Week 1: Getting started with Haskell

Hans Georg Schaathun

25th April 2017

Time	Topic	Reading
8.15-	Introductory Lecture	(Thompson Chapter 1-3
9.00-	Install software	
	Tutorial 1: Getting Started	
11.45-	Lunch break	
12.15-	Recap/discussion of Tutorial 1	
	Lecture: Recursion and problem solving	Thompson Chapter 4, 10.2)
13.00-	Tutorial 2–4: Using functions	
	If time, continue on Tutorial 5	

We will focus on Tutorials 1–4 in class. You should aim to complete all five tutorials by next Friday, *especially* if you find it challenging.

This PDF document is available in an HTML version at `http://www.hg.schaathun.net/FPIA/week01.html`.

1 Software Installation

We will be programming in Haskell for this module. Hence you will need to install the Haskell platform one way or another.

All our examples and instructions will assume that you use some kind of standard Unix system, typically linux. We know that many students use more obscure operating systems, and we will try to help. However, it is your own responsibility to make it work if you use non-standard systems.

We have prepared a virtual machine image which you may try if you want to try out linux on a non-linux system. In our experience though, most students find it easier to install the Haskell-platform on their system, and live with quirks, than to install a virtualbox and the virtual machine. It is your choice.

1.1 Install the Haskell Platform

You can download and install the Haskell platform from `https://www.haskell.org/platform/`. Many linux distros also have it available in the standard repos. For instance, in Debian you can do

```
apt-get install haskell-platform
```

You will probably have to install a number of cabal packages. For instance we need `easyplot` later today. You can install it using the following command:

```
cabal install easyplot
```

Other packages can be installed as the need arises. Which packages you need may depend on your OS and how you install the Haskell platform.

You will also need a text editor to edit source code. Windows Notepad is a bad choice because it does not understand unix-style linebreaks. Most other editors should work well, even on files imported from other operating systems. Popular examples include gedit, vim, and emacs.

Easyplot depends on gnuplot which has to be installed as a standalone problem. On Debian/Ubuntu this is straight forward:

```
apt-get install gnuplot
```

On recent releases on Mac OS you may have to instal an X server, such as XQuartz, to make gnuplot work.

1.2 Install Virtualbox

You need to download and install virtualbox from `https://www.virtualbox.org/wiki/Downloads`, following the official instructions for you OS.

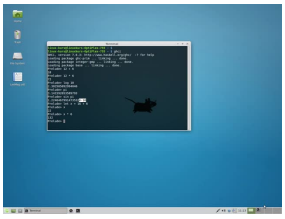
When you have virtualbox, you can download and add the VM for the module.

1. Download the VM from `http://www.hg.schaathun.net/FPIA/fpiavm.zip`
2. Unzip the archive and put in a suitable place.
3. Start virtualbox.
4. Import the fpia VM (Machine → Add)
5. Start the VM
6. Log in using
   - Username: `fpia`
   - Password: `NeuralNetwork`

2 Tutorial 1: Getting Started

2.1 Step 1: Getting started

First of all, we need to get to grips with the Haskell interpreter, that is GHCi.



Do the following.

1. Open a terminal window
2. Start `ghci` on the command line
3. Use GHCi to calculate the following expressions:

```
12 * 412 - 15
12 * (412 - 15)
1 / (2*pi)
cos pi
2^2
2**0.5
```

   Do you get the results you expect?

4. Use GHCi to calculate the following expressions:

$$7 + 37 \cdot 63 = \tag{1}$$
$$3 \cdot (12 + 9) = \tag{2}$$
$$11^{16} = \tag{3}$$
$$\frac{\cos \pi}{2} = \tag{4}$$
$$3^{12} \mod 19 = \tag{5}$$
$$0.4^4 = \tag{6}$$

   Do you get the results you expect?

5. ghci has a number of interpreter commands which are not part of the Haskell language. Such commands start with a :. Have a look at the list of available commands by typing

```
:help
```

   You will see some of the listed commands in use later.

6. Exit ghci. You can use either Ctrl-D or the `:quit` command to exit.

2.2 Step 2: Using a script

Although ghci makes a neat calculator out of the box, that's not quite what we want in this module. To solve complex problems we need to write our own programs, that is, define our own functions. To do this, we use separate text files, called scripts or modules, such as the following FirstScript.hs file.

```
1 -- (C) 2016: Hans Georg Schaathun <hasc@ntnu.no>
2
3 module FirstScript where          -- Module header
4
5 double :: Integer -> Integer      -- Function declaration (type)
6 double x = 2*x                    -- Function definition
```

The first line, and other text following a double dash (`--`), are comment which is ignored by GHC/GHCi. The module header (Line 3) gives the module name which has to match the file name. The function *declaration* (Line 5) gives the type of a new function. The `double` function takes an integer as input and gives an integer as output. Mathematically, we would write `double` : $\mathbb{Z} \to \mathbb{Z}$. The last line *defines* the `double` function. The return value is two times the input value `x`.

