Install You a Haskell for Great Good!
Informatics 1 – Introduction to Computation
Functional Programming Tutorial 1

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

Week 2
due 12:00 on Tuesday 24 September 2024
tutorials on Thursday 26 September and Friday 27 September 2024

You will not receive credit for your coursework unless you attend the corresponding tutorial session. Please email your tutor 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.
Inf1A tutorials are for learning, and — as an exception to the University's rules on good scholarly practice — you are encouraged to collaborate and to seek help from others when you get stuck. But you must always understand any solution you hand in, and be able to explain it to your tutor or a fellow student.
Please do not publish solutions to these exercises on the internet or elsewhere, to avoid others copying your solutions.

Welcome

Welcome to your first functional programming tutorial! This document will explain how to get started writing Haskell.

As with all Inf1A tutorials, you should complete all work in advance of the tutorial, and submit by the deadline above.

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 GHCi. ('GHC' stands for 'Glasgow Haskell Compiler'.)

The tutorial consists of the following parts:

0. **Install you a Haskell** In the first part you will set up the system and get to know the basic tools for programming.
1. **Getting Started** The second part consists of some simple exercises where you will write some arithmetic functions in Haskell.
2. **Optional Material: Chess** Part three is an optional exercise where you will compose and manipulate images of chess pieces.

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, go to the link below.

https://www.haskell.org/ghcup/

Then follow the installation instructions for your platform. If you're unsure where to begin, here is a YouTube video explaining the setup (the instructions are for Windows, but steps on other platforms are very similar). The interactive installer (GHCup) will set up GHC and Cabal for you. It will optionally allow you to also install HLS and Stack. This is what all these names mean:

- GHC (Glasgow Haskell Compiler) A tool that translates human-readable code into machine-code that a computer can run.
- Cabal (A Haskell build tool) Cabal is used to structure Haskell projects, build them, run them, define dependencies, etc.
- Stack (A Haskell build tool) An alternative to Cabal.
- HLS (Haskell Language Server) You won't use HLS directly, instead your code editor may use it in the background to provide you with a better experience while editing Haskell code (like syntax highlighting and auto-completion).

Choose to install all of them. Additional platform-specific dependencies may also appear (e.g., MSYS2 on Windows); select to install those too.

Next, install the `QuickCheck` package by running the following in your terminal:

```
$ cabal update
$ cabal install --lib QuickCheck
```

You can use Haskell with any text editor as shown in this short video.

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

Type "`ghci`" in your terminal.

(a) Try "`3 + 4 * 5`" and "`(3 + 4) * 5`". Does arithmetic in Haskell work as expected?

(b) Find the length of a string by typing "`length "This is a string"`".

(c) Reverse a string by typing "`reverse "Madam I'm Adam"`".

(d) Check that `QuickCheck` is installed correctly by typing "`import Test.QuickCheck`". This should not produce an error message.

Integrated Development Environments (IDEs)

There is also the option of integrating Haskell and GHCi into your favorite code editor. This is an optional step that some students might wish to pursue. Here, we point to some popular choices:

- Haskell mode for Emacs: https://haskell.github.io/haskell-mode/.
- Haskell extension for Visual Studio Code: https://marketplace.visualstudio.com/items?itemName=haskell.haskell.

1 Getting Started

Your tutorial pack zip file should unpack to 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) Type the command "`:set -package containers`". (This is required in recent versions of Haskell because of differences in the Haskell library. In older versions, like the one installed on DICE, it does no harm.)

(b) Load the file `Tutorial1.hs` by typing "`:load Tutorial1`".

(c) 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.

(d) Save and reload the corrected file.

(e) Use GHCi to display

  i. the value of `double 21`

  ii. the type of `double`, by using the command "`:type double`"

  iii. the type of `double 21`

(f) What happens if you ask GHCi to evaluate `double "three"`?

(g) 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 BCE. 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 (325–265BCE) 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 test equality).

