These are notes for the class *Five Programming Languages in Ten Days*, which was taught at Canada/USA Mathcamp in 2012 by Nic Ford and Asilata Bapat. If you're not reading them as part of that class, you should know that they were written as a supplement to the lectures that were given in the class, and not as a replacement for them. Read at your own risk.

1 Scala

Scala is built on top of the Java Virtual Machine, so when you're writing Scala code, you'll have access to all the classes in Java's standard libraries. In particular, a lot of Java code is very easy to turn into Scala.

But Scala can do so much more than this. It has tools built in to easily facilitate *functional programming*, a style of programming in which variables are constants, functions do nothing but compute and return a value, and the entire execution chain consists of evaluating some functions on some inputs and seeing what happens. Scala will serve as our bridge between the imperative and functional styles, and once we move on to the languages that come after it, the familiar Java-like imperative scaffolding will start to fall away. Let's get started.

1.1 Values, Arithmetic, and Control Structures

You can open the Scala interactive interpreter by going to a terminal and typing `scala`. Here you can type expressions and have Scala evaluate them for you:

```
scala> 1+1
res0: Int = 2

scala> "hello"
res1: java.lang.String = hello

scala> 5+3+"1"
res2: java.lang.String = 81

scala> 5+(3+"1")
res3: java.lang.String = 531
```

You can also put some code in a file, say `test.scala`, and run it all at once by going to a terminal and typing `scala test.scala` (or whatever the file is called.) If you want to load code from an external file in the interactive interpreter, you can type (for example) `:load test.scala`.

There are two quick things to notice here: one is that strings are Java strings. This is something that happens in Scala sometimes; Java types will sort of creep in from the background in places where the designers of the language thought it would be appropriate.

The second thing to notice is that, while Scala cares a lot about types, it's willing to to change the types of objects in situations where it would make sense. So when you try to add an integer to a string, it's willing to turn the integer into a string in order to make that happen.

The central idea of functional programming is that functions shouldn't have side-effects — that is, the only thing a function should do when it's called should be to compute a value to return and return it. But Scala isn't a purely functional language; it's a hybrid. So there are functions that have side-effects:

```
scala> println("hey there")
hey there
```

Like in Java, there is an `if` in Scala, but it works a little differently. The expression "`if (condition)` *a* `else` *b*" is an expression, just like `1+3`, and it evaluates to *a* if the condition is true and *b* otherwise:

```
scala> 1 + (if (1 < 2) 1 else 2)
res0: Int = 2
```

Of course, the expressions can also have side-effects:

```
scala> if (1 == 1)
     | {
     |   println("in here!")
     |   println("for real!")
     | }
in here!
for real!
```

Note that, like in Java or other C-like languages, you can put multiple instructions into a code block using curly braces.

You can declare variables using either `val` or `var`. The difference is that variables declared using `val` are *immutable*, that is, they can never be changed once they're assigned:

```
scala> var a = 1
a: Int = 1

scala> val b = 2
b: Int = 2

scala> a+b
res0: Int = 3

scala> a = 4
a: Int = 4

scala> a+b
res2: Int = 6

scala> b = 8
<console>:5: error: reassignment to val
       b = 8
```

```
            ^
```

Because Scala is a hybrid language that allows you to easily use functional and imperative styles, they make it easy to create both mutable and immutable variables. When we graduate to languages that are more fully committed to the functional paradigm, we'll find it correspondingly more difficult to make variables mutable.

Scala also has standard C-like loops, like `while` and `for`:

```
scala> var i = 1
i: Int = 1

scala> while (i < 40)
     | {
     |   println(i*i)
     |   i += 1
     | }
1
4
9
[...]
1444
1521

scala> var j = 0
j: Int = 0

scala> for (k <- 1 until 4) j += k

scala> j
res0: Int = 6
```

Notice that the range `1 until 4` excludes 4 and includes 1. The expression `1 until 4` is an example of a *range*:

```
scala> val r = 1 until 4
r: Range = Range(1, 2, 3)

scala> val s = 1 to 10
s: Range.Inclusive = Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> r.start
res0: Int = 1

scala> r.end
res1: Int = 4

scala> r.step
```

```
res2: Int = 1

scala> s.end
res3: Int = 10

scala> s by 2
res4: Range = Range(1, 3, 5, 7, 9)

scala> s
res5: Range.Inclusive = Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> 0 to 8 by 3
res6: Range = Range(0, 3, 6)
```

Note that by is an operator that takes a range and returns a new one with a different step, but (like most operators and functions in Scala) doesn't change the value of its parameter.

1.2 Collections

Scala also has *tuples*, which are arbitrary combinations of elements, maybe of different types:

```
1  scala> val t = (1, "two")
2  t: (Int, java.lang.String) = (1,two)
3
4  scala> t._1
5  res0: Int = 1
6
7  scala> t._2
8  res1: java.lang.String = two
9
10 scala> t._3
11 <console>:6: error: value _3 is not a member of (Int, java.lang.
       String)
12        t._3
13          ^
```

(Here ._ isn't a special operator; we're accessing the member _1 of the object t. You can see evidence of this in the interpreter's response to line 10.)

You can use tuples to make assignments to multiple variables at once using Scala's pattern-matching system:

```
scala> val (a, b) = t
a: Int = 1
b: java.lang.String = two
```

This is a feature that we'll explore more deeply with the languages we encounter later in the course.

Scala supports a few other basic collections. A *list* is an ordered collection of variable length:

```
scala> val a = List(1,2,3)
a: List[Int] = List(1, 2, 3)

scala> a(2)
res0: Int = 3

scala> 4::a
res1: List[Int] = List(4, 1, 2, 3)

scala> (5::7::3::Nil)(1)
res2: Int = 7
```

In Scala, the built-in constant `Nil` refers to an empty list, and `::` is the list construction operator, which takes an entry and another list and returns a new list with the entry at the head. Lists are store internally as a *head*, which is the single element at the beginning of the list, and a *tail*, which is another list, containing everything but the head. This goes on down the list until we get to the empty list, or `Nil`:

```
scala> val b = List(7,2,4)
b: List[Int] = List(7, 2, 4)

scala> b.head
res0: Int = 7

scala> b.tail
res1: List[Int] = List(2, 4)

scala> b.tail.head
res2: Int = 2

scala> b.tail.tail
res3: List[Int] = List(4)

scala> b.tail.tail.tail
res4: List[Int] = List()
```

Scala also supports *sets*, which are unordered collections:

```
scala> val a = Set(1,2,3,4)
a: scala.collection.immutable.Set[Int] = Set(1, 2, 3, 4)

scala> a+5
res0: scala.collection.immutable.Set[Int] = Set(5, 3, 1, 4, 2)
```

```

scala> a-3
res1: scala.collection.immutable.Set[Int] = Set(1, 2, 4)

scala> a-8
res2: scala.collection.immutable.Set[Int] = Set(1, 2, 3, 4)

scala> a ++ Set(1,2,10)
res3: scala.collection.immutable.Set[Int] = Set(10, 3, 1, 4, 2)

scala> a -- Set(1,2,3,4)
res4: scala.collection.immutable.Set[Int] = Set()

scala> a ** Set(4,5,6)
res5: scala.collection.immutable.Set[Int] = Set(4)

scala> Set(1,2,3) == Set(1,3,2)
res6: Boolean = true

scala> List(1,2,3) == List(1,3,2)
res7: Boolean = false

scala> a.contains(3)
res8: Boolean = true
```

And maps, which are unordered correspondences between keys and values:

```
scala> val b = Map(1 -> "one", 12 -> "twelve", 122 -> "one hundred
    twenty-two")
b: scala.collection.immutable.Map[Int,java.lang.String] = Map(1 ->
    one, 12 -> twelve, 122 -> one hundred twenty-two)

scala> b(12)
res0: java.lang.String = twelve

scala> b(4)
java.util.NoSuchElementException: key not found: 4
  at scala.collection.Map$class.default(Map.scala:169)
  at scala.collection.immutable.Map3.default(Map3.scala:22)
  at scala.collection.Map$class.apply(Map.scala:80)
  at scala.collection.immutable.Map3.apply(Map3.scala:22)
  at .<init>(<console>:6)
  at .<clinit>(<console>)
  at RequestResult$.<init>(<console>:3)
  at RequestResult$.<clinit...

scala> (b + (4 -> "four"))(4)
```

```
res2: java.lang.String = four
```

Scala has a special type, called `Any`. Every type inherits from `Any`; accordingly, a list of `Any`s can contain objects of any type:

```
scala> List(1, 2, "three")
res0: List[Any] = List(1, 2, three)
```

Another special type, called `Unit`, has only one object, called `()`. `Unit` is used as the return type of a function that isn't meant to provide any information by its return value, kind of like `void` in C-like languages. For example, the function `println` that we encountered before is of type `Unit`.

1.3 Functions

You can define functions in Scala like you can in imperative languages, but there are important differences. In many imperative languages (including C, Java, and Python) functions output values using the keyword `return`. But since Scala is a functional language, returning values is by far the most important thing that a function can do. Accordingly, you specify what a function returns just by writing the return value by itself:

```
scala> def add1(x: Int): Int = x + 1
add1: (Int)Int

scala> add1(3)
res0: Int = 4
```

There are a couple things to notice about the definition of `add1` here. First, the syntax is rather different from the syntax for defining functions in most C-like languages, including in Java. The equal sign in particular is mandatory, and easy to forget. The second thing to notice is the colons and the types. The "`x: Int`" means that the variable `x` has type `Int` (this is what would be written `Int x` or something in statically typed C-like languages) and the "`: Int`" outside the parentheses gives the return type of the function.

(Scala makes a distinction between *functions* and *methods*, and the definition of `add1` here is technically a method. We won't be going deeply enough into Scala to care about the difference, but feel free to ask if you'd like to know more about this.)

These are called *type annotations*. You actually could have been using them after any expression but most of the time Scala's type inference system will figure things out for you. But there's nothing syntactically wrong about being explicit, or even redundant, with type annotations. For example, these lines all act exactly the same:

```
val s = "Hello"
val s: String = "Hello"
val s = "Hello": String
val s:String = ("Hello": String): String
```

Type annotations for the parameters to a function, however, are mandatory. The annotation for the return type isn't, i.e., you could omit the second "`: Int`" in the definition of `add1` above.

It's customary to introduce functional programming languages by showing how to define the factorial function in them. Since Scala is a hybrid, we're going to define the factorial in a few different ways and compare them. The first will be the imperative way:

```
def fact(n: Int) =
{
    var r = 1
    for (i <- 1 to n)
      r *= i
    r
}
```

This is full of functional no-nos like mutable state and functions that do more than compute a return value, but since Scala is not purely functional, it's relatively easy (and sometimes desirable) to write functions this way. Note the use of curly braces to define a block of code, and the fact that the return value appears alone on the last line.

A more purely functional way to write this function is to use recursion:

```
def fact(n: Int): Int =
  if (n == 0)
    1
  else
    n * fact(n - 1)
```

Notice that, although it's split up into several lines to make it easier to read, the stuff after the equal sign consists of just one expression. Also, since the factorial function given here is defined in terms of itself, Scala requires an annotation for the return type.

One of the things that makes functional programming special is that they allow you to use functions in the same way that you use other objects. For example, lists in Scala have a method called `map` that takes a function as a parameter and returns the list you get by applying the function to everything in the list:

```
scala> List(0,1,2,3,4) map fact
res0: List[Int] = List(1, 1, 2, 6, 24)
```

To facilitate programming in this style, Scala makes it easy to define *anonymous functions*, functions that are defined without being assigned a name. For example:

```
scala> List(0,1,2,3,4) map (x => x*x)
res0: List[Int] = List(0, 1, 4, 9, 16)
```

The expression "`x => x*x`" refers to a function that takes a parameter called `x` and returns its square. If you only need to use each parameter once, there's an even shorter syntax:

```
scala> List(0,1,2,3,4) map (_ + 2)
res0: List[Int] = List(2, 3, 4, 5, 6)
```

Each _ in the expression refers to a parameter to a function, and the underscores are bound to the parameters to the function in the order that they appear. For example:

```
scala> val f: (Int, Int) => Int = _ - 2 * _
f: (Int, Int) => Int = <function>

scala> f(10, 4)
res0: Int = 2
```

(You might notice the difference between the type of `f` here and the type of `fact` above: this type has an arrow in it and the other one doesn't. This is related to the difference between methods and functions mentioned earlier — the one with the arrow is the type of a function, not a method — and again, you should ask if you want to know more about this.)

There's an even shorter way to write the factorial function in Scala, and it gives us an excuse to introduce some more language features that demonstrate some of the power of functional programming:

```
def fact(n: Int) = (1 /: (2 to n))(_ * _)
```

The operator `/:` is called the *folding* operator, and it deserves some explanation. The folding operator needs three things: a list, an initial value, and a function for combining values. What it does is, starting with the initial value, applies the function to the current value and the first entry in the list, getting a new value, then moves on to the rest of the list. So here, we'll start with 1, then apply the multiplication function to 1 and the first element of the list, giving us 2. Then we'll apply it again to 2 and the next element of the list — which is 3 — and get 6. This will continue until we run out the list.

Also, notice the syntax of `/:`. It takes an initial value and a list and returns a function. This method then takes a function, which it uses to fold the list into the value. This process — having a function which returns another function that takes more parameters — is called *currying*, and it's something that will become more important with some of the other languages we'll encounter later.

Recursion is also the usual functional way to write functions that operate on lists. For example, consider the following two functions that each compute the length of a list:

```
def length1(a: List[Any]) =
{
    var r = 0
    a foreach (_ => r += 1)
    r
}

def length2(a: List[Any]): Int =
  if (a == Nil)
    0
  else
    1 + length2(a.tail)
```

The imperative version has some new things in it which are worth commenting on. The `foreach` method of lists is just like `map`, except that it expects a function that returns unit. Things like += that modify variables return `Unit` (note that this is different from many C-like languages) so the function used there does the trick. Since we don't care about the parameter to the function, we can refer to it with the placeholder _.

The second version doesn't use any temporary variables or mutable state, and once we move on to more functional programming languages it's going to be the only game in town. The tail of a list has length one less than the list itself, so we can use this fact to recursively compute the length. We just need the base case: the length of the empty list is zero.

We can also build lists recursively using the list construction operator. For example:

```
1  scala> def squaresbetween(x: Int, y: Int): List[Int] =
2       | {
3       |   if (x > y)
4       |     Nil
5       |   else
6       |     (x*x) :: squaresbetween(x + 1, y)
7       | }
8  squaresbetween: (Int,Int)List[Int]
9
10 scala> squaresbetween(4,9)
11 res0: List[Int] = List(16, 25, 36, 49, 64, 81)
```

(The curly braces are only necessary to make the interactive interpreter not stop reading input after line 4. The function body is still just one expression long.)

1.4 Classes and Objects

A Java program is a collection of classes, and almost everything in Java is thought of as an *object*, a collection of data together with functions (called *methods* in Java) for manipulating that data. A *class* is a type of object, which specifies that sorts of data and methods an object is supposed to have, and objects are created by *instantiating* classes.

Scala operates on a similar philosophy, and it also allows you to define classes and objects in a way that is similar to, but also different from, the way Java does. Consider the following class definition:

```
class Rational(n: Int, d: Int)
{
    def this(x: Int) = this(x, 1)

    private def gcd(x: Int, y: Int): Int =
6          if (y == 0)
7              x
8          else
9              gcd(y, x % y)
10
11     private val num = n / gcd(n, d)
12     private val den = d / gcd(n, d)
13
14     def toDouble: Double = num.toDouble / den.toDouble
15     override def toString: String = num + "/" + den
16
17     def +(rhs: Rational) = new Rational(num * rhs.den + rhs.num * den
       , den * rhs.den)
18     def -(rhs: Rational) = new Rational(num * rhs.den - rhs.num * den
           , den * rhs.den)
19     def *(rhs: Rational) = new Rational(num * rhs.num, den * rhs.den)
20     def /(rhs: Rational) = new Rational(num * rhs.den, den * rhs.num)
21 }
```

There are a lot of things going on here, so let's take them one at a time. This defines a class called `Rational`, designed to represent rational numbers. The parameters after the name of the class make it look kind of like a function, and in fact it is: when we define the class, we also define a *constructor*, a function for creating objects of this class. In this case, our constructor takes two integers `n` and `d` and produces the rational number n/d, so the expression `new Rational(3,5)` evaluates to the rational number 3/5.

On line 3, we define another constructor which takes just one number and returns the corresponding `Rational`. We define it in terms of the other constructor. (In both Java and Scala, `this` is a keyword that refers to the specific instance of the class which is running the method in which it appears.)

Starting on line 5 we have a private method, and starting on line 11 we have some private data members. Methods and data that are *private* are meant to be used only by the class itself and not by any other piece of code that uses the class. Trying to invoke the method `gcd` on some object of type `Rational` would cause an error, as would trying to access the member `num` or `den`. Having private elements in a class feeds in to a philosophy of object-oriented programming called *encapsulation*, the idea of which is to hide details about the implementation of a class from the end user so that they can be changed without affecting how the class is used.

