The **Haskell platform** is the single, easily installable package for setting up your Haskell development environment.

The Haskell Platform includes the following:

* The Glasgow Haskell Compiler (GHC)
* An interactive interpreter (GHCi)
* The stack tool for managing Haskell projects
* A bunch of useful Haskell packages

The Haskell Platform can be downloaded from [www.haskell.org/downloads#platform](www.haskell.org/downloads%23platform).

You can use any text editor or IDE, the recommendation is that you install a Haskell plugin/extension for that editor.

The Glasgow Haskell Compiler is the main compiler for Haskell. The job of the compiler is to transform human-readable source code into machine-readable binary. The main benefit of a compiler over an interpreter is that because the compiler transforms code in advance, it can perform analysis and optimization of the code you’ve written.

Haskell is strongly typed; this ensures that a program will throw errors when there is a wrong expression during compilation. There’s an adage that if it compiles, it works.

The interactive mode of GHC is GHCi. We use the command ghci to start it and :q to exit on terminal.

**Note**: Prior to version 8 of GHCi, function and variable definitions needed to be prefaced with a let keyword. This is no longer necessary, but many Haskell examples on the web and in older books still include it:

GHCi> let f x = x + x

GHCi> f

2

GHCi is mainly used to test and interact with programs that you are writing.

Loading file to GHCi

1. Pass filename as an argument to ghci:

$ ghci hello.hs

1. Use :load or :l command in the interactive session.

After loading you can call any expressions you have defined in your file.

**Note**: Unlike compiling files in GHC, your files don’t need a main in order to be loaded into GHCi.

**Writing and working with Haskell code**

In Haskell it is impossible to debug your code by printing values.

It is also strict about the correctness of your code, wrong code won’t compile.

Haskell strongly rewards taking time and thinking through problems before running programs.

The trick to writing Haskell code with minimal frustration is to write code in little bits, and play with each bit interactively as it’s written.

**Functional Programming**

Most programming paradigms use the Von-Nueman style of compution.

Functional programming attempts to liberate programming from the von Neumann style.

Functional programming focuses on computation and not computers, you solve problems simply by describing them.

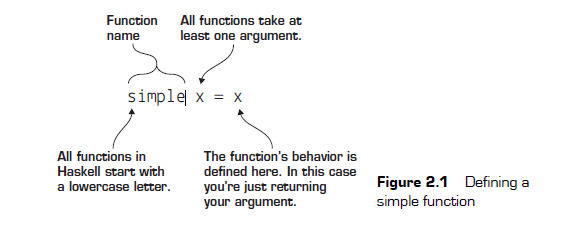
By focusing on computation, not computers, functional programming allows the programmer access to powerful abstractions that can make many challenging problems much easier to solve.

The price of this is that getting started can be much more difficult because it is very different and abstract.

Just as C is the nearly perfect embodiment of the von Neumann style of programming, Haskell is the purest functional programming language you can learn.

**Functions**

The behavior of functions in Haskell comes directly from mathematics. In math, we often say things like f(x) = y, meaning there’s some function f that takes a parameter x and maps to a value y. In mathematics, every x can map to one and only one y.



The simple function takes a single argument x and then returns this argument untouched. Notice that unlike many other programming languages, in Haskell you don’t need to specify that you’re returning a value. In Haskell, functions must return a value, so there’s never a need to make this explicit.

All functions in Haskell follow three rules that force them to behave like functions in math:

* All functions must take an argument.
* All functions must return a value.
* Anytime a function is called with the same argument, it must return the same value: *referential transparency*.

Lambda calculus by Alonzo Church is where you represent everything as functions. Church’s goal for creating this system of logic was to resolve some problems in the mathematical field of set theory. Unfortunately, lambda calculus didn’t solve these problems, but something more interesting came out of Church’s work. It turns out that lambda calculus allows for a universal model of computation, equivalent to a Turing machine!

This discovery of the relationship between lambda calculus and computing is called the Church-Turing thesis. The wonderful thing about this discovery is that you have a mathematically sound model for programming!

In Haskell you can mathematically prove that programs will do exactly what you expect and have the limitless abstractions that mathematics allows because you are not limited by engineering decisions in the language.

The aim of functional programming is to bring the power of mathematics to the programmer in a usable way.

Value of functional programming.

It is safe: because functions return the same value for the same input every time it is evaluated. Programs are safe when they always behave exactly the way you expect them to and you can easily reason about their behavior. A safe programming language is one that forces your programs to behave as expected. **Pure functions**

Unsafe languages use impure functions these are functions with side effects. Sides effects are created by changing a program’s state. When you change a value in your programming environment, you’re changing the program's state.

Haskell doesn’t allow functions to have side effects, which explains why all Haskell functions must take an argument and return a value. If Haskell functions didn’t always return a value, they’d have to change a hidden state in the program; otherwise, they’d be useless. If they didn’t take an argument, they’d have to access a hidden one, which would mean they’re no longer transparent.

This small property of Haskell’s functions leads to code that’s dramatically easier to predict.

**Variables**

There are no variables in Haskell only definitions.

Definitions are constants once a value is set you or the program cannot change it.

In imperative languages variable order matters because you can always reassign the value of something after you’ve assigned it. In Haskell, because of referential transparency, this isn’t an issue.

However, In GHCi you are allowed to reassign variables. This is a special case so that you don’t have to always restart ghci every time you want to experiment with a different variable.

Lambda functions

These are anonymous functions usually denoted by \ which represents the Greek lamda (λ).

Each time you use your lambda expression, you have to redefine it. This makes sense, because you have no name to call it by! Lambda functions are useful but are designed to exist for only a short while. In general, if a named function will do the job, it’s better to use one.

GHCi> (\x -> x) 4

4

GHCi> (\x -> x) [1,2,3]

[1,2,3]

Although the lambda function is messier than the original where, it’s also more powerful! The where statement makes everything much easier to understand, but it’s also syntactically wrapped up in your function. There’s no way to just pull out a where section. This clearly isn’t the case with your lambda expression. You pasted it into place and could just as easily pull it out. Your lambda function is an expression, a self-contained chunk of code, all on its own.

Haskell has an alternative to where clauses called **let** expressions. A let expression allows you to combine the readability of a where clause with the power of your lambda function.

Figure 3.3 shows the sumSquareOrSquareSum function using let.