

Install You a Haskell for Great Good!

Informatics 1 – Introduction to Computation

Functional Programming Tutorial 1

Banks, Burroughes, Heijltjes, Fehrenbach, Lehtinen, Melkonian
Sannella, Scott, Vlassi-Pandi, Wadler, Yeum

Week 2
due 4pm Wednesday 30 September 2020
tutorials on Friday 2 October 2020

Welcome

Welcome to your first functional programming tutorial! This document will explain how to get started writing Haskell. *Please go through the entire worksheet in advance of the tutorial; answer the questions when needed or take some personal notes.*

The main purpose of the present tutorial is to familiarize yourselves with writing and using Haskell. You will be shown how you can use your favorite text editor to write a Haskell program and run it with the interactive Haskell interpreter GHCi¹

The tutorial consists of the following parts:

- 0. Install you a Haskell** In the second part you will set up the system and get to know the basic tools for programming.
- 1. Getting Started** The third part consists of some simple exercises where you will write some arithmetic functions in Haskell. Read this beforehand. You will do the exercises during this tutorial.
- 2. Optional Material: Chess** Part four is an optional exercise where you are being asked to compose and manipulate images of chess pieces.

Attendance at tutorials is obligatory; please send email to lambrose@ed.ac.uk if you cannot join your assigned tutorial.

Good Scholarly Practice: Please remember the good scholarly practice requirements of the University regarding work for credit. You can find guidance at the School page

<http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct>.

This also has links to the relevant University pages. Please do not publish solutions to these exercises on the internet or elsewhere, to avoid others copying your solutions.

¹‘GHC’ stands for ‘Glasgow Haskell Compiler’ or ‘The Glorious Glasgow Haskell Compilation System’

0 Install you a Haskell

Open a terminal and type “`ghci --version`” to see if you have “`ghci`” already installed on your computer. If not, please follow the link below in order to install the Haskell Platform by picking the correct version (Windows, OS X for Mac, Linux) and choosing the minimal installation:

<https://www.haskell.org/downloads#platform>

Once you have installed “`ghci`” install the QuickCheck package by running in your terminal:

```
$ cabal update
$ cabal install QuickCheck
```

(If cabal’s version is ≥ 3 , then you need to instead run `cabal install --lib QuickCheck`)

You can use Haskell with any text editor as shown in [this short video](#), which will be also demonstrated during the tutorial.

We will begin by using the Haskell REPL (read-eval-print-loop), by running the “`ghci`” command in a terminal. This interactive environment is usually provided by GHCi, the interactive Haskell compiler/interpreter. At any time you can type “`:help`” at the prompt to see all the available commands.

Exercise 1

Start the Haskell REPL by just typing “`ghci`” in your terminal.

- (a) Type “`3 + 4`” at the prompt. What does it say?
- (b) Try “`3 + 4 * 5`” and “`(3 + 4) * 5`”. Does arithmetic in Haskell work as expected?
- (c) Find the length of a string by typing “`length "This is a string."`”
- (d) Reverse the previous string using the...“`reverse`” function.

Integrated Development Environments (IDEs)

There is also the option of integrating Haskell & GHC into your favorite code editor. This is an optional step that some students might wish to pursue, since the landscape of Haskell IDEs is rapidly changing. Here, we point to some popular choices:

- Haskell mode for Emacs: <http://haskell.github.io/haskell-mode/>.
- Haskell package for Atom: <https://atom.io/packages/atom-haskell>.
- Haskell extension for Visual Studio Code: <https://marketplace.visualstudio.com/items?itemName=haskell.haskell>.

1 Getting Started

Your Github repository should contain a folder which in turn contains the files `Tutorial1.hs` and `PicturesSVG.hs`. Use your favorite editor to open the `Tutorial1.hs` file.

Below the introductory comments and the phrase `import Test.QuickCheck`, which loads the QuickCheck library that we will use later, you will find the type signature and an implementation of the `double` function.

Open your terminal and “cd” to the folder where you have the exercises, next start the REPL in your terminal by typing “ghci”.

Exercise 2

- (a) Load the file `Tutorial1.hs` into the REPL by using “`:load Tutorial1`”.
- (b) Part of the definition (the line `double x = x + x`) is incorrectly indented: it should be vertically aligned with its type signature (the line above). Edit this line to correct the indentation.
- (c) Save and reload the corrected file as shown in the video.
- (d) Use the REPL to display
 - i. the value of `double 21`
 - ii. the type of `double`, by using the command “`:type functionName`”
 - iii. the type of `double 21`
- (e) What happens if you ask the REPL to evaluate `double "three"`?
- (f) Complete the definition of `square :: Int -> Int` in `Tutorial1.hs` so it computes the square of a number (you should replace the word “`undefined`”). Reload the file and test your definition.

