Haskell Lecture Notes

Zachary May

August 13, 2014

1 Introducing Haskell

- Haskell is a **statically-typed**, **non-strict**, **pure functional** programming language.
  - **Functional:** Conceptually, computation proceeds via the application of functions to parameters rather than by sequential instructions manipulating values in memory. Functions are first-class values.
  - **Pure:** The functions in question are more like mathematical functions than procedures. They map values in an input domain to values in an output domain. Pure functions have no side effects. These features are valuable when reasoning about and testing our code.
  - **Non-strict:** By default, Haskell uses a lazy evaluation strategy. Expressions do not need to be evaluated until the results are needed. For example, Haskell can cleanly represent infinite lists because the language only evaluates such an expression as needed.
  - **Statically-typed:** The type of every expression is known at compile time, preventing run-time errors caused by type incompatibilities. This feature prevents things like Java's `NullPointerException`, because a function that claims it returns a value of a specific type must live up to that promise. Returning the Haskell equivalent of `null` is a compile-time type error. Additionally, Haskell makes

use of a technique called **type inference** to figure out the types of most things without needing explicit type annotations.

2 A Sample Program

```
sumSquares :: Int -> [Int] -> Int                    1
sumSquares count numbers =                           2
    sum (take count (map (^2) numbers))              3
                                                     4
printSquares :: IO ()                                5
printSquares =                                       6
   print $ sumSquares 10 [1..]                       7
```

- The type signature in line 1 describes a function with two parameters: an `Int` and a list of `Int`s. It returns an `Int`. In general, the type after the last `->` is the return value, and the others are the parameters.
- Haskell would actually infer a more general type than what we see in the type annotation on line 1. Haskell programmers usually include type annotations as a form of machine-checked documentation.
- Haskell functions are defined with an equational syntax as seen in lines 2-3: `sumSquares` applied to the parameters `count` and `numbers` is equal to the expression on the right-hand side of the equation.
- Line 3 shows us several examples of **function application**. The notation for function application is lightweight: simple juxtaposition of terms. The parentheses here are for grouping only.
- Working from the inside out, `map` applies a function to each element in a list, producing a new new list containing the resulting values.
- `(^2)` is called a **section**, a shorthand for an anonymous function whose parameters fills in the blank for a binary operator, exponentiation in this case. So `map (^2)` transforms a list of numbers into their squares.
- Given a number n and a list, `take` returns the first n items of the list, or the whole list if it has fewer than n elements. `sum` takes a list of numbers and returns the sum.
- Lines 5-7 define another function, `printSquares`. It takes no parameters, and its return type, `IO ()`, is quite interesting.

- Look back at the type `[Int]`. The list type `[]` is a **parameterized** type. That is, list itself is not a concrete type, but a **type constructor**. We need another type, the parameter, to create a concrete type. Similarly, `IO` is a type constructor.
- Here we instantiate `IO` with the type `()`, the **unit type** with only a single value, also written as `()`. The type `IO ()` represents an **I/O action** with no interesting return value. The `IO` type constructor is the technique Haskell uses to perform I/O, inherently impure, in a world of pure functions, via a far more general abstraction called the **monad**.
- Consider the expression `[1..]`. This expression represents the infinite list of positive integers. However, because Haskell evaluates expressions lazily, merely describing the value is not enough to force the run-time to evaluate the entire thing. In fact, in this case, evaluation of `[1..]` is only forced by `print` asking for the value produced by `sum`, which demands the values yielded by `take count`, and so on, on demand.
- Notice the `$` in line 7. Pronounced "applied to", it is an operator with extremely low precedence that breaks up the very high precedence of function application, allowing us to avoid nesting parentheses. We could have defined `sumSquares` using `$` as: `sum $ take count $ map (^2) numbers`.

3 Basic Haskell Types

In this section, we take a deeper look at Haskell's type system.

- Haskell offers basic primitive data types: [1]
  - `Int` and `Integer`: machine-sized and arbitrary precision integers, respectively.
  - `Float` and `Double`: single- and double-precision floating point numbers.
  - `Char`: Single Unicode characters, implemented in GHC as 31-bit ISO 10646 characters.
  - `Bool`: Boolean values (but see below; `Bool` is actually a composite type).

- Additionally, we have several composite data types:
  - Linked lists: . The type is written `[a]` where `a` is another type.
  - `String`: character strings; `String` is literally just a **type synonym** for `[Char]`.
  - Tuples: k-element tuples, $k \geq 2$. The type of an n-element tuple is `(a1, a2, ..., an)` where `a1`, `a2`, ..., and `an` are other types.
- Programmer-defined data types are introduced with the `data` keyword.
- Type synonyms can be introduced with the `type` keyword. `type EmailAddress = String` says that the identifier `EmailAddress` can be used interchangeably with the type `String`. The advantage is that `EmailAddress` is more descriptive.
- The keyword `newtype` creates a more controlled sort of type synonym.
  - If we want a type to describe e-mail address values but do not want it to be interchangeable with `String`s in general, we can define a new type that simply tags a `String` value: `data EmailAddress = EmailAddress String`.
  - Wrapping the `String` in the `EmailAddress` wrapper comes with some amount of syntactic and run-time overhead each time we want to access the underlying `String`.
  - Instead we can use `newtype EmailAddress = EmailAddress String`. Haskell's type checker treats this type exactly like a type introduced with the `data` keyword, but the compiler generates code that disregards the wrapper, as though we had just used a type synonym.
- Although Haskell can infer the types of most all expressions, types can be stated explicitly with a type annotation using `::`. For example:

```
nothing :: [String]                                    1
nothing = []                                           2
                                                       3
moreNothing = []                                       4
```

  - We define two values, `nothing` and `moreNothing`.
  - Although the equational definitions are identical, we explicitly define the type of `nothing` to be of type `[String]`, a list of strings, with the type annotation on line 1.

    - Without an explicit type annotation, Haskell infers the most general type of `moreNothing`, `[a]`. (`[a]` is a **polymorphic type** which we will discuss later.)

- Haskell's lists and tuples are specific examples of the language's **algebraic type system**.
- Algebraic data types were introduced in the Hope programming language in 1980. [3]
- An algebraic type system generally offers two sorts of types:
    - **Product types**: A data type with one or more fields.
        - Also referred to as a Cartesian product.
        - Tuples are the archetypal product type.
        - The cardinality of a product type is the product of the sizes of the types of its fields.
        - E.g., `(Bool, Bool)`, the type of 2-tuples of two Boolean values, has a total of $2 \times 2 = 4$ possible values.
    - An example:

```
data DimensionalValue =                          1
    DimensionalValue Float Dimension             2
```

        - `DimensionalValue` represents a `Float` value tagged with a unit of measure of type `Dimension`. We will see later how we might describe that type.
        - The `data` keyword introduces a data type definition.
        - The identifier before the `=` is the name of the new type.
        - Following the `=` are the type's constructor definitions. The first identifier is the **data constructor**, followed by arbitrarily many field declarations.
        - By convention, if there is only one data constructor, it has the same name as the type itself.
        - Our `DimensionalValue` type is equivalent to a tuple `(Float, Dimension)`, but we have given it a distinct and descriptive name.
    - **Sum types**: A data type with one or more alternatives.
        - Also referred to as a disjoint union.
        - Enumerations are the archetypal sum types
        - An example:

```
data Dimension = Seconds
               | Meters
               | Grams
```

- As before, `data` introduces a new type, here named `Dimension`.
- The vertical pipe `|` separates the various alternatives.
- Each alternative is given as a data constructor definition as described above. Here we define three simple data constructors, `Seconds`, `Meters`, and `Grams`.
- These identifiers can be used as literal values of the type `Dimension`.

- The cardinality of a sum type is the sum of the sizes of the types of its alternatives.
- The type `Dimension` has $1 + 1 + 1 = 3$ possible values.
- Haskell's `Bool` data type is defined as a sum type with data constructors `True` and `False`.

- The power of algebraic data types comes when we combine the two: sums of products and products of sums:

```
data PlaneTicket
    = PlaneTicket Section MealOption

data Section | Coach
             | Business
             | FirstClass

data MealOption = Regular
                | Vegetarian

data TravelDetails = Train
                   | Automobile
                   | Plane PlaneTicket
```

- Here we define several types that might describe the domain model of a travel-agency application.
- `PlaneTicket` is a product type over two sum types: the section (`FirstClass`, `Business`, or `Coach`) and the meal option (`Regular` or `Vegetarian`).
- `TravelDetails` is a sum type over two singleton data constructors `Train` and `Automobile` and a unary product alternative that tags `PlaneTicket` details with the data construc-

tor `Plane`
- How many possible values are there for the `TravelDetails` type? $1 + 1 + 1 \times (3 \times 2) = 8$.

4 Polymorphic types

- Earlier, we described the list and tuple types in terms of other, unspecified data types:
  - `[a]` is the type of lists with elements of some type `a`.
  - `(a, b)` is the type of 2-tuples with first element of some type `a` and second element of some type `b`.
- Here, `a` and `b` are **type variables**.
- Lexically, type variables must begin with a lowercase letter. Concrete data types and data constructors must begin with an uppercase letter.
- Data types that contain type variables are called **polymorphic types**.
- This type of polymorphism is known as **parametric polymorphism**: substituting the concrete type `Char` for the **type parameter** `a` in `[a]` gives the **concrete type** `[Char]`.
- Parametric polymorphism is distinct from the **inclusion polymorphism** seen in object-oriented programming.
- This example shows how we might implement our own linked-list and 2-tuple types:

```
data List a = Nil                                    1
            | Cons a (List a)                        2
                                                     3
data Pair a b = Pair a b                             4
                                                     5
data UniformPair a = UniformPair a a                 6
```

  - Introducing type variables on the left-hand side of the `=` indicates that we are defining a polymorphic types. `List` is parametric in a single type variable `a` and `Pair` is parametric in two type variables, `a` and `b`.

