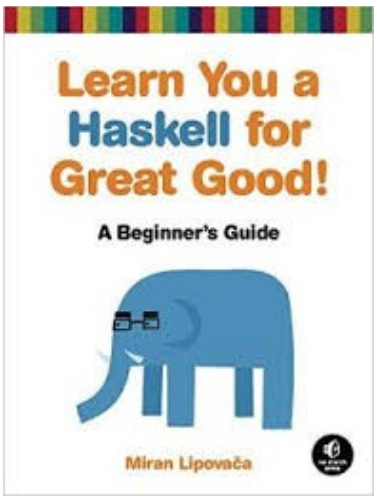
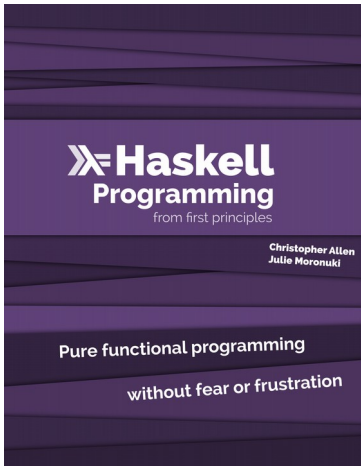
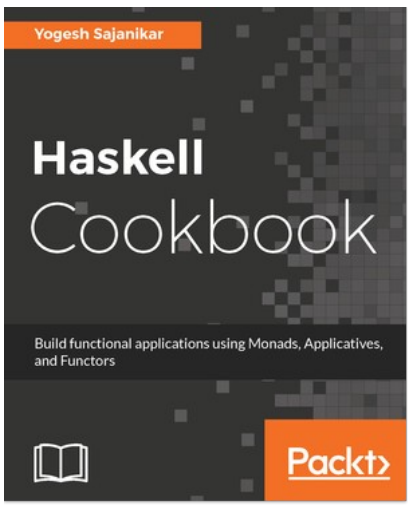
Monday Morning Haskell
Beginners Checklist

Essential Tools and Knowledge for Starting Your Haskell Journey







Basics

Haskell Platform

- If you've read part 1 of our Liftoff series, you should already be set with this! But if you haven't, the simplest way to get Haskell installed on your machine is to go to haskell.org/platform and follow the directions for your operating system. This will install GHC, Cabal, and Stack!
- GHC is the Glasgow Haskell Compiler, the most common compiler used for the Haskell programming language.
- Cabal is the build system that allows you to organize your project so that it will automatically download the libraries you need.
- The Stack tool adds a further layer on top of Cabal. It further refines the package organization and download process by using groups of libraries that are known to work well together and not have conflicts. It also makes sandboxed builds the default options, so it is easier to have multiple Haskell projects on the same computer. You can learn more about Stack by enrolling in our free Stack Mini-Course!

Hackage

- Hackage is the main repository for open source Haskell libraries. It also is the source of any documentation for those libraries. This is always the best place to look first if you need to figure out what the type of a function is, and if there's any other context on that function you need to know about.
- A word of warning! If you use Google to search for a particular module, you'll probably get Hackage links as your first hits. But Google doesn't know what version of a package you're using in your project! So it might link you to out-of-date documentation. To figure out if you've got the right version, first run the command **`stack pkg-list`** from your project and figure out what version you're using. Then, on Hackage, click the "Contents" link in the top right, and you'll be able to see what version you're using in bold in the middle of the page (and click on a past version if necessary).

Hayoo

- Hayoo is a search engine for Haskell libraries. It is an excellent resource for finding documentation when you come across a function in someone's code and you don't know where it comes from. It is particularly helpful for finding operator functions that are difficult to Google.

Tools

HLint

- In Haskell, there are often different ways of writing the same expression. Some ways definitely lead to more cluttered syntax. But you won't always be sure about the cleanest way to write your code that will still work. HLint is a tool that can teach you a lot about best practices with Haskell's syntax. It will tell you when you have redundant parentheses, or redundant uses of the `$` operator. It can even show you when there's a library helper function that you could be using!
- If you have Stack installed, you can get HLint very easily by using **`stack install hlint`**. You can either do this at a project specific level by running it inside the directory, or at a global level by running from your home directory. Then you can run **`hlint .`** to run it recursively over your current directory. Otherwise you can use it on a specific directory.

HPack

- When Cabal builds your Haskell project, it uses a file (typically called {yourproject}.cabal) to determine what libraries it needs to download, and which pieces of your project depend on which libraries.
- It also should list all the different modules of your project. And when you add a new library dependency to a module, you'll have to add that as a dependency in the cabal file, often in a couple different places!
- The HPack library solves a lot of the problems with cabal files. Instead of editing the cabal file directly, you'll edit a file called package.yaml, with a simpler language in a format that is easier to understand.
- You can install HPack the same way you install HLint, by running **`stack install hpack`**. Then in your project directory you can run **`hpack`** to convert your cabal file to the `package.yaml` file.
-

Text Editor Plugins

- No matter what language you're programming in, it helps a TON to have a good development environment. The exact set of features you'll get depends on your choice of text editor obviously. But Haskell-specific tools can often give you good bonuses like being able to examine the types of expressions in your code or get auto-completion for library expressions.
- For editors like Vim, Emacs, or Atom, you'll want to look for plugins named "Haskell Mode". In general, you'll need to make sure the **`hlint`** and **`ghc-mod`** commands are installed on your system. Furthermore, you'll need to make sure the version of **`ghc-mod`** corresponds to the version of GHC for your project.
- If you use Visual Studio, then you'll want to take a look at either "Haskero" or "Haskelly" to enhance your IDE for use with Haskell.
- Vim Haskell Mode Tutorial
- Emacs Haskell Mode
- Atom Haskell Mode

Books

Learn You a Haskell for Great Good

- *Learn You a Haskell* is a great language introduction you can get for free online! It teaches a lot of core concepts of the language in a fun and engaging way. It'll take you all the way through learning some of the most common monads and solving problems using a more functional approach.

The Haskell Book

- The Haskell Book (*Haskell Programming from First Principles*) is a more thorough introduction to Haskell. It covers a lot of the same material as LYAH, but it goes into more depth and also provides exercises, which are a vital part of trying things hands on and improving your skills.

Haskell Cookbook

- The Haskell Cookbook is generally a more advanced book, but it does start from first principles and work up from there. If you want to go all the way from beginner to writing fully fledged programs, this book can take you there.

Blogs

FP Complete

- FP Complete is a great site with a lot of cool material. Their blog tends to focus on Dev Ops, but they also have a lot of different resources for Haskell beginners as well!

School of Haskell

- School of Haskell has an awesome array of tutorials covering a wide array of different skill sets. They have a few different beginner tutorials but they go all the way through some pretty advanced techniques like web programming.

Language Basics Review

Here's a quick review of the language basics we went over in the Liftoff series.

Fundamentals

- All the code you write consists of expressions.
- Every expression has a type.
- Functions are special expressions that take inputs and give an output.
- You can import code from Haskell modules into GHCI. You can also run a Haskell module with the “runghc” command if it has a `main :: IO ()` function. For projects, you should use Stack or Cabal.

Syntax

- If Statements are expressions like anything else. They **must** have an else branch. Both branches must be expressions with the same type. The whole “If” statement then has this type.

```
myIfStatement :: Int -> Int
myIfStatement a = if a <= 2
  then a + 2
  else a - 2
```

- When there are many different conditions, you can use guards instead of nested if statements.

```
myGuardStatement :: Int -> Int
myGuardStatement a
  | a <= 2 = a + 2
  | a <= 6 = a
  | otherwise = a - 2
```

- You can also use pattern matching to run different code based on the structure of an input argument.

```
myFunction :: [Int] -> Int
myFunction [a] = a + 3
myFunction [a,b] = a + b + 1
myFunction (1 : 2 : _) = 3
myFunction (3 : 4 : _) = 7
myFunction xs = length xs
```

- Case statements allow you to use pattern matching within a function.

```
myFunction :: Bool -> [Int] -> Int
myFunction usePattern xs = if not usePattern
  then length xs
  else case xs of
    [a] -> a + 3
    [a,b] -> a + b + 1
    (1 : 2 : _) -> 3
    (3 : 4 : _) -> 7
    _ -> 1
```

- Intermediate values can be declared within a “where” clause or a “let” statement.

```
mathFunction :: Int -> Int -> Int -> Int
mathFunction a b c = diff1 + diff2 + prod + a
  where
    diff1 = c - a
    diff2 = b - a
    prod = a * b * c

-- Using let
mathFunction :: Int -> Int -> Int -> Int
mathFunction a b c =
  let diff1 = c - a
      diff2 = b - a
      prod = a * b * c
   in diff1 + diff2 + prod + a
```

Data Types

- You can declare a data type with the “data” keyword, followed by a capitalized type name.

```
data Task = …
```

- Each data type has 1 or more constructors. These have capitalized names and are then followed by a list of 0 or more types.

```
data Task = BasicTask String Int
```

- Different constructors for the same type can store different kinds of data!

```
data Task =
  BasicTask String Int |
  ComplexTask String Int Location
```

- Types can be parameterized. You can leave a variable type to be filled in by the user and then use that type within your constructors.

```
data Task a =
  BasicTask String a |
  ComplexTask String a Location
```

- You can name the different fields of a constructor using record syntax.

```
data Task = BasicTask
  { taskName :: String
  , taskLength :: Int
  }
```

- The type keyword allows you to create a type synonym. This does not have compile time implications and you can use the original name and the synonym interchangeably.

```
type TaskTuple = (Int, String, Task)
```

- The newtype keyword allows you to wrap a type in a way that causes compile time implications. You **cannot** use the types interchangeably. However at runtime, the types will be the same.

```
newtype TaskLength = TaskLength Int
```