Do the following:

1. Make a new directory to keep your notes and files for this first tutorial. E.g.

   ```
   mkdir Week01
   ```

2. Change into the new directory. E.g.

   ```
   cd Week01
   ```

3. Download the file FirstScript.hs file and put it in the current directory (`Week01`).

4. Check the contents of the new file by opening it in your editor, e.g.

   ```
   gedit FirstScript.hs
   ```

5. Start `ghci` from the command line

6. Load the script using the `:load` command:

   ```
   :load FirstScript
   ```

   Note that the extension `.hs` may be omitted.

7. Test the function by typing

   ```
   double 5
   ```

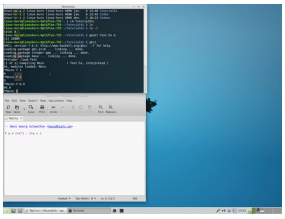
8. Try the following

```
double 2.5
```

What happens? Remember the function declaration. What input does `double` accept?

2.3 Step 3: Making a script

Now, you are going to make your own simple script/module from scratch.



1. Make sure that you are in the `Week01` directory which you used in the previous step. You can do this with the following command which prints the absolute path to the current directory:

   ```
   pwd
   ```

2. Open an editor to edit a Haskell file. E.g.

   ```
   gedit MyFirstScript.hs
   ```

   The name of a Haskell file (script) have to end with `.hs` and should start with an upper-case letter. E.g. MyFirstScript.hs.

3. Start with a comment at the top of your file to claim authorship, e.g.

   ```
   -- John Doe <john@doe.nowhere.invalid>
   ```

   Comments are ignored by the compiler or interpreter, and are purely a help for human readers.

4. Define a new module by adding a module header under the comment

   ```
   module MyFirstScript where
   ```

   Note that the module name must match the filename without the .hs extension.

5. Define your first function inside the

   ```
   greet :: String -> String
   greet name = "Hello, " ++ name
   ```

6. Save the file

7. Start ghci from the command line

8. Load your script, e.g.

```
:l MyFirstScript
```

`:l` is shorthand for `:load`.

9. Test the function

```
greet "John"
```

10. You can add more functions to your module. Do not close ghci. Use a different window to add a new function to your module.

```
square :: Int -> Int
square n = n*n
```

11. Returning to ghci, reload the module using

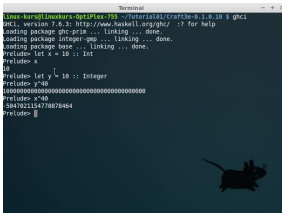
```
:reload
```

12. Test the new function:

```
square 2
square 5
```

2.4 Step 4: Working with Integers

One of the fundamental data types is integers. In fact Haskell has two integer types, namely Int and Integer. We shall make a little experiment to see the difference.



1. Open your text editor with a new file, IntTest.hs is a good name.
2. Define a function to do exponentiation of Integer.

```
pow1 :: Integer -> Integer -> Integer
pow1 x y = x^y
```

3. Define another function to do exponentiation of Int.

```
pow2 :: Int -> Int -> Int
pow2 x y = x^y
```

4. Open ghci and load the module you just created.
5. Test the two functions, with small and big arguments.

```
pow1 2 10
pow2 2 10
pow1 2 65
pow2 2 65
```

```
pow1 3 39
pow2 3 39
```

What happens? Why do pow1 and pow2 give different answers? Discuss answers with your class mates.

2.5 Step 5: Working with Booleans

Boolean is another fundamental data type, with two possible values, True or False.

1. Open your editor to create a new Haskell module, e.g. BoolTest.hs
2. Add the customary comment to identify yourself as author and the module header.
3. Add a couple of functions:

   ```
   myNot :: Bool -> Bool
   myNot False = True
   myNot True = False

   myAnd :: Bool -> Bool -> Bool
   myAnd True True = True
   myAnd _ _ = False
   ```

   Both functions are standard logical operators, the negation (not) and the conjunciton (and).
4. Start ghci, load your module, and test the above functions, e.g.