(b) Load the file and 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 `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 or ends with `prop` (short for "property") 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 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`:

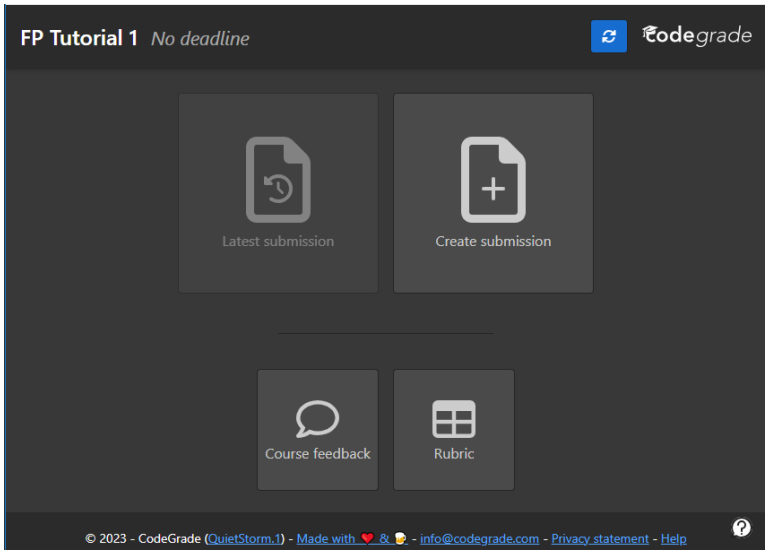
```
Main> verboseCheck prop_triple
```

Submitting your work

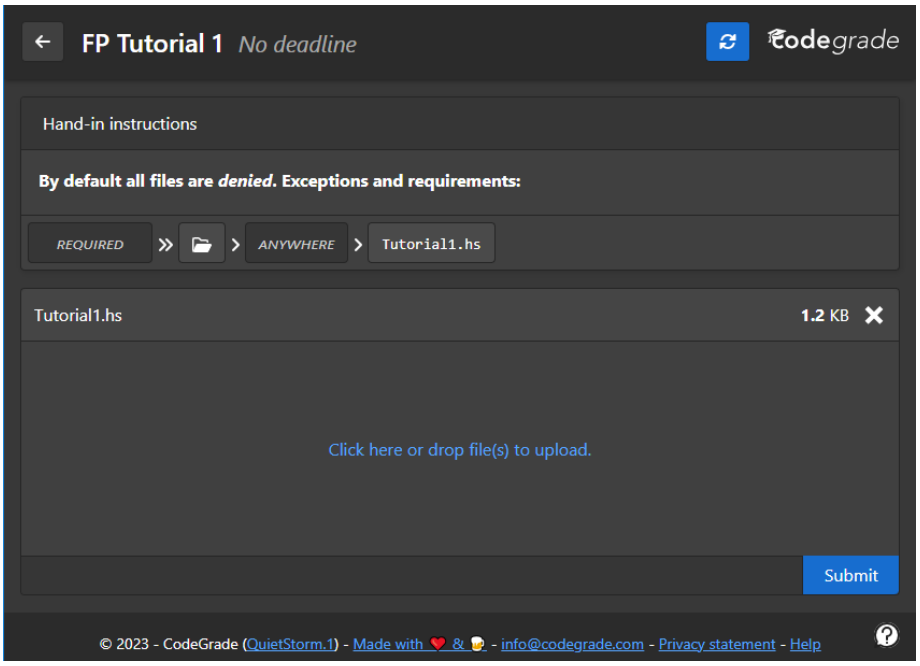
Exercise 6

Once you have completed all the above exercises, submit your work on CodeGrade. Don't worry if you want to also try the optional exercises – you can submit as many times as you like before the deadline. Submitting early will allow you to catch errors and have a backup on-time submission just in case.

