An Introduction To Haskell

By: Ryan Abel

Table of Contents

Overview 3

Features 4

Monads 4

Types And Type Classes 5

Syntax 6

Pattern Matching 7

Guards 7

Modules 8

Real World Applications 8

Conclusion 9

References 10

Appendix 11

# Overview

Programming languages come in all shapes and forms. They follow different paradigms, appeal to different crowds, and each have their own benefits and drawbacks. Recently, developers have begun to see the benefits of functional programming. One language in particular, Haskell, has gotten attention from a lot of developers.

Haskell is named after logician Haskell Curry. Curry is known as one of the pioneers of combinatory logic. Haskell was first released in 1990 and was inspired by the release of Miranda in 1985. At the time, functional languages were just beginning to gain popularity. The increase in interest resulted in a plethora of new functional languages. At a functional conference in 1987 the powers that be decided the functional community needed a common language. There was a strong consensus at this meeting that more widespread use of this class of functional languages was being hampered by the lack of a common language. It was decided that a committee should be formed to design such a language. The result of this committee was Haskell.

Since its inception, Haskell has versioned several times to add additional features and refactor previous ones. Haskell is considered a purely functional language. Therefore any function call has no side effects and all data structures are immutable. A function call is guaranteed to return the same result every time. Haskell implements lazy evaluation. Lazy evaluation states that functions are only evaluated when their return value is needed by another function. This also helps to avoid repeating the same evaluation multiple times. Haskell is a strong statically typed language. All variables must use a type and type checking is enforced at compile time (similar to Java).

# Features

## Monads

The most unique feature of Haskell is monads. Monads can be tough to comprehend and fully understand. Monads are an abstract structure that allows programmers to develop at a very general level. A monad is a wrapper type around another type. The wrapper adds a structure to the original type. This lets a programmer combine function calls of an inner type to create more sophisticated functions without necessarily worrying about the original type. An example monad type is Haskell is Maybe. The maybe data type looks like this:

|  |
| --- |
| data [Maybe](http://www.haskell.org/ghc/docs/6.12.2/html/libraries/base-4.2.0.1/Data-Maybe.html#t%3AMaybe) a |
| |  | | --- | | = [Nothing](http://www.haskell.org/ghc/docs/6.12.2/html/libraries/base-4.2.0.1/Data-Maybe.html#v%3ANothing) | | | [Just](http://www.haskell.org/ghc/docs/6.12.2/html/libraries/base-4.2.0.1/Data-Maybe.html#v%3AJust) a | |

The maybe data type adds a structure to another data type (“a”) that allows the inner type to be nothing or the result of Just a. It can be hard to see how this may be useful. Take an example such as finding the square root of a number. The function has an obvious result for a value like 16. However, what if the value is a negative integer? Typical paradigms may lead a developer to throw an exception. Instead Haskell would define the return type as Maybe Integer. If the return type is an integer, the function will return it. Otherwise it will return the data type Nothing. The finished function might look like this:

isqrt :: Integer -> Maybe Integer

isqrt x = isqrt' x (0,0)

where

isqrt' x (s,r)

| s > x = Nothing

| s == x = Just r

| otherwise = isqrt' x (s + 2\*r + 1, r+1)

## Types And Type Classes

## 

Haskell is a strongly typed static language. All variables must have declared types and type checking is completed at compile time. In most languages, data types can be restricting and prevent a programmer from seeing problems in a general way. However with monads, Haskell helps solve this problem. The use of monads frees the developer from the constraints of types and lets them focus on solving the problem in a more general manner. Haskell also has type classes. Every type fits into a type class. Type classes help define a types behavior. In other words, a function that returns the result of a comparison must take two parameters that are of the Eq (Equal) type class.

## Syntax

Although Haskell is a functional language, it looks very little like many functional languages. Languages such as Common Lisp and Clojure are memorable because of the barrage of parenthesis. Haskell looks more like CoffeeScript or Ruby in its syntax. Parentheses on function calls are optional as they are in Ruby. Expressions (+, -, etc.) are infix notation. One commonality Haskell shares with LISP languages is the use of lists. Lists are the main data structure in the language. Lists operations are easy to use. For example, combining two lists is as simple as:

1. ghci> [1,2,3,4] ++ [9,10,11,12]
2. [1,2,3,4,9,10,11,12]

And adding to the beginning of a list is as easy as:

1. ghci> 5:[1,2,3,4,5]
2. [5,1,2,3,4,5]

Lists can also be created conditionally. For example, if a developer wanted a list that was all the odd values from one to ten it may look like this:

1. ghci> 5:[x | x <- [1..10], x ‘mod’ 2 == 0]
2. [1,3,5,7,9]

This creates a list of values ‘x’ where x is a value between one and ten and the result of x mod two must be zero.

## Pattern Matching

Functions in Haskell use pattern matching. Pattern matching allows the developer to execute a certain part of the body based on the parameter passed in. When the function is called, the pattern is evaluated from top to bottom until a match is found. It is important to make sure there is some sort of “catch all” in a pattern. If a function is called with a parameter that does not match, the pattern an exception will be thrown.

## Guards

Haskell also has a feature called guards. Guards are the same as an if, else if, else statement except that the syntax looks much cleaner. A guard will execute the first statement that evaluates to true in a series of guards. Here is an example:

1. bmiTell :: (RealFloat a) => a -> a -> String
2. bmiTell weight height
3. | weight / height ^ 2 <= 18.5 = "You're underweight, you emo, you!"
4. | weight / height ^ 2 <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
5. | weight / height ^ 2 <= 30.0 = "You're fat! Lose some weight, fatty!"
6. | otherwise                 = "You're a whale, congratulations!"

Haskell allows easy function passing via partially applied functions. A function can be called without all of its parameters. At this point it becomes a partially applied function, in other words it becomes a new function. That function can then be passed to other functions as a parameter. It is best illustrated with an example.

1. multThree :: (Num a) => a -> a -> a -> a
2. multThree x y z = x \* y \* z

Calling multThree 2 4 8 is exactly the same as calling ((multThree 2) 4) 8. The second way creates a partially applied function with a two, then four is applied to that function, creating yet another function. Finally, eight is applied and the function is evaluated.

## Modules

Haskell does not define classes, but it does have modules. Modules are a set of similar functions. They are used in other code by Import statements. Typically in Haskell a program is defined as a bunch of module. The main module loads up all other necessary modules and uses the functions defined in the other modules. This helps guide programmers to creating writing clean, separated code.

# Real World Applications

Haskell has many uses in real world development. One good candidate for a Haskell program is one that is computationally heavy. Haskell is great at parallelization, therefore lending itself to this type of environment. Parallelization also makes Haskell well suited for environments that need to handle huge amounts of data. Because Haskell is a general-purpose programing language, it can be used in a wide range of projects. AT&T uses Haskell to automate the processing of Internet abuse and complaints. Facebook and Google both use Haskell on internal projects. Merril Lynch uses Haskell for backend data transformation and loading. These projects alone show how versatile Haskell is.

# Conclusion

In conclusion, Haskell is a very interesting language to try to learn and understand. It has many caveats, but after learning them it is apparent how powerful it can be. It is easy to see why Haskell’s popularity has continued to grow steadily and why so many programmers are highly recommending learning it.

# 

# References

<http://research.microsoft.com/en-us/um/people/simonpj/papers/history-of-haskell/history.pdf>

<http://www.haskell.org/haskellwiki/Haskell_in_industry>

<http://www.haskell.org/haskellwiki/Applications>

<http://ertes.de/articles/monads.html#section-4>

Many code snippets provided by: <http://learnyouahaskell.com/chapters>

# Appendix