   ```
   myNot False
   myNot True
   myAnd False False
   myAnd False True
   myAnd True False
   myAnd True True
   ```

   Are you happy with the results?
5. Add functions for logical or (myOr) and exclusive or (myXOR) to your module using pattern matching and literals. There is more than one way of doing it, so just think about the mathematical (logical) meaning and do it your way.
6. Reload your module and test the new functions. Do they work as intended?

2.6 Step 6: Working with Floating Point Numbers

Many of our algorithms work on real numbers. Unfortunately, the computer does not support real numbers. Instead we have to work with *floating point numbers*. There are two floating point

data types in common use. In Haskell, they are called Float and Double, but they are defined by the CPU architecture and not by the language. Let's explore this

1. Why cannot computers work with real numbers?
2. Start ghci
3. Compare the following expressions

```
2**500 :: Double
2**500 :: Float
2**(-500) :: Double
2**(-500) :: Float
```

The floating point types permit a couple of non-numeric values, namely ±Infinity, and NaN (Not a Number).

4. Evaluate each of the following Haskell expressions using GHCi. What value do you expect to get? What value do you actually get?

```
1/0
0/0
isNaN (log (-1))
(log (-1)) == (log (-1))
(1/0)*0
isInfinite (0/0)
isInfinite (1/0)
```

NaN values will usually just propagate through floating point operations, so that NaN in gives NaN out. However, some operations have unpredictable results, so you should check for NaN whenever there is a risk that NaN has occured.

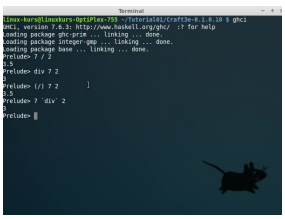
5. What value do you get from the following expressions:

```
ceiling (log (-1))
ceiling (0/0)
floor (0/0)
```

2.7 Step 7: Prefix and in-fix notation

We have seen a couple of functions. For instance:

```
cos 1
pow1 2 10
pow2 2 10
```



They are evaluated by writing the function name first, and then a string of arguments. This is called *prefix notation*, since the function name prefixes the arguments.

We have also seen a couple of operators. For instance

```
2 + 2
3 * 5
6 / 2
```

They are evaluated by writing the operator symbol between two *operands*. This is called *in-fix notation*.

1. Try the following expressions in GHCi and compare:

   ```
   2 + 2
   (+) 2 2
   ```

2. Try the following expressions and compare:

   ```
   3 * 5
   (*) 3 5
   ```

3. Try the following expressions and compare:

   ```
   6 / 2
   (/) 6 2
   ```

4. Make sure that you have loaded the module defining `pow1` and `pow2`. Try the following expressions and compare:

   ```
   pow1 6 2
   6 `pow1` 2
   ```

5. Try the following expressions and compare:

   ```
   pow2 2 3
   2 `pow1` 3
   ```

6. Try the following expressions and compare:

   ```
   32 `mod` 10
   mod 32 10
   ```

7. Try the following expressions and compare:

   ```
   32 `div` 10
   div 32 10
   ```

8. Discuss *What is the difference between an operator and a function?*

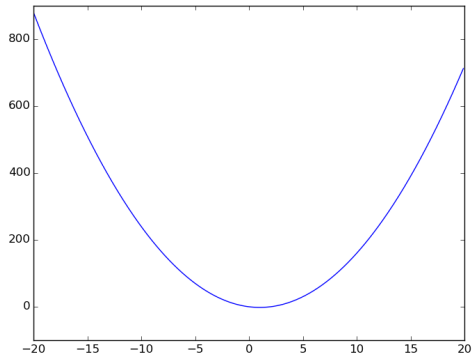


Figure 1: Plot of $f(x) = 2x^2 - 4x$.

3 Tutorial 2: Studying functions in Haskell

3.1 Worked example

We can use Haskell to study the behaviour of mathematical functions. The function $f(x)$ is defined as

$$f(x) = 2x^2 - 4x$$

and plotted in Figure 1.

1. To study the function, we need to define it as a function. Create a module in a file `NumTutorial.hs`, with the following definition

```
module NumTutorial where

f :: Double -> Double
f x = 2*x**2 - 4*x
```

2. In this exercise, we will use the easyplot library. We install it with the following commands before we start GHCi.

```
cabal update
cabal install easyplot
```

3. `easyplot` depends on gnuplot, which you also have to install. If you run Debian/Ubuntu, try the following.

```
apt-get install gnuplot
```

   Otherwise please google for installation instructions for your OS.