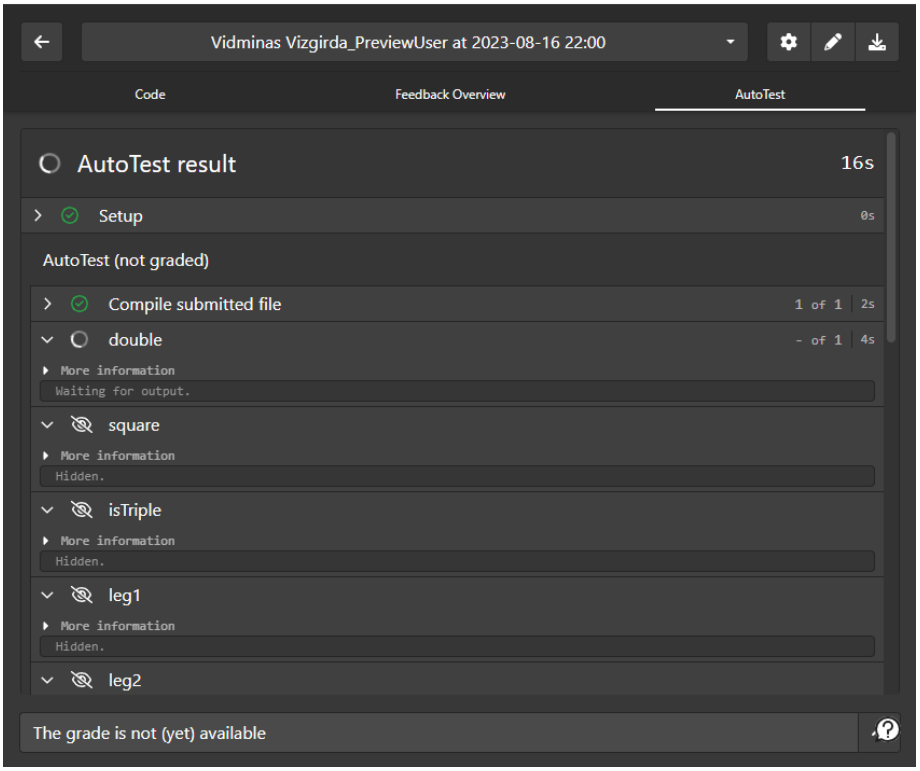
1) Go to the course Learn page and, under the week 1 folder and tutorial 1 materials subfolder, find the link to CodeGrade. When opened, the submission box should look like this:



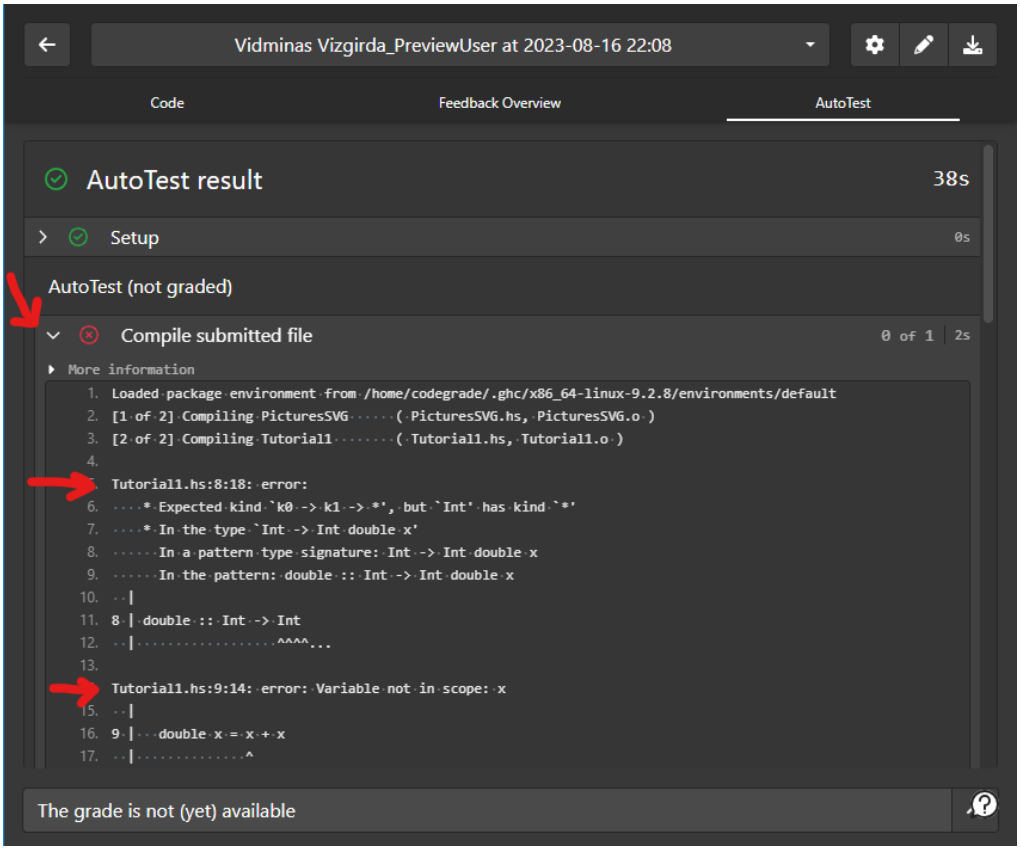
2) Click "Create submission". This will present hand-in instructions (which files are expected) and a space to upload files. For tutorial 1, only one file is expected: `Tutorial1.hs`. Upload the file with your solutions and click Submit.



