

# Functional Programming

## Assignment 0: Introduction

Ruud Koot, Alejandro Serrano Mena, Frank Staals

In this exercise we will guide you through the basics of writing, compiling and running a Haskell program, as well as introduce the DOMjudge system you that will use to submit your weekly programming assignments. Some basic familiarity with using a computer and the command line is assumed.

### 1 Installation and set-up

In order to follow this course you need to install “Haskell” first. In particular, you will need GHC (the compiler) and stack (a build tool you will use for the final project). You can download GHC and stack from <https://www.haskell.org/downloads/>. How to get the Platform to your computer depends on your operating system; so select the right operating system and follow the appropriate instructions.

After the installation is complete, open a terminal or command line,

- *Windows*: you need to open the **Command Prompt**. You can find it in the Start Menu (use the Search functionality).
- *Mac OS X*: in this operating system you need to open the **Terminal**, which you can find in the *Applications* folder.
- *Linux*: once again, this changes depending on the distribution. Usually, anything which includes **Terminal** or **Term** in the name is a safe choice.

Now write the following line of text and press *Enter*:

```
ghc --version
```

You should receive a message like the following (the version number may change):

```
The Glorious Glasgow Haskell Compilation System, version 8.6.5
```

If you get an error message about the program or command not being found, restart your computer and try again. If the problem persists, ask one of the lab assistants.

**Code editors.** Since you are going to spend a great part of this course (and of the rest of your life?) dealing with code, it is good to use a text editor geared towards this purpose. Choosing a code editor is like choosing a car — people discuss endlessly about which is the best and makes you more productive. But the minimum bar is to provide *syntax highlighting*, that is, to colour your code in a way that helps you to identify the structure and makes it easier to notice simple errors like a parenthesis which is not closed.

- Graphical: arguably simpler to use if you already know Visual Studio or a similar IDE.
  - Visual Studio Code <https://code.visualstudio.com/>. After downloading the program, click on the *Extensions* tab (the fifth icon in the left side), search for *Haskell Syntax Highlighting* and click *Install*.
  - Atom <https://atom.io/>, with the `language-haskell` package.
  - Notepad++ <https://notepad-plus-plus.org/download>.
- Keyboard-oriented: harder to grasp at first, but an amazing tool once you learn them
  - Emacs: the official website at <https://www.gnu.org/software/emacs/> provides installers for Windows, for Mac OS X the version at <https://emacsformacosx.com/> is recommended. You can find Emacs on any Linux distribution as the `emacs` package. After installing Emacs, you still need to download and configure a Haskell *mode*. There are as many ways to do so as stars in the sky, contact Victor or Frank to help you on this adventure.
  - Vim: the official website at <https://www.vim.org/> provides links for Windows and Mac OS X. Once again, any Linux distribution comes with a `vim` package, most of them even have them installed by default.

## 2 Hello, world!

Use your favourite editor to create a source file containing the following program:

```
module Main where
main = putStrLn "Hello, world!"
```

You can give the file any name ending in `.hs`, such as `HelloWorld.hs`, but for the rest of the exercise we will assume that you named the file `Main.hs`. Note that, like in Java and C#, it is always a good idea to make sure the name of your file and the name of your module are identical, otherwise the compiler will complain when you try to import this module from another one.

In Haskell the *Main* module—the one containing your *main* function, the function that will be invoked when you execute your compiled program—is a bit of an exceptional case in this respect, in that the module name must *always* be *Main*, but so long as you do not try import it from another module—which you generally won’t—you can give it another file name. If there is no good reason to do so, however, you probably shouldn’t and just name your program `Main.hs`.

**Step 1: Compilation.** The first step is to open the terminal. Then, you need to move to the folder in which your files live, by running:

```
cd path/to/your/folder
```

Then you can compile your program by invoking the following command from the command line.

```
ghc --make Main.hs
```

This command will compile both the *Main* module, as well as any other modules it depends on.

**Step 2: Execution.** The compilation has produced an executable. In Linux and Mac OS X you run it by typing:

```
./Main
```

whereas in Windows you need to run:

```
Main.exe
```

For the rest of the assignment we'll write `Main` for the name of the executable. You should choose which variant of the name you need to use depending on your operating system. Running your program will give the output:

```
Hello, world!
```

Apart from compiling and then running a program, we can also run the program interactively from an interpreter:

```
ghci Main
```

You will now be presented with the prompt

```
*Main>
```

Type in `main` and press enter to start the program, again resulting in the output:

```
Hello, world!
```

Using the interpreter is more convenient when you are developing your program, as you can invoke any function defined in your program and pass it any arguments you desire. When you compile your program it will always be the function `main` that gets called.

### 3 Interaction with the outside world

Shouting a message to the outside world without bothering to listen to its response is somewhat boring. What we are really interested in is *interaction* with the outside world.

This can be achieved using the `interact` function from Haskell's standard library (also called the *Prelude*). The function `interact` is an IO action (IO stands for "input/output") that takes another function as its argument. This concept—passing a function as an argument to another function—may be unfamiliar if you have only programmed in *imperative programming languages*, such as Java or C#, before, but is one of the cornerstones of the *functional programming* paradigm, as is reflected in its name. The function `interact` is thus an example of a so-called *higher-order function* and we shall become intimately familiar with them during this course.

But what does the function `interact` do? First, it reads a string from the *standard input*—by default your keyboard, but below we shall see how we can redirect the standard input to read from a file instead. Next, it applies the function you passed as an argument to `interact` to the string it just read from the standard input, to transform it into another string. Finally, it prints the transformed string to the *standard output*—by default your screen, but below we shall see how we can redirect the standard output to write to a file instead.

