Prof. Dr. Peter Thiemann

Janek Spaderna
janek.spaderna@pluto.uni-freiburg.de

Winter Term 2022/23

Issued 2022-11-25
Discussion on 2022-12-02

Functional Programming
https://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2022/

Exercise Sheet 6

Now that you have learned about I/O in Haskell you can create executables which actually do interesting things. The last page of the exercise sheet contains a short description how to create and run executables with stack and cabal.

Exercise 1 (Numbers game)

In the Numbers game, the computer tries to guess a user-selected number between 1 and 100. Below is an example. Texts in *cursive* after the `>` prompt are user input.

```
Choose a number between 1 and 100!
Is it 50?
> greater
Is it 75?
> smaller
Is it 62?
> smaller
Is it 56?
> yes
I won in 4 attempts!
```

Implement this game using `IO` actions and `do` notation. A set of predefined `IO` actions in the `Prelude` is documented in section *Simple I/O operations*.

Exercise 2 (Stack calculator interface)

In the first exercise sheet we implemented a simple stack calculator. This calculator was missing a crucial component: a command line interface!

Using `IO` extend your calculator with a command line interface. Each line read from the user (e. g. "push 3" or "add") should correspond to a command executed in the calculator. Display the stack after each step. "exit" should terminate the program

Exercise 3 (Simple grep)

The command line tool `grep` is a near irreplaceable utility on *nix systems.[1] The goal of this exercise is to write a very basic Haskell version of the "fixed-strings" mode: patterns are interpreted not as regular expressions but are searched for literally. On many systems this mode is available either through invoking `fgrep` or by providing the `-F` flag to `grep`.

1. Write a function `contains :: String -> String -> Bool` to check whether a string contains a given pattern. Order the parameters such that ``s1 `contains` s2`` has the naturally expected behaviour. Test your implementation using QuickCheck.

2. Write the function `grepString :: Bool -> String -> String -> String` which filters the lines in the second string based on the first string parameter, the search string. The `Bool` parameter indicates if matched or unmatched lines should be kept.

   Note Take a look at the `lines` and `unlines` functions defined in the `Prelude`. It is possible to write `grepString` without mentioning the third parameter by name.

[1] https://www.man7.org/linux/man-pages/man1/grep.1.html

3. Write the `main` action. It should parse the arguments, read and filter the input, and finally write the result back to the terminal.

   You can access the command line arguments through `getArgs` from `System.Environment`. In short, your program should follow this synopsis:

   ```
   usage: ... [-v] pattern [file]
   ```

   If the `-v` flag is present, inverted mode should be activated, and if no file is given the input should be read from standard input. The `System.Exit` module provides functions to abort with a non-zero exit code.

   Note If you want to test your executable from GHCi you should use the `:main` command instead of running the `main` action: The former lets you specify command line arguments, whereas you have to jump through some hoops to provide these using the latter form.

Creating Executables

You can have multiple executables per project. The configuration looks a bit different between cabal and stack. As usual, only modify the `*.cabal` file if you're using cabal, and only modify `package.yaml` if you're using stack.

cabal

```
executable example
  main-is: Main.hs
  other-modules:
    M1
    M2.A
  build-depends:
    base,
    QuickCheck,
  hs-source-dirs: exe
  default-language: Haskell2010
```

This describes an executable named `example` whose source code lives in the `exe` directory (relative to the `.cabal` file) and requiring the base and QuickCheck libraries.

The `main` action should be defined in `exe/Main.hs`. Additional code lives in the files `exe/M1.hs` and `exe/M2/A.hs`.

You can run your executable using `cabal run example -- args...`

If you have multiple executables you may want to start a GHCi session in the context of a specific one: `cabal repl example`

Refer to the cabal package description documentation for more information.

stack

```
executables:
  example:
    main: Main.hs
    source-dirs: exe
    dependencies:
      - base
      - QuickCheck
```

This describes an executable named `example` whose source code lives in the `exe` directory (relative to the `package.yaml` file) and requiring the base and QuickCheck libraries. Any dependencies declared on the top-level are inherited.

The `main` action should be defined in `exe/Main.hs`. Additional modules can live inside the source directory and its subdirectories and don't need to be listed.

You can run your executable using `stack run example -- args...`

If you have multiple executables you may want to start a GHCi session in the context of a specific one (note the leading colon!): `stack ghci :example`

Multiple executables should be defined inside one `executables:` block. Refer to the hpack documentation (specifies how the YAML file gets turned into a `.cabal` file) and the cabal package description documentation for the meaning of the fields.