3) Once submitted, CodeGrade will run automatic tests on your code. You can see the results by opening the AutoTest tab at the top. These can take some time to run, especially if many students are submitting at the same time; this is another reason to submit early. If your solution compiles correctly, the first autotest will pass like in the screenshot below. Hidden tests will also run, but their results will only become visible after the deadline.



4) If there are errors in your submitted code, some autotests will fail. You can expand and collapse test output (including error messages) by clicking the > arrow to the left of each step:



Error messages can be cryptic, but they usually say which exact place the error occured and looking at the code in that place can help understand what's going on. In this case, the first error doesn't make much sense unless you've seen it before, but it starts with `Tutorial1.hs:8:18` which means the error occured in file Tutorial1.hs, line 8, column 18.

Line 8 (displayed just below the error message) is the line `double :: Int -> Int`, which seems correct. Compiler errors (not just in Haskell) are often caused by the line before or after where the error occurs. In this example, the problem was caused by the line after, which is the wrongly indented `double x = x + x` (see exercise 1).

5) To resubmit, click the back (←) arrow in the top left, and press "Create submission" again.

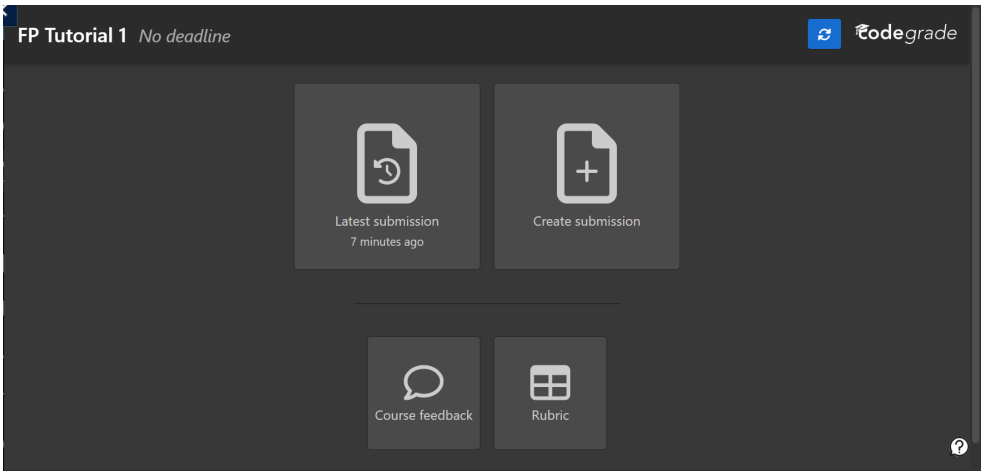
6) **Note:** If you are running out of time before the submission deadline, you can submit the **incomplete** exercise, but first make sure that the REPL can **load it without errors**. Otherwise, it will be more difficult for markers to run your code. To do this, turn any broken code into harmless commentary by putting two dashes (`--`) in front of it, for example:

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

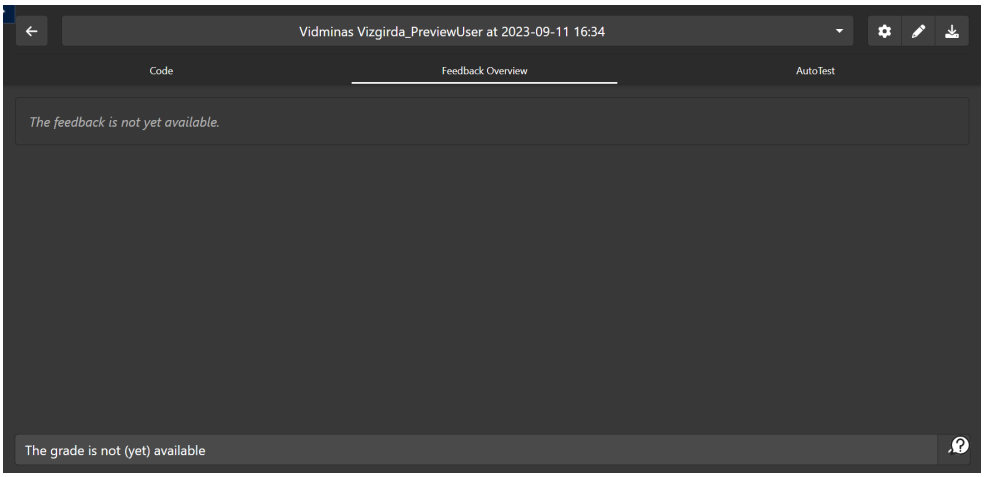
Reviewing your feedback

When you submit your work, you will receive feedback and a grade from your tutor. To review feedback, visit the same CodeGrade submission link and follow these steps:

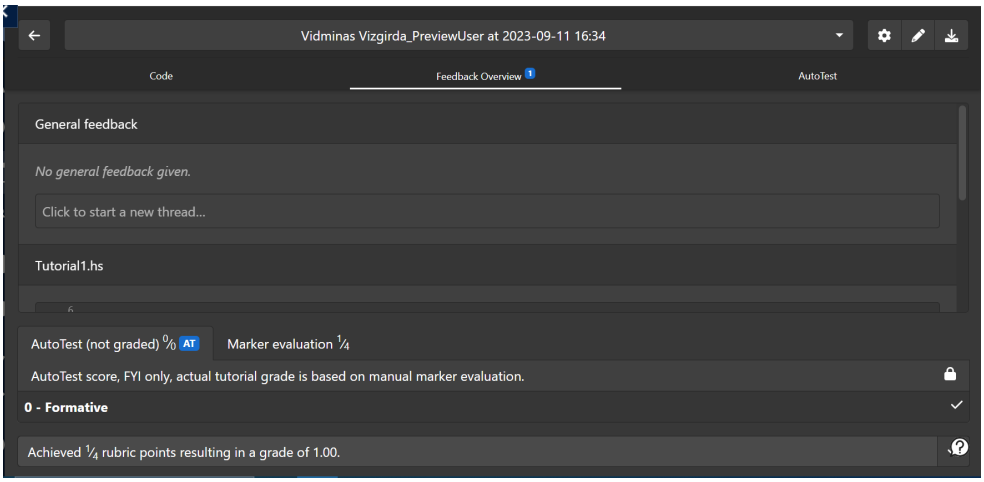
1) Click on “Latest submission” to review your last submission.



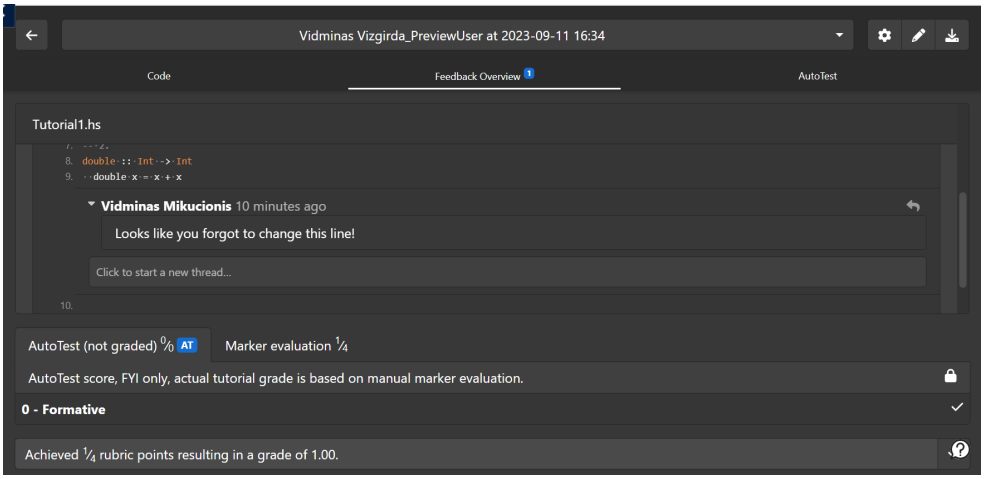
2) Switch to the Feedback Overview tab at the top. Before the deadline, the only feedback that will be available will be in the AutoTest tab, telling you whether your code successfully compiles. If you try the Feedback Overview tab, it will be blank:



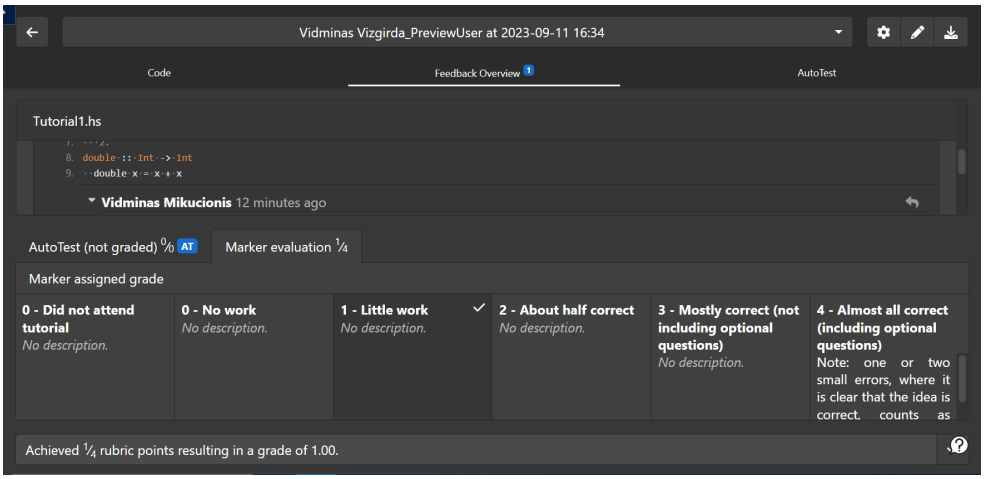
3) After the deadline, when your tutor has marked your work and the grades are published, your feedback will be available in the Feedback Overview tab. Your marker may or may not provide a general comment (in the example below there isn’t one):



4) You will find any inline comments by scrolling below the general feedback:



5) Finally, your grade is available at the bottom of the screen, in the "Marker evaluation" rubric section. In the example below, 1/4 marks ("little work") was awarded.



2 Optional Material: Chess

Following the Common Marking Scheme, a student with good mastery of the material is expected to get 3/4 points. This section is for demonstrating exceptional mastery of the material. It is optional and worth 1/4 points.

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`.

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

`bishop`	A bishop	
`king`	A king	
`knight`	A knight	
`pawn`	A pawn	
`queen`	A queen	
`rook`	A rook	

Board squares

`blackSquare`	A black (grey) square*	
`whiteSquare`	A white square	

* 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

Exercise 7

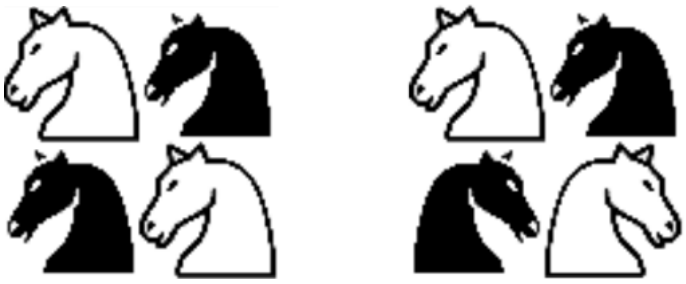
Ask GHCi to show the types of these functions and write them down.

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 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.

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 9

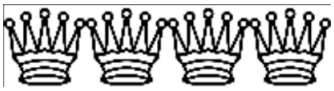
(a) Write a function `twoAbove` that places a picture above an inverted copy of itself.

(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:

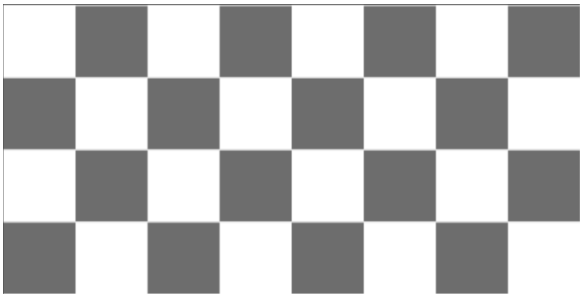
- When a problem says "using the function `repeatH`" or "using the picture `emptyRow`" you must use the named function or picture; a solution that does not is wrong. You can use other functions or pictures as well.
- You are free to define intermediate functions or pictures if that makes it easier to define the solution to an exercise.

Exercise 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: