Discovering Haskell

FIT2102 Programming Paradigms

S2 2017 Week 6

Preamble

Welcome to your week 6 tutorial, first in Haskell! The goal of this session is to learn about the basic syntax of the language.

First of all you will need to install Haskell. [1]

Next, you need to actually *set up* the code. To do so, you will use a tool called *stack* [2] which takes care of all the nitty gritty details. In a nutshell, stack creates a contained environment based on the information provided in the file `stack.yaml`, this allows you to work on multiple projects with different, and possibly conflicting, requirements at the same time.

To test stack, from a command prompt in the home directory (`cmd.exe` on windows, `terminal` on mac and linux) run:

```
$ stack new haskellPlatformTestProj
$ cd haskellPlatformTestProj
$ stack setup
$ stack build
```

The first time you run stack setup it will take a minute or so (it is installing a local version of GHC under `C:\SR` or `~/.stack`) – later invocations will use the cached version and will therefore be very quick.

The default stack installation (`stack new`) will create a number of files in a project directory. The file `stack.yaml` contains build requirements, while `my-project.cabal` contains the code requirements – external libraries and the like. The file you will have to worry about is `src/Lib.hs` where the logic resides, `app/Main.hs` is used to build an executable to run, and the test will not be used as of now.

`stack.yaml` contains build requirements, while `my-project.cabal` contains the code requirements – external libraries and the like. The file you will have to worry about is `src/Lib.hs` where the logic resides, `app/Main.hs` is used to build an executable to run. The file `test/Spec.hs` contains code to run test cases.

For the Week 6 lab class, we have provided you with a `stack` project ready to go. The starter code is in `src/week6`. There are three files in this directory: `Pair.hs`, `List.hs` and `BinTree.hs`. Currently the functions in

[1] The easiest way to is to get Haskell Platform for your operating system.
This should come with all the tools you will need to compile, run, and test Haskell code.

[2] Stack comes with the Haskell Platform (there's a checkbox to tick on the Haskell Platform installer). Optionally, you can download it separately from the stack website and install it for your operating system.

these files are left undefined. Your task is to complete the functions. So that the tests pass.

After unzipping the folder, `cd` into it from the command line and run `stack setup`. If all goes well, after this step, your environment will be set up and you can start hacking. To run code within the environment, use:

```
$ stack ghci
```

You should see something similar to:

```
$ stack ghci
...
Configuring GHCi with the following packages: haskell-tutes
Using main module: 1. Package `haskell-tutes' component exe:
haskell-tutes with ...\week6\app\Main.hs
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help
[1 of 3] Compiling BinTree
[2 of 3] Compiling List
[3 of 3] Compiling Pair
Ok, modules loaded: Pair, List, BinTree.
[4 of 4] Compiling Main
Ok, modules loaded: Pair, List, BinTree, Main.
Loaded GHCi configuration from ...
*Main BinTree List Pair>
```

This is the Glasgow Haskell Compiler interactive interpreter, you will be able to load and run your code directly in there. This interactive console has tab completion, information on the code loaded, etc.

For example, you should be able to create a Pair from REPL:

```
*Main BinTree List Pair> Pair 2 1
Pair 2 1
```

If not, the tutors are here to help.

In every file of the tutorial, you will see functions with a type signature and an `undefined` in the body, you need to replace this with your code. [3]

[3] Comments starting with >>> are called *doctests*, to verify they work run:

```
$ stack test
```

If your code is complete, they should all pass, if not it means you still have work to do.

Exercise 1: Pairs

A `Pair` is an element comprising of two `Int`, that is two natural numbers.

Your goal is to use *pattern matching* to access the elements in a pair, or multiple pairs, in order to apply function to them.

Exercise 2: Binary Tree

A `BinTree` is a *recursive data structure*, which means that each element in a binary tree `is` a binary tree. Each node in a binary is therefore either a `Nil`, or empty tree; or a `Node`. A `Node` has three elements:

1. an integer value;
2. a left sub-tree; and
3. a right sub-tree.

Your task is to write basic functions on a binary tree.

Exercise 3: List

A `List` is an extended container, implemented as a record [4], which holds information about its inner list:

- `size`: the size of the list;
- `elems`: the actual list;
- `low`: the lowest element in the list;
- `high`: the highest element in the list.

Your goal is to implement functions on this data structure while keeping the members up to date; e.g., if you add an element to the list, you have to update the size, and potentially the `low` or `high` members.

[4] Record syntax supports pattern matching as well as accessors.