

Sample Demonstration

Here are some demonstrations of the expected behaviour of your `grep` function. The sample file is `sect4.lhs`, the script containing example functions from section 4 of the lecture notes.

```
? grep "of" "sect4.lhs"
This is the script for section 4 of the lecture notes.
Some definitions of exclusive-or.
> offset = ord 'A' - ord 'a'
> capitalize ch = chr (ord ch + offset)

? grep ")$" "sect4.lhs"
> fact n = if n==0 then 1 else n * fact (n-1)
> factorial n = if n==0 then 1 else n * factorial (n-1)
> xOr x y = (x || y) && not (x && y)
> xOr' = (/=)
> capitalize ch = chr (ord ch + offset)

? grep "!~" "sect4.lhs"
> (~=) :: Float -> Float -> Bool
> x ~= y = abs (x - y) <= 1.0e-6

? grep "^> *z" "sect4.lhs"
> zeroInPeriod :: Int -> Bool
> zeroInPeriod 1 =
> zeroInPeriod n =
>      zeroInPeriod (n-1) || sales n == 0

? grep "~==)" "sect4.lhs"
> xOr' = (/=)
> (~=) :: Float -> Float -> Bool
> (=#=) :: Float -> Float -> Bool
```