4. Open GHCi

```
ghci
```

5. Load the module

```
:l NumTutorial
```

6. Test the function `f`

```
f 0
f 1
f (-1)
```

   Do the results look reasonable?

7. We import the EasyPlot library so that we can use it

```
import Graphics.EasyPlot
```

   You will see that the prompt changes, to indicate loaded modules.

8. The following command plots the function `f`.

```
plot X11 ( Function2D [] [] f )
```

   The second argument to `plot` is a data set. The `Function2D` function creates such a data set from a function.

   Note If you run Windows you need to replace `X11` with `Windows` (unless you have an X server running). There is a similar `Aqua` option for Mac OS, but Mac ofthen has an X server as well.

   If the plot window closes before you manage to see it, try the following variant instead (Windows):

```
plot' [Interactive] Windows ( Function2D [] [] f )
```

9. The two empty brackets in the arguments for `Function2D` are lists of options. We can add a title and restrict the range, as follows:

```
:set +m
plot X11 ( Function2D
           [Title "The f function"]
           [Range (-20) (20)] f )
```

   The `:set +m` command is necessary to allow one statement to span multiple lines. The plot statement has been split over three lines just to make it easier to read.

10. The `X11` argument to `plot` says that the plot be shown in an X11 window. It is possible to plot to file, as follows:

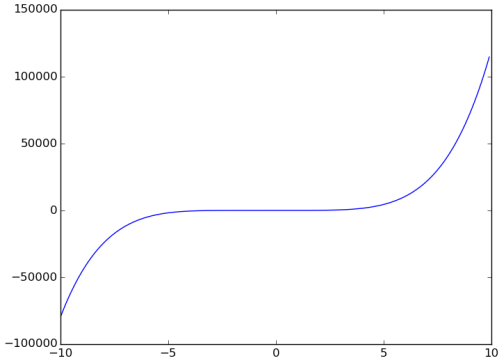


Figure 2: Plot of $g(x) = x^5 + x^4 \cdot 2 + x^2 \cdot 4 + x - 10$.

```
:set +m
plot (PDF "fplot.pdf") ( Function2D
          [Title "The f function"]
          [Range (-20) (20)] f )
```

11. Leave GHCi (use the `:q` command).
12. Open the PDF plot, using

    ```
    okular fplot.pdf &
    ```

If you want to learn more about EasyPlot, you find documentation on Hackage.

3.2 Practice Problem

Let's study a second function $g(x)$. A plot is shown in Figure 2, and the definition is

$$g(x) = x^5 + x^4 \cdot 2 + x^2 \cdot 4 + x - 10$$

Please implement and plot this function in Haskell, as follows:

1. Add a definition for a function `g` in the file `NumTutorial.hs`.
2. Open GHCi and load `NumTutorial`.
3. Evaluate $g(0)$, $g(1)$, and $g(-1)$ to test the function `g`. Do the results look reasonable?
4. Import `Graphics.EasyPlot`
5. Plot the function `g` on the range $-10 \geq x \geq 10$ in a window. Give the curve a meaningful title.
6. Plot the `g` function to a PDF file (`gplot.pdf`).

7. Leave GHCi (use the `:q` command) and open the PDF plot in `okular` (or another PDF viewer).

4 Tutorial 3: Recursion and Problem Solving

In this section we will continue the study of the two functions $f(x)$ and $g(x)$ from the previous tutorial.

Reading: Simon Thompson, Chapter 4 and 10.2

4.1 Worked Example: Optimisation

A minimisation problem has the form

$$\min_x f(x)$$

for some function $f(x)$. The problem is to find the smallest possible value for $f(x)$. Often, we also want to find the corresponding value of x.

One simple minimisation algorithm is obtained by imagining a ball rolling on the curve. It will always run down-hill, and eventually it will stop in a local minimum. We will implement this algorithm to find the x value minimising the function $f(x)$ from the previous tutorial.

1. Open the module from the previous tutorial, with the definition of the `f` function.
2. We need to decide how far the ball should roll per iteration. For the time being we make it a constant, so add the following line to you module

```
delta = 0.01
```

3. Then we decide on the bounds for the search. We search for a minimum in the interval $(-20, 20)$, and add the following lines to the module.

```
low = -20
high = 20
```

4. We shall implement the function `fmin` which takes a starting point x_0 as input, and returns a (local) minimum. First we add the type declaration to our module, with one input and one output:

```
fmin :: Double -> Double
```

5. Now we can start adding definitions for `fmin`. Using guards, we can define one case at the time. First of all, if the ball has reached the lower or upper bound, this bound value is returned:

```
fmin x0 | x0 - delta < low      = low
        | x0 + delta > high     = high
```

This constitues a *base case* for recursion.

