Functional Programming - Lab 1

Based on the Informatics 1 — Functional Programming @ University of Edinburgh

Welcome

Welcome to your first functional programming exercise! This document will explain how to get started writing Haskell. 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. Chess
   Part three is an exercise where you will compose and manipulate images of chess pieces.

It is important to complete all three parts and start getting used to the computer labs.

The system

The first part of the exercise will explain how to set up and use the GHCi Haskell interpreter.

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 7.10.3: http://www.haskell.org/ghc/  :? for help
   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.

Editor

You can use any text editor. Editors with syntax highlighting for Haskell (gedit, gvim, Emacs) are preferable.

The usual working cycle would be Editing a file and the (re)loading it into GHCi

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.

Getting started

The current directory should contain the files `labweekexercise.hs`, `labweekchess.hs` and `PicturesSVG.hs`.

Open the file `labweekexercise.hs`. 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. 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 a2 +b2 = c2. For example, (3, 4, 5) is a Pythagorean triple, since 32 +42 = 9+16 = 25 = 52.

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.

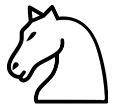
Chess

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`. Load it into GHCi and type this at the prompt:

```
Main> render knight
```

The webpage (which refreshes every second) should now show a picture of a white knight chess piece:



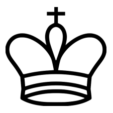
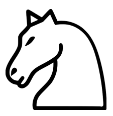
Note: If you don't see the pictrue you were rendering try refreshing the webpage.

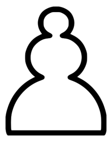
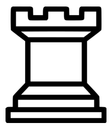
The tutorial file labweekchess.hs is able to draw pictures using the module `PicturesSVG`, contained in the file `PicturesSVG.hs`, by means of the line:

```
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

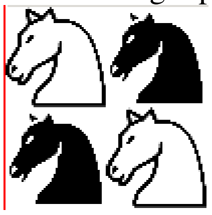
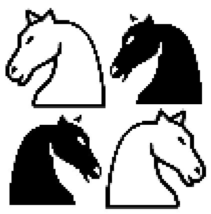
7. Ask GHCi to show the types of these functions.

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

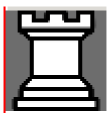
8. Use the knight picture and the above transformation functions to create the following two pictures:

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)
```

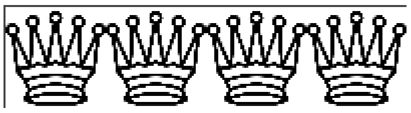


You can use `over` to put any picture on top of another, but the result looks best if you simply put pieces on squares.

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)
```

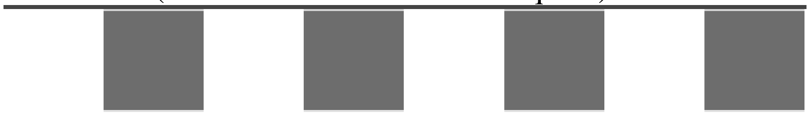


Notes:

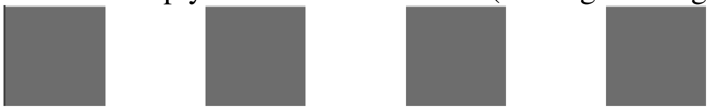
- When a problem says "... using the function (or picture) `foo`," you *must* use the function `foo`. A solution that does not use that function will not be accepted, but of course you can use other functions as well.
- Unless an exercise says you can't, you are free to define intermediate functions,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

Exercises

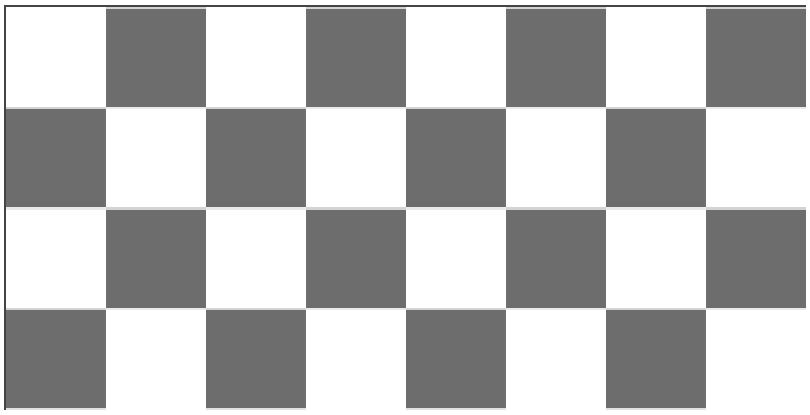
9.
   a. Using the `repeatH` function, create a picture `emptyRow` representing one of the empty rows of a chessboard (this one starts with a white square).

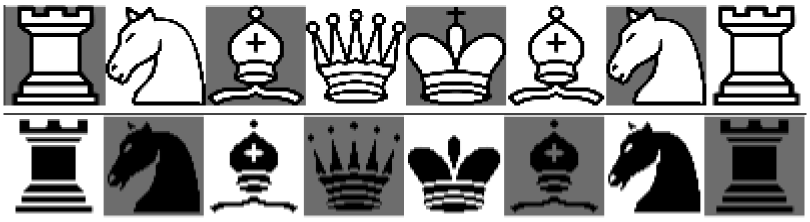
   b. Using the picture `emptyRow` from the last question, create a picture `otherEmptyRow`, representing the other empty rows of a chessboard (starting with a grey square).

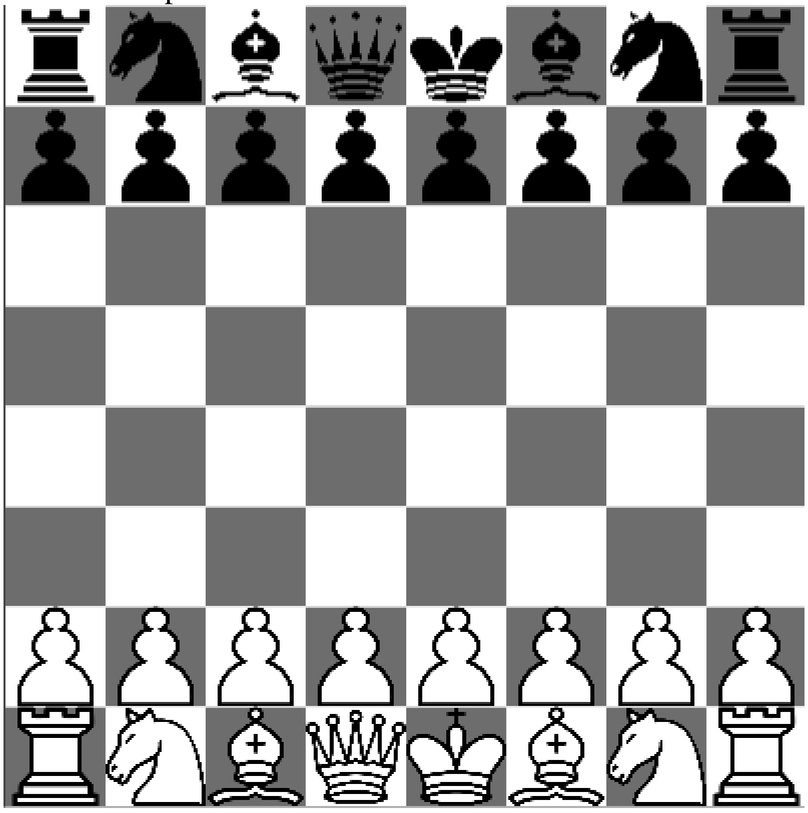
   c. Using the previous two pictures, make a picture `middleBoard` representing the four empty rows in the middle of a chessboard:

   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:



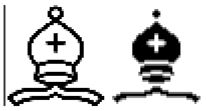
Functions

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`:

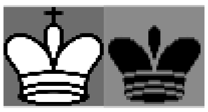
```
twoBeside :: Picture -> Picture
twoBeside x = beside x (invert x)
```

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

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

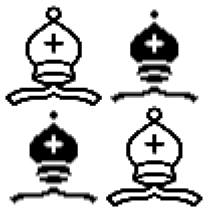


```
Main> render (twoBeside (over king blackSquare))
```



Exercises

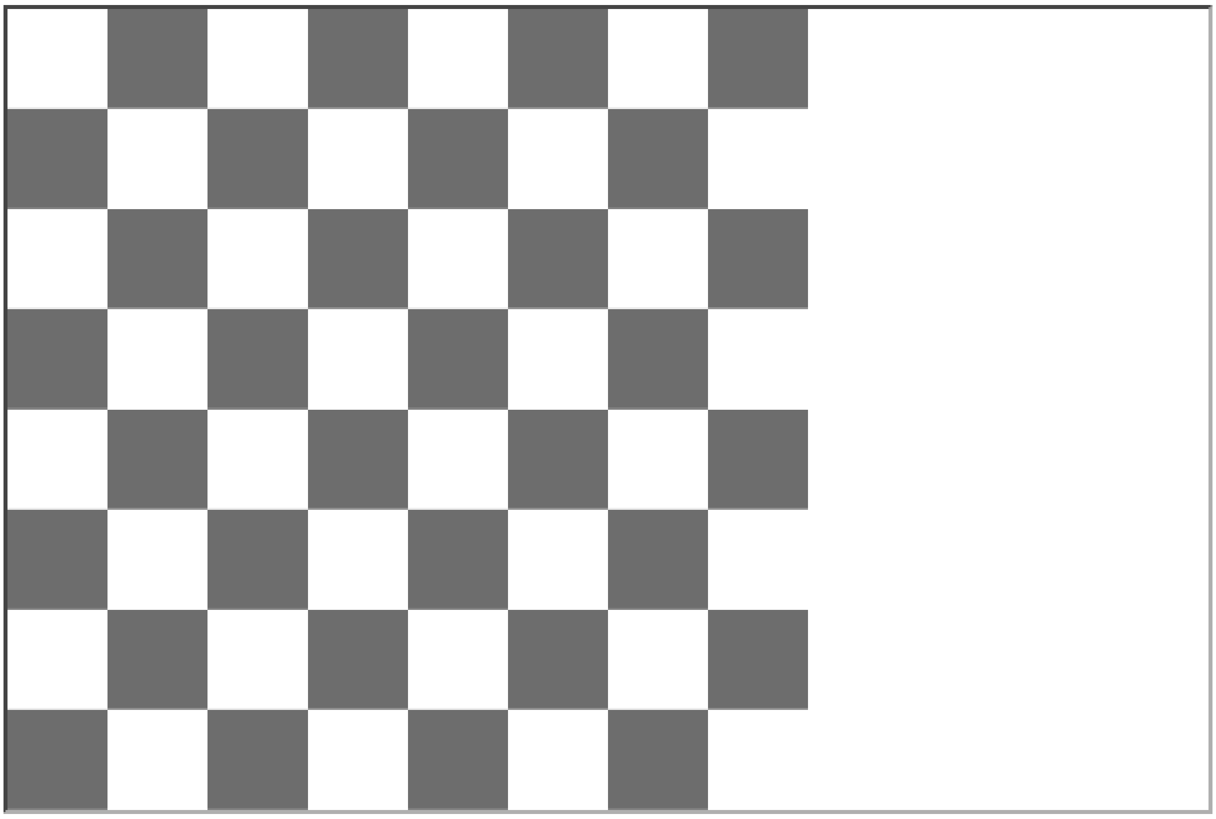
10. a. Write a function `twoAbove` that places a picture above an inverted copy of itself;
    b. Write a function `fourPictures` that puts four pictures together as shown below. You may use the functions `twoBeside` and `twoAbove`.



Extra credit exercise

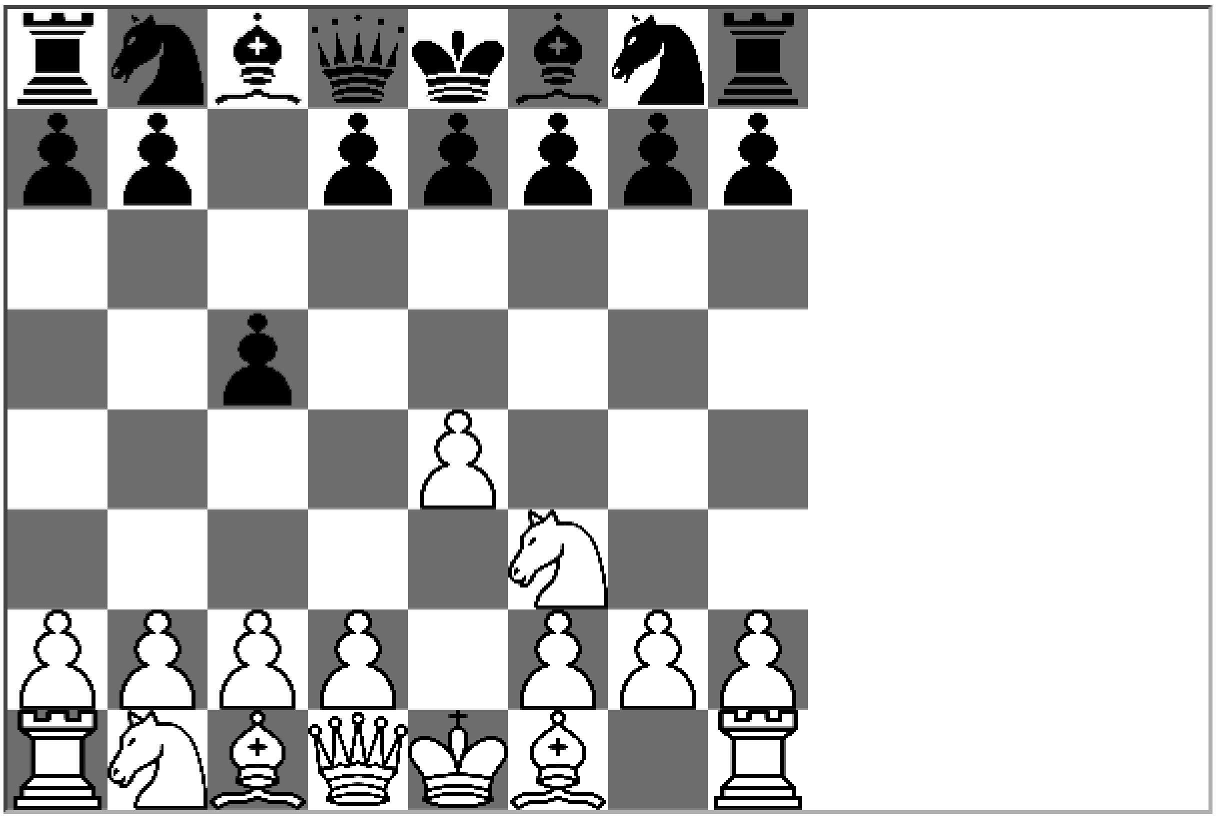
a. Create a picture `emptyBoard` describing the empty board.

```
Prelude> render emptyBoard
```



b. Given the description of a chess position in the FEN notation, define a picture representing it. Use the `emptyBoard` picture created above. For example, the `e4c5Nf3` picture is described by the **"rnbqkbnr/pp1ppppp/8/2p5/4P3/5N2/PPPP1PPP/RNBQKB1R"** FEN string and it is rendered to:

```
Prelude> render e4c5Nf3
```



In the coming laboratories we will learn how to write a function which turns any such FEN description into a picture.