Haskell was designed as a practical, purely functional programming language. It's famous for its monads and its type system, but I keep coming back to it because of its elegance. Haskell makes coding a real joy for me.

```
-- Single line comments start with two dashes.
{- Multiline comments can be enclosed
in a block like this.
-}

------------------------------------------------------
-- 1. Primitive Datatypes and Operators
------------------------------------------------------

-- You have numbers
3 -- 3

-- Math is what you would expect
1 + 1 -- 2
8 - 1 -- 7
10 * 2 -- 20
35 / 5 -- 7.0

-- Division is not integer division by default
35 / 4 -- 8.75

-- integer division
35 `div` 4 -- 8

-- Boolean values are primitives
True
False

-- Boolean operations
not True -- False
not False -- True
1 == 1 -- True
1 /= 1 -- False
1 < 10 -- True

-- In the above examples, `not` is a function that takes one value.
-- Haskell doesn't need parentheses for function calls...all the arguments
-- are just listed after the function. So the general pattern is:
-- func arg1 arg2 arg3...
-- See the section on functions for information on how to write your own.

-- Strings and characters
"This is a string."
'a' -- character
'You cant use single quotes for strings.' -- error!

-- Strings can be concatenated
"Hello " ++ "world!" -- "Hello world!"

-- A string is a list of characters
```

```haskell
['H', 'e', 'l', 'l', 'o'] -- "Hello"
"This is a string" !! 0 -- 'T'


-----------------------------------------------------
-- Lists and Tuples
-----------------------------------------------------

-- Every element in a list must have the same type.
-- These two lists are the same:
[1, 2, 3, 4, 5]
[1..5]

-- Ranges are versatile.
['A'..'F'] -- "ABCDEF"

-- You can create a step in a range.
[0,2..10] -- [0, 2, 4, 6, 8, 10]
[5..1] -- This doesn't work because Haskell defaults to incrementing.
[5,4..1] -- [5, 4, 3, 2, 1]

-- indexing into a list
[1..10] !! 3 -- 4

-- You can also have infinite lists in Haskell!
[1..] -- a list of all the natural numbers

-- Infinite lists work because Haskell has "lazy evaluation". This means
-- that Haskell only evaluates things when it needs to. So you can ask for
-- the 1000th element of your list and Haskell will give it to you:

[1..] !! 999 -- 1000

-- And now Haskell has evaluated elements 1 - 1000 of this list...but the
-- rest of the elements of this "infinite" list don't exist yet! Haskell won't
-- actually evaluate them until it needs to.

-- joining two lists
[1..5] ++ [6..10]

-- adding to the head of a list
0:[1..5] -- [0, 1, 2, 3, 4, 5]

-- more list operations
head [1..5] -- 1
tail [1..5] -- [2, 3, 4, 5]
init [1..5] -- [1, 2, 3, 4]
last [1..5] -- 5

-- list comprehensions
[x*2 | x <- [1..5]] -- [2, 4, 6, 8, 10]

-- with a conditional
[x*2 | x <- [1..5], x*2 > 4] -- [6, 8, 10]
```

```haskell
-- Every element in a tuple can be a different type, but a tuple has a
-- fixed length.
-- A tuple:
("haskell", 1)

-- accessing elements of a pair (i.e. a tuple of length 2)
fst ("haskell", 1) -- "haskell"
snd ("haskell", 1) -- 1

----------------------------------------------------
-- 3. Functions
----------------------------------------------------
-- A simple function that takes two variables
add a b = a + b

-- Note that if you are using ghci (the Haskell interpreter)
-- You'll need to use `let`, i.e.
-- let add a b = a + b

-- Using the function
add 1 2 -- 3

-- You can also put the function name between the two arguments
-- with backticks:
1 `add` 2 -- 3

-- You can also define functions that have no letters! This lets
-- you define your own operators! Here's an operator that does
-- integer division
(//) a b = a `div` b
35 // 4 -- 8

-- Guards: an easy way to do branching in functions
fib x
  | x < 2 = 1
  | otherwise = fib (x - 1) + fib (x - 2)

-- Pattern matching is similar. Here we have given three different
-- definitions for fib. Haskell will automatically call the first
-- function that matches the pattern of the value.
fib 1 = 1
fib 2 = 2
fib x = fib (x - 1) + fib (x - 2)

-- Pattern matching on tuples:
foo (x, y) = (x + 1, y + 2)

-- Pattern matching on lists. Here `x` is the first element
-- in the list, and `xs` is the rest of the list. We can write
-- our own map function:
myMap func [] = []
myMap func (x:xs) = func x:(myMap func xs)
```

```haskell
-- Anonymous functions are created with a backslash followed by
-- all the arguments.
myMap (\x -> x + 2) [1..5] -- [3, 4, 5, 6, 7]

-- using fold (called `inject` in some languages) with an anonymous
-- function. foldl1 means fold left, and use the first value in the
-- list as the initial value for the accumulator.
foldl1 (\acc x -> acc + x) [1..5] -- 15

----------------------------------------------------
-- 4. More functions
----------------------------------------------------

-- partial application: if you don't pass in all the arguments to a function,
-- it gets "partially applied". That means it returns a function that takes the
-- rest of the arguments.

add a b = a + b
foo = add 10 -- foo is now a function that takes a number and adds 10 to it
foo 5 -- 15

-- Another way to write the same thing
foo = (+10)
foo 5 -- 15

-- function composition
-- the (.) function chains functions together.
-- For example, here foo is a function that takes a value. It adds 10 to it,
-- multiplies the result of that by 4, and then returns the final value.
foo = (*4) . (+10)

-- (5 + 10) * 4 = 60
foo 5 -- 60

-- fixing precedence
-- Haskell has another operator called `$`. This operator applies a function
-- to a given parameter. In contrast to standard function application, which
-- has highest possible priority of 10 and is left-associative, the `$` operator
-- has priority of 0 and is right-associative. Such a low priority means that
-- the expression on its right is applied as the parameter to the function on its left.

-- before
even (fib 7) -- false

-- equivalently
even $ fib 7 -- false

-- composing functions
even . fib $ 7 -- false


----------------------------------------------------
-- 5. Type signatures
----------------------------------------------------
```

```haskell
-- Haskell has a very strong type system, and everything has a type signature.

-- Some basic types:
5 :: Integer
"hello" :: String
True :: Bool

-- Functions have types too.
-- `not` takes a boolean and returns a boolean:
-- not :: Bool -> Bool

-- Here's a function that takes two arguments:
-- add :: Integer -> Integer -> Integer

-- When you define a value, it's good practice to write its type above it:
double :: Integer -> Integer
double x = x * 2

----------------------------------------------------
-- 6. Control Flow and If Expressions
----------------------------------------------------

-- if expressions
haskell = if 1 == 1 then "awesome" else "awful" -- haskell = "awesome"

-- if expressions can be on multiple lines too, indentation is important
haskell = if 1 == 1
            then "awesome"
            else "awful"

-- case expressions: Here's how you could parse command line arguments
case args of
  "help" -> printHelp
  "start" -> startProgram
  _ -> putStrLn "bad args"

-- Haskell doesn't have loops; it uses recursion instead.
-- map applies a function over every element in an array

map (*2) [1..5] -- [2, 4, 6, 8, 10]

-- you can make a for function using map
for array func = map func array

-- and then use it
for [0..5] $ \i -> show i

-- we could've written that like this too:
for [0..5] show

-- You can use foldl or foldr to reduce a list
-- foldl <fn> <initial value> <list>
foldl (\x y -> 2*x + y) 4 [1,2,3] -- 43
```

```haskell
-- This is the same as
(2 * (2 * (2 * 4 + 1) + 2) + 3)

-- foldl is left-handed, foldr is right-
foldr (\x y -> 2*x + y) 4 [1,2,3] -- 16

-- This is now the same as
(2 * 1 + (2 * 2 + (2 * 3 + 4)))

-----------------------------------------------------
-- 7. Data Types
-----------------------------------------------------

-- Here's how you make your own data type in Haskell

data Color = Red | Blue | Green

-- Now you can use it in a function:


say :: Color -> String
say Red = "You are Red!"
say Blue = "You are Blue!"
say Green =  "You are Green!"

-- Your data types can have parameters too:

data Maybe a = Nothing | Just a

-- These are all of type Maybe
Just "hello"    -- of type `Maybe String`
Just 1          -- of type `Maybe Int`
Nothing         -- of type `Maybe a` for any `a`

-----------------------------------------------------
-- 8. Haskell IO
-----------------------------------------------------

-- While IO can't be explained fully without explaining monads,
-- it is not hard to explain enough to get going.

-- When a Haskell program is executed, `main` is
-- called. It must return a value of type `IO ()`. For example:

main :: IO ()
main = putStrLn $ "Hello, sky! " ++ (say Blue)
-- putStrLn has type String -> IO ()

-- It is easiest to do IO if you can implement your program as
-- a function from String to String. The function
--    interact :: (String -> String) -> IO ()
-- inputs some text, runs a function on it, and prints out the
-- output.
```

```haskell
countLines :: String -> String
countLines = show . length . lines

main' = interact countLines

-- You can think of a value of type `IO ()` as representing a
-- sequence of actions for the computer to do, much like a
-- computer program written in an imperative language. We can use
-- the `do` notation to chain actions together. For example:

sayHello :: IO ()
sayHello = do
   putStrLn "What is your name?"
   name <- getLine -- this gets a line and gives it the name "name"
   putStrLn $ "Hello, " ++ name

-- Exercise: write your own version of `interact` that only reads
--           one line of input.

-- The code in `sayHello` will never be executed, however. The only
-- action that ever gets executed is the value of `main`.
-- To run `sayHello` comment out the above definition of `main`
-- and replace it with:
--   main = sayHello

-- Let's understand better how the function `getLine` we just
-- used works. Its type is:
--    getLine :: IO String
-- You can think of a value of type `IO a` as representing a
-- computer program that will generate a value of type `a`
-- when executed (in addition to anything else it does). We can
-- store and reuse this value using `<-`. We can also
-- make our own action of type `IO String`:

action :: IO String
action = do
   putStrLn "This is a line. Duh"
   input1 <- getLine
   input2 <- getLine
   -- The type of the `do` statement is that of its last line.
   -- `return` is not a keyword, but merely a function
   return (input1 ++ "\n" ++ input2) -- return :: String -> IO String

-- We can use this just like we used `getLine`:

main'' = do
    putStrLn "I will echo two lines!"
    result <- action
    putStrLn result
    putStrLn "This was all, folks!"

-- The type `IO` is an example of a "monad". The way Haskell uses a monad to
-- do IO allows it to be a purely functional language. Any function that
```

```haskell
-- interacts with the outside world (i.e. does IO) gets marked as `IO` in its
-- type signature. This lets us reason about what functions are "pure" (don't
-- interact with the outside world or modify state) and what functions aren't.

-- This is a powerful feature, because it's easy to run pure functions
-- concurrently; so, concurrency in Haskell is very easy.


--------------------------------------------------
-- 9. The Haskell REPL
--------------------------------------------------

-- Start the repl by typing `ghci`.
-- Now you can type in Haskell code. Any new values
-- need to be created with `let`:

let foo = 5

-- You can see the type of any value with `:t`:

>:t foo
foo :: Integer

-- You can also run any action of type `IO ()`

> sayHello
What is your name?
Friend!
Hello, Friend!
```

There's a lot more to Haskell, including typeclasses and monads. These are the big ideas that make Haskell such fun to code in. I'll leave you with one final Haskell example: an implementation of quicksort in Haskell:

```haskell
qsort [] = []
qsort (p:xs) = qsort lesser ++ [p] ++ qsort greater
    where lesser  = filter (< p) xs
          greater = filter (>= p) xs
```

There are two popular ways to install Haskell: The traditional Cabal-based installation, and the newer Stack-based process.

You can find a much gentler introduction from the excellent Learn you a Haskell or Real World Haskell.