- The two type variables called `a` in the definitions of `List` and `Pair` are distinct. That is, the scope of a type variable is a single `data` definition.
- What is the difference between our definition of `Pair` and `UniformPair`? `UniformPair` is parametric in only one type variable so both of its elements must be of the same type.
- We see that `List` is a sum of products: A `List` containing elements of type `a` is either the empty list `Nil` or it is a value of type `a` followed by another `List` containing elements of type `a`. Thus, `List a` is a recursively-defined data type.
- Let us also make a distinction here between:
  - a concrete type, like `List Integer` or `(String, Dimension)` that has no type variables;
  - a polymorphic type like `List a` that has one or more type variables;
  - a type constructor like `List` that, if applied to a concrete type, yields a concrete type, and if applied to a type variable yields a polymorphic type.
    - Data constructors are applied to values to produce new values. For example, `Cons` must be applied to a value of type `a` and a value of type `List a` to produce a value of type `List a`. `Nil` is a **nullary data constructor** is already a value of type `List a` any more value
    - Analogously, type constructors are applied to types to produce new types. For example, `List` applied to the type `Integer` produces the type `List Integer`.

5 Function Types

- The examples we have looked at so far are for the types of values. However, Haskell supports **first-class functions**: functions can be passed as parameters into functions and be be returned as the result of a function.
- That is to say, in Haskell, functions **are** values. So how do we describe their types?
- First, we never actually define new function types with `data`, although we can define synonyms for function types with `type`.

- The one true function type constructor is `->`, as in `a -> b`, the polymorphic type of functions with domain `a` and co-domain `b`.
  - What does the function type `a -> a` represent? Functions with identical domain and co-domain.
  - With no other information about the type `a`, what sort of function can have the type `a -> a`? The identity function.
- The functions described by `->` appear to only have one parameter, the type on the left of the `->`. Haskell has operations (read: functions) like addition that take two parameters, so how can we describe the type of such a function?
- Recall that functions can return other functions as their result. Haskell models multi-parameter functions with single parameter functions that return a new function ready to consume more parameters. This technique is called **currying**, named for the logician Haskell Curry.

```
add x = \y -> x + y
```

  - We define `add` as a function that takes a single parameter `x`.
  - It returns an anonymous function, introduced by `\` (meant to suggest the Greek λ). Its parameter is called `y`. The result of this anonymous function is the sum of `x + y`.
  - When calling `add`, the actual parameter provided for the formal parameter `x` is preserved in a **closure** that, along with the body of the anonymous function, makes up the function value we return.
- Haskell does not actually inconvenience us by requiring this notation. We can just define `add` as:

```
add x y = x + y
```

- However, Haskell really is using currying under the hood. As such, we can **partially apply** functions. Even with the simple definition, `add 5` is not an error, it returns a function value ready to accept another parameter and add it to 5.
- Now the type of `add` should be more clear. Assuming we are only adding `Integers`, it must be `Integer -> (Integer -> Integer)`.
- `->` is right associative, so we can simplify this to just `Integer -> Integer -> Integer`.

- In this form, we can view the type after the last `->` as the return type of the function and all the other types as the types of the function's parameters.
- We still need parentheses for grouping if one of the parameters is a function:
  - Consider the function `map :: (a -> b) -> [a] -> [b]`.
  - What are the types of the parameters and return value of `map`? The first parameter is a function with domain `a` and co-domainb. The second parameter is a list of `a`s. The result is a list of `b`s.
  - How is that different from `map' :: a -> b -> [a] -> [b]`? `map'` takes three parameters (an `a`, a `b`, and a list of `a`s) and returns a list of `b`s.
  - To what extent can you infer the semantics of `map` from its type alone?
- In general, we call functions that have one or more functions as their parameters or that return functions as their result **higher-order functions**. As we will see, they central to more advanced techniques in functional programming.

6 Ad-Hoc Polymorphism with Type Classes

- At the machine level, adding two integers is quite a different operation from adding two floating-point numbers. High-level languages, in an effort to hide this detail, **overload** the semantics of the addition operator to support these distinct operations using the same operator.
- In a strongly-typed language like Haskell, what might the type of `(+)` be?
  - In Haskell, infix binary operators are just syntactic sugar for functions of two parameters. When referring to binary operators outside of their normal infix notation, Haskell requires them to be surrounded by parentheses.
- `Int -> Int -> Int` or similar is insufficient, since the type of `(+)` needs to be general enough to describe adding together two operands of many different numeric types.

- `a -> a -> a` seems promising: two operands and a result, all of the same type. However, this type signature **unifies** with `TravelDetails -> TravelDetails -> TravelDetails` and addition of that type does not make sense.
- What happens if we ask the Haskell compiler to infer the type of `(+)`?
  - The most popular Haskell compiler, GHC, has a REPL interface called GHCi.
  - The command `:t` *`expression`* asks GHCi to infer the type of *`expression`*.
  - `:t (+)` yields the inferred type: `(Num a) => a -> a -> a`.
- `(Num a) => ...` is a **class constraint** on the type signature that follows the `=>`.
- Normally, a free type variable in a type signature can be unified with any type at all.
- A class constraint on a type variable restricts the types that it can be unified with to types that are **instances** of the named **type class**.
- So `(+) :: (Num a) => a -> a -> a` says that `(+)` is a function of two operands and result all of some type `a` where `a` is an instance of the type class `Num`.
- A type class acts a bit like a **interface** in object-oriented programming. It acts as a contract: any type that is an instance of a type class must implement certain methods to qualify.
- The terminology may be a bit confusing:
  - In an OO language, an object is an instance of a class which might implement an interface.
  - In Haskell, a value has a type which might be an instance of a type class.

7 Basic Typeclasses from The Haskell Prelude

Haskell offers quite a bit of functionality in its standard library. In particular, the **Prelude**, the set of type and function definitions imported automatically

into every program, defines

One of the most basic typeclasses is `Eq`, consisting of types that implement an equality-testing operation. Here is how it is defined:

```
class Eq a where
      (==), (/=)  ::  a -> a -> Bool

      x /= y  = not (x == y)
      x == y  = not (x /= y)
```

- The `class` keyword introduces a type class definition, followed by the name of the type class and a type variable that we will use in the description of the type class's interface. Think of this type variable as a formal parameter in a function definition.
- The `Eq` type class defines two required operations, equality and inequality. In this case, the two have exactly the same signature, so the type annotation is shared.
- If we asked Haskell to infer the type of `(==)`, what would we get? `(==) :: (Eq a) => a -> a -> Bool`.
- Then we see two equational function definitions. These are default implementations for `Eq`'s operations.
- This means we do not have to define both `(==)` and `(/=)`. Each is defined in terms of the other, so an implementation for one is enough. The compiler will complain if neither is implemented.
- Because we can define default implementations, type classes are actually more like **abstract classes** in OO languages.

Typeclasses can themselves have class constraints. Here is the definition of `Ord`, which describes operations available for totally ordered data types:

```
class Eq a => Ord a where
    compare               :: a -> a -> Ordering
    (<), (>=), (>), (<=)  :: a -> a -> Bool
    max                   :: a -> a -> a
    min                   :: a -> a -> a

data Ordering = LT | EQ | GT
```

- Class constraints in a type annotation, as in line 1 above, require that the constrained type variable be an instance of the given typeclass.
- In this case, for a type to be an instance of `Ord`, it must also be an instance of `Eq`. It should be pretty clear why that is necessary.
- The full definition of `Ord` gives default implementations for all these operations so that an instance need only implement either `compare` or `(<=)`.

There are several other typeclasses worth mentioning:

- `Show` instances can be turned into a `String` representation with `show :: (Show a) => a -> String`.
- `Read` instances know how to undo the process and turn a string into a value.
- `Bounded` instances are types with a smallest and largest values, given as two polymorphic constants `minBound, maxBound :: (Bounded a) => a`.
- `Enum` instances are sequentially ordered types. Given a value in that sequence, we can use `succ, pred :: (Enum a) => a -> a` to get the next or previous value. We can use `enumFromTo :: (Enum a) => a -> a -> [a]` to get a list containing the elements in the sequence between a start value and an end value, inclusive.

Haskell's standard library also defines a hierarchy of numeric typeclasses.

- We saw that GHC would infer the type `(Num a) => a -> a -> a` for the `(+)` operator. That, along with `(*)`, `(-)` (the binary subtraction operator), `negate` (for unary negation), and a couple of others define the most basic interface for numeric types.
- `Fractional` extends `Num` with division in `(/)` and reciprocation in `recip`.
- `Floating` extends `Fractional` with real-valued logarithms, exponentiation, trigonometric functions and even `(Floating a) => pi :: a`, the polymorphic constant π.

The full numeric hierarchy is even richer and there is plenty of detail in the Prelude's typeclasses that we have glossed over. Full details are available in [1, section 6.4].

8 Creating New Typeclass Instances

Haskell typeclasses are **open**, meaning that we can define new instances of typeclasses defined in the Prelude or elsewhere.

Let's see how we can implement some of the Prelude's basic typeclasses for a simple type.

```
data Section = Coach
             | Business
             | FirstClass

instance Eq Section where
    FirstClass == FirstClass = True
    Business   == Business   = True
    Coach      == Coach      = True
    _          == _          = False

instance Ord Section where
    x          <= y          | x == y = True
    Coach      <= _                  = True
    Business   <= Coach              = False
    Business   <= FirstClass         = True
    FirstClass <= _                  = False

instance Show Section where
    show Coach      = "Coach"
    show Business   = "Business"
    show FirstClass = "FirstClass"
```

- An instance declaration begins with the keyword `instance`, followed by equational definitions for the various functions defined for the class.
- The definition of `(==)` for `Eq` is straightforward. We define the function

in four cases. In the first three equations, we enumerate the cases where values could be considered equal and the last equation is a catch-all: the underscore character matches any value, so any case not matched by the first three equations will get caught by the fourth and will return `False`.

- We define an `Ord` instance by enumerating the ways in which `Section` values can be ordered. The first equation uses a **guard**: `x` and `y` will match any values, but the match is only successful if the **guard expression** evaluates to `True`.
- The `Show` instance is trivial: we simpliy define a string value to return for each of `Section`'s three data constructors.

