Functional Programming Course

Written by Tony Morris

With support from NICTA Limited and contributions from individuals (thanks!)

Introduction

The course is structured according to a linear progression and uses the Haskell programming language to learn programming concepts pertaining to functional programming.

Exercises are annotated with a comment containing the word "Exercise." The existing code compiles, however answers have been replaced with a call to the Haskell `error` function and so the code will throw an exception if it is run. Some exercises contain tips, which are annotated with a preceding "Tip:". It is not necessary to adhere to tips. Tips are provided for potential guidance, which may be discarded.

The exercises are designed in a way that requires personal guidance, so if you attempt it on your own and feel a little lost, this is normal. All the instructions are not contained herein.

Getting Started

1. Install the Glasgow Haskell Compiler (GHC) version 7 or higher.
2. Change to the directory containing this document.
3. Execute the command `ghci`, which will compile and load all the source code. You may need to set permissions on a file, `chmod 600 .ghci`.
4. The `Intro` module does not contain exercises. Its purpose is to demonstrate the structure of a project. The first recommend exercise is `Course.List`.
5. Edit a source file to a proposed solution to an exercise. At the `ghci` prompt, issue the command `:reload`. This will compile your solution and reload it in the GHC interpreter. You may

use `:r` for short.

Tips after having started

1. Some questions take a particular form. These are call *WTF questions*. WTF questions are those of this form or similar:

   - What does ____ mean?
   - What does the ____ function mean?
   - What is a ____ ?
   - Where did ____ come from ?
   - What is the structure of ____ ?

   They are all answerable with the `:info` command. For example, suppose you have the question, "What does the `swiggletwoop` function mean?" You may answer this at GHCi with:

   ```
   > :info swiggletwoop
   ```

   You may also use `:i` for short.

2. Functional Programming techniques rely heavily on types. This reliance may feel foreign at first, however, it is an important part of this course. If you wish to know the type of an expression or value, use `:type`. For example,

   ```
   > :type reverse
   ```

   ```
   [t] -> [t]
   ```

   This tells you that the `reverse` function takes a list of elements of some arbitrary type (`t`) and returns a list of elements of that same type. Try it.

   You may also use `:t` for short.

3. GHCi has TAB-completion. For example you might type the following:

   ```
   > :type rev
   ```

   Now hit the TAB key. If there is only one function in scope that begins with the characters `rev`, then that name will auto-complete. Try it. This completion is context-sensitive. For example, it doesn't make sense to ask for the type of a data type itself, so data type names will not auto-

complete in that context, however, if you ask for `:info`, then they are included in that context. Be aware of this when you use auto-complete.

This also works for file names:

```
> readFile "/etc/pas"
```

Now hit the TAB key. If there is only one existing filename on a path that begins with `/etc/pas`, then that name will auto-complete. Try it.

If there is more than one identifier that can complete, hit TAB twice quickly. This will present you with your options to complete.

4. Follow the types.

   You may find yourself in a position of being unsure how to proceed for a given exercise. You are encouraged to adopt a different perspective. Instead of asking how to proceed, ask how you might proceed while adhering to the guideline provided by the types for the exercise at hand.

   It is possible to follow the types without achieving the desired goal, however, this is reasonably unlikely at the start. As you become more reliant on following the types, you will develop more trust in the potential paths that they can take you, including identification of false paths.

   Your instructor must guide you where types fall short, but you should also take the first step. Do it.

Running the tests

Some exercises include examples and properties, which appear in a comment above the code for that exercise. Examples begin with `>>>` while properties begin with `prop>`.

The solution to the exercise must satisfy these tests. You can check if you have satisfied all tests with cabal-install and doctest. From the base directory of this source code:

```
> cabal update> cabal install doctest> cabal configure --enable-tests> cabal build> cabal test
```

Alternatively, you may run the tests in a single source file by using `doctest` explicitly. From the base directory of this source code:

```
> doctest -isrc -Wall -fno-warn-type-defaults <filename.hs>
```

Note: There is a bug in GHC 7.4.1 where for some configurations, running the tests will cause an unjustified compiler error.

Progression

It is recommended to perform some exercises before others. The first step is to inspect the introduction modules.

- `Course.Id`
- `Course.Optional`
- `Course.Validation`

They contain examples of data structures and Haskell syntax. The next step is to complete the exercises in `Course.List`.

After this, the following progression of modules is recommended:

- `Course.Functor`
- `Course.Apply`
- `Course.Applicative`
- `Course.Bind`
- `Course.Monad`
- `Course.State`
- `Course.StateT`
- `Course.Extend`
- `Course.Comonad`
- `Course.ListZipper`
- `Course.Parser`
- `Course.MoreParser`
- `Course.JsonParser`
- `Course.Interactive`
- `Course.Lens`
- `Course.Anagrams`
- `Course.FastAnagrams`
- `Course.EditDistance`
- `Course.BKTree`

- `Course.Cheque`

After these are completed, complete the exercises in the `projects` directory.

References

- The Haskell `error` function
- Glasgow Haskell Compiler