Next, starting on line 14, we have two public methods. You can tell that they're public because the word `private` doesn't appear next to them, and you can tell that they're methods because they're declared with the word `def`. Scala methods don't have to take parameters; unlike in some C-like languages, you don't need to follow the name by an empty pair of parentheses. If you define a method in this way, it is also invoked without any parentheses. Note that this is indistinguishable from accessing a data member.

Notice that `toString` starts with the word `override`. In the Java world, every class inherits from the class `Object`. (If you're not familiar with this concept, *inheritance* is a way for objects of one class to also be considered objects of another class. For example, if we had classes `Person` and `Student`, it would make sense for `Student` to inherit from `Person` so that, for example, a list of `Persons` would also be allowed to contain `Students`. Scala supports inheritance, but we won't be exploring it very deeply.) `Object` already defines a method called `toString`, so since we're defining it again, we have to tell Scala that we're overriding the old definition. Unlike in Java, this is mandatory.

Finally, we define four public methods for doing arithmetic, which are relatively straightforward. When the new `Rationals` are created by these methods, they're going to be put in lowest terms by the constructor, so there's no need to do it in the method itself.

We can use this class in the interpreter, and it works how you might expect. (Note: If you want to use the interactive interpreter's `:load` functionality, you should move the opening curly brace onto the same line as the class definition so it doesn't think the class definition has ended before that line.)

```
scala> val r = new Rational(10, 45)
r: Rational = 2/9

scala> r + new Rational(1, 2)
res0: Rational = 13/18

scala> new Rational(5, 6) * new Rational(2)
res1: Rational = 5/3

scala> r.toDouble
res2: Double = 0.2222222222222222
```

Notice how the `toString` method is automatically used by the interactive interpreter to decide how to display the output.

There's one more thing to point out here. The methods `+` and `*` here are invoked like they're operators, without a dot or parentheses or anything. You might think this is because they share names with predefined operators, but this actually has nothing to do with it. In Scala, any method that only takes one parameter can be invoked this way — in fact, we did it already with the `map` and `foreach` operators above. In Scala, even numbers are objects, and the operations we perform on them are actually method calls. That is, the expression `1+2` is actually read as `(1).+(2)`, that is, the result of invoking the method `+` on the object `1` with the parameter `2`.

2 Clojure

Just like Scala, Clojure is built on top of the Java Virtual Machine. When writing code in Clojure, you will have access to all of Java's standard libraries. But Clojure is designed to be much more in the functional programming vein. We will first focus on Clojure as a standalone language, and then say something about how to access Java libraries from within Clojure.

Clojure is considered to be a dialect of Lisp, which was one of the original languages influenced by lambda calculus. Hence most of Clojure's basic syntax is very similar to Lisp syntax. This makes Clojure a great transition into the world of functional programming: one of the most salient features of Clojure's syntax is its utter simpliciy. Most of the code is written in so-called *s-expressions*, which are parenthesized (and possibly nested) lists.

2.1 Arithmetic, Strings, and Functions

Clojure code can be compiled, but we will start with the Clojure interpreter, which is also knows as the *REPL* or *read-eval-print loop*. You can open the Clojure REPL by typing `clojure` in a terminal. To exit the REPL, hit `Ctrl-D` or `(System/exit 0)`.

You can now type expressions in the REPL to have them evaluated.

```
user=> (+ 1 1)
2

user=> (+ 1 2 3)
6

user=> (+ (+ 1 2) 3)
6

user=> (+ 1 1.0)
2.0
```

The first thing to observe in these examples is the s-expression notation for evaluating functions. As seen in the third example, s-expressions can be nested. Even though types aren't specified while defining or executing a function, there are some internal types which redirect to Java types. Moreover, there is some type coercion, as seen in this example.

```
user=> (type (+ 1 1))
java.lang.Integer

user=> (type (+ 1 1.0))
java.lang.Double
```

An s-expression consists of a sequence of sub-expressions, each of which is itself an s-expression. To evaluate a function, we write an s-expression whose first element is the name of the function and the remaining elements are the arguments. Usually, the arguments to functions are evaluated in an unspecified order. Let us now look at some expressions involving strings.

```
user=> "hello, world!"
"hello, world!"

user=> (+ "hello " "world!")
java.lang.ClassCastException: java.lang.String cannot be cast to java
    .lang.Number (NO_SOURCE_FILE:0)

user=> (str "hello " "world!")
"hello world!"

user=> (str 1 "2" (+ 3 4))
"127"
```

Hence we see that the + function only works on numeric values, and concatenation of strings is done by `str`. Moreover, we see some Java lurking in the background: the error message shows up is a Java exception.

Just like Scala, Clojure is not a purely functional language. Hence it supports some functions with side effects.

```
user=> (println "hello, world!")
hello, world!
nil
```

In this example, the side effect is that the string “hello, world!” gets printed to the REPL. The function output is represented as `nil`, which corresponds to `Null` in Java.

Now we will explore how to define new variables and functions in Clojure. Variables are defined using the `def` special form, whose output is the curently defined variable. Functions are defined using the `defn` form, whose output is the currently defined function. Here are some examples.

```
user=> (def x 5)
#'user/x

user=> (def y 10)
#'user/y

user=> (defn add-x-and-y [] (+ x y))
#'user/add-x-and-y

user=> add-x-and-y
#<user$add_x_and_y__1 user$add_x_and_y__1@baa466>

user=> (add-x-and-y)
15

user=> (def x 10)
#'user/x
```

```

user=> (add-x-and-y)
20
```

Observe that after defining the function `add-x-and-y`, we can refer to the function by its name. Hence the output of the expression `add-x-and-y` is the function itself. Since it is a function with no arguments, there are no arguments specified in the `def` statement (indicated by the empty square brackets). Also observe that if we redefine `x` to be 10, then the output of the function changes to the new correct value, because the function is evaluated again after being called. We can define functions with arguments as follows.

Notice that the variables inside the function definition are just names for the arguments of the function and have nothing to do with the global variables `x` and `y` that we defined earlier. However, as in the third example if `x` is the only argument specified and I refer to `y`, then in this case `y` refers to the global variable already defined.

```
user=> (defn square [x] (* x x))
#'user/square

user=> (square 2)
4

user=> (defn my-multiply [x y] (* x y))
#'user/my-multiply

user=> (my-multiply 4 6)
24

user=> (defn add-to-y [x] (+ x y))
#'user/add-to-y

user=> (add-to-y 3)
13
```

There may be multiple expressions inside a function definition. The value of the evaluated function is then the value of the last expression. There is also an optional documentation string that can be specified just after the name of the string, which can be accessed later using `doc`.

If no expressions are given in the definition, the function evaluates to `nil`. Here are some examples.

```
user=> (defn square "Squares the input." [x] (* x x))
#'user/square

user=> (doc square)
-------------------------
user/square
([x])
  Squares the input.
```

```
nil

user=> (square 3)
9

user=> (defn foo "Redefines y to be x, then squares x." [x] (def y x)
     (* x x))
#'user/foo

user=> (foo 19)
361

user=> y
19

user=> (defn bar [a])
#'user/bar

user=> (bar)
java.lang.IllegalArgumentException: Wrong number of args passed to:
    user$bar (NO_SOURCE_FILE:0)

user=> (bar 3)
nil
```

It is also possible to define anonymous functions in Clojure. One way to do this is the `fn` form. These anonymous functions can be used within other expressions. A shorter way is to use the `%` syntax, which is best learned via examples.

```
user=> ((fn [x] (even? x)) 4)
true
user=> ((fn [x y] (- x y)) 5 9)
-4
user=> (#(even? %) 4)
true
user=> (#(- %1 %2) 5 9)
-4
```

2.2 Control Structures and Data Structures

Let us explore the control structures in Clojure. First, the booleans are `true` and `false`. As we saw earlier, there is also a “nothing” value called `nil`. Normally if there is a sequence of expressions, say as the arguments of a function or in a `def` statement, then they are evaluated in an unspecified order. To evaluate expressions in the order given, use the `do` special form.

This is especially useful in conditionals. The syntax of an `if` statement is `(if <condition> <then> <else>)`. If `<condition>` is evaluates to something other than `false` or `nil`, then the output is the `<then>` expression. Otherwise, the output is the `<else>` expression.

```
user=> (if true 5 6)
5

user=> (if false 5 6)
6

user=> (if nil 5 6)
6

user=> (if (< 1 2)
           (do (print "five = ") 5)
           (do (print "six = ") 6))
five = 5

user=> (if (> 1 2)
           (do (print "five = ") 5)
           (do (print "six = ") 6))
six = 6
```

If only one branch is needed, the `when` form can be used.

```
user=> (when (< 1 3) 5)
5

user=> (when (> 1 3) 5)
nil
```

So far we have seen lists in Clojure. Other common data structures are: vector, set, and map. We will talk about lists, vectors, and sets.

A list is an ordered collection of items, which is usually accessed top-down. We have seen lists as the format in which variables are defined and functions are evaluated. A list is constructed by the syntax `(list <items>)`. Given a list `lst`, there are functions to return the first object, namely `(first lst)`, and the remaining list, namely `(rest lst)`.

```
user=> (def lst (list 1 2 3 4))
#'user/lst

user=> lst
(1 2 3 4)

user=> (first lst)
1

```

```
user=> (rest lst)
(2 3 4)
```

A new item can be attached in front of an existing list using the function `cons`. Two lists can be appended using `concat`.

```
user=> (cons 1 nil)
(1)

user=> (cons 1 '(2))
(1 2)

user=> (concat '(1 2 3) '("a" "b" "c"))
(1 2 3 "a" "b" "c")
```

A vector is also an ordered collection of items, but vectors are like arrays, and accessing elements of a vector is faster than accessing elements of a list. Hence vectors are commonly used as a data structure. Moreover, many functions that work on lists also work on vectors. Here are some examples.

```
user=> (def vect1 (vector 1 2 3 4))
#'user/vect1

user=> vect1
[1 2 3 4]

user=> (def vect2 [5 6 7 8])
#'user/vect2

user=> vect2
[5 6 7 8]

user=> (nth vect1 0)
1

user=> (nth vect1 4)
java.lang.ArrayIndexOutOfBoundsException: 4 (NO_SOURCE_FILE:0)

user=> (first vect2)
5

user=> (last vect2)
8

user=> (cons 1 [1 2 3])
(1 1 2 3)

```

```
user=> (concat [1 2] [2 2 1])
(1 2 2 2 1)

user=> (count [1 2])
2
```

There is a function called `count` which finds the length of a collection. Similarly there is also a function called `conj` which adds an element to a collection. But it behaves differently on different types of collections, as seen below.

```
user=> (conj [1 2 3] 4)
[1 2 3 4]

user=> (conj '(1 2 3) 4)
(4 1 2 3)
```

Set is another data structure. Here is how to construct sets.

```
user=> (set '(1 1 2 3 4))
#{1 2 3 4}

user=> (set ["a" "b" 1 1 "b" "c" "a"])
#{1 "a" "b" "c"}
```

An exciting feature of Clojure is its ability to create *lazy* lists. These are lists whose elements don't get evaluated until absolutely necessary. In particular, the list can be infinitely long. Here is a basic example.

```
user=> (def my-lst (repeat 1))
#'user/my-lst

user=> (take 5 my-lst)
(1 1 1 1 1)

user=> (take 10 my-lst)
(1 1 1 1 1 1 1 1 1 1)
```

To create functions that use lazy lists, the `lazy-seq` macro is used. We can create some fancy functions this way.

```
user=> (def ones (lazy-seq (cons 1 ones)))
#'user/ones

user=> (take 10 ones)
(1 1 1 1 1 1 1 1 1 1)
```

```

user=> (def numbers
            (lazy-seq
              (cons 1 (map (fn [x] (+ x 1)) numbers))))
#'user/numbers

user=> (take 5 numbers)
(1 2 3 4 5)
```

2.3 Recursion and iteration

Clojure supports recursive functions. Here is an example.

```
user=> (defn factorial [n]
             (if (= n 0)
                 1
                 (* n (factorial (- n 1)))))
#'user/factorial

user=> (factorial 0)
1

user=> (factorial 5)
120
```

However, this is not very efficient, because if `n` is large, then the successive values `n`, `n-1`, etc. need to be placed on the evaluation stack in order, which may result in a stack overflow exception. A more efficient way to write the same function recursively is the `loop` and `recur` combination. This function is still recursive, but consumes constant stack space.

```
user=> (defn better-factorial [n]
             (loop [num n, prod 1]
                   (if (zero? num)
                       prod
                       (recur (dec num) (* prod num)))))
#'user/better-factorial
```

Let us go through this definition step-by-step. The `loop` special form defines the local variables `num` and `prod`, which have initial values set to `n` and `1` respectively. We will take the input and successively decrement it, keeping track of the product accumulated so far.

The variable `prod` keeps track of the product accummulated so far. The variable `num` keeps track of the current number. If we are ever at a stage when `num` is zero, then we return the total product accumulated. If `num` is non-zero, then we decrement it and multiply `prod` by `num`. Then we send the new values of `num` and `prod` to the body under the `loop` statement, by using the construction `recur`.

2.4 Data structure processing

In this section we will briefly describe some built-in functions to process data structures in Clojure. The `filter` function takes in a predicate and a sequence, and outputs a list consisting only of those elements of the original sequence that satisfy the predicate.

```
user=> (filter #(even? %) '(1 2 3 4 5 6))
(2 4 6)

user=> (filter #(even? %) [1 2 3 4 5 6])
(2 4 6)
```

The `map` function takes in a function `f` of k arguments and a k sequences. It outputs the sequence obtained by applying `f` to each tuple of corresponding elements of the k sequences. If the k sequences have different lengths, it outputs the shorter of the two.

```
user=> (map #(* % %) '(1 2 3))
(1 4 9)
user=> (map #(* %1 %2) '(1 2 3) '(3 4 5 6))
(3 8 15)
```

The `reduce` function takes in a function `f` of two arguments, an optional start value, and a sequence. It applies `f` to the start value and the first element of the sequence, then `f` to the result and the second element of the list, and so on, and outputs the final result. If no start value is given, reduce starts by applying `f` to the first two elements of the sequence.

```
user=> (reduce #(+ %1 %2) 1 '(1 2 3))
7

user=> (reduce #(+ %1 %2) [1 2 3])
6

user=> (reduce #(if (> %1 %2) %1 %2) [1 3 2])
3

user=> (reduce #(if (> %1 %2) %1 %2) '(1 3 2))
3

user=> (reduce #(if (> %1 %2) %2 %1) [1 3 2])
1
```

2.5 Interacting with Java

As mentioned briefly before, Clojure can access Java libraries, and Clojure code can be compiled into Java bytecode to be able to run on a Java Virtual Machine. Here are some simple examples of how Java libraries can be accessed to run on the Clojure REPL.

```
user=> (java.lang.System/currentTimeMillis)
1342406906630

user=> (java.lang.System/nanoTime)
127762484559483

user=> (java.lang.Math/PI)
3.141592653589793

user=> (java.lang.Math/log java.lang.Math/E)
1.0
```

3 Haskell

Haskell is a lot like the languages we've already examined, but more so. Scala and Clojure have functional components, but Haskell is *purely* functional: no functions are allowed to have any side-effects, ever. Later on, we'll catch a glimpse of *monads*, which are how Haskell deals with this seemingly absurd restriction, and we'll see that it's not nearly as strange as it sounds. In particular, this restriction allows the language to fully implement *lazy evaluation*. Since no function can do anything except compute a value to return, there's no need to evaluate any expression until you actually need the value.

Haskell is, in some sense, the ultimate functional programming language, for better or for worse. Laziness, along with some other features like a wonderfully designed type system with good type inference, pattern matching, and concise syntax give a Haskell user the strong impression that things have all been done "the right way." Still, the restrictions that have to be put into place to allow this to happen can sometimes make some problems harder to solve, and the price you have to pay for lazy evaluation is that it's not always clear when or how many times functions are going to be executed.

3.1 Types and Data Structures

The compiler we're using for this class is called GHC, and like the previous two, there is an interactive interpreter. It's called GHCi, and you can get to it by typing `ghci` at a terminal.

Just as before, we can make GHCi evaluate expressions for us. You'll notice that Haskell's syntax is more normal than Clojure's, and more concise than Scala's.

```
Prelude> 2 + 3 * 4
14
Prelude> "this is " ++ "a string"
"this is a string"
Prelude> head [5, 2, 7, 8]
5
```

The expression `[5, 2, 7, 8]` is a list, and `head` is a function that does the same thing as the method by the same name that we encountered in Scala. Note that functions in Haskell are applied just by putting the arguments after the name of the function.

Unlike Scala (and to some extent Clojure) Haskell isn't built on the object-oriented programming paradigm; there are no methods to be seen here. Functions that operate on data structures are just functions, like `head`, and they're not invoked through any given object. But, like Scala, Haskell has a *static type system*, meaning that Haskell needs to know the type of every expression at compile time, and this prevents you from applying functions in situations where it makes no sense. (The object-oriented goal of encapsulation is handled, to some extent, by Haskell's module system, which you should investigate if you're interested.)

Also like Scala, Haskell has type inference, which means that most of time you don't need to explicitly label the types of expressions. You can ask for the type of an expression in GHCi by typing `:t` before the expression:

```
Prelude> :t True
```

```
True :: Bool
Prelude> :t (1 == 2) || (7 < 9)
(1 == 2) || (7 < 9) :: Bool
Prelude> :t 'c'
'c' :: Char
Prelude> :t [True, False, True]
[True, False, True] :: [Bool]
Prelude> :t "string"
"string" :: [Char]
```

Notice that the type of a list of x's is just written `[x]`, and that a string is the same thing as a list of `Char`s. Trying this on numbers gives us a very different-looking response from GHCi:

```
Prelude> :t 7
7 :: (Num t) => t
Prelude> :t 2.72
2.72 :: (Fractional t) => t
```

`Num` and `Fractional` are examples of what Haskell calls *type classes*. A type class is a collection of types that can all be used in the same set of functions. For example, everything in the type class `Num` can be added, subtracted and multiplied, along with some other things. `Fractional` is a *subclass* of `Num`, meaning that every type in the `Fractional` class is also in the `Num` class. In addition to everything other `Num` types can do, `Fractional` types can also be divided using the `/` operator. (There is also a built-in integer division function called `div`.)

The types that were assigned in the code listing above contain *type variables*. The type inference system produces types like this when it's unable to completely deduce the type of an expression. The type `(Num t) => t` should be read as something like "For any type `t` in the class `Num`, this could have type `t`." We can see type variables popping up in even more complicated types:

```
Prelude> :t [1, 2, 3]
[1, 2, 3] :: (Num t) => [t]
Prelude> :t []
[] :: [a]
Prelude> :t head
head :: [a] -> a
```

In the last two examples, we see that type variables need not be assigned to a class at all: the `head` function works on any sort of list, no matter what type is in it, and it produces whatever type was in the list.

Lists are composed of a head and a tail, just as in Scala, and there is a list construction operator called `:` that takes an element and a list and returns the list whose head and tail are the parameters:

```
Prelude> 3:[6,2,8]
[3,6,2,8]
Prelude> []:[[1,2],[3,4]]
[[],[1,2],[3,4]]
```

```
Prelude> 4:2:9:2:[]
[4,2,9,2]
Prelude> []:[]
[[]]
```

Note that the roles of the symbols : and :: are switched from Scala.

Haskell also has tuples that work just like tuples in Scala:

```
Prelude> :t ('a', 3, [1, 2])
('a', 3, [1, 2]) :: (Num t, Num t1) => (Char, t, [t1])
```

Tuples belong to different types depending on how many things are in them and what type each thing has.

3.2 Functions

You can declare variables in GHCi using `let`:

```
Prelude> let a = 3
Prelude> a + 1
4
```

If you put code in another file and want to load it in GHCi, you can use `:l filename` and it will run all the code in the file. If you do this, though, you shouldn't precede declarations with `let`:

```
--In test.hs
a = 6

--In GHCi
Prelude> :l test
[1 of 1] Compiling Main             ( test.hs, interpreted )
Ok, modules loaded: Main.
*Main> a
6
```

We can define functions using a nearly identical syntax:

```
Prelude> let f x y = x + 2 * y
Prelude> f 7 1
9
```

Again, if you're loading code from a file, you should omit the `let`. Recursive functions work just like you would expect:

```
Prelude> let fact n = if n == 0 then 1 else n * fact (n-1)
Prelude> fact 6
720
```

In the definition of `f` above, you can see that just as the name of the function is separated from its first parameter by a space, so is the first parameter from the second. This is a reflection of the fact that all functions in Haskell are *curried.* Currying a function of two arguments means turning it into a function of one argument that returns a function to take the other argument. That is, we can do things like this:

```
Prelude> let g x y = x + y
Prelude> g 2 3
5
Prelude> let part = g 7
Prelude> part 3
10
```

When we write "`g 7`" we've plugged in `7` for `x`, but we don't have a value yet for `y`. So what we get isn't a number, it's a function which need the value of `y` and then returns the number. The `7` is sort of "suspended" inside `part`, waiting for the second parameter so it can be added to it.

We can see evidence of currying in the types of the functions:

```
Prelude> :t g
g :: (Num a) => a -> a -> a
```

That type could also be written `(Num a) => a -> (a -> a)`. That is, `g` is a function that takes a numeric type `a`, and returns a function that takes `a`'s to `a`'s. So the expression `g 2 3` is secretly read as `(g 2) 3`: we're applying the function `g 2` to the parameter `3`.

Anonymous functions can be defined using relatively concise notation. Note the presence of currying here too:

```
Prelude> (\x -> [x]) 3
[3]
Prelude> let f = (\a -> \b -> b - a) 9
Prelude> f 3
-6
```

Haskell has some syntax for treating operators in this way too. Operators in Haskell, like `+`, `<`, `&&`, and `:`, are actually functions just like `g`, just with a different syntax. We can treat them as functions by surrounding them with parentheses:

```
Prelude> :t (+)
(+) :: (Num a) => a -> a -> a
Prelude> (+) 3 5
```

```
8
```

We can partially apply an operator by sticking a parameter on one side of it inside the parentheses:

```
Prelude> let f = (<3)
Prelude> f 5
False
Prelude> f 1
True
Prelude> let g = (9-)
Prelude> g 4
5
Prelude> g 10
-1
Prelude> (((+7) 2)*) 8
72
```

As in Scala, there are a lot of built-in functions for interacting with lists, many of which take functions as parameters. We'll demonstrate some here; for more information, you should check the documentation online. It's a good exercise to try implementing these yourself, especially after you've read the next section.

```
Prelude> length [4, 8, 9, 0]
4
Prelude> [0,2,4,6,8] !! 3
6
Prelude> take 5 [2,3,7,5,1,0,9,3,7,4,8]
[2,3,7,5,1]
Prelude> map (\x -> x + 1) [4, 5, 6]
[5,6,7]
Prelude> let even x = x ‘mod‘ 2 == 0
Prelude> filter even [1..20]
[2,4,6,8,10,12,14,16,18,20]
Prelude> foldl (+) 1 [2,4,6]
13
Prelude> any even [3,6,9,11]
True
Prelude> all even [3,6,9,11]
False
```

3.3 Useful Features for Function Definitions

One of Haskell's most powerful features is *pattern matching*. It allows you to define a function for values that look a certain way. For example, we can write a function that adds up all the numbers in a list like this (in a file):

```
sum [] = 0
sum (x:xs) = x + sum xs
```

The first line defines the value of the function on the empty list, and the second line defines the value of the function on any list that looks like `x:xs`, that is, any list that has a head and a tail. Since every list either has a head and a tail or is empty, we've defined `sum` on every list. Note the placement of parentheses: in Haskell, function application binds tighter than everything else, so we need to put parentheses around the pattern `x:xs`, but not around the function application `sum xs`.

We can use `case` expressions to do pattern matching outside of a function definition:

```
--In test.hs
n = case [1, 2, 3] of
      [] -> 0
      x:_ -> x

--In GHCi
Prelude> :l test
[1 of 1] Compiling Main             ( test.hs, interpreted )
Ok, modules loaded: Main.
*Main> n
1
```

(The pattern _ is used when we want to indicate that something belongs there but we don't care to give it a name.)

We can use *guards* to augment patterns:

```
--In test.hs
abs' 0 = 0
abs' x
  | x > 0 = x
  | x < 0 = -x

--In GHCi
Prelude> :l test
[1 of 1] Compiling Main             ( test.hs, interpreted )
Ok, modules loaded: Main.
*Main> abs' 5
5
*Main> abs' 0
0
*Main> abs' (-3)
3
```

The guard "`| x > 0`" means that the definition that follows applies only in the case that `x` is greater than zero, and similarly for the next one. Notice that the first pattern isn't strictly necessary; we could have included it by changing one of the later lines. (This function is called `abs'` because there's already a built-in function called `abs`. We have to put `-3` in parentheses because otherwise Haskell thinks we're subtracting `3` from `abs'`.)

The special guard `otherwise` matches anything not caught by the other guards:

```
--In test.hs
describeList xs
  | length xs == 0 = "empty"
  | length xs < 3 = "kinda small"
  | length xs < 10 = "medium-sized"
  | otherwise = "really big"

--In GHCi
Prelude> :l test
[1 of 1] Compiling Main             ( test.hs, interpreted )
Ok, modules loaded: Main.
*Main> describeList [4,2,10,9]
"medium-sized"
*Main> describeList [1..20]
"really big"
```

It's often helpful to be able to define temporary variables and functions for later use. Haskell offers two ways to do this, called `let` and `where`. They look and act very similar:

```
--In test.hs
silly x = [y, f x, y, f x] where
  y = 2 * x
  f a = 3 * a

silly' x =
  let
    y = 2 * x
    f a = 3 * a
  in
    [y, f x, y, f x]

--In GHCi
*Main> silly 3
[6,9,6,9]
*Main> silly' 3
[6,9,6,9]
```

The main difference between these, aside from where the definitions go, is that `let` is an expression and `where` is just something that goes onto the end of a definition. That is, we can do this with `let`:

```
Prelude> 1 + (let x = 2 in x * 3)
7
```

and there's no corresponding construction with `where`.

3.4 Datatypes

Haskell allows you to define your own data structures to use as types. These constructions, called *datatypes*, have some things in common with classes in object-oriented languages, but with some differences. They tend to be a lot smaller, and they don't have any methods; the things that would be methods in a language like Scala are instead implemented as functions that take the corresponding object as a parameter.

You can define a datatype with a `data` declaration:

```
--In a file
data Foo = Bar | Baz Int Int | Qux [Foo]

--In GHCi
*Main> :t Bar
Bar :: Foo
*Main> :t Baz 3 5
Baz 3 5 :: Foo
*Main> :t Baz
Baz :: Int -> Int -> Foo
*Main> :t Qux [Qux [Bar, Baz 2 3], Baz 0 9]
Qux [Qux [Bar, Baz 2 3], Baz 0 9] :: Foo
```

The `data` line declares a type called `Foo`. Objects of type `Foo` can be constructed in one of three ways: `Bar` is a constant of type `Foo`, `Baz` is a function that turns two integers into a `Foo`, and `Qux` is a function that turns a list of `Foo`s into a `Foo`.

You can do pattern matching on datatypes just like you can on lists:

```
--In a file
describe Bar = "Bar"
describe (Baz m n) = "Baz " ++ show m ++ " " ++ show n
describe (Qux xs) = "Qux ( " ++ descriptions xs ++ ")" where
  descriptions [] = ""
  descriptions (f:fs) = describe f ++ " " ++ descriptions fs

--In GHCi
*Main> describe (Baz 2 9)
"Baz 2 9"
*Main> describe (Qux [Qux[Bar, Baz 2 3], Baz 0 9])
"Qux ( Qux ( Bar Baz 2 3 ) Baz 0 9 )"
```

Datatypes can take other types as parameters. For example, we could define a datatype for binary trees by typing:

```
data BinaryTree a = Leaf a | Branch (BinaryTree a) (BinaryTree a)
```

Here a binary tree is either a leaf, in which case it's labelled with an object of whatever type `a` is, or it's a branch, in which case it has two binary trees under it, one going to the left and one going to the right. In fact, lists can be thought of as a datatype themselves. That is, except for the special bracket syntax, it's as if lists we defined like this:

```
data List a = Empty | Colon a (List a)
```

The pattern matching we used on lists should then be thought of as being of the same type as the what we used with `describe` above.

All the variables we've been using up to now have begun with a lower-case letter, but the names of the functions used to construct types (called *constructor functions*) start with upper-case letters. This is actually not just a convention; it's a restriction that's imposed by the compiler.

3.5 Lazy Evaluation

One of the most entertaining features of Haskell is its *lazy evaluation*. This means that, in the process of evaluating some expression, Haskell will try as long as possible to avoid carrying out a computation it might not end up having to do. This can lead to a lot of efficiency gains. On the more obvious side, in an expression of the form:

```
if (condition that's false) then
    really hard computation
else
    really easy computation
```

the hard computation doesn't have to be performed because the condition is false. This type of thing is actually done in most programming languages that have this type of control structure. On the slightly less obvious side, in code like:

```
l = [some really long list]
m = map (some function) l
...
some code that requires us to know head m
```

the function doesn't actually need to be applied to everything in the list, so Haskell won't bother, even in the declaration of `m`. Then later, when we need to know `head m`, Haskell will finally actually grab the head of `l`, apply the function to it, and give the result.

Notice that this is only possible because of the restriction that functions can't have side-effects. If applying the function to the elements of `l` caused some side-effect to happen, the programmer almost certainly would have expected that to happen when `m` is defined, and not at some arbitrary time hundreds of lines later. If functions in Haskell could have side-effects, then lazy evaluation would have crazy and unpredictable results.

Another advantage of lazy evaluation is that it allows us to have data structures that contain or refer to themselves! For example, Haskell supports *infinite lists*, which can be defined in a remarkably simple way:

```
Prelude> let naturals = 0 : map (+1) naturals
Prelude> head naturals
0
Prelude> naturals !! 7
7
Prelude> take 10 naturals
[0,1,2,3,4,5,6,7,8,9]
Prelude> take 10 (filter even naturals)
[0,2,4,6,8,10,12,14,16,18]
```

If I actually asked GHCi to print out `naturals`, it would never stop, but as long as I only ask for things that can be guaranteed to terminate in a finite amount of time, lazy evaluation allows us to only pay attention to the part of the list we care about.

Incidentally, the list we called `naturals` can also be speicified in a simpler way. We've used Haskell's *range* construction a few times already: for example, `[1..6]` evaluates to the list `[1, 2, 3, 4, 5, 6]`. We can also leave off the second number to get an infinite list, so `naturals` could have been defined as simply `[0..]`.

As a slightly more complicated example, we'll build the sequence of factorials as an infinite list. To do this, we'll use the built-in function `zipWith`, which is defined like this:

```
zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

This function takes a function and two lists and produces the list that you get by applying the function to the pairs of elements of the two lists. For example:

```
Prelude> zipWith (+) [1, 2, 3] [6, 7, 8]
[7,9,11]
```

Then we can define our factorials like this:

```
Prelude> let facts = 1 : zipWith (*) facts [1..]
Prelude> take 8 facts
[1,1,2,6,24,120,720,5040]
```

It's a good exercise to study this example until you're convinced you understand it.

3.6 Monads

Like the other programming languages we've seen, Haskell has some basic functions to deal with input and output. For example, the putStrLn function takes a string and prints it, like the `println` function from Scala and Clojure:

```
Prelude> putStrLn "Hello, side-effects!"
Hello, side-effects!
```

There's another function, called `getLine`, which waits for a string from the user and then returns it. You can string these together using the following syntax:

```
Prelude> do x <- getLine; putStrLn ("What you said was: " ++ x ++
    "!")
I thought Haskell wasn't supposed to have side-effects
What you said was: I thought Haskell wasn't supposed to have side-
    effects!
```

At this point you should be objecting: Haskell wasn't supposed to have side-effects, but here we seem to be writing functions with side-effects. But have no fear, everything is fine. Even though these lines look like instructions, they're actually expressions, just expressions of a type we haven't encountered yet:

```
Prelude> :t putStrLn
putStrLn :: String -> IO ()
Prelude> :t getLine
getLine :: IO String
```

`IO` is an example of a *monad*. An object of type `IO a` is a representation of a sequence of input/output ("I/O") operations that, in the end, returns an `a`. So `putStrLn` takes a string and returns a sequence of I/O operations that return nothing; and `getLine` is a sequence of I/O operations that returns a string.

So the `do` syntax above isn't actually a sequence of instructions at all. It's an expression which evaluates to an `IO` object, and when the interpreter is asked to evaluate it, it just performs the corresponding sequence of operations.

A monad may be thought of as a *container* that is easy to get into, but not necessarily possible to get out of. A monadic type `m a` is then any type that is encapsulated in the container `m`, which is a monad. Computations on a monadic type `m a` can be performed just like computations on the type `a`, but there is a catch. To perform any computation, you must enter the container `m`, perform a computation within the container, and then exit the container, the output being another monadic type `m b` for the same monad `m`.

We'll get used to monads with a simpler example, called `Maybe`. A monad is any datatype that belongs to the `Monad` type class. Any type in the `Monad` type must satisfy certain axioms, and we will illustrate these first with the help of the `Maybe` monad. The datatype `Maybe` is defined like this:

```
data Maybe a = Just a | Nothing
```

It is meant to be a container that represents the result of a computation that might produce an `a` or might fail to produce anything. For example, the built-in function `find` (which lives in the module `Data.List`) returns a `Maybe`:

```
Prelude> import Data.List
Prelude Data.List> :t find
find :: (a -> Bool) -> [a] -> Maybe a
Prelude Data.List> find (<4) [5, 2, 0, 9]
Just 2
Prelude Data.List> find (<4) [5, 12, 10, 9]
Nothing
```

Let us talk about the axioms that the datatype `Maybe` satisfies, that make it a monad.

1. There is a function `return :: a -> (Maybe a)` that inserts any element of type `a` into the `Maybe` container, by sending any input `x` to `Just x`.

2. There is a operator >>= (pronounced 'bind'), which takes in an input of type `Maybe a` and a function `f :: a -> Maybe b`, and returns an output of type `Maybe b` by "applying" `f` to the input. This is not literally possible, because the input may be `Nothing`. In this case, the output is simply `Nothing`. If the input is `Just x`, the output is `Just f(x)`.

   For example, what if you had a computation that depended on the result of `find`? You could wrap everything in a sequence of `case` expressions, but this would get cumbersome very quickly. Instead we can just use >>= as follows.

```
Prelude Data.List> (find (>2) [3, 2, 1]) >>= (\x -> Just (x + 2))
Just 5
Prelude Data.List> (find (>5) [3, 2, 1]) >>= (\x -> Just (x + 2))
Nothing
```

3. On any input `x`, the output of `(return x) >>= Just` is `Just x`. In general for a monadic type `m a` and a constructor `k :: a -> m a`, we must have `(return x)>>= k = k x`.

4. The result of the expression `(Just x)>>= return` is `Just x`, and the result of `Nothing >>= return` is `Nothing`. So binding `return` keeps the input unchanged.

5. Finally, the bind operation is associative in a sense that we will illustrate with the help of an example: the output of `(Just x)>>= (\y -> (f y >>= g))` is the same as the output `((Just x)>>= f) >>= g`.

There are two more functions that are optionally part of the `Monad` type class, namely `(>>)` and `fail`. You can read about these in the documentation.

The `do` syntax lets us represent bind in a slightly more readable way. The example `find (> 2) [3,2,1] >>= Just (x + 2)` could be rewritten as follows:

```
do x <- find (>2) [3, 2, 1]; return (x + 2)
```

In other words, we try to find a number in `[3, 2, 1]` that's bigger than 2, store it in `x`, and then return `x + 2`. When you're typing these in a separate file you can use line breaks instead of semicolons, and as the number of things being strung together gets longer the benefits of the `do` syntax become apparent:

```
nnn--Version 1
(find (>2) [3, 2, 1]) >>= (\x ->
  (find (>x) [4, 5, 6]) >>= (\y ->
    Just (x + y)))

--Version 2
do
   x <- find (>2) [3, 2, 1]
   y <- find (>x) [4, 5, 6]
   return x + y
```

Now that we know the basic structure of a monad, we return to the `IO` monad. Our examples from before work exactly the same way. The following two expressions mean exactly the same thing:

```
-- Version 1
putStr "Enter a string: " >>= (\_ ->
  getLine >>= (\s ->
    return (length s) >>= (\n ->
      putStrLn ("Your string has " ++ show n ++ " characters."))))

--Version 2
do
   putStr "Enter a string: "
   s <- getLine
   n <- return (length s)
   putStrLn "Your string has " ++ show n ++ " characters."
```

And it's clear which one is easier to understand. (Note that, since `putStr` is an `IO ()`, we don't need to care about its return value, and the `do` syntax has a shortcut for this.)

Using `do` and the `IO` monad (along with other monads for different imperative-style tasks) it's possible to write Haskell code that looks a lot like it belongs in an imperative language. Still, because monads are objects just like any other object, we can maintain the no-side-effects restriction that makes Haskell's lazy evaluation work so well.

4 Erlang

Erlang has a lot of things in common with the languages we've examined so far — it's a functional language with lists, pattern matching, and first-class functions. As far as these features are concerned, the only new thing to understand is the syntax. Because of this, we'll go over these aspects of the language very quickly.

The reason we're examining it has to do with its support for *concurrency*, that is, multiple processes that run at the same time. We'll find that, unlike in a lot of other languages, Erlang manages to make concurrency pretty simple. Let's dive in.

4.1 Erlang Syntax Rundown

You can get an interactive interpreter by typing `erl` from a terminal. The first strange thing you'll notice is that every expression has to end with a period:

```
1> 7.
7
2> 2+3.
5
```

Erlang has numbers, strings, lists (denoted by square brackets), and tuples (denoted by curly braces). The list construction operator (the thing corresponding to `::` in Scala, `cons` in Clojure, and `:` in Haskell) is written with a pipe (`|`) where both arguments are in square brackets:

```
1> [1, 2, 3].
[1,2,3]
2> {4, [2, 2], [3, 4, 4]}.
{4,[2,2],[3,4,4]}
3> [4|[2,2,0]].
[4,2,2,0]
4> [6,3|[5,0,9]].
[6,3,5,0,9]
5> [[1,2]|[3,4]].
[[1,2],3,4]
```

The first and third expressions are lists, and the second is a tuple containing a number and two lists in order. Notice that the pipe syntax lets you put more than one element at the front of the list, and that lists can hold different types of values. As in Haskell, strings are secretly lists.

You can assign values to variables using a very simple syntax. The thing to remember here is that variables have to begin with capital letters, the opposite of the restriction we faced in Haskell:

```
1> Num1 = 2.
2
2> X = 8.
8
3> X + Num1.
```

```
10
```

Identifiers that begin with lower-case letters denote `atoms`, fixed values that act kind of like strings that can't be taken apart. Two atoms are the same if and only if they are denoted by the same string:

```
1> apple.
apple
2> X = banana.
banana
3> X == apple.
false
4> X == banana.
true
```

To define functions we're going to put some code in a file and load it in the interpreter. We'll put the following code into a file called `test.erl`. (The lines that start with a `%` are comments.)

```
% This is in test.erl
-module(test).
-export([fact/1]).

fact(0) -> 1;
fact(N) -> N * fact(N-1).

% Everything from here on is in the interactive interpreter.
1> c(test).
{ok,test}
2> test:fact(8).
40320
```

The first line defines the name of the *module* that will hold all the code we're writing, and the second line provides a list of the functions we'll *export*, that is, make available to the user of the module. The `/1` after the name of the function gives the number of parameters that the function takes; functions aren't curried by default in Erlang. Pay attention to the semicolon separating the different parts of the function definition.

When we compile the code in `test.erl` by typing `c(test)`, we can execute our function by typing the module name, a colon, and then the name of the function. You can't define functions using this syntax in the interactive interpreter, but you can replicate it using the anonymous function syntax (which we will talk about) and the `case` expression (which we won't).

Erlang also has an `if`, but it works more like guards in Haskell than the `if` from other languages we've looked at:

```
1> X = 3.
3
2> if
```

```
2>   X == 0 -> zero;
2>   X > 0 -> positive;
2>   true -> negative
2> end.
positive
```

The conditions are separated from each other with semicolons, and Erlang evaluates the expression by going through the conditions one by one until finding one that evaluates to `true`. This is why sticking `true` at the end allows you to catch any cases that don't get caught by anything higher; it acts like `otherwise` in Haskell's guards. Pay attention to the syntax here; it's easy to screw up.

Like the other languages we've seen, there are functions which take and return other functions:

```
1> lists:map(fun(X) -> X*X end, [0,1,2,3,4,5,6]).
[0,1,4,9,16,25,36]
2> lists:map(fun test:fact/1, [0,1,2,3,4,5,6]).
[1,1,2,6,24,120,720]
3> F = fun(X) -> -X end.
#Fun<erl_eval.6.13229925>
4> lists:map(F, [0,1,2,3,4,5,6]).
[0,-1,-2,-3,-4,-5,-6]
```

You can define anonymous functions this way using `fun`; don't forget the `end`. Also note the funny syntax for referring to compiled functions. The `lists` module has a lot of familiar functions for dealing with lists, like `filter/2` and `foldl/3`. Documentation can be found on the Internet.

You can use pattern matching to write functions that go through lists, and the list construction syntax to build lists recursively in functions. This works just like it does in the languages we've already examined, so we won't go into much detail about it here.

4.2 Concurrency

In this section we'll scratch the surface of how Erlang does concurrency. There are three pieces, and they're all very simple: one creates a new process, one sends a message to a process, and one receives a message. The first is the built-in `spawn` function, which takes a function as its parameter and starts a separate process to run it. It returns a *process ID*, a handle that can be used to refer to the process later on. (The definition of the function `other/0` will be discussed in a moment.)

```
1> c(test).
{ok,test}
2> P = spawn(fun test:other/0).
<0.40.0>
```

The `spawn` function can also take a list as a third parameter which contains any parameters you want passed to the function you're creating a process for. Then we can send messages to the other thread using the `!` operator:

```
3> P ! message.
Received a message that I wasn't ready for. Trying again.
message
4> P ! something.
Received message: something! Stopping.
something
5> P ! something.
something
```

(The messages look like they're being repeated because `P ! x` returns `x`, and the interpreter is displaying the return value.) So what happened here is that we sent the process the message `message`, then the message `something`, and once it received the second message, the process ended. Let's look at how the function `other/0` is implemented:

```
-module(test).
-export([other/0]).

other() ->
  receive
    something -> io:format("Received message: something! Stopping.~n
        ");

    something_else -> io:format("Received message: something_else!
        Stopping.~n");

    _ -> io:format("Received a message that I wasn't ready for.
        Trying again.~n"),
          other()
  end.
```

Processes can deal with messages using a `receive` block, which waits until a message is received by the process before evaluating anything. The syntax for `receive` looks a little like the definition of a function. The message can be any Erlang value at all, and the part before the `->` is a pattern that's matched against the message.

The function `io:format` takes a string and displays it; the `ñ` is a line break. You can string expressions together using `,`.

The messages, as we mentioned before, can be any Erlang expression at all, including something rather complicated. A common practice is to use this to include the process ID of the sender so that the receiving process can send a message back. Here's a trivial example:

```
% In test.erl
-module(test).
-export([oneadder/0]).

oneadder() ->
  receive
```

```
    {Source, X} -> Source ! X + 1
  end.