We can imagine that defining instances for these typeclasses would be quite similar for any algebraic data type. It seems trivial to automatically construct an `Eq` instance for any simple sum type. Furthermore, because of the recursive nature of algebraic data types, it would be easy to extend that idea to arbitrary sum-of-products types.

```
data S = P1 T1_1 ... T1_K1                                                    1
       | P2 T2_1 ... T2_K2                                                    2
         ...                                                                  3
       | PN TN_1 ... TN_KN                                                    4
                                                                             5
instance Eq S where                                                          6
    P1 u1_1 ... u1_k1 == P1 v1_1 ... v1_k1 = u1_1 == v1_1 && .. && u1_k1
    P2 u2_1 ... u2_k2 == P2 v2_1 ... v2_k2 = u2_1 == v1_1 && .. && u2_k2
    ..                                                                       9
    PN un_1 ... uN_kN == PN vN_1 ... v1_kN = uN_1 == vN_N && .. && uN_kN
    _ == _ = False                                                           11
```

- This is pseudocode for the general form of an `Eq` instance for a sum of N constructors that are each a product of K_N values.
- In words, two `S` values are equal if their data constructors are equal and each pair of constituent values are equal.

In fact, Haskell offers the ability to **derive** typeclass instances, and not just for `Eq`.

```
data Section = Coach                                    1
             | Business                                 2
             | FirstClass                               3
             derving (Eq, Ord, Show)                    4
```

- The `deriving` keyword instructs the compiler to automatically derive instances for the typeclasses that follow.
- The Haskell 98 standard can derive instances for `Eq`, `Ord`, `Enum`, `Bounded`, `Show`, and `Read`.
- In automatically derived instances of `Ord`, `Enum`, and `Bounded`, the order of declaration of the data constructors is used. So our derived `Ord` instance for `Section` still returns `True` for `Coach <= FirstClass`.

Because the derived definitions are recursive, we might not always be able to derive instances when the constituents of product types do not support the operations we need:

```
data Section = Coach                                    1
             | Business                                 2
             | FirstClass BeverageOption                3
             derving (Eq, Ord, Show)                    4
                                                        5
data BeverageOption = Wine                              6
                    | Beer                              7
                    | Soda                              8
```

- Here, we cannot automatically derive an `Eq` instance for `Section` because `BeverageOption` is not an instance of `Eq`. We cannot determine if two values using the `FirstClass` constructor are equal because we have no way of checking two `BeverageOption` values for equality.
- Similarly, we could not derive an `Ord` instance since we have no notion of ordering on `BeverageOption`s.
- In this case, adding a `deriving` clause to our definition of the `BeverageOption` type would resolve the problem.

9 Maybe, Lists, and The Functor Typeclass

Now let us consider a type defined in the Haskell Prelude, `Maybe`:

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

- `Maybe` is the Haskell version of the **option type**. It offers us a way to represent a value that might not exist. `Nothing` is the null value, and the `Just` constructor wraps an actual value.
- For example, we might want a function that parses an integer value from a string to have the return type `Maybe Integer`, since the parse might fail.
- Compare this to Java's type system where `null` is a possible value for any reference type. `null` is a `Person` even though `null` does not respond to any of `Person`'s methods–or any methods at all!
- In Haskell, on the other hand, a function that claims to return a `Person` always returns a full-fledged `Person` (barring exceptional failure) and a function that sometimes returns a `null`-like value must declare that in its type, e.g., `String -> Maybe Integer`.

The safety we get when we use an option type is nice, but it comes with some inconvenience: If I have a value of type `Maybe Integer`, how do I add five to it? In general, how do I unwrap a value of the form `Just x` to get at `x`? It is not difficult in principle:

```
parseInteger :: String -> Maybe Integer
# Implemented elsewhere

example1 :: String -> Integer
example1 str = case parseString str of
                  Just x  -> x + 5
                  Nothing -> 0

example2 :: String -> Maybe Integer
example2 str = case parseString str of
                  Just x  -> Just (x + 5)
                  Nothing -> Nothing
```

- However, this code has some shortcomings:
  - In `example1`, we are making an assumption about how to handle the error case (returning zero if the parse failed) that is now interwoven with the independent process of adding five.
  - Both examples repeat the code to test both the `Just` and `Nothing` case. Moreover, if we used `Maybe` frequently (which is encouraged), this would start to get rather annoying.

We will reject `example1` because we really would like to maintain the orthogonality of dealing with `Maybe` values and our actual operation. However we can use higher-order functions to factor out repetition we see in analyzing `Maybe`s.

```
applyToMaybe :: (a -> b) -> (Maybe a) -> (Maybe b)          1
applyToMaybe f Nothing  = Nothing                           2
applyToMaybe f (Just x) = Just (f x)                        3
```

- `applyToMaybe` factors out handling the `Nothing` and `Just` cases of `Maybe` values.
- We also get some vocabulary for lifting normal function application into the world of `Maybe` values.
- In our add five example, we can now just use `applyToMaybe (+5) $ parseString str`

Here is another example. We have seen Haskell's basic, homogeneous list type. It offers us a way to represent a collection of zero or more values of some type.

We have also seen the function `map :: (a -> b) -> [a] -> [b]` that applies a function to each element in a list and returns the results collected into a new list.

```
map :: (a -> b) -> [a] -> [b]                               1
map f []     = []                                           2
map f (x:xs) = f x : (map f xs)                             3
```

If we compare `applyToMaybe` and `map`, we see some important similarities:

- Both functions have an empty case and a case where one or more values are unwrapped, a function applied, and the result(s) wrapped back up.
- If we ignore the special case of Haskell's list type syntax, the functions have analogous types of the form `(a -> b) -> f a -> f b`

9.1 The Functor Typeclass

In fact, this pattern is codified in Haskell with the `Functor` type class:

```
class Functor f where                                    1
    fmap :: (a -> b) -> f a -> f b                       2
```

- A `Functor` instance is always a polymorphic data type with a single type parameter.
- At one level we can think of `Functor`s as simply mappable containers.
- At another level, we can think of them as values in some sort of context, where `fmap` lifts function application into that new context.

9.1.1 Functor Instances

Here are some instances of the `Functor` type class:

- `Maybe`
  - We can think of `Maybe a` as a context representing a value of type `a` with the possibility of failure.
  - In this context, we can think of `fmap` as creating new functions that know how to propagate these failure states.
- **Lists, i.e., `[]`**

  - The implementation of `fmap` for lists is literally just the Prelude's `map` function.
  - From the context perspective, we can think of lists as non-deterministic values; i.e., the result of applying the function `(* 2)` to the non-deterministic value that might be one of `[1, 2, 3]` would be the non-deterministic value that might be one of `[2, 4, 6]`.

- `Tree`
  - Mapping over a collection makes sense for trees, but what might `Tree` represent from the perspective of values in a context?
  - Interestingly, while mapping over elements in a set seems reasonable enough, Haskell's `Set` type cannot be directly declared an instance of `Functor`. Because `Set` is implemented via balanced binary trees, it has an `Ord` constraint on the types it can contain. This extra constraint is incompatible with the general `Functor` definition; we would need `fmap` to have the type `(Ord a, Ord b) => (a -> b) -> f a -> f b`.
  - `Map`, however can be a `Functor`. Rather, maps with keys of type `k`, i.e., `Map k` can be a `Functor`. Haskell `Map`s are represented using balanced binary trees over the key type `k`, so there is still an `Ord` constraint, `Functor` cares about the type of the values, not the type of the keys.

- `((->) e)`
  - This type looks a big strange. Haskell's syntax does not allow it, but read this type as `(e ->)`.
  - Concretely, the type of the `fmap` implementation here would be `(a -> b) -> (e -> a) -> (e -> b)`
  - [5] describes `((->) e)` as "a (possibly infinite) set of values of `a`, indexed by values of `e`," or "a context in which a value of type e is available to be consulted in a read-only fashion."
  - If we have a predicate `isOdd :: Int -> Bool`, and `fmap` it over a function `length :: String -> Int` that returns the length of its parameter, we get a new function of type String -¿ Bool that returns whether or not the `String`'s length is odd.
  - From the context perspective, `fmap isOdd` takes us from `Int`s indexed by `String`s to `Bool`s indexed by `String`s.

9.1.2 Functor Laws

For the Haskell type system, anything that implements `fmap :: (a -> b) -> f a -> f b` is perfectly suitable as an instance of `Functor`. However, the concept of functors come to us from the branch of mathematics called category theory, where functors must satisfy certain laws. In Haskell terms:

```
     fmap id == id                                    1
fmap (g . h) == (fmap g) . (fmap h)                   2
```

- Mapping the identify function over the contents of a `Functor` just gives back the original `Functor`.
- `fmap` distributes over function composition.
- Ultimately, these two laws just mean that a well-behaved `Functor` instance only operates on the contents of the `Functor`, leaving its structure unchanged.

Consider this badly-behaved instance definition for lists taken from [5].

```
instance [] where                                     1
    fmap g []     = []                                2
    fmap g (x:xs) = g x : g x : fmap g xs             3
```

- This implementation of `fmap` duplicates all the output values: `fmap (+1) [1, 2, 3]` returns `[2, 2, 3, 3, 4, 4]`.
- The first law is broken because `fmap id [1, 2, 3]` returns `[1, 1, 2, 2, 3, 3]` rather than `[1, 2, 3]`.
- The second law is broken because `fmap ((+1) . (*2)) [1,2,3]` returns `[3, 3, 5, 5, 7, 7]` rather than `[3, 5, 7]`.

Although Haskell's type system is quite powerful, it is in general undecideable whether a `Functor` instance satisfies the two laws described above, so that requirement cannot be checked at compile time. Since other Haskell code in the standard libraries and elsewhere will expect new `Functor` instances to

be well-behaved, it is the responsibility of the programmer to prove, at least to their own satisfaction, that their implementation satisfies those laws. We will look at more typeclasses later on, each with their own laws, and this caveat applies to them as well.

10 Applicative Functors

When we looked at the `Functor` type class, we saw in `fmap` a way for us to lift normal functions into the domain of computational contexts, the `Functor` instances, where they can operate on values in those contexts.

But recall that in Haskell, functions are themselves first-class values. So how do we use a function that is itself in a computational context? This question is answered by the concept of applicative functors, realized in Haskell with the `Applicative` type class.

Consider this scenario. We are given an `Integer` with the possibility that it might not actually be there, i.e., `Maybe Integer`. We are also given a function to apply to that value, again with the possibility that it might not actually be there, i.e., `Maybe (Integer -> Integer)`. Of course, since either the function or the parameter might be `Nothing`, the return value needs to be able to propagate this possibility of failure. How would we accomplish that?

```
maybeApplyToMaybe :: Maybe (Integer -> Integer)          1
                  -> Maybe Integer                       2
                  -> Maybe Integer                       3
maybeApplyToMaybe (Just f) (Just x) = Just $ f x         4
maybeApplyToMaybe _ _ = Nothing                          5
```

There are three cases:

- When we get both a function and an parameter, we can unwrap them from their `Maybe` wrapper, apply the function to the parameter, and return the result wrapped back up.

- In any other case, either the function or its parameter is `Nothing`, so we return `Nothing`.

What happens if we want to use a function with two or more parameter in this way? We do not actually have to write any more code: Haskell's default of curried functions gives us native partial application of functions and `maybeApplyToMaybe` gets that for free.

Say we have a function `add :: Integer -> Integer -> Integer`. The expression `add 5` has type `Integer -> Integer`.

If we look back at line 4 of the previous example, we apply our unwrapped function to the unwrapped parameter, and return the result wrapped back up. Since the partial application `add 5` returns a function of type `Integer -> Integer`, `maybeApplyToMaybe (Just add) (Just 5)` returns a value of type `Maybe (Integer -> Integer)`.

10.1 The `Applicative` Type Class

We saw in the previous section that `map` for lists and `applyToMaybe` shared a common pattern, which led us to the general `Functor` type class. The same idea works here, and the resulting type class is called `Applicative`, short for **applicative functor**.

```
class Functor f => Applicative f where                    1
    pure  :: a -> f a                                     2
    (<*>) :: f (a -> b) -> f a -> f b                     3
```

The `Applicative` type class has its own class constraint: every instance of `f` of `Applicative` must also be an instance of `Functor`. In fact, most of the standard library's `Functor` instances are also `Applicative`s as well.

Importantly, though, `Applicative` and its functions are not included in the Prelude and must be imported manually from the `Control.Applicative` module.

10.2 `Applicative`'s Functions

If we look back at `maybeApplyToMaybe :: Maybe (Integer -> Integer) -> Maybe Integer -> Maybe Integer`, we see simply a monomorphic instance of the more general, polymorphic type of `(<*>)`. In fact, Haskell would have inferred a more general type for `maybeApplyToMaybe`: `Maybe (a -> b) -> Maybe a -> Maybe b` and our implementation is essentially the standard library's implementation of `(<*>)` for `Maybe`:

```
instance Applicative Maybe where                    1
    pure x             = Just x                     2
    Just f <*> Just x = Just $ f x                  3
    _ <*> _            = Nothing                    4
```

That means we can scrap our 17 character function name and rewrite `maybeApplyToMaybe (Just (+5)) (Just 2)` as `Just (+5) <*> Just 2`.

In general, `(<*>)` is the function (used infix like an operator) that takes a function in some `Applicative` context `f` and a value in the same context and handles the plumbing of unwrapping the function and the value, applying the function to the value, and returning the result wrapped back up in the `f` context.

We have not said much about the other half of the `Applicative` class, but it is quite simple and the type is very telling. What makes sense for the type `a -> f a`?

We are getting a value of any type and returning a value of that type in the context described by `f`. Without knowing anything about the type `a` of the parameter, we cannot modify it in any way. So, from the type alone, we can surmise that `pure` probably injects a value into the context described by `f` in some default way.

What might `Maybe`'s implementation of `pure` look like? It is literally just `Just`!

10.3 Building a Better `fmap`

It might seem that all we really get from `Applicative` is a way to factor out the unwrapping required to apply a function in a context to a value in a context, but there is more here.

Consider the operator `(<$>)` provided by the `Control.Applicative` module. It that takes a function, injects it into `f` with `pure`, and then uses `(<*>)` to apply it to a value in `f`.

```
(<$>) :: (a -> b) -> f a -> f b
f <$> v = pure f <*> v
```

The type of `(<$>)` should look familiar: it is the same type as `fmap`. In fact, assuming both `Functor` and `Applicative` laws (which we will see in a moment) the two are synonyms: `g <$> pure x == fmap g $ pure x`. The infix version is only introduced for stylistic reasons, suggesting the relationship with `(<*>)`.

What we really get out of `Applicative` is a better version of `fmap`.

Suppose we called `fmap (+) (Just 3)`. This is not an error, we will just partially apply `(+)` to the wrapped value `3` and get back basically `Just (3+)`, a function value inside a `Maybe` context, exactly where we were at the beginning of this section.

Now that we have seen how `Applicative`s work, we have the tools needed to finish `fmap`ping a function with two parameters: `fmap (+) (Just 3) <*> Just 5`, in `Applicative` terms: `pure (+) <*> Just 3 <*> Just 5`, or even more idiomatically: `(+) <$> Just 3 <*> Just 5`.

From a practical perspective, we could write a function that attempted to build a `PlaneTicket` value based on optional `Section` and `MealOption` values:

```
createPlaneTicket
    :: Maybe Section
    -> Maybe MealOption
    -> Maybe PlaneTicket
```

```
createPlaneTicket section meal =                                    5
    PlaneTicket <$> section <*> meal                                6
```

`createPlaneTicket` uses the language of `Applicative` to succinctly lifts the `PlaneTicket` data constructor into the `Maybe` context where the `MealOption`s and `Section`s might not exist.

In fact, the type that Haskell would actually infer for `createPlaneTicket` is `(Applicative f) => f Section -> f MealOption -> f PlaneTicket` and would work for any `Applicative`, such as lists, which we will take a look at shortly.

10.4 `Applicative` Laws

There are four laws that `Applicative` instances should follow, with the same motivations and caveats described when we discussed the `Functor` laws.

- **Identity**: `pure id <*> v == v`
- **Homomorphism**: `pure g <*> pure x == pure (g x)`
- **Interchange**: `g <*> pure x == pure ($ x) <*> g`
- **Composition**: `g <*> (h <*> k) == pure (.)  <*> g <*> h <*> k`
- **Functor Instance**: `fmap g x == pure g <*> x`

The **identity law** can be thought of as putting an upper bound on what can actually happen inside the plumbing of the `Applicative` implementation. If that plumbing does anything that fails to preserve identity, it is not a proper `Applicative`.

The intuition behind the **homomorphism law** is that these operations are just lifting function application into `Applicative` contexts. If we have a function `f` and a value `x`, inject each into the context via `pure` and apply them via `(<*>)`, we should get the same thing as if we had injected `f x` into the context directly.

The **interchange law** is a bit tricky. To start, remember that the `($)` operator is just function application with a very low precedence. So `($ y)` is a function that takes a function and applies it to `y`. What the interchange law is trying to express is that the order in which we evaluate the function and its parameter should not matter in a proper `Applicative` instance.

We can think of the **composition law** as formalizing an associative property for `(¡*¿)` in terms of Haskell's standard function composition operator `(.)`. Here, `g :: f (b -> c)` and `h :: f (a -> b)` are two functions inside an applicative functor `f` and `k :: f a` is a value in the same applicative functor.

Finally, the **functor instance law** describes how an `Applicative` instance should behave relative to its `Functor` instance and is required for the equivalence between `(<$>)` and `fmap` to hold for an `Applicative` instance.

[5][section 4.2] offers a bit more detail on the `Applicative` laws.

10.5 `Applicative` and Lists

When we looked at `Functor`s, our two canonical examples were `Maybe` and lists, but we have not really mentioned lists yet in this section. The problem is not that lists are not `Applicative`s, but that there are two perfectly reasonable ways to implement the `Applicative` instance for lists!

Lists are a context that support zero or more values. So suppose we had a list of functions and wanted to apply them (in the `Applicative` sense) to some values also in a list context:

```
[(+1), (*2), (^3)] <*> [4, 5, 6]
```

We could certainly interpret this as pair-wise application, applying `(+1)` to `4`, `(*2)` to `5`, etc.

However, recall that we could view lists not just as a container of zero or more values but as a kind of non-deterministic value where `[4, 5, 6]` represents a value that might be any one of those numbers. In this interpretation, it

might make more sense to do apply each function from the left-hand list to each value value in the right-hand list.

The result is a list containing the possible values when a non-deterministic function is applied to a non-deterministic value, yielding a total of 9 possible values in this example.

In fact, the Haskell library's `Applicative` instance for lists uses the latter interpretation. The implementation looks like this:

```
instance Applicative [] where
    pure x    = [x]
    gs <*> xs = [ g x | g <- gs, x <- xs ]
```

- `pure` injects a value into the list context by creating a singleton list containing that value.
- `(<*>)` applies each function from the left-hand list to each value in the right-hand list as discussed. It does so via Haskell's **list comprehension** syntax.

What about the pair-wise version of `Applicative` for lists? Due to language constraints, the list type cannot have two implementations for the same type class. Instead, Haskell offers type called `ZipList` that wraps a normal list but offers a different `Applicative` instance:

```
newtype ZipList a = ZipList { getZipList :: [a] }

instance Functor ZipList where
    fmap f (ZipList xs) =
        ZipList $ map f xs

instance Applicative ZipList where
    pure x = ZipList $ repeat x
    (ZipList gs) <*> (ZipList xs) =
        ZipList $ zipWith ($) gs xs
```

- The `newtype` keyword defines a type synonym that is checked at compile time but discarded so there is no run-time overhead. A `ZipList`

can never be used as a normal list, but there is no additional overhead. Record syntax is used here to automatically create a function `getZipList` to translate normal lists to `ZipLists`.