Pythagorean Triples

Pythagoras was a Greek mystic who lived from around 570 to 490 BC. He is known to generations of schoolchildren as the discoverer of the relationship between the sides of a right-angled triangle. There is little evidence, however, that Pythagoras was a geometer at all. Early references to Pythagoras make no mention of his putative mathematical achievements, but refer instead to his pronouncements on dietary matters (he prohibited his followers from eating beans) or his less cerebral achievements such as biting a snake to death.

Whether or not Pythagoras had anything to do with the discovery of the theorem that bears his name, it was evidently known in antiquity. A stone tablet from Mesopotamia which predates Pythagoras by 1000 years, “Plimpton 322”, appears to contain part of a list of “Pythagorean triples”: positive integers corresponding to the lengths of the sides of a right-angled triangle. Back with the Greeks, Euclid (325 – 265BC) described a method for generating Pythagorean triples in his famous treatise *The Elements*.

In this part of the exercise we’ll be taking a more modern approach to the ancient problem, using Haskell to generate and verify Pythagorean triples.

First, a formal definition: a *Pythagorean triple* is a set of three integers (a, b, c) which satisfy the equation $a^2 + b^2 = c^2$. For example, $(3, 4, 5)$ is a Pythagorean triple, since $3^2 + 4^2 = 9 + 16 = 25 = 5^2$.

Exercise 3

Write a function `isTriple` that tests for Pythagorean triples. You don’t need to worry about triples with sides of negative or zero length.

- (a) Find the skeleton declaration of `isTriple :: Int -> Int -> Int -> Bool` and replace `undefined` with a suitable definition (use ‘`==`’ to compare two values).

- (b) Load the file into the REPL. Test your function on some suitable input numbers. Make sure that it returns `True` for numbers that satisfy the equation (such as 3, 4 and 5) and `False` for numbers that don't (such as 3, 4 and 6).

```
Main> isTriple 3 4 5
True
Main> isTriple 3 4 6
False
```

Next we'll create some triples automatically. One simple formula for finding Pythagorean triples is as follows: $(x^2 - y^2, 2yx, x^2 + y^2)$ is a Pythagorean triple for all positive integers x and y with $x > y$. The requirements that x and y are positive and that $x > y$ ensure that the sides of the triangle are positive; for this exercise, we will forget about these constraints.

Exercise 4

Write functions `leg1`, `leg2` and `hyp` that generate the components of Pythagorean triples using the above formulas.

- (a) Using the formulas above, add suitable definitions of

```
leg1 :: Int -> Int -> Int
leg2 :: Int -> Int -> Int
hyp  :: Int -> Int -> Int
```

to your `Tutorial1.hs` and reload the file.

- (b) Test your functions on suitable input numbers. Verify that the generated triples are valid.

```
Main> leg1 5 4
9
Main> leg2 5 4
40
Main> hyp 5 4
41
Main> isTriple 9 40 41
True
```

QuickCheck

Now we will use QuickCheck to test whether our combination of `leg1`, `leg2`, and `hyp` does indeed create a Pythagorean triple. QuickCheck can try your function out on large amounts of random data, which it creates itself. It's always a good idea to thoroughly test your code, and a better idea to have an automatic way to do that! But before we start using QuickCheck, we will try to get a flavour of what it does by testing your functions manually.

Exercise 5

The function `prop_triple`—by convention, the name starts with `prop`(erty) to indicate that it is for use with QuickCheck—uses the functions `leg1`, `leg2`, `hyp` to generate a Pythagorean triple, and uses the function `isTriple` to check whether it is indeed a Pythagorean triple.

- (a) How does this function work? What kind of input does it expect, and what kind of output does it generate?
- (b) Test this function on at least 3 sets of suitable inputs. Think: what results do you expect for various inputs?
- (c) Type the following at the REPL-prompt (mind the capital 'C'):

```
Main> quickCheck prop_triple
```

The previous command makes QuickCheck perform a hundred random tests with your test function. If it says:

```
OK, passed 100 tests.
```

then all is well. If, on the other hand, QuickCheck responds with an answer like this:

```
Falsifiable, after 0 tests:  
5  
6
```

then your function failed when QuickCheck tried to evaluate it with the values 5 and 6 as arguments—when testing manually, that would be:

```
Main> prop_triple 5 6  
False
```

If this happens, at least one of your previous functions `isTriple`, `leg1`, `leg2` and `hyp` contains a mistake, which you should find and correct.

2 Optional Material: Chess

Please note that optional exercises **do** contribute to the final mark. If you don't do the optional work and get the rest mostly right you will get a mark of 3/4. To get a mark of 4/4, you must get almost all of the tutorial right, including the optional questions.

In this final part of the tutorial we will get more familiar with Haskell, by drawing pictures of chess pieces on a board.

First, open the file `showPic.html` in your web-browser. Next, assuming you have loaded `Tutorial1.hs` in the REPL, type this at the prompt:

```
Main> render knight
```

Now refresh the webpage, and a picture of a white knight chess piece should appear:











Note: When you draw another image, you will need to refresh the webpage to view it.