% In the interpreter.
1> c(test).
{ok,test}
2> Adder = spawn(fun test:oneadder/0).
<0.40.0>
3> Adder ! {self(), 5}.
{<0.33.0>,5}
4> receive X -> X end.
6
```

We spawn a process for the function `oneadder` and send it the number `5` together with the sender's process ID, which is the output of the built-in function `self`. This way, the function can send a message back to the original process.

Whenever a process sends a message to another process, Erlang adds the message to the receiver's *message queue*, and ordered list of messages that have to be received. The receiver doesn't wait for the sender to read the message, so in this last example, the spawned process ends right after it sends its message to `Source`. The message queue allows multiple messages to pile up before being dealt with; they'll just be received in the order they were sent:

```
% In test.erl
-module(test).
-export([onetwoadder/0, sender/0]).

onetwoadder() ->
  receive
    {Source, X} ->
      Source ! (X + 1),
      Source ! (X + 2)
  end.

getnum() ->
  receive
    X -> io:format("Received ~p.~n", [X])
  end.

sender() ->
  P = spawn(fun test:onetwoadder/0),
  P ! {self(), 4},
  getnum(),
  getnum().

% In the interpreter
1> c(test).
{ok,test}
```

```
2> test:sender().
Received 5.
Received 6.
ok
```

It's a good exercise to think about what would have happened if there were one more call to `getnum` in the body of `sender`, or if there were one fewer.

The thing that makes Erlang's concurrency so much easier to use than that of other languages is its commitment to the idea that there shouldn't be any mutable state. Whenever two different processes have access to the same mutable data, there are always conflicts about which process is able to change what when, and these conflicts are resolved through the use of "locks," which grant exclusive access to the data for as long as one process needs it. This in turn can create "deadlocks," where two process are both waiting for each other to allow them to proceed. By doing away with mutable state altogether — that is, by programming in a functional style — we can neatly avoid all of these problems.

5 Prolog

Prolog is a logic programming language. This means that constraint satisfaction is a basic feature of prolog. We will see how this works in practice by looking at examples.

You can start prolog by typing `prolog`. You can exit the interpreter by typing `halt.`. You will have to write the code in a separate file, and then load the file into the prompt by typing `consult('<filename>').`.

5.1 Facts and Rules

Let us create a sample file that contains some *facts*. A fact defines an n-ary relation. Relations are specified in our file one after the other. For example:

```
man(harry).
man(draco).
man(ron).
woman(hermione).
woman(ginny).
```

We have defined some 2-ary relations, as follows:

```
brother(fred, george).
sister(ginny, ron).
sister(ginny, fred).
sister(ginny, george).
```

After consulting the file in the interpreter, we can ask for facts. Evaluating each true fact outputs `true`. Moreover, we can also query relations by putting in variables for some of the arguments. (Remember that variables start with a capital letter, just like in Erlang.)

Here are some examples.

```
?- brother(fred,X).
X = george.
?- parent(X,fred), man(X).
X = arthur.
?- parent(X,fred), woman(X).
X = molly.
?- brother(ginny,X).
false.
```

Observe that prolog doesn't assign `X` to the value that satisfies the constraint. It simply tells you that some value satisfies the constraint. If nothing satisfies the constraint, it outputs `false`.

As you can see, we can try to satisfy multiple constraints by simply listing them in order. What happens if there is more than one solution to a constraint? The interpreter outputs a possible solution and waits for the user to either accept it or move to the next one. (Accept by pressing the return key and move to the next one by pressing the semicolon key.) If the user moves on, then prolog backtracks and tries to choose a different value.

```
?- parent(X,fred).
X = arthur ;
X = molly.
?- woman(X).
X = hermione ;
X = ginny ;
X = molly.
```

So far we only had a set of facts to work with. Now we define *rules*, which tell the interpreter how to combine the facts to get new facts. This is how relations are constructed. For example, we can write the following.

```
father(X,Y) :- parent(X,Y), man(X).
mother(X,Y) :- parent(X,Y), woman(X).
```

Like in Haskell and Erlang, the underscore _ is a wildcard that matches anything.

```
?- father(arthur,_).
true
```

Prolog has extensive pattern matching, just like Haskell and Erlang. There is a special operator, written = and pronounced unification, which carries out pattern matching without evaluating the arguments. Here are some examples.

```
?- X = 3.
X = 3.
?- a = a.
true.
?- a = b.
false.
?- X = a.
X = a.
?- a = X.
X = a.
?- X = 3+4.
X = 3+4.
```

Observe that atoms like `a` are only equal to themselves, and don't match any other atom. Also observe that = does not assign values to variables, since `X` did not get assigned to 3 after the first expression was evaluated.

Also observe the peculiarity that `X = 3+4.` outputs `X = 3+4.`, and not `X=7.`. This is because the unification operator does not evaluate its arguments. Pattern matching can be more complicated:

```
?- a(Y,Z) = a(3,4).
Y = 3,
Z = 4.
```

So why is unification useful if there is no assignment? In a composite prolog statement, prolog remembers the unification rules and then applies them. Here is an example.

```
?- X=Y, a(Z) = a(Y), X=2.
X = 2,
Y = 2,
Z = 2.
```

There is a special predicate called `is`, which behaves similar to the unification operator, but it evaluates its arguments. This will finally allow us to do arithmetic calculations in the interpreter.

```
?- X is 1+2.
X = 3.

?- X is (3 + 4)*9.
X = 63.

?- X = 3 + 'four'.
X = 3+four.

?- X is 3 + 'four'.
ERROR: is/2: Arithmetic: ‘four/0’ is not a function
```

Recall from the examples that prolog tries to find all possible matches to a rule query, one at a time. Hence there is a stack that keeps track of the matchings and constraints so far. As we match new variables, we push to the stack with the values of these variables. At the end, if we reject the match, then prolog backtracks by popping the stack and choosing a different option if possible.

Sometimes we would like prolog to stop trying to find more values of a variable. To prune the search tree at a point, the `!` operator (pronounced cut) is used. If we use a cut after an expression, then the first matched value will be bound to all variables for the duration of the current computation. Here is an example.

```
?- sister(ginny,X), !.
X = ron.

?- sister(ginny,X).
X = ron ;
X = fred ;
X = george.
```

In the first statement, the matcher stopped at the first match, while in the second statement, the matcher was willing to backtrack even after finding one match.

5.2 Functions, recursion, lists, etc.

Let us now try to define a "normal" function, such as the factorial. At this point we should say something about functions in prolog. Remember that the basic objects in prolog are relations. A function is a special kind of relation, and is treated no differently than a relation. Let us try to define a function called `add1` that adds 1 to its argument. Normally it would be defined by saying something to the effect of `add1 X = X + 1`. In prolog however, `add1` would be a 2-ary relation, where `add1(X,Y)` is true if and only if $Y = X + 1$. The definition is then as follows.

```
add1(X,Y) :- Y is (X+1).
```

Let us look at a more complicated example, which is written recursively.

```
factorial(0,1).
factorial(A,B) :-
    A > 0,
    C is A-1,
    factorial(C,D),
    B is A*D.
```

Observe that we had to introduce the local assignment of `C` to `A-1`. Typing `A-1` directly into the factorial does not work. In this version of factorial, the stack size grows linearly with the input. But we can write an iterative factorial which will make use of a tail call optimization, as follows.

```
ifact(X,Y) :- ifact-helper(X,1,Y).
ifact-helper(0,Y,Y).
ifact-helper(X,T,Y) :-
    C is X-1,
    D is X*T,
    ifact-helper(C,D,Y).
```

For example, we have the following.

```
?- ifact(4,X).
X = 24
?- ifact(5,120).
true
?- ifact(5,X).
X = 120
```

Prolog has good support for lists. You can extract the head and tail of a list simply by pattern-matching.

```
?- [X|_] = [1,3,4].
X = 1.
?- [_|T] = [1,3,4].
T = [3, 4].
```

Similarly there are rules called `append`, `member`, `takeout` that do the obvious things. There are also rules called `union` and `intersection`.