Functional Programming 2013/2014
Assignment 0: Introduction

Ruud Koot

September 16, 2014

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 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`.[1]

You can compile your program by invoking the following command from the command line:[2]

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

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

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
```

[1]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`.

[2]On the university computers you *must* open a command line using Start > Standard Applica-

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.

2 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[3] 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 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* $String \to String$.

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
```

Compile the program (running it from the interpreter is not going to work

tions > Computing Sciences > Haskell > Cabal!

[3]IO stands for "input/output."

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
```

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**

$main = interact\ (unlines \circ map\ reverse \circ lines)$

Note that we will often try very hard to make out programs look very pretty on paper (including in the exercises, the lecture notes, and on exams) by replacing some of the operators used in the source code by their correct mathematical symbols. The actual text that you need to enter in your source file would be:

```
module Main where

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

This program makes use of the *function composition* operator $\circ$ to glue three functions together before they are passed to $interact$. Note that, just like in mathematics, composed functions should be read from right-to-left.

The first function, $lines$, will split a string into a list of strings (denoted $[String]$). Its type is thus $String \to [String]$.

The second function, $map\ reverse$, will reverse all of the strings contained inside a list. Its type is thus $[String] \to [String]$. Note that this function is actually another instance of a higher-order function (in this case map) applied to a second function ($reverse$).

The third function, $unlines$, takes a list of strings and concatenates them together with a newline character in between.

In this instance it may be more apparent what the program does if we could compose functions from left-to-right. We can do so as follows:

module $Main$ **where**

import $Control.Arrow$

$main = interact\ (lines \ggg map\ reverse \ggg unlines)$

Running either of the two programs 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
```

3 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`.

4 Handing it in

You can submit your solution through the webform at: `http://science-vs89.science.uu.nl/fp/`. Select Assignment0 as the problem.

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 Ruud (`r.koot@uu.nl`) 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 You solution is correct. Note that the system only checks for functional correctness. Your exercise will also be looked at by a human, who may decide to deduct points for bad coding style.

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 You 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 you 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.