6. If we find smaller function value to the left of `x0`, i.e. at `x0-delta`, we move left and call the function recursively with the new starting point. Similarly, we move right if this gives a smaller function value.

```
fmin x0 | f (x0 - delta) < f x0  = fmin (x0 - delta)
        | f (x0 + delta) < f x0  = fmin (x0 + delta)
```

7. If neither moving left nor right reduces the function value, we must be in a local minimum which is then returned:

```
fmin x0 | otherwise              = x0
```

8. Save your module, start GHCi, and load (or reload) the module.

9. Test the `fmin` function, as follows

```
fmin 0
```

10. Discuss: does it matter what value you give as starting point?

4.2 Practice Problem: Bisection Method

In this tutorial we shall implement the *Bisection Algorithm* which is a simple numerical approach to equation solving. Consider the following equation as an example

$$0 = x^5 + x^4 \cdot 2 + x^2 \cdot 4 + x - 10$$

Note that the right hand side is the function $g(x)$, so we can write $0 = g(x)$ instead. Let's not discuss whether a solution can be found analytically. We want to find some solution numerically.

A couple of observations are easy to make.

1. $g(-10) < 0$
2. $g(10) > 0$
3. $g(x)$ is continuous

It follows from these three observations that $g(x)$ has at least one root in the interval $-10 < x < 10$. Without considering the possibility of multiple roots, we want to find *one* such root.

You can look up the *Bisection Method* in most undergraduate introductions to calculus. Let $g(x)$ be a **continuous** function and (l, u) an interval so that $g(l)$ and $g(u)$ have different sign; then $g(x)$ has a zero in the interval (l, u). The *Bisection Method* finds the midpoint m in the interval (l, u). If $g(l)$ and $g(m)$ have different sign, then there must be a zero in the interval (l, m) and we use bisection *recursively* on this interval. Otherwise, we call it recursively on the interval (m, u).

4.2.1 Tasks

1. Open your editor and create a new Haskell module for this problem. Choose an appropriate name for the module.

2. Implement the function $g(x)$ described above in Haskell. The type description should be

   ```
   f :: Double -> Double
   ```

3. We are going to implement a `bisect` function, which can be used to find a zero of `f` with a call like this:

   ```
   bisect (-10) 10
   ```

   The arguments are resp. the lower and upper bound of the search interval and have type Double.

   - Write the type signature of `bisect` into your module file.

4. The `bisect` function has to be a recursive function, so we need two cases. Use `l` and `u` to denote the bound of the search interval `(l,u)`.

   Base case If the difference `l-u` is very small, we can just take the average of `l` and `u` as the approximate solution. Write a guarded expression for the base case, e.g.

   ```
   bisect l u | u-l < eps = ...
   ```

   choose an appropriate value for `eps` and add a defintion after the equal sign.

   Recursive case If the search interval is larger, we split it in two, and find which half must contain a root. Then we *recursively* continue the search in the relevant half. The recursive call, with two different cases using guards, can look something like this.

   ```
   bisect l u | fl*fm < 0 = bisect l m
              | otherwise = bisect m u
                      where ...
   ```

   You need to complete the `where` clause to define the midpoint `m` and the function values `fm` ($g(m)$) and `fl` ($g(l)$).

5. Start ghci and load the module that you have written.

6. Test your bisection function by finding the root of $g(x)$ in the interval $(-10, 10)$. I.e. evaluate the following

   ```
   bisect (-10) 10
   ```

   Does the answer look reasonable when you compare to the plot of $g(x)$ in the previous tutorial?

5 Tutorial 4: Higher-Order Functions

5.1 Worked example: The optimisation problem

In Tutorial 3, we implemented minimisation of a single pre-defined function $f(x)$. It would be obviously, be better if we could use `fmin` on any function $h(x)$. Then the function h needs to be an argument to the function.

1. First, let's update the type decalaration, by adding a function parameter. The first argument must have the same type as `f`, i.e. `Double -> Double`. I.e. change the type declaration to

```
fmin :: (Double -> Double) -> Double -> Double
```

2. Secondly, we need to add the parameter in the definition, and every reference to `f` must be replaced by a reference to the parameter name `h`, as follows:

```
fmin h x0 | x0 - delta < low       = low
          | x0 + delta > high      = high
          | h (x0 - delta) < h x0 = fmin h (x0 - delta)
          | h (x0 + delta) < h x0 = fmin h (x0 + delta)
          | otherwise              = x0
```

3. Save the module and reload the module it in GHCi.

4. Test the function

```
fmin f 0
```

   Do you get the same answer as the first time?

5.2 Practice problem: The bisection algorithm

Now you have seen how to make `fmin` a higher-order function. Now you do the same to the `bisect` function, so that it can find zeros of arbitrary continuous functions.

1. We want to change the `bisect` function so that it can be used like this

```
bisect f (-10) 10
```

   The first argument is the function for which we want to find a zero, with type `Double -> Double`, making `bisect` a higher-order function. The other two arguments are as before.

   a) What is a higher-order function?

   b) Change the type signature of `bisect` into your module file.

2. Update the implementation of `bisect` to become a higher-order function.

   a) Add the function argument `f` as the first argument on the right hand side of the definition?

   When the symbol `f` is used as an argument, `f` on the right hand side will refer to the argument and not to the global definition. Thus no changes are required on the right hand side.

3. Test the revised bisection function by finding the root of $g(x)$ in the interval $(-10, 10)$. I.e. evaluate the following

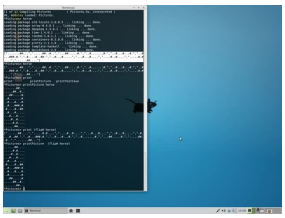
   ```
   bisect f (-10) 10
   ```

   Is the answer the same as before?

6 Tutorial 5 (Optional): The Pictures DSL

6.1 Step 1: The Pictures Example

A major benefit of functional programming is that we often get a description which closely resembles natural or mathematical language as we would use it in a textbook or report. Functional languages are often used to define domain-specific languages (DSL), which support simple and intuitive definition of problems and solutions within a specific domain. Simon Thompson gives a simple example in the Pictures module, which gives a simple DSL for image operations.



1. Download the source code for the textbook. You can do this on the commandline using:

   ```
   wget https://hackage.haskell.org/package/Craft3e-0.1.0.10/Craft3e-0.1.0.10.tar.gz
   ```

2. Unpack the tarball.

   ```
   tar xzf Craft3e-0.1.0.10.tar.gz
   ```

3. Unpacking the tarball gives you a new directory with all the source code for the textbook. Change into this directory:

   ```
   cd Craft3e-0.1.0.10
   ```

4. Use the following command to see the contents of the directory.

   ```
   ls
   ```

   Don't worry about all the different files. We are really only interested in `Pictures.hs`.

5. Start `ghci` inside the Craft3e directory
6. Load the Pictures module

   ```
   :load Pictures
   ```

   Note that the `.hs` extension is not required.
7. Print (render) the `horse` image as ASCII graphics

   ```
   printPicture horse
   ```

   Does it look like a horse?
8. Try the standard `print` function too:

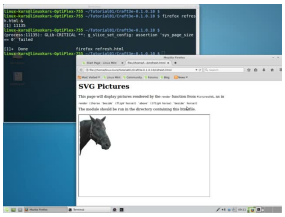
   ```
   print horse
   ```

   This shows the raw data representing the image. You see a list of strings, each string enclosed in double quotes, and the list enclosed in square brackets with commas to separate the elements.

We will use the Pictures module for several exercises later.

6.2 Step 2: Pictures with SVG rendering

The ASCII arts used above isn't too pleasing to the eye. There is a variation rendering SVG graphics as well, namely the PicturesSVG module. It requires a browser on the side for the display. Test it, and see that it works.



1. Make sure that Craft3e-0.1.0.10 is the current directory.
2. Open the file `refresh.html` in a web browser, e.g.

   ```
   firefox refresh.html &
   ```

   The ampersand (&) makes the program start in the background, so that it does not block the command prompt.
3. Start ghci and load the PicturesSVG module. You can do this in two steps, as with the Pictures module above, or you can do both in a single step giving the module name as an argument to ghci:

   ```
   ghci PicturesSVG
   ```
4. Render the horse picture

   ```
   render horse
   ```

5. Take each of the images which you printed in ASCII in the previous step, and render them as SVG.

6.3 Step 3: Playing with Pictures

Do Exercises 2.1-2.4 in Simon Thompson's book.

6.4 Step 4: Designing Problems

Do Exercises 4.25-4.30 in Simon Thompson's book.

7 Solutions

1. Tutorial 1
2. Tutorial 3–4