Getting Started
Informatics 1 – Functional Programming: Lab Week Exercise

Burroughes, Heijltjes, Wadler

Due: Friday 3 Oct. (5pm)
Reading assignment: Chapters 1–3 (pp. 1–52)

Please attempt the entire worksheet in advance of the tutorial, and bring with you all work, including (if a computer is involved) printouts of code and test results. Tutorials cannot function properly unless you do the work in advance.

You may work with others, but you must understand the work; you can't phone a friend during the exam.

Assessment is formative, meaning that marks from coursework do not contribute to the final mark. But coursework is not optional. If you do not do the coursework you are unlikely to pass the exams.

Attendance at tutorials is obligatory; please let your tutor know if you cannot attend.

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, the interactive Haskell interpreter GHCi[1], and the "`submit`" command used to hand in your electronic coursework.

The exercise consists of three 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.

It is important to complete all three 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 there.

[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 6.8.3: http://www.haskell.org/ghc/  :? for help
   Loading package base ... linking ... done.
   Prelude>
   ```

   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:

`:load` *filename*	Load a file containing Haskell definitions
`:reload`	Reload the most recently-loaded file
`:type` *expression*	Display the type of *expression*
`:?`	Display a list of commands

Exercises

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

Emacs

To start up the text editor Emacs, simply type "`emacs`" at the shell prompt or select "Emacs Text Editor" from the "Applications ⇒ Programming" 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

Emacs and GHCi

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.

Setting up Emacs

The Emacs installation on DICE is set up to use *Haskell mode* for editing files whose names end with `.hs`. 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, just type the following at a comand prompt (do this now):

```
cat /home/infteach/.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. Subsequently, typing `ctrl-c ctrl-r` causes Emacs to reload the 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 `labweekexercise.hs` from the course website:

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

and open the file in Emacs. Below 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. (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 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 $3^2+4^2 = 9+16 = 25 = 5^2$.

Exercises

4. Write a function `isTriple` that tests for Pythagorean triples.

   (a) Find the skeleton declaration of `isTriple :: Int -> Int -> Int -> Bool` and replace `undefined` with a suitable definition.

(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 $x > y$.

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.

As a final note, if you also want to know all of the values that QuickCheck tries *successfully*, use `verboseCheck` instead of `quickCheck`:

```
Main> verboseCheck prop_triple
```

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 use this opportunity to see how it works—it might save you valuable time at the exam.

Once you have completed working on the Haskell file[2], `labweekexercise.hs`, you need to submit it with the `submit` command. For *this* exercise, you should use the command as follows:

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

The meaning of the arguments is as follows:

`inf1`:	the code for your current year, informatics 1;
`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/s08...../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/s08...../Desktop/labweekexercise.hs
```

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

```
submit 2.65.4-1 usage: to submit an exercise:
submit <year> <course> <exercise-number> <file1> <file2>...
where the <exercise-number> is some integer and <file> can
be a regular file or a directory.

The exercise name should be one of:
lab
 for course inf1-fp of year inf1.


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.

[2]If 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 2.65.4-1 usage: to submit an exercise:
submit <year> <course> <exercise-number> <file1> <file2>...
where the <exercise-number> is some integer and <file> can
be a regular file or a directory.


You need to specify which year you are in.
Valid 'Year' choices are:
unit     (courses: unit)
msc     (courses: tts-5 tpas tm-5 tdd sbm sapm-5 rtse rl...
inf2    (courses: inf2d inf2c inf2b inf2a)
inf1    (courses: inf1-op inf1-fp inf1-da inf1-cl)
cs4     (courses: tpl slip segp sapm-4 qsx proj ppls-4...
cs3     (courses: st seoc sdp pi os lsi ip fps ec dsi dbs ct...
cs1     (courses: cpmt cp1 cl1)
cg1     (courses: hc1 cfcs1)
ai4     (courses: tts-4 tm-4 proj masws-4 la-5 la-4 ...
ai3     (courses: pi lp kri ivr ics gagp ailp abs)
test    (courses: test5 test4 test3 test2 test1)


SUBMISSION DID NOT HAPPEN!
```

When you think you have successfully submitted the file, you can check with the `show submissions` command. Typing "`show submissions inf1`" at the prompt gives an overview like this:

```
EXERCISE      LAST
YEAR   MODULE     NAME      MODIFIED         SIZE(bytes)    NAME
================================================================================
inf1   inf1-fp    lab   Thu, 26, Sep   8:15      3589    labweekexercise.hs
```

Note that `show submissions` displays every file for every exercise that *you* have submitted; it does not show files submitted by other users.