Note that this does put some restrictions on the kinds of functions you can pass to `interact`: they should take exactly one string as their argument, and also return a string as their result. Formally, we write that the argument of `interact` should be of the type `String → String`.

Time for an example:

```
module Main where
main = interact reverse
```

Additionally, create a file called `in.txt`, containing the text:

```
eb ot
eb ot ton ro
noitseuq eht si taht
```

If you use a Windows machine, do *not* use Notepad to copy this text, as it will not correctly interact with the Haskell program. The solution is simple: just use another editor.

Compile the program (running it from the interpreter is not going to work correctly!), and run it, while redirecting the standard input to read from the file `in.txt`. On Windows machines this can be achieved using the command:

```
Main < in.txt
```

In recent versions of Windows, the command prompt has been replaced by PowerShell. In that case, you should use:

```
Get-Content in.txt | ./Main
```

On Mac and Linux machines you need to use the command:

```
./Main < in.txt
```

This will give the output:

```
that is the question
or not to be
to be
```

Almost, but not quite right. We reversed the complete file, instead of reversing the lines one at a time. Let us try again:

```
module Main where
  main = interact work
  where work text = unlines (map reverse (lines text))
```

Now instead of a function from the *Prelude*, we introduce our own *work* function. This function takes a *text* and applies three consecutive operations:

1. The first function, *lines*, will split a string into a list of strings (denoted  $[String]$ ). Its type is thus  $String \rightarrow [String]$ .
2. The second function, *map reverse*, will reverse all of the strings contained inside a list. Its type is thus  $[String] \rightarrow [String]$ . Note that this function is actually another instance of a higher-order function (in this case *map*) applied to a second function (*reverse*).
3. The third function, *unlines*, takes a list of strings and concatenates them together with a newline character in between.

**Tabs versus spaces.** You have sure noticed that the **where** keyword is indented with respect to *main*. This is required by the rules of the language. In this course, every time you need to indent further, you should use *two spaces*. In most editors, if you press the Tab key, you insert those characters instead. Check it, because you should *not* use tabs for indentation.

Running the program will give us the desired output:

```
to be
or not to be
that is the question
```

If you want to save the results to a file instead of printing it on the screen, you can instead invoke the program with the command:

```
Main < in.txt > out.txt
```

As we have discussed above, one of the main features of functional programming is treating functions as elements which can be manipulated. In this case, we can rewrite our program into a nicer form by using the *function composition* operator  $\circ$ .<sup>1</sup> Note that, just like in mathematics, composed functions should be read from right-to-left.

```
main = interact work
where work = unlines  $\circ$  map reverse  $\circ$  lines
```

---

<sup>1</sup>Although it looks fancy on the text, the  $\circ$  operator is written in code as a simple dot.

Note that we have dropped the argument *text* from the definition of *work*. We are no longer stating how *work* acts on an argument, but just defining a new function as the combination of others.

We can even go one step further and replace the *work* function by its definition:

```
main = interact (unlines ◦ map reverse ◦ lines)
```

## 4 The exercise

Modify the program given above to have it—instead of printing each line of the input on a separate line—print the lines with slashes in between. Thus for the input:

```
eb ot
eb ot ton ro
noitseuq eht si taht
```

the program should give as output:

```
to be / or not to be / that is the question
```

Hint: you can use the function *intercalate* from the library *Data.List* (which you will have to **import**). See <http://hackage.haskell.org/packages/archive/base/latest/doc/html/Data-List.html#v:intercalate>.

## 5 Handing it in

In this course we use DOMjudge. Consult the course website (under *Assignments*) for the precise URL.

You can log in using your Solis ID; if this does not work, you have probably not registered on time for this course, and will now have to notify the lecturer about this. You should always submit your source file, not a compiled executable. Your solution should be graded promptly, giving you one of the following results:

**PENDING** Your solution is still being graded. You will need to have a bit of patience. If this takes more than a few minutes, something is probably wrong. Either try again later and/or bug the teaching assistants.

**CORRECT** Your solution is correct. Note that the system only checks for functional correctness.

**WRONG-ANSWER** Your program did not produce the expected output. Something is wrong with your program: you need to fix this and try again.

**NO-OUTPUT** Sort of like WRONG-ANSWER, except that your program did not seem to produce any output at all.

**COMPILE-ERROR** Your program contains a mistake that did not even allow it to be compiled correctly. Fix the mistake and try again. Note that if you click on your submission in the webform, you will be taken to another screen which contains the full output of the compiler. This may help you to diagnose the problem.

**RUN-ERROR** Your program caused an exception at run-time. Perhaps you tried to take the *head* of an empty list, forgot a pattern in a **case**-statement, or divided by zero.

**TIMELIMIT** Your program took an extraordinary long time to finish or did not finish at all. You might have introduced an infinite loop somewhere in your program.

### 5.1 Style checks

A **CORRECT** output from DOMjudge only means your program gives the right output for our tests. In order to help you getting a good style, we use a tool called HLint which tells you when something

can be improved.

In order to see these suggestions, you have to go to your latest submission, which appears in the initial DOMjudge page. Once the submissions moves from **PENDING** to another status, click on *Compiler output*. The first block there corresponds to HLint. Take those suggestions into account and submit your assignment again.

You are not required to follow *every* single suggestion from HLint. However, if you decide not to follow it, consider writing a small comment in the code explaining your decision.