- Our `ZipList` type needs to be a `Functor` before it can be an `Applicative`, so we define that here, delegating to the list type's `map` function.
- `zipWith` takes two lists and applies a function pair-wise to the elements of those lists. In this case the function is `($)`, which we have seen previously. So the `ZipList` instance of `Applicative` is implementing pair-wise application.
- Because `zipWith` truncates the result to the length of the shorter of its two parameter, it makes sense for `pure` to inject values into the `ZipList` context by creating an infinite list via `repeat`.
- If we used the same `pure` implementation as normal lists, `pure g <*> [1..] == [g 1]` and the functor instance law no longer holds.

10.6 Summary

In this section we have looked at applicative functors and Haskell's `Applicative` type class. We have seen how `Applicative` offers an abstraction for applying functions even when the functions themselves were wrapped up in a context just as `fmap` allowed us to apply bare functions to values in a context.

In the next section we will discuss the `Monad` type class and see how it further extends the notion of computational contexts that we have built up via `Functor` and `Applicative`.

11 Monads

Consider the following Java-like pseudocode:

```
String emailDomain = user.getContactInfo()                    1
                          .getEmailAddress()                  2
                          .getDomain();                       3
```

We have several domain objects:

- `User`, which has method `getContactInfo()` that returns the user's contact information with the type
- `ContactInfo`, which as a method `getEmailAddress()` that returns the associated email address with type
- `EmailAddress`, which has a method `getDomain()` which returns the the domain portion of that email address as a `String`.

Now suppose that a `User`'s `ContactInfo` is optional so that `getContactInfo()` might return `null`. Likewise, a `ContactInfo` record's `EmailAddress` is optional so that `getEmailAddress()` might return `null`.

That makes the above code snippet dangerous. The `getEmailAddress()` and `getDomain()` calls could be performed on null references, causing a `NullPointerException`. If uncaught, the program crashes.

We could try this:

```
String emailDomain;
ContactInfo contactInfo;
EmailAddress emailAddress;

contactInfo = user.getContactInfo();

if ( contactInfo != null )
{
    emailAddress = contactInfo.getEmailAddress();

    if ( emailAddress != null )
    {
        emailDomain = emailAddress.getDomain();
    }
    else
    {
        emailDomain = null;
    }
}
else
```

```
{                                                                    21
    emailDomain = null;                                              22
}                                                                    23
```

We have managed to propagate possible `null` values through the chain of method calls, but at the cost of a great deal of boilerplate code.

Of course, we have already seen how Haskell's `Maybe` type lets us encode the possibility of `null`-like values explicitly. Suppose we have analagous Haskell types and functions. The above code translates to something like:

```
userEmailDomain :: User -> Maybe String                              1
userEmailDomain user = case getContactInfo user of                   2
    Nothing          -> Nothing                                      3
    Just contactInfo -> case getEmailAddress contactInfo of
        Nothing    -> Nothing                                        5
        Just email -> getDomain email                                6
```

Although we are now explicit about the possibility of a null result, we have not really addressed the issue of boilerplate. If we had to chain together even more calls, our code would quickly stair-step right off the screen.

We can see a pattern, however. When we apply `getContactInfo` to `user`, we do a pattern match. If we got an actual value (the `Just` case), we take that value and pass it on to the `getEmailAddress` call. In the `null` case, though, we short-circuit the chain of function calls and just return `Nothing`. The same strategy is used when we try to pass the result of `getEmailAddress` into `getDomain`.

We can generalize this pattern:

```
chainMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b                   1
Nothing  `chainMaybe` f = Nothing                                    2
(Just x) `chainMaybe` f = fx                                         3
```

Note: Here we juse backticks around the function name to cause Haskell treat the function like an infix operator, not unlike using parentheses around an infix operator causes Haskell to treat it like a regular, prefix function.

Having factored out the pattern matching to handle both cases, we can rewrite our stair-stepped chain of function calls:

```
userEmailDomain :: User -> Maybe String                    1
userEmailDomain user =                                     2
(Just user) `chainMaybe` getContactInfo                    3
            `chainMaybe` getEmailAddress                   4
            `chainMaybe` getDomain                         5
```

Now our code is as readable as the original Java chain of method calls, but with the `null`-safety of the clunky second attempt.

11.1 The `Monad` Type Class

Not surprisingly, Haskell offers a type class to describe this pattern in polymorphic terms:

```
class Monad m where                                        1
    (>>=)  :: m a -> (a -> m b) -> m b                     2
    (>>)   :: m a -> m b -> m b                            3
    return :: a -> m a                                     4
    fail   :: String -> m a                                5
                                                           6
    m >> k = m >>= \_ -> k                                 7
```

- `(>>=)` is the monadic chaining operator. It is a polymorphic, infix equivalent of `chainMaybe`, which has the same definition as `Maybe`'s implementation of `(>>=)`.
- `(>>)` is a special case of `(>>=)` where the value passed into the right-hand function is simply ignored. It is given a default implementation in terms of `(>>=)` on line 5.
- `return` is the `Monad` class's general method for injecting a value into a monadic wrapper. Code that uses the `Monad` interface cannot use specific constructors like `Just` on line 3 of our final `userEmailDomain` implementation, so `Monad` instances implement `return` to define the behavior. If `return` sounds familiar, it is because it is actually identical to `pure` from `Applicative`. We will discuss this further in a moment.

- `fail` offers a way for `Monad` instances to short-circuit evaluation when a computation has failed. Use of `fail` is discouraged in general because some `Monad`s, including `IO`, implement `fail` by raising a fatal error. `Maybe`'s implementation of `fail` returns `Nothing`.

11.2 `Monad` and `Applicative`

We mentioned before that `return` and `pure` were basically identical. In fact, conceptually, `Applicative` is a superclass of `Monad`. In fact, the hierarchy from `Functor` to `Applicative` to `Monad` represents progressively more flexible operations on values inside some context.

However, while `Functor` and `Monad` were part of the Haskell standard library as described in the Haskell 98 standard [1], applicative functors were not introduced until 2008 in the paper *Applicative Programming with Effects* [7] by McBride and Paterson.

Altering the standard library's definition of `Monad` to include an `Applicative` class constraint would break existing user-defined `Monad` instances that did not offer an `Applicative` implementation. However, a proposal to make this change is likely to be implemented in the near future. For that reason, newer versions of GHC will issue warnings when `Monad` instances are declared without accompanying `Applicative` instances.

In the mean time, all the `Monad` instances we will discuss have accompanying `Applicative`.

11.3 The List Monad

Now that we have seen the basic definition of the `Monad` type class and seen a simple instance in `Maybe`, we can look at a more complex example: lists.

Recall that lists can be viewed as simple containers or as computational contexts supporting non-deterministic values. The list monad is based on this non-deterministic value perspective.

- The `Functor` instance for lists lifted function application into the domain of non-deterministic values.
- The `Applicative` instance for lists (i.e., not the `ZipList` instance) introduces the ability to apply non-deterministic function values to non-deterministic values.
- Finally, we can think of the `Monad` instance for lists as lifting computation in general into the domain of non-deterministic values.

Here is how the list instance of `Monad` is defined:

```
instance Monad [] where                                    1
    return x = [x]                                         2
    xs >>= f = concat (map f xs)                           3
    fail _   = []                                          4
```

- `return` is, again, equivalent to `pure` and simply returns a singleton list containing the given element.
- `fail` returns the empty list.
- `(>>=)` first maps `f :: a -> [a]` over `xs` yielding a value of type `[[a]]`, i.e., a list of lists. Then `concat :: [[a]] -> [a]` concatenates each of those lists into a single list. For example, `concat [[1, 2], [3, 4]] == [1, 2, 3, 4]`.

Suppose we wanted to work with a square root function that non-deterministically returned both the postive and negative square roots of its parameter:

```
sqrt' :: Double -> [Double]                                1
sqrt' x = [sqrt x, negate $ sqrt x]                        2
```

Now, when we evaluate `[4.0, 9.0] >>= sqrt'`, we map `sqrt'` over the list, yielding `[[2.0, -2.0], [3.0, -3.0]]`. Then `concat` is applied, yielding `[2.0, -2.0, 3.0, -3.0]`.

Suppose we try to evaluate `[16.0, 81.0] >>= sqrt' >>= sqrt'` (note that `(>>=)` is left associative) so we can simplify this expression:

```
([16.0, 81.0] >>= sqrt') >>= sqrt'                                    1
== [4.0, -4.0, 9.0, -9.0] >>= sqrt'                                   2
== [2.0, -2.0, NaN, NaN, 3.0, -3.0, NaN, NaN]                         3
```

When `sqrt'` is applied to one of the negative intermediate values, the result is `NaN`, since we cannot take the real square root of a negative. We would like to extend our non-deterministic square root function to deal with that case.

```
sqrt'' :: Double -> [Double]                                          1
sqrt'' x | x >= 0.0  = [sqrt x, negate $ sqrt x]                      2
         | otherwise = []                                             3
```

We use *guard clauses* to deal with the two cases. If the parameter is non-negative, yield two possible values, otherwise, yield no values at all.

Now, if we evaluate `[16.0, 81.0] >>= sqrt'' >>= sqrt''` we get `[2.0, -2.0, 3.0, -3.0]`. From the perspective of non-deterministic values, these are the values that could result from applying `sqrt''` twice to the non-deterministic value `[16.0, 81.0]`. When applying `sqrt''` to negative values, the return value of `[]` represents a path of computation that has failed, and no trace of it shows up in the final result.

11.4 The `Monad` Laws

Like `Functor` and `Applicative`, there are laws that govern how `Monad` instances should behave:

- **Left identity**: `return a >>= f == f a`
- **Right identity**: `m >>= return == m`
- **Associativity**: `(m >>= f) >>= g == m >>= (x -> f x >>= g x)`

The left and right identity laws describe are primarily concerned with the neutral behavior of `return`. The associativity law, with the behavior of sequences of monadic actions linked together with `(>>=)`.

Interestingly, none of these laws look much like identity or associativity as we might recall from algebra. However, if we introduce a new (but related) operator, they do:

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
(f >=> g) x = f x >>= (\y -> g y)
```

The operator `(>=>)` acts like the standard function composition operator `(.) :: (a -> b) -> (b -> c) -> a -> c` and if we rewrite the above laws in terms of `(>=>)`, the names look far more appropriate:

- **Left identity**: `return >=> f == f`
- **Right identity**: `f >=> return == f`
- **Associativity**: `(f >=> g) >=> h == f >=> (g >=> h)`

11.5 The `State` Monad

In the imperative programming paradigm, our programs essentially operate by mutating global state. To swap the values of two variables `a` and `b`, we might store the value in `a` into a temporary variable `t`, store the value in `b` into `a`, then store the value in `t` into `b`. Our programs just shuffle bit patterns around in memory.

In the end, Haskell programs are doing the same thing, but we are interested in expressing our programs in terms of higher-level constructs. However, the mutating state is a powerful tool and the `State` monad allows us to do emulate this style of programming by providing the framework through which a value of some type can be passed through a sequence of monadic actions, possibly being replaced along the way. We do not actually write values to locations in memory, but the end result is similar, while being built on the same monadic abstraction we have looked at so far.

Here is the definition of the `State` type and its `Monad` instance.

```
newtype State s a = State { runState :: s -> (a,s) }
```

```
instance Monad (State s) where                                    3
    return x = State $ \s -> (x,s)                                4
    (State h) >>= f = State $ \s -> let (a, newState) = h s       5
                                        (State g) = f a           6
                                    in  g newState                7
```

The `State` type constructor takes two type parameters: `s` is the type of the state value and `a` is the result type. Just as we might call `Maybe Integer` an `Integer` value in `Maybe`'s context of possible failure, `State String Integer` might be referred to as an `Integer` value in the context of a stateful computation, where the state is a `String` value.

The `newtype` definition of `State` uses Haskell's **record syntax** to created a **named field**. A `State` value is actually a wrapper around a function of type `s -> (a, s)`. The record syntax automatically creates a function `runState` which pulls that function out of the wrapper so that it can be applied to an initial state value of type `s`, returning the result of type `a` and the final state, again of type `s`.

The `State` type is polymorphic in two type variables, but recall that the definition of the `Monad` type class had only a single type variable. As line 3 suggests, it might be more appropriate to think of it as the `State s` monad, with each concrete type (`State [Integer]`, `State String`, etc.) as being separate, incompatible monads that happen to have identical implementations.

Intuitively, a value in the `State` monad represents a unit of stateful computation. However, it is important to emphasize that such a value is fundamentally just a function. Specifically, it is a function that takes an initial state and performs some computation that results in a value and a new state.

`State`'s implementation of `return` takes a value `x` and returns the simplest `State` context possible: a function that takes any initial state and returns `x` along with the initial state unaltered.

We will now consider `State`'s implementation of `(>>=)`.

The left-hand parameter of `(>>=)` is a `State` value, and we use pattern matching to bind the wrapped function to the identifier `h`.

The right-hand parameter of `(>>=)` is a function `f :: a -> State s b`.

We expect the result to be a `State` value, and indeed we see an anonymous function wrapped in the `State` constructor.

The anonymous function takes a formal parameter `s`, the incoming initial state, and uses `let` to make bindings for some intermediate values. The function `h` is applied to the incoming intial state and we bind the resulting pair to `(a, newState)`. We can think of this step as forcing the evaluation of the left-hand stateful computation before directing the result of that computation into the function on the right-hand side.

The next line binds the result of applying the right-hand side operand, `f`, to `a`, the result of the left-hand computation. The result of the expression `f a` is a `State` value wrapping a function, which is bound to `g`.

Finally, our anonymous function will apply `g` to newState.

Intuitively, the `State` implementation of `(>>=)` is the plumbing for a pipeline of computations that take an initial state and result in a value, along with the final state. Furthermore, any of these `State` computations can be further composed in just the same way.

This is a bit abstract, so an example may be helpful. This example is adapted from a very entertaining introduction to Haskell called *Learn You a Haskell for Great Good* [8].

The state that we will use in our computations will be a stack, implemented as `[Integer]`. We will define `push` and `pop` actions and use them in a small sample program.

```
type Stack = [Integer]

pop :: State Stack Integer
pop = State $ \(x:xs) -> (x, xs)

push :: Integer -> State Stack ()
push x = State $ \xs -> ((), x:xs)
```

- In line 1, we use `type` to define a type synonym.
- Consider the type of `pop`. The type signature is opaque: `pop` is just a stateful computation, working with a stack, resulting in an Integer.
- The result value is a function. It takes an initial state (our `Stack`) and returns a pair: the result value and the updated state value. This is exactly the right sort of type signature to wrap in the `State` constructor.
- Note how pattern matching in the anonymous function definition binds the head and tail of the `Stack` parameter to identifiers we use in the body. However, this pattern match is non-exhaustive (empty lists will not match), but we will address that later. This will result in a run-time error, unfortunately. We will address this later.
- `push` takes an `Integer`, the value to be pushed onto the stack, and returns a stateful computation. Again, the underlying result is a function wrapped in the `State` constructor.
- Although it looks strange, recall that `()` is the unit type. Its one value (also spelled `()`) is used when we do not really care about the result value. In this case, the result of a `push` is irrelevant, we just want the stack to get updated.

Now we can use our `push` and `pop` actions:

```
add :: State Stack ()                                                    1
add = pop >>= (\x -> pop >>= \y -> (push (x + y)))                       2
                                                                         3
simpleMath :: State Stack Integer                                        4
simpleMath = push 2 >>                                                   5
             push 2 >>                                                   6
             add >>                                                      7
             pop                                                         8
                                                                         9
result = runState simpleMath []                                          10
```

- We define a new action `add` in terms of `pop` and `push`. In conjunction with `(>>=)`, we use the two anonymous functions to bind the results of the two `pop` actions before pushing their sum back onto the stack.

- In essence, we have extended our language of `Stack` actions to include a new stack-based addition operator.
- Then in `simpleMath`, we use that language to write `Stack` manipulation code that has the appearance of imperative code.

In order to actually run `simpleMath`, we call `runState` on the action we want to execute and the initial state. As described, `runState` unwraps the underlying function inside the `State` wrapper and that function is immediately applied to the initial state, giving as a result value and a final state.

11.6 `do`-Notation

If we look back at the definition of `add`, we see a mess of odd-looking operators, anonymous functions, and nested parentheses. That would be frustrating to deal with and Haskell offers an alternative.

All monads support the use of **`do`-notation**, syntactic sugar that makes monadic code much more readable. An example:

```
add' = do
    x <- pop
    y <- pop
    push (x + y)

simpleMath' = do
    push' 2
    push' 2
    add'
    pop
```

These definitions have exactly the same type and semantics as the previous implementations that used `(>>=)` explicitly.

The Haskell compiler simplifies `do`-notation systematically, using the following basic patterns:

- Write up left-arrow de-sugaring.

```
before = do
    op1
```

```
after = op1
```

```
before = do
    op1
    op2
    op3
```

```
before = op1 >>
    do
        op2
        op3
```

```
before = do
    x <- op1
    op2
    op3
```

```
before = op1 >>
    do
        op2
        op3
```

- Fix tables.

11.7 The `IO` Monad

As a pure functional language, Haskell functions are unable to have side effects, including modifying program state and I/O.

We have seen how the `State` monad allows us to write code that models state manipulation and a similar approach is used for I/O actions with the `IO` monad.

Intuitively, we can think of the `IO` monad as a special case of the `State` monad where the state value being modified represents the outside world. An action in the `IO` monad that reads a string from the keyboard can be thought of as taking the current state of the outside world and returning a pair containing the string entered, plus the new, modified state of the outside world, with the keyboard input consumed.

Just as values of type `State a` can be thought of as stateful computations resulting in a value of type `a`, values of type `IO a` can be thought of as actions that reach outside of the pure Haskell execution model to perform I/O and yield a value of type `a`. The `IO` action described above would have type `IO String`. An `IO` action that printed a string to the screen and yielded no interesting result would have type `IO ()`.

Recall as well that Haskell is a lazy language. `IO` actions are decoupled from

the actual execution of the I/O they describe. For example, in the `IO String` action described above, no characters are read from the keyboard until they are demanded by evaluating an expression that needs them.

Here is an example. We will prompt the user for a file name (trying repeatedly until they entire the name of a file that exists). Then we will read the contents of the file, process it with a pure function that reverses the individual words, and print the output to the screen.

```
import Control.Applicative
import System.Directory

process :: String -> String
process = unlines
        . map unwords
        . map (map reverse)
        . map words
        . lines

reverseWords :: IO ()
reverseWords = do
    putStr "Enter a file name: "
    fileName <- getLine
    fileExists <- doesFileExist fileName
    if fileExists then do
        process <$> readFile fileName >>= putStr
    else do
        putStrLn "File does not exist. Try again."
        reverseWords

main :: IO ()
main = reverseWords
```

- We begin by importing modules. We need `Control.Applicative` for `(<$>)`, `System.Directory` `doesFileExist`.
- We begin by defining `process`. This is a pure function of type `String -> String`.
    - `process` is written as pipeline of pure functions linked together by the function composition operator `(.)`. Data moves through

the pipeline from right to left (or, as formatted here, bottom to top).

- `lines` breaks a `String` into a list of `String`s on newline characters.
- We break each of these lines into its constituent words with `map words`.
- We now have a list of lists of `Strings` (`[[String]]`). We reverse each of these individual strings with `map (map reverse)`. We need two instances of `map` since we are working with nested lists.
- With the words reversed, we reassemble the lines with `unwords` and reassemble the `String` as a whole with `unlines`.

- Note how we have separated the pure processing code from the I/O code. We are better able to reason about and test `process` because we know that the type system will enforce this separation.
- `reverseWords` is the `IO` action that describes the user interaction we want to perform.
  - We use `do`-notation, which is available for any monad.
  - The `IO` actions `putStrLn` and `putStr`, both of type `String -> IO ()` write a string to the console with and without a newline, respectively.
  - `getLine` reads a line of input from the console. We bind the result to the identifier `fileName` with the left-arrow notation.
  - Now we check whether the file exists. We use `doesFileExist :: FilePath -> IO Bool`, where `FilePath` is a descriptive type synonym for `String`.
  - We bind the result to the identifier `fileExists`. Note that we cannot use the value `doesFileExist fileName` directly in an `if` statement because it requires a value of type `Bool`, not `IO Bool`. This restriction is Haskell's type system enforcing the separation of pure and impure code.
  - If the file does not exist, we print a message and recursively call `reverseWords`.
  - If the file does exist, we will process the contents of the file.
    - The expression `readFile fileName` has type `IO String`, so we cannot directly apply `process` which requires a parameter of type `String`.

  - However, because `IO` is also an instance of `Applicative`, we can use `(<$>)` to enable `process` to operate on `IO String`.
  - The expression `process <$> readFile fileName` is itself an `IO` action of type `IO String`, and we use explicit monadic binding with `(>>=)` to pipe the its result into `putStr` to write the processed result to the console.
- Finally, we define an `IO` action `main` to just call `reverseWords`. When the Haskell compiler produces an executable, it looks for an action of type `IO ()` named `main` to be the entry point into the program.

11.8 Conclusion

We have now looked at a hierarchy of type classes, `Functor`, `Applicative`, and `Monad`. These classes successively refine the notion of actions within some computational context.

`Functor` offers us the higher-order function `fmap` which lifts pure functions into a context. We can think of `fmap sqrt` as being polymorphic in the type of context it operates on:

- In the `Maybe` functor, it takes the square root of the number if it exists, and propagates the lack of value if not.
- In the list functor, it takes the square root of any number of values.
- `State`, which we have only considered as a monad, is also a `Functor`. In this case, `fmap sqrt` can be applied to a stateful computation, resulting in a new stateful computation that leaves the state value unchanged but takes the square root of the number yielded by another original stateful computation.
- Similarly, in `IO`, `fmap sqrt` can be applied to an I/O action yielding a number to create a new I/O action yielding the square root of that number. So `fmap sqrt $ readLn` is an I/O action that attempts to read a string from the console, parse it as a floating-point number, and yields the result.

`Applicative` extends this idea, with `(<*>)` allowing us to use function values that are themselves inside a computational context.

Finally, `Monad` builds on this notion of computational contexts the notion of chaining together function applications with `do`-notation and the underlying `(>>=)` operator. Chaining together actions within each monad can be seen as programming with a constrained type of side effect.

- `Maybe` offers computation with the side effect of short-circuit failure: `(>>=)` binds two actions that might fail. If the first succeeds, the second will be able to operate on the result. Otherwise, the entire evaluation fails.
- The list monad offers computation with the side effect of non-determinism: `(>>=)` binds two actions that can each result in many possible values. The second action returns a list containing all the results it gets from operating on all the results from the first action.
- `State` offers computation with the side effect of mutable state: `(>>=)` abstracts away the process of an action yielding a value along with the current state to a second action, simulating a sequence of imperative actions that are mutating some shared state.
- Finally, `IO` deals with the most general sort of side effect: interaction with the world outside Haskell's pure evaluation: binding `IO` actions with `(>>=)` sequences their execution.

In the next section, we will look at how we can compose different types of effects using monad transformers.

12 Monad Transformers

We have seen how monads describe a general framework for programming with side effects in an otherwise pure functional language.

However, although we can use `IO` for input and output and `State` to track mutable state, we cannot use them together to write code that does both at

the same time. We would like to be able to compose two monads so that we can write code that takes advantage disparate types of side effects.

We will look at one solution to this problem: *monad transformers*.

Consider the following problem. We want to read lines of text from the console until the user enters a blank line while keeping a running count the number of lines read and saving the longest line entered so far.

We clearly need `IO` to read from the console, but updating the statistics we need seems like a good use case for `State`. Since `State` is not strictly necessary, we can implement this only using only `IO`:

```
data Stats = Stats { count :: Integer                                  1
                   , longest :: String }                               2
                   deriving (Show)                                     3
                                                                       4
lineStats :: IO Stats                                                  5
lineStats = runStats $ Stats { count = 0, longest = "" }               6
                                                                       7
runStats :: Stats -> IO Stats                                          8
runStats stats = do                                                    9
    line <- getLine                                                    10
    if line == ""                                                      11
        then return stats                                              12
        else runStats $ updateStats line stats                         13
                                                                       14
updateStats :: String -> Stats -> Stats                                15
updateStats line stats =                                               16
    Stats { count = newCount, longest = newLongest }                   17
        where newCount   = 1 + count stats                             18
              newLongest =                                             19
                if length line > (length $ longest stats)              20
                    then line                                          21
                    else longest stats                                 22
```

- Line 1 introduces a record type to track the statistics we are interested in.
- Line 5 defines a function that will begin the process, starting with an initial state.

- On line 8 we define a recursive `IO` action called `runStats`. It takes a `Stats` value and yields a possibly updated `Stats` value.
    - We get a line of text from the console, binding the result to the identifier `line`.
    - If the line is empty, we stop and yield the statistics gathered so far.
    - In this context, we appear to be using `return` like we would in an imperative language. However, in Haskell, `return` is just a function that wraps a value inside a monadic context and has nothing to do with terminating execution of a procedure. We only use `return` because we need the result of this expression to be of type `IO Stats`.
    - If the line is not empty, we recursively call `runStats` with updated statistics.
- Line 15 defines a helper function that takes a `String` and updates a `Stats` value accordingly. It increments the count and keeps the given `String` if it is longer. Note that we are not mutating the original `Stats` value, only using its constituents to build a new one.

This solution works, but it has the drawback that we were responsible for keeping track of the state value, explicitly passing it through recursive calls to `runStats`. We want compose `IO` and `State` to give us a monad that gives us mutable state while still allowing the I/O operations we require.

12.1 The Monad Transformer Library

The Haskell standard library includes a framework for composing two or more monads, called the Monad Transformer Library.

Monad transformers work by layering an interface for monadic operations of one type on top of a base monad. We saw earlier how to use the `State` monad to write stateful code that manipulated a stack of integers. Here we will extend that example, wrapping the `IO` monad with `State` to give us state manipulation and I/O at the same time.

```
import Control.Monad.Trans                                    1
import Control.Monad.Trans.State                              2
                                                              3
type Stack = [Integer]                                        4
                                                              5
type StatePlusIO a = StateT Stack IO a                        6
```

- We need to import `Control.Monad.Trans` to access some basic transformer-related functions, namely `lift` which we will see below.
- The `Control.Monad.Trans.State` module gives us the transformer for `State`.
- On line 4, we define a type synonym for the state values our combined monad will track.
- On line 6, we define a type synonym, `StatePlusIO`, for our combined monad.
- We build our combined monad using the type `StateT s m a`:
  - By convention the transformer equivalent of a monad is suffixed with "`T`". Thus, `State`'s transformer equivalent is `StateT`.
  - The first type variable, `s`, is the type for the state values, `Stack` in the example.
  - The second type variable, `m`, is the type of the underlying monad, `IO` in the example.
  - The third and final type variable, `a`, is just the result type of values yielded by operations in the monad. We do not specify it here, since we may want operations of type `StatePlusIO Integer` or `StatePlusIO ()`, etc.

Now we have a new monadic type that has the same stack manipulation categories we saw earlier but with the capability of performing I/O. We can write some basic actions to interact with this new monad:

```
push :: Integer -> StatePlusIO ()                             1
push x = do                                                   2
    stack <- get                                              3
    put (x : stack)                                           4
    lift . putStrLn $ "Pushed " ++ show x                     5
```

```
                                                                    6
pop :: StatePlusIO Integer                                          7
pop = do                                                            8
    (x:xs) <- get                                                   9
    put xs                                                          10
    lift . putStrLn $ "Popped " ++ show x                           11
    return x                                                        12
                                                                    13
readPush :: StatePlusIO ()                                          14
readPush = do                                                       15
    lift $ putStr "? "                                              16
    x <- lift $ readLn                                              17
    push x                                                          18
                                                                    19
add :: StatePlusIO ()                                               20
add = do                                                            21
    x <- pop                                                        22
    y <- pop                                                        23
    push (x + y)                                                    24
                                                                    25
calculator :: StatePlusIO ()                                        26
calculator = do                                                     27
    readPush                                                        28
    readPush                                                        29
    add                                                             30
```

- The functions `push` and `pop` operate just as they did before, but use `lift` to turn the `IO` action `putStrLn` into an action in the `StatePlusIO` monad.
- The general type of `lift` is `m a -> t m a`. Haskell can infer both the inner and outer monad based on the environment that `lift` is used in.
- In this case, `m` is `IO` and `t` is `StateT Stack`.
- So `lift` takes `putStrLn :: String -> IO ()` and gives us `String -> StateT Stack IO ()`.
- The `readPush` action uses our stateful framework plus I/O to read an integer from the console and push the result onto the stack. Again, `lift` brings the `readLn` action from the underlying `IO` monad up into our combined monad.

Just as with the examples we saw using a simple `State` monad, the actions we define for `StatePlusIO` will not do anything until we run them with `runStateT`:

```
ghci> runStateT calculator []
? 3
Pushed 3
? 5
Pushed 5
Popped 5
Popped 3
Pushed 8
((),[8])
```

To run a `State` action, we used `runState :: State s a -> s -> (a, s)`, which took a `State` action and an initial state and returned the result along with the final state. The type of `runStateT` has a slightly different type: `StateT s m a -> s -> m (a, s)`. We still provide a monadic action (in this case, built from the `StateT` transformer) and an initial state, but the resulting pair is wrapped inside the underlying monad.

In the case of `StatePlusIO`, the result of `runStateT` is a value in the `IO` monad.

The monad transformer library allows us to compose more than just two monads. In the following example, we will extend `StatePlusIO` with globally accessible, read-only configuration using `ReaderT` to give our stack manipulation code a list of values that it will refuse to store:

```
import Control.Monad.Trans
import Control.Monad.Trans.Reader
import Control.Monad.Trans.State

type Stack = [Integer]
type Config = [Integer]
type StateReaderIO a = StateT Stack (ReaderT Config IO) a

push :: Integer -> StateReaderIO ()
push x = do
    blacklist <- lift ask
    if x `elem` blacklist then
```

```
    liftIO . putStrLn $ "Forbidden: " ++ show x          13
else do                                                  14
    stack <- get                                         15
    put (x : stack)                                      16
    liftIO . putStrLn $ "Pushed " ++ show x              17
```

- We add an additional import this time, `Control.Monad.Trans.Reader` for the `Reader` monad's transformer equivalent `ReaderT`.
- We also define a type synonym for our configuration value, in this case a list of integers.
- The definition of our new combined type is similar to before, but instead of layering `StateT` over just `IO`, we are building on top of the monad we get from layering `ReaderT` over `IO`. We are composing three different monads to create the combined monad `StateReaderIO`.
- The `push` function receives the biggest change:
  - In the `Reader` monad, the function `ask` yields the read-only value carried along with the computation.
  - Since `ReaderT` is not the top-level monad, we have to use `lift` to bring `ask` (an action in the `ReaderT Config IO` monad) up into the full `StateReaderIO` monad.
  - We bind this result to the identifier `blacklist` and if the value we are attempting to push is in that list, we alert the user that the operation is forbidden. Otherwise, we push the value onto the stack as before.
  - We use a new function `liftIO` to lift the `IO` action into our full combined monad. Because `lift` can only bring an action up a single level in the monad stack, we would actually need `lift . lift` (`lift` composed with itself) to bring an `IO` action up two levels into `StateReaderIO`.
  - Because `IO` is often the base monad on top of which several monad transformers are layered, `liftIO` was included in the monad transformer library to lift `IO` actions to the top level no matter where in the stack `IO` is actually located.

Of course, we need a function to evaluate an action in our combined `StateReaderIO` monad. Recall that `runStateT` ran our `StateT`-based action and resulted in a value within the underlying `IO` monad.

In this case, `runStateT` will take a `StateReaderIO` action and an initial state, but give us a value in the `ReaderT Config IO` monad. Thus we need to use `runReaderT` to fully evaluate a `StateReaderIO` action to get a result in the `IO` monad.

Below we demonstrate `readPush` in our new monad, where `[1,2,3]` is the blacklist of values that `push` will reject and `[]` is the initial state.

```
ghci> runReaderT (runStateT readPush []) [1,2,3]
? 4
Pushed 4
((),[4])

ghci> runReaderT (runStateT readPush []) [1,2,3]
? 2
Forbidden: 2
((),[])
```

12.2 Monadic Parsing

Parsing text is a frequently encountered problem in computing. Simple recursive descent parsing is almost identical to computations in the `State` monad. A parser for some type `t` takes some initial state, the input buffer, and returns a value of type `t` and the unconsumed input.

However, parsing does not always succeed. For this reason, we would like to extend the `State` monad with the possibility of failure. In this case, we would like failures to come with some explanation. Rather than use `Maybe`, we will use `Either e t`, where `e` is the type of the error value and `t` is the type of a successful result.

`Either` has kind `* -> * -> *`, so it cannot be a monad. However, if we apply one of the type parameters, we get a proper monad: `Either e` for some error type `e`.

```
import Control.Applicative
import Control.Monad
import Prelude hiding (Either, Left, Right)
```

```
                                                                    4
data Either e t = Left e                                            5
                | Right t                                           6
                deriving (Eq, Show)                                 7
                                                                    8
instance Functor (Either e) where                                   9
    fmap f (Left e) = Left e                                        10
    fmap f (Right t) = Right (f t)                                  11
                                                                    12
instance Applicative (Either e) where                               13
    pure = Right                                                    14
    Left e <*> _ = Left e                                           15
    _ <*> Left e = Left e                                           16
    Right f <*> Right t = Right (f t)                               17
                                                                    18
instance Monad (Either e) where                                     19
    return = Right                                                  20
    Left e >>= f = Left e                                           21
    Right t >>= f = f t                                             22
```

- `Either` has two data constructors, `Left` for errors, `Right` for values.
- The `Functor`, `Applicative`, and `Monad` instances are all analogous to the ones we saw for `Maybe`, but rather than dealing with `Nothing` values, the failure case is `Left` wrapping an error value.
- We avoid `Monad`'s `fail` function since it takes a `String` and we want to be able to use any type for errors. The default implementation (causing a fatal error) will be used. As a result, our code will use `Right` to signal errors.

Just as the `State` monad has its transformer equivalent `StateT`, `Either` has its transformer, called `ExceptT`. We will use `ExceptT` on top of `State` to build our parsing monad.

```
import Control.Applicative ((<$>), (<*>))                           1
import Control.Monad.State                                          2
import Control.Monad.Trans.Except                                   3
import Data.List (intercalate)                                      4
import Prelude hiding (error)                                       5
                                                                    6
```

```
type ParseError = String                                          7
                                                                  8
type Parser a = ExceptT ParseError (State String) a               9
                                                                  10
runParser :: (Parser a)                                           11
          -> String                                               12
```

- On line 7, we define `ParseError` as a synonym for `String`.
- On line 9, We define a type synonym for our parser type. `Parser a` is much easier to read than the alternative, but ultimately our parser type is just stateful computation modified with failure with error messages.
- Lines 11-12 define the function to drives our `Parser` computations. It takes a parsing computation and an initial state and returns the result (a value or error message) plus the unconsumed input.

Now we can write some basic parsing primitives:

```
runParser = runState . runExceptT                                 1
                                                                  2
char :: Char -> Parser Char                                       3
char c = do                                                       4
    buffer <- get                                                 5
    case buffer of                                                6
        []     -> throwE "Unexpected end of input"                7
        (x:xs) -> if c == x                                       8
                  then do                                         9
                    put xs                                        10
                    return x                                      11
                  else                                            12
                    throwE $ "Expecting "                         13
                          ++ show c                               14
                          ++ ", found "                           15
                          ++ show x                               16
                                                                  17
string :: String -> Parser String                                 18
string = mapM char                                                19
                                                                  20
(<|>) :: Parser a -> Parser a -> Parser a                         21
left <|> right = do                                               22
```

```
    buffer <- get                                                    23
    catchE left (\_ -> do                                            24
        put buffer                                                   25
        right)                                                       26
                                                                     27
many :: Parser a -> Parser [a]                                       28
many parser = ((:) <$> parser <*> many parser)                       29
          <|> (return [])                                            30
                                                                     31
manyOne :: Parser a -> Parser [a]                                    32
manyOne prs = (:) <$> prs <*> many prs                               33
```

- `runParser` is a convenience function that composes running a `State` computation with the `ExceptT` monad transformer on top of it.
- The function `char` tries to parse a single character `c`. We `get` the current remaining parse buffer. If the buffer is empty, our parse fails. If there is at least one character in the buffer, we check whether it is equal to `c`. If so, update the state with the remainder of the buffer and yield the character. Otherwise, our parse fails.
- The `string` parser tries to parse a given string. We implement this using the general monadic function `mapM :: Monad m => (a -> m b) -> [a] -> m [b]`. This function sequences the application of a function over each element in a list, yielding the collecting the results. In this case, the function is `char :: Char -> Parser Char`. Since `String` is equivalent to `[Char]`, we sequence the parsing of each character in the string, yielding the parsed string. If parsing any of the individual characters fails, parsing the entire string fails.
- The choice operator `(<|>) :: Parser a -> Parser a -> Parser a` attempts the left-hand parser using `catchE` which runs a computation that might raise an exception and, if an exception occurs, passes the exception value into another action. In this case, we ignore the exception value, replace the initial parser state and try the right-hand parser.
- `many` parses zero or more repetitions of the given parser, yielding a list of the results. It use `(<|>)` chooise between no parses (`return []` on the right) or one or more. This definition uses applicative notation to lift `(:)` to build lists of parse results in a way that is more succinct than the equivalent `do`-notation.

- `manyOne` is like `many`, but requires at least one successful application of the given parser.

With these simple primitives and the general library functions we get for `Applicative` and `Functor` we can write recursive descent parsers. The following simple parser parses nested parentheses, yielding a tree structure:

```
data Tree = Leaf
          | Node [Tree]
          deriving (Eq, Show)

parens :: Parser Tree
parens = do
    char '('
    inside <- many parens
    char ')'
    if inside == []
        then return Leaf
        else return (Node inside)
```

```
ghci> runParser parens "()"
(Right Leaf,"")

ghci> runParser parens "((()()))"
(Right (Node [Leaf,Node [Leaf]]),"")

ghci> runParser parens ""
(Left "Unexpected end of input","")
```

References

[1] Simon Peyton Jones, et al., *Haskell 98 Language and Libraries: The Revised Report*, 2002.

[2] Paul Hudak, et al., "A History of Haskell: Being Lazy With Class", 2007.

[3] R.M. Burstall, D.B. MacQueen, D.T. Sannella, "Hope: An Experimental Applicative Language", 1980.

[4] Ralf Hinze, Simon Peyton Jones, "Derivable Type Classes", *Proceedings of the Fourth Haskell Workshop*, 227–236, 2000.

[5] Brent Yorgey, "The Typeclassopedia", *The Monad.Reader*, 13, 17-68, 2009.

[6] GHC Documentation, , 2014.

[7] Conor McBride, Ross Paterson, "Applicative Programming with Effects" *Journal of Functional Programming* 18:1, 1-13, 2008.

[8] Milan Lipovača, *Learn You a Haskell for Great Good*, 2011.

[9] Simon Peyton Jones, "Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell", 2010.