

Getting Started

Informatics 1 – Functional Programming: Lab Week Tutorial

Burroughes, Heijltjes, Scott, Wadler, Banks, Sannella, Lehtinen

Due: Friday 30th Sep. (5pm)

Welcome

Welcome to your first functional programming exercise! This document will explain how to get started writing Haskell. You will be shown how to use the text editor Emacs, which you have been introduced to last week, with the interactive Haskell interpreter GHCi¹, and the “**submit**” command used to hand in your electronic coursework.

The exercise consists of four parts:

- 1. The system** In the first part you will set up the system and get to know the basic tools for programming.
- 2. Getting started** The second part is a simple exercise where you will write some arithmetic functions in Haskell.
- 3. Submitting your work** In the third part you will be shown how to submit your solutions to the Haskell exercises.
- 4. Chess** Part four is an exercise where you will compose and manipulate images of chess pieces.

It is important to complete all four parts and start getting used to the computer labs and DICE machines, since the exams for the course, which are programming tests, will be held here. In particular, part three of this exercise teaches how electronic work, including the exam to this course, must be submitted. Therefore:

Note: Completing part **3**, submitting your work, is a compulsory requirement.

¹‘GHC’ stands for ‘Glasgow Haskell Compiler’

1 The system

The first part of the exercise will explain how to set up and use the Emacs text editor together with the GHCi Haskell interpreter. But before we put them together, we will look at GHCi on its own.

GHCi

GHCi is an implementation of the Haskell programming language. GHCi is interactive: it evaluates each Haskell expression that you type and prints the result. You can start GHCi by typing “ghci” at a shell prompt.

Exercises

1. Open a terminal window (right-click, then “open in terminal”) and type “ghci”. After you press “enter”, the screen should display:

```
GHCi, version 7.8.4: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
```

This is the interactive environment of GHCi. We will let it do some simple arithmetic first.

- (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?

The interactive environment can handle any Haskell expression, not just arithmetic:

```
Prelude> length "This is a string"
16
Prelude> reverse "This is not a palindrome"
"emordnilap a ton si sihT"
```

In addition to Haskell expressions, GHCi understands a number of *commands*, for example:

<code>:load filename</code>	Load a file containing Haskell definitions
<code>:reload</code>	Reload the most recently-loaded file
<code>:type expression</code>	Display the type of <i>expression</i>
<code>:?</code>	Display a list of commands

Exercises

2. Find the command to *quit* GHCi and use it.

Emacs

Emacs is the recommended program for editing Haskell code, because it can work very well together with GHCi to edit and run Haskell programs. In this section we will set you up for doing so. If you have missed last week’s introduction to Emacs, you might want to look at that first. It can be found on the Systems Introduction website: www.inf.ed.ac.uk/teaching/courses/inf1/system.

To start up the text editor Emacs, simply type “emacs” at the shell prompt or select “Emacs” from the “Applications ” menu.

Emacs uses two types of commands: “Control” and “ESCape”. Control commands are used by holding down the control-key (`ctrl`) and then pressing the command’s letter key; for escape commands, press and release the escape key and then press the letter key. Many commands are sequences of simple commands, which are used simply by giving one command after the other.

Some of the commands you will use most are:

ctrl-x ctrl-s	saves the current file
ctrl-x ctrl-w	saves the current file under a new name
ctrl-x ctrl-c	exits Emacs (it asks to save the file)
ctrl-y	paste (“yank”) a selected piece of text
ctrl-c ctrl-l	loads a Haskell file into GHCi
ctrl-c ctrl-c	stops a runaway Haskell computation
ctrl-x o	switches from the file to the GHCi buffer

Emacs and GHCi

Setting up Emacs

The Emacs installation on DICE is set up to use *Haskell mode* for editing `.hs` files. In Haskell mode, Emacs provides several facilities to help you program. Some of the features described in this section are switched off by default, but you can enable them all by modifying the Emacs initialization file. To do this, you need to add the following lines to a file called `~/.emacs`

```
(add-hook 'haskell-mode-hook 'turn-on-haskell-doc-mode)
(add-hook 'haskell-mode-hook 'turn-on-haskell-indentation)
```

Do this now by either opening `~/.emacs` with your favourite text editor and copy pasting the above lines, or by running this at the command line:

```
cat /home/infteach/AllPreviousFiles/.emacs >> ~/.emacs
```

Among the many useful features available in Haskell mode are:

Syntax highlighting Emacs understands the basic structure of Haskell code, and uses different colours to indicate which words refer to type names, variables, numbers, etc.

Help with indentation Haskell is sensitive to the way code is indented: a single misplaced space or tab character can change a correct program into an incorrect one, or even change the meaning of your program without warning from GHCi. Emacs can help here: repeatedly pressing the **TAB** character cycles through the possible indentation levels for the line of code under the cursor, making it easy to avoid whitespace-related errors.

Context-sensitive help You’ll often need to use functions from the Haskell standard library in your programs. When the cursor is at the name of a standard library function Emacs displays the function’s type in the mini-buffer.

Running GHCi Typing **ctrl-c ctrl-l** while editing Haskell code causes Emacs to start GHCi running *within Emacs*, then to load the current file into GHCi. GHCi’s prompt appears in an Emacs buffer, where you can input Haskell expressions for evaluation using Emacs’ editing commands.

If GHCi detects an error when you load your program Emacs will move the cursor to the part of the code where the error occurs.

2 Getting started

Download the file `Labweek.zip` from the course website:

<http://www.inf.ed.ac.uk/teaching/courses/inf1/fp/#tutorials>

and unpack it. It should contain the files `labweekexercise.hs`, `labweekchess.hs` and `PicturesSVG.hs`.

Open the file `labweekexercise.hs` in Emacs. Below the introductory comments and the phrase `import Test.QuickCheck`, which loads the QuickCheck library that we will use later, you should see the following function definition:

```
double :: Int -> Int
double x = x + x
```

Exercises

3. (a) 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.
- (b) Load the file `labweekexercise.hs` into GHCi. (by typing `ctrl-c ctrl-l`, see instructions under “Emacs and GHCi” and “Emacs”, above.) Use GHCi to display
 - i. the value of `double 21`
 - ii. the type of `double`
 - iii. the type of `double 21`
- (c) What happens if you ask GHCi to evaluate `double "three"`?
- (d) Complete the definition of `square :: Int -> Int` in `labweekexercise.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 triple of positive integers (a, b, c) which satisfy the equation $a^2 + b^2 = c^2$. For example, $(3, 4, 5)$ is a Pythagorean triple, since $3^2 + 4^2 = 9 + 16 = 25 = 5^2$.

Exercises

4. 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 GHCi. 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 that.

Exercises

5. 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 `labweekexercise.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. But before we start using it, we will try to get a flavour of what it does by testing your functions manually.

Exercises

6. The function `prop_triple`—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 GHCi-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.

When you're done

If you have completed the exercises and written out all of the functions in `labweekexercise.hs`, add your name and matriculation number to comments at the start of the file and continue to the next section, which will demonstrate how to submit your solutions for marking.

Note: If you are running out of time (the deadline for submission is Friday, 5pm), you can submit the incomplete exercise, but first make sure that GHCi can load it without errors. To do this, turn the offending code into harmless commentary by putting two dashes (`--`) in front of it.

3 Submitting your work

This section demonstrates the “`submit`” command, used to submit your work electronically. Most importantly, you will have to use it to submit your exams. In general you will not be asked to submit your tutorial exercises, so so use this opportunity to see how it works. To make sure everyone is ready for the exams, completing this part of the exercise is compulsory.

Once you have completed working on the Haskell file² `labweekexercise.hs`, you need to submit it with the `submit` command. For *this* exercise, you should use the command as follows:

```
submit inf1-fp lab labweekexercise.hs
```

The meaning of the arguments is as follows:

```
inf1-fp:  the code for the course, informatics 1 – functional programming;
lab:      the code for the current exercise;
```

Each exercise has an individual code for submission, which you will be given when you are asked to submit. After you type in the command, the following dialogue will pop up:

```
Submit the following for exercise lab, module inf1-fp of the inf1 course.
/afs/inf.ed.ac.uk/user/s16..../Desktop/labweekexercise.hs
Is this correct (y/n: n aborts)?
```

The path to the file will probably differ, but it should reflect the location of the file you want to submit—check this if you are unsure, for instance by typing “`pwd`” at a shell prompt. Note that you might also see the short version of the path: `/home/Desktop/labweekexercise.hs`. If all is correct you can answer with “`y`”, and you will see:

```
Submission of the following for exercise lab, module inf1-fp
of the inf1 course succeeded:
/afs/inf.ed.ac.uk/user/s16..../Desktop/labweekexercise.hs
```

If you get one of the codes for the year, course or for the exercise wrong, you will instead see something like the following:

```
submit 3.4.3-1 usage:
  to submit an exercise:
submit <course> <exercise> <file1> <file2>...
where <exercise> is the short name of an exercise and <file> can
be a regular file or a directory.
```

You need to specify which exercise you are submitting.

The exercise name should be one of:

```
lab
  for course inf-fp.
```

SUBMISSION DID NOT HAPPEN!

Note that `submit` will always tell you if the submission completes or fails — you will see either “SUBMISSION DID NOT HAPPEN!” or “Submission of ... succeeded”.

Exercises

7. Submit your file `labweekexercise.hs` now.

When you have successfully submitted a file, you should receive a confirmation email from the submit system.

²If you did not complete the exercises, you can submit the incomplete file—it is more important to submit than to complete the exercises.

Getting more information

If you just type the command “submit” you will be given a list of the valid options (in the output below, long lines are truncated):

```
submit 3.4.3-1 usage:
  to submit an exercise:
submit <course> <exercise> <file1> <file2>...
where <exercise> is the short name of an exercise and <file> can
be a regular file or a directory.
```

You need to specify which course you are submitting the exercise for.

Choose from one of these:

```
test1 tts tspl tdd tcm st sp slip seoc selp sdp sapm rtn rss rlsc rl rc qsx
proj ppls pon pmr pm pi pa os nr nlu nip nc mt mpp2 mlpr mip1 mi mdi masws
lsi lp ivr it irr irp iqc inf2d inf2c-se inf2c-cs inf2b inf2a inf1-op inf1-fp
inf1-da inf1-cl inf1-cg ijp iar iaml hci fnlp fmt2 fmt1 ext exc es ds dmr
dmmr dme diss dip die dbs dapa ct cs lp cs cp copt comn coc cnv cn cmc cg
cdi1 cd ccs ccn cav car ca bio2 bio1 av asr ar anlp ale1 ailp agta ads adbs
abs
```

SUBMISSION DID NOT HAPPEN!

Chess

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, open the file `labweekchess.hs` in emacs. Load it into GHCi and 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 `labweekchess.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 GHCi can't find a module, see if the problem is solved by putting your files in the same directory (folder).

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				
pawn	A pawn				
queen	A queen				
rook	A rook				

* The black square is grey so that you can see the black pieces on it.

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

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

Exercises

8. Ask GHCi to show the types of these functions.

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))
```

Exercises

9. 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.

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.

Exercises

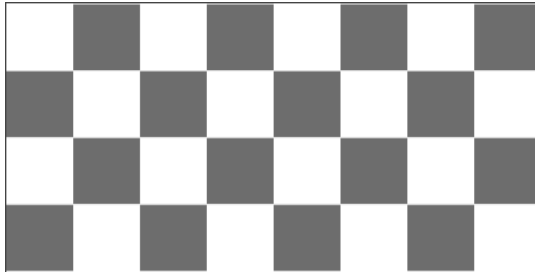
10. (a) Using the `repeatH` function, create a picture `emptyRow` representing one of the empty rows of a chessboard (this one starts with a white square).



- (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).



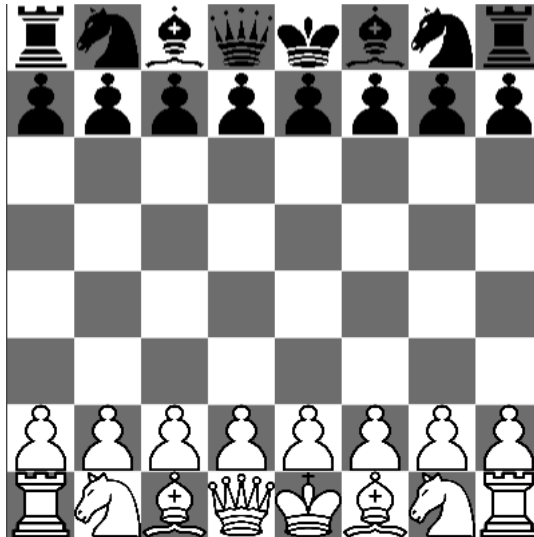
- (c) Using the previous two pictures, make a picture `middleBoard` representing the four empty rows in the middle of a chessboard:



- (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:



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))
```



Exercises

11. (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`.