The tutorial file `Tutorial1.hs` is able to draw pictures using the *module* `PicturesSVG`, contained in the file `PicturesSVG.hs`, by means of the line:

```
import PicturesSVG
```

Note: If you get an error that GHC can't find module `PicturesSVG`, make sure that you have invoked `ghci` from the directory `tutorial1`, which should contain the file `PicturesSVG.hs`.

All in all the `PicturesSVG` module includes all chess pieces and white and grey squares to create a chessboard, and some functions to manipulate the images. The following tables show the basic pictures:

Chess pieces			Board squares		
bishop	A bishop		blackSquare	A black (grey) square*	
king	A king		whiteSquare	A white square	
knight	A knight		* The black square is grey so that you can see the black pieces on it.		
pawn	A pawn				
queen	A queen				
rook	A rook				

All the basic pictures above have the type `Picture`. Below are the functions for arranging pictures:

<code>flipV</code>	reflection in the vertical axis
<code>flipH</code>	reflection in the horizontal axis
<code>invert</code>	change black to white and vice versa
<code>over</code>	place one picture onto another
<code>beside</code>	place one picture next to another
<code>above</code>	place one picture above another
<code>repeatH</code>	place several copies of a picture side by side
<code>repeatV</code>	stack several copies of a picture vertically

Exercise 6

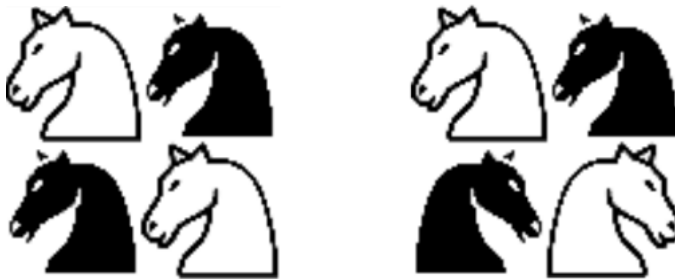
Ask the REPL to show the types of these functions and write them down.

Try applying the functions in various combinations to learn how they behave (for instance: what happens if you put pictures of different height side by side). Just as with the simple picture `knight`, you can see the modified pictures by using the `render` function. You'll probably need some parentheses, for example:

```
Main> render (beside knight (flipV knight))
```

Exercise 7

Use the `knight` picture and the above transformation functions to create the following two pictures:



Feel free to use convenient intermediate pictures.

The fourth function, `over`, can place a piece on a square, like this:

```
Main> render (over rook blackSquare)
```



You can use `over` to put any picture on top of another, but the result looks best if you simply put pieces on squares.

Functions

In the previous section we have used the built-in functions to arrange ever larger pictures. Now we will use them to construct more complicated functions. First, take a look at the function `twoBeside`:

```
twoBeside :: Picture -> Picture
twoBeside x = beside x (invert x)
```

It takes a picture and places it beside an inverted copy of itself:

```
Main> render (twoBeside bishop)
```

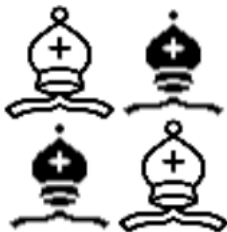


```
Main> render (twoBeside (over king blackSquare))
```



Exercise 8

- (a) Write a function `twoAbove` that places a picture above an inverted copy of itself:
- (b) Write a function `fourPictures` that puts four pictures together as shown below. You may use the functions `twoBeside` and `twoAbove`.



The full chessboard

Next, we will build a picture of a fully populated chessboard. The functions `repeatH` and `repeatV` create a row or column of identical pictures, in the following way (try this out):

```
Main> render (repeatH 4 queen)
```



Notes:

- When a problem says “...using the function (or picture) `foo`,” you *must* use the function `foo`. A solution that does not use that function will not be accepted, but of course you can use other functions as well.
- Unless an exercise says you can’t, you are free to define intermediate functions, or pictures in this case, if that makes it easier to define the solution to an exercise.

Exercise 9

- (a) Using the `repeatH` function, create a picture `emptyRow` representing one of the empty rows of a chessboard (this one starts with a white square).



- (b) Using the picture `emptyRow` from the last question, create a picture `otherEmptyRow`, representing the *other* empty rows of a chessboard (starting with a grey square).



- (c) Using the previous two pictures, make a picture `middleBoard` representing the four empty rows in the middle of a chessboard:



- (d) Create a picture `whiteRow` representing the bottom row of (white) pieces on a chessboard, each on their proper squares. Also create a picture `blackRow` for the top row of (black) pieces. You can use intermediate pictures, but try to keep your knights pointing left. The pieces should look like this:



- (e) Using the pictures you defined in your answers to the questions above, create a fully-populated board (`populatedBoard`). It will be helpful to make pictures `blackPawns` and `whitePawns` for the two rows of pawns. The result should look like this:

