LEARNING HASKELL WITH NETWORK PROGRAMMING

CONTENTS

FOREWORD

Haskell, as a language, has a bit of a loaded reputation.

A lot of programmers assume that Haskell is that “mathey language”, and once they hear mention of functors, monoids, monads, and arrows, they end up with the conclusion that it is a a tool that might have some interesting ideas, but is too academic to be suitable for any kind of practical use.

This is an easy viewpoint to have. There is a steep, non-trivial learning curve to the minutia of functional programming, and Haskell is among the most dogmatic of the functional languages; to understand Haskell is to understand the basics of algebra, Category Theory, Type Theory, and Set Theory. Consequently, a lot of people assume that people learn Haskell so that they can look smart.

While this is certainly true (and is why I picked up the language initially), functional programming has started to become fairly popular in the last couple years, and many people have started to appreciate the benefits of concepts like immutability, a clear type system, and the mathematical safety that programming in a functional style provides. The rise in popularity in languages like Scala, Rust, and F#, as well as the newer functional features being added to Java 8, C++, and C# has shown us that there is a clear market for functional programming in the field of computer science.

Learning Haskell is about more than just the language. Haskell has become the poster-child for the testing ground of the cutting edge features of computer science.

New methods of parallelism, thread safety, parsing, and nearly every other new programming theory comes over to Haskell first.

But what you get with Haskell isn't just access to functional features. Indeed, if you wanted an excuse to play with functional programming, the barrier of entry to do so is much lower with JavaScript.

What you get from Haskell is a purely devoted, unambiguous entry to a functional way of structuring your programs, with all the benefits and annoyances that come with that. You get all the frustration of having to avoid state, but also all the benefits of guarantees that your functions will work exactly the same after being run a trillion times. You get all the frustration of having to use recursion instead of loops, but all the benefit of knowing that you'll never be forced to deal with a `null` reference error.

What you get with Haskell is something you do not get with many other languages: guarantees. Mathematics is one of the very few things in life that boasts in its proofs and correctness and by taking a purely mathematical approach allows you to reap all the benefits of all that scary topics that I mentioned before. And with those guarantees, you find that Haskell is a language *especially* well-suited for production systems that need to *keep working.*

Upon learning Haskell, it's not uncommon for a person to re-evaluate and change their entire coding process to use a more functional style, or to start utilizing the functional features of their favorite language.

Being forced to view things from a different, strange angle can be a godsend in how you approach problems; sometimes approaching certain problems from a functional perspective can greatly simplify the problem. Recursion is a useful, often-underutilized tool that becomes easier to understand once confronted with a world without loops. Optional types give compile-time checks to that completely infamous `null` reference exception. Higher-order functions give a level or modularity that is nearly impossible to replicate with traditional imperative coding styles.

There are plenty of reasons to learn functional style, and many of them will be described in this book, but if you are to take anything out of this book, it should be that there is value in the extravagant and theoretical. I hope you enjoy it, and find that something like Haskell can open the door to the other interesting new paradigms in this whacky field we call "Computer Science".

CHAPTER 1

GETTING SHIT INSTALLED

1.1 Introduction

The shit that's always the hardest fucking part of getting into a langauge is getting the compiler and tooling set up. Well, no, that's a fucking lie, there are lots of parts that are probably harder, but by the time you get to them you're probably already committed to the language so there's no use in complaining.

1.1.1 GHC

The most important thing you'll need to do Haskell development is The Glorious Haskell Compiler, or GHC for short. There are other Haskell compilers out there, like YHC or UHC, but I don't know anyone that uses them, and I can't be bothered to try them out myself. When people say "Haskell", they usually seem to mean "GHC"

1.1.2 Bullshit about GHC

While GHC is what you need to to compile your Haskell program, it can be pretty frustrating between different point versions, and especially irritating between different platforms. For the love of God, do *not* just install GHC from the command line in your favorite operating system's package manager, since this will almost certainly be wrong, and confuse you.

There's also the fact that GHC is *only* the compiler. Handling dependencies, managing the project files, and handling information about the project itself.

To make sure it's done correctly, you'll need to install either the Haskell Platform (the wrong way), or Stack (the much better way).

1.2 The Shitty Old Way

Back in the old days, back in my youth, the main way to get Haskell installed on your machine was to download and install the Haskell Platform.

The Haskell Platform includes GHC, Cabal, and a thirty-two of the most commonly used packages with GHC, and was the de-facto method of getting all of Haskell on your computer.

1.2.1 What the fuck is Cabal?

As mentioned before, GHC is *just* a compiler. I'm not saying that it's a bad thing, it's just limited in scope. Cabal was the first real attempt to for a utility to manage Haskell projects.

Cabal will give you a structure to your project, allowing you a means of creating your project without having to define any kind of makefiles or build scripts. It will also handle library dependencies, including basic transitive dependency support (though it can get confused if your transitive dependencies have conflicting versions).

1.2.1.1 Initializing the folder and sandbox To make a new project with Cabal, you should need to make a new folder, and initialize it as a Cabal project. If you don't want projects to fuck each other up, you should utilize the Cabal Sandbox feature.

```
mkdir my_new_project
cd my_new_project
cabal sandbox init
```

1.2.1.2 Sandboxing Sanboxing, though not strictly required, makes your life a lot simpler. By default, Cabal will install libraries into the global space. This is okay if you only have one project, but if you have multiple Haskell project, it's very easy to get dependency conflicts.

If you plan on having multiple Haskell projects, it's effectively a requirement to use a Sandbox, so as to minimize a bunch of dependency collisions, known by assholes like me as "Cabal Hell".

1.2.1.3 Initializing the Project Once you've initialized the project, you will need to run `cabal init` and go through the optoins.

```
cabal init
```

But there's not much point to doing that shit anymore since they finally solved all these problems and invented the Stack application. At this point you really can just say "Fuck it, just install Stack".

1.3 What the hell is Stack?

Stack is a Haskell program by FPComplete to make it easier to make other Haskell programs.

You'll actually find that to be a bit of a pattern in the Haskell universe. It feels like half of all the programs for it are just there to make you feel cool because you're using a Haskell program, or to try to talk you into making more Haskell programs

Anyway, what exactly does Stack do? In a nutshell: it manages your Haskell your entire Haskell project.

Using Stack, you can have your dependencies, structure, compilation, and even your compiler. In the process, you get something that man had previously only dreamed of: reproducible builds.

This is new shit, since it used to be really fucking difficult to get different versions of GHC installed on the same machine, especially if you decided it would be a good idea to use a Linux package manager.

1.4 Installing Stack

Ok, enough rambling about this Stack shit, I'm getting tired of talking about it anyway.

To go about installing it on a sane Mac or Linux platform, you should be able to do something to the effect of

```
curl -sSL https://get.haskellstack.org/ | sh
```

If you're on a shitty operating system like Windows, you don't have access to tools that were invented in the last forty years like "pipe". I guess it's just too advanced for Seattle.

What you'll need to do is just go to the haskellstack.org website, and download the Windows installer.

By the time you read it, this has probably changed, so you'll have to look up the updated URL anyway. I don't really know why I decided to use a printed medium to write a book about a fucking functional language that's hellbent on constantly updating. It's just going to get outdated.

1.5 Using Stack

So now you should have stack installed, and all its glory is available to you. Unless you installed it wrong. Don't exclude that possibility, it will be your downfall.