Programming in Haskell – Project Assignment

UNIZG FER, 2014/2015

Handed out: December 5, 2014. Due: January 14/21, 2015

1 Introduction

The task is to implement a Haskell-based shell, `hash`. It should be capable of performing tasks both inline and from text files called `.hash` scripts. The syntax for both will be the same.

You will have until the final week of classes to implement your version of `hash`. You will present your project in a show-and-tell presentation to the teaching staff. Points will be given for functionality – 80% for the basic functionality and 20% for 10 points worth of advanced functionality. Additionally, extra points (over 100%) might be given for additional functionality or especially elegant code. To pass this aspect of the course, you need to have at least 50% of the maximum points for the project.

Furthermore, the project has to be developed with some caveats. It will be hosted in a **public** Github repository and must be formatted as a cabal project. Use the `README.md` file provided in the PUH repository. You may add content to it. Use the checklist provided to keep track of milestones.

You may use any module you wish to aid you in the development process, as long as it does not solve the task in its entirety or solve a large part of it. If you are uncertain if it does so, ask.

2 Basic Functionality

The basic functionality listed below **must** be present in any version of `hash`. It does not have to be implemented perfectly, but it has to compile and at least mostly do what it is supposed to do. Many functionalities listed below are simplifications of various Unix tools. If you are uncertain how they work, please RTFM (which is to say consult the man pages).

B1. *(2 pts)* Abstracted command calling scheme

You are to make adding new commands simple. For example, you can create a single directory where all the executable Haskell command files are kept and dynamically call them when a user tries to execute a command in `hash`. Alternatively, create a `(Name, Command)` dictionary which can be easily modified. Extra points may be given for especially creative or elegant solutions.

B2. *(6 pts)* Conditional branching – `if-then-else(-fi)`

The conditional branching has to support **at least** a single `if` and a single `if-else` pair. You do not have to implement both numeric (`>`, `<`, ...) and string (`-eq`, `-gt`,

. . .) comparison. You may pick only one. Furthermore, you do not have to support nested `if-then-else` statements.

Example (your syntax may be different):

```
if $a > 9000 then
    echo "It's over 9000!";
else
    echo "It's too low.";
fi
```

B3. *(3 pts)* Basic file manipulation commands `mv` (move), `cp` (copy), `rm` (remove), `create`. They have to support both absolute and relative paths.

`mv src target` ⇒ Moves a source file to the target file
`mv src ... directory` ⇒ Moves one or more files to a target directory
`cp src target` ⇒ Copies a source file to the target file
`cp src ... directory` ⇒ Copies one or more files to a target directory
`rm file ...` ⇒ Removes one or more files (not directories)
`create file ...` ⇒ Creates one or more empty files

B4. *(3 pts)* Basic directory manipulation commands `mv` (move), `cpdir` (copy), `mkdir` (make directory), `rmdir` (remove directory). They have to support both absolute and relative paths.

`mv dir ... dir` ⇒ Moves one or more directories to a target directory
`mv dir new_name` ⇒ Renames the directory to `new_name`
`cpdir dir ... directory` ⇒ Copies one or more directories to a target directory
`mkdir dir ...` ⇒ Creates a new empty directory
`rmdir dir ...` ⇒ Removes an empty directory

B5. *(3 pts)* Basic file system navigation commands `ls` (list directory contents), `pwd` (print working/current directory), `cd` (change directory). These commands have to support both absolute and relative paths.

`ls [dir]` ⇒ Lists the contents of the given directory, or the current one if no argument is given.
`pwd` ⇒ Prints the current directory (working directory)
`cd [dir]` ⇒ Changes the working directory to a given one, or the user's home directory if no argument is given.

B6. *(5 pts)* Stream redirection.

`hash` commands have to read from the `stdin` and write to the `stdout` by default, but the user has to be able to redirect input and output from and to files. Support the `>` (redirect output to file), `<` (redirect input from file) and `>>` (appends output to file) operators.

```
echo "Overwriting" > file.txt
echo "Appending" >> file.txt
foo < input.txt
```

B7. *(2 pts)* Basic comment syntax

For example, lines beginning with `#` or whitespace followed by `#` are comments and are ignored by `hash`. You do not have to support comments appearing after meaningful content.

```
# This is a comment
This is a command > file.txt # Doesn't have to be a comment
    # Another comment
```

B8. *(3 pts)* Variables, marked in a way to make them different from commands and string literals – for example prefix them with `$`.

```
a=10
echo $a
# Prints 10
```

B9. *(2 pts)* A basic `cat` command that dumps the contents of a variable number of files (one or more) to the screen (or a file, if redirected).

B10. *(3 pts)* A basic `echo` command that repeats user input to the screen, replacing any instances of variables with their values.

```
a=67
echo "$a bottles of beer on the wall"
# Prints "67 bottles of beer on the wall"
```

3 Advanced Functionality

The functionality listed below is advanced and not mandatory. However, you need to implement at least some of it. Before each bullet, its worth in points is listed. You need to implement as least 10 points worth of advanced functionality and those 10 points will translate to 20% of the score. Any additional advanced functionality will not be scored for the mandatory 50% to pass the course, but may be awarded extra points. You can suggest your own utilities and shell functions, which we may accept and assign them points, or refuse.

This section of the document will be updated if and when new functionality suggested by you (or invented by us) is added.

A1. *(4 pts)* A basic grep command that matches lines containing a given string literal. It should support the `-v`, `-i`, `-o`, `-n` and `-c` flags.

A2. *(5 pts)* Command piping. Using some special operator (for example `|`) `hash` should be capable of chaining commands so that the output of one command is made input into another.

A3. *(3 pts)* A basic `chmod` command for editing file and directory permissions. It should take a rwx permission mask and a list of files and adjust their permissions. (Hint: `System.Posix.Files`)

A4. *(5 pts)* Glob expansion of names in the shell.

A5. *(8 pts)* Some form of loops – `while`, `for` or `foreach`. Multiple loop implementations are not scored separately.

A6. *(8 pts)* Forking processes. Some command or special operator should be capable of starting another command in a new process. You also have to implement stopping or killing the new process in some way.

A7. *(3 pts)* Implement a `hexdump` command that, given a file, prints out its byte contents as hexadecimal numbers.

A8. *(2 pts)* Support `.hashrc` files. These files contain `hash` syntax that should be executed before running a hash script or starting an interactive session. They should be no matter in which directory `hash` is started – for example, always look in the user's home directory.

A9. *(3 pts)* Implement a `ping` command. It should perform the same task as a vanilla (flagless) version of `ping`.

4 Project Structure

Your code should be structured and split into modules according to functionality. Modularisation of code is especially important in larger tasks such as this one. Below is an example of a document structure you could use, consisting of directories and .hs files. You may use the structure and functions provided, or do it differently if you think you have a better idea.

```
|- Hash.hs
|- Main.hs
|- Language
    |- Expressions.hs
    |- Exec.hs
    |- Commands.hs
|- Parsing
    |- HashParser.hs
```

The modules mentioned above contain these functions and data structures:

4.1 module Hash.Language.Expressions

```
-- Contains the data structures describing the structure of the language itself
-- You are free to use a different structure, as long as it describes a similar
-- enough language sufficiently well.

-- A command which performs something - can be a command that takes arguments
-- or an assignment.
data Cmd = Cmd { name   :: Expr  -- The command name (can be a variable)
               , args   :: [Expr] -- The command arguments
               , inDir  :: Maybe Expr -- A redirected input fp
               , outDir :: Maybe Expr -- A redirected output fp
               , append :: Bool -- If redirected, is it appending?
               }
          | Assign { var    :: Expr -- Assignment target
                   , val    :: Expr -- A value to assign to a variable
```

```
                  }
          deriving Show

-- A bottom-level expression
data Expr = Var  String -- A named variable
          | Str  String -- A mere string, the peasant of expressions
            deriving (Eq, Show)

-- A comparison operation
data Comp = CEQ Expr Expr -- ==
          | CNE Expr Expr -- /=
          | CGE Expr Expr -- >=
          | CGT Expr Expr -- >
          | CLE Expr Expr -- <=
          | CLT Expr Expr -- <
    | CLI Expr -- A wrapped expression literal - True if nonempty
            deriving (Eq, Show)

-- Something that evaluates to a truth value
data Pred = Pred    Comp -- A wrapped comparison
          | Not     Pred -- Negation
          | And     Pred Pred -- A binary logical and
        | Or      Pred Pred -- A binary logical or
          | Parens  Pred  -- An expression in parentheses
            deriving (Eq, Show)

-- A conditional branching expression - if-then or if-then-else
-- If-then with a condition and a list of actions
data Conditional = If { cond  :: Pred -- Predicate to satisfy
                      , cthen :: [Cmd] -- Actions if satisfied
                      }
-- An if-then-else with a condition and two possible paths
                 | IfElse { cond  :: Pred  -- Predicate to satisfy
                          , cthen :: [Cmd] -- Actions if satisfied
                          , celse :: [Cmd] -- Actions otherwise
                          }
                  deriving Show

-- A top-level expression, wrapping either a conditional expression or a
-- command
data TLExpr = TLCmd Cmd
            | TLCnd Conditional
              deriving Show
```

4.2 module Hash.Language.Exec

```
-- A model of a command which is waiting for arguments and a state to run
type Command      = [String] -> ScriptState -> IO ScriptState
```

```
-- A table of variables, in fact a map of (Name, Value) pairs.
type VarTable     = M.Map String String

-- A command table - abstracted command execution, (contains command name,
-- command) pairs. Simplest, but hardly the best way to implement this.
type CommandTable = M.Map String Command

-- A script state containing the last output, current working directory and
-- the current table of variables.
data ScriptState  = ScriptState { output   :: String
                                , wd       :: FilePath
                                , vartable :: VarTable
                                } deriving Show

-- Runs a set of commands for a given command table. If this is the first
-- command in the chain, it is given a FilePath and constructs a new, initially
-- blank, ScriptState. Otherwise, it is given the state as left by the previous
-- command's execution.
runHashProgram :: CommandTable -> Either FilePath ScriptState -> [TLExpr]
                    -> IO ScriptState

-- Calculates the result of a top-level command execution
runTopLevel :: CommandTable -> ScriptState -> TLExpr -> IO ScriptState

-- The rest of the module should consist of similar functions, calling each
-- other so that each expression is parsed by a lower-level function and the
-- result can be used in a higher-level function. The Command table and state
-- are passed around as necessary to evaluate commands, assignments and
-- variable substitution. A better way to pass around variables would be to
-- use the State monad or even the StateT monad transformer to wrap IO into it.
```

4.3 module Hash.Language.Commands

```
-- A map of (command name, command pairs), used to abstract command
-- execution and make adding new commands relatively easy
commands :: Data.Map String Command
```

4.4 module Parsing.HashParser

```
-- Contains the parsers that take a string and produce an executable list of
-- TLExpr constructs. We recommend Parsec for parsing.
```

4.5 module Hash

```
-- The top-level module. Connects parsing to execution and adds interaction
-- with the user / reading from file.

-- Runs a .hash script
runScript :: FilePath -> IO ()
```

```
-- Communicates with the user and performs hash commands line by line
runInteractive :: IO ()
```

5 Parsing

There are a variety of parsing libraries available for parsing. However, we recommend Parsec, a parser combinator library. It combines simple parsers to create more complex ones. It is a useful, industrial-strength utility, but it takes some experience to use properly. We will provide a brief tutorial. We will also try to have an extra class about Parsec during one of the HA review meetups.

6 Submission Format

The end result of your project should be an executable called `hash` (with possible system-dependent extensions). Running it without arguments should start an interactive `hash` environment, while giving it a file path as a single argument should cause it to run the file. Alongside the executable, you should present a `Hash` library, which should expose only the functions needed to run `hash`, for example:

```
runInteractive :: IO ()
runScript :: FilePath -> IO ()
```

7 Submission Date

There will be two separate submission dates:

1. Thursday, January 14, 2015
2. Thursday, January 21, 2015

Both will likely be in the evening. If you are unable to attend, contact us and we will try to work something out.