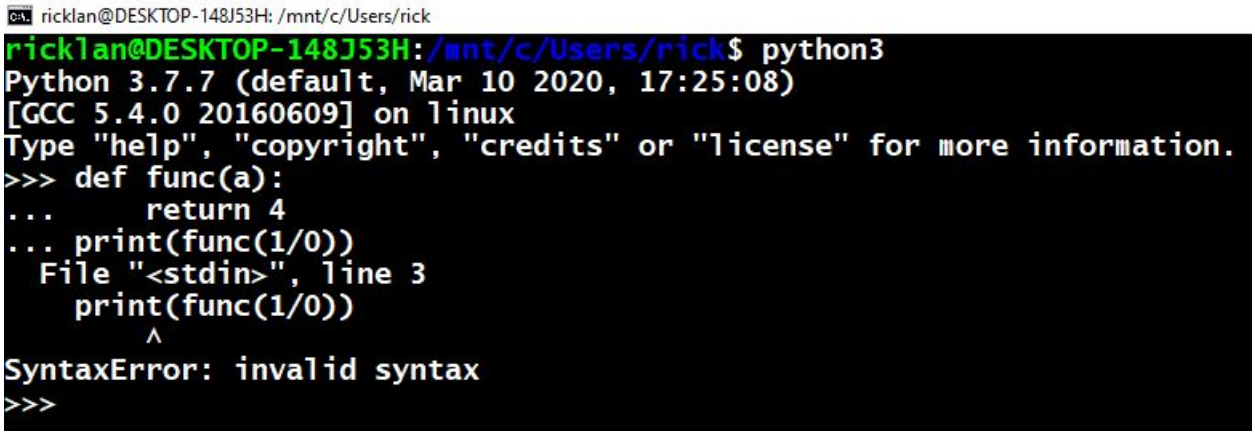
Introduction:

- Haskell is a widely used purely functional language. **Functional programming** is based on mathematical functions.
- Haskell is a lazy language. By lazy, we mean that Haskell won't evaluate any expression without any reason. When the evaluation engine finds that an expression needs to be evaluated, then it creates a thunk data structure to collect all the required information for that specific evaluation and a pointer to that thunk data structure. The evaluation engine will start working only when it is required to evaluate that specific expression. As a consequence, in Haskell, many short-circuiting operators and control constructs are user-definable whereas in other languages you're stuck with what's hardwired.
  E.g. Suppose we define f(x) = 4. Now, what does f(1/0) equal to?
  Most languages will do call by value, meaning that they will evaluate 1/0 first, which will give them an error. However, because Haskell is lazy, it doesn't evaluate 1/0 yet and will just plug in as-is. Oh x is unused, so f(1/0) = 4.
  In the pictures below, it shows a Python program and a Haskell program that tries to do the same thing, namely, create a function and have it return 4 and then call the function with the argument 1/0. In Python, this causes an error while in Haskell, it doesn't.



- A Haskell application is nothing but a series of functions.
- In conventional programming language, we need to define a series of variables along with their type. In contrast, Haskell is a **strictly typed language**. This means that the Haskell compiler is intelligent enough to figure out the type of the variable declared, hence we need not explicitly mention the type of the variable used.

Comments:

- Comments in Haskell are denoted as:
  1. Single line: **--**
  2. Multi line: **{- .. -}**

Do Notation:

- A do-block combines together two or more actions into a single action.
- **Note:** In a do-block, you don't use the keyword "in".

Variables:

- The left-hand side is the name of the value. Furthermore, = is used to declare the expression that is bound to the name on the left side (value definition).
  E.g. a = 3
- Haskell variables are immutable.
  E.g. If you do something like:
  a = 3
  a = a + 1
  print(a)
  You will get an error or your print(a) will not run.
- We can name part of the computation using **let** or **where**.
- There are 2 mains ways **let** is used in Haskell:
  1. This form is a **let-expression**, which is shown below:
     let [<definition>] in <expression> is an expression and can be used anywhere.
     E.g. let x = 5 in x + 1
  2. This form is a **let-statement**. This form is only used inside of do-block, and does not use **in**.
     E.g.

```
main = do
    let a = 3
    print(a)
```

     Note: in must be used in conjecture with **let**. It has no meaning on its own.
- **where** is part of a definition and is special syntax. **where** is bound to a surrounding syntactic construct, like the pattern matching line of a function definition.
  E.g. y = x + 1 where x = 5
- E.g. Consider the code and output below:

```
differenceOfSquaresV1, differenceOfSquaresV2 :: Integer -> Integer -> Integer

differenceOfSquaresV1 x y =
    let minus = x - y
        plus = x + y
    in minus * plus

differenceOfSquaresV2 x y = minus * plus
    where
        minus = x - y
        plus = x + y
```

```
*Main> differenceOfSquaresV1 3 2
5
*Main> differenceOfSquaresV2 3 2
5
*Main>
```

putStr, putStrLn and print:

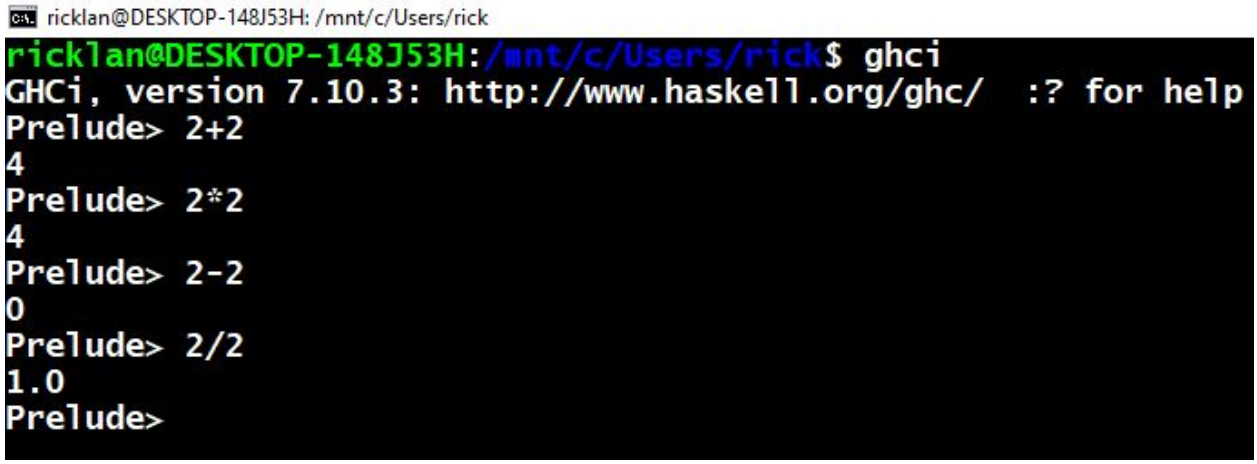
- putStr will print a string without a newline character at the end.
- putStrLn will print a string with a newline character at the end.
- print will just print whatever is in the parentheses.
- E.g. Consider the code and output below:

Basic Data Types:

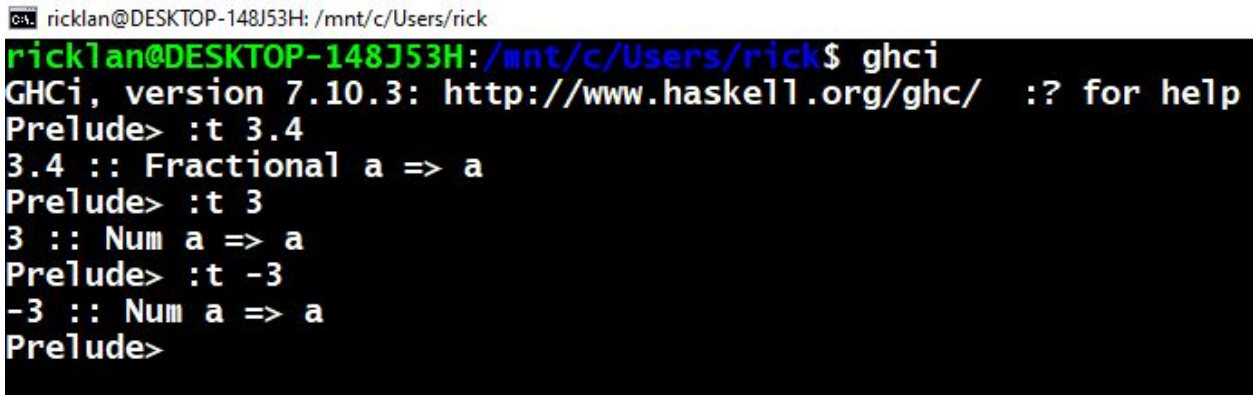
1. **Numbers:**

- Haskell is intelligent enough to decode some number as a number. Therefore, you need not mention its type externally as we usually do in case of other programing languages.

- E.g. Consider the code and output below:



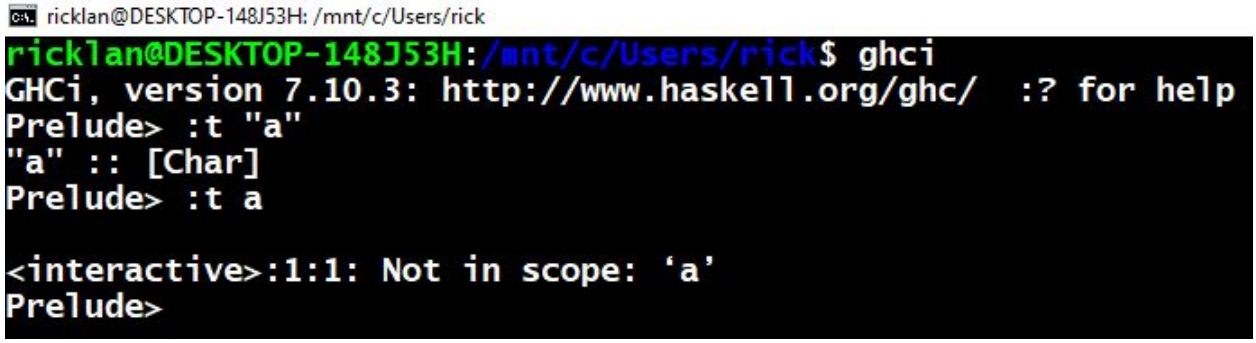
- **Note: :t** is to include the specific type related to the inputs.
- E.g. Consider the code and output below:



Notice how it shows the type of the input.

2. Characters:

- Like numbers, Haskell can intelligently identify a character given in as an input to it.
- E.g. Consider the code and output below:

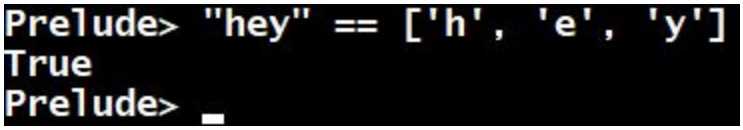


Note: The error message "<interactive>:1:1: Not in scope: `a'" means that the Haskell compiler is warning us that it is not able to recognize your input. Haskell is a type of language where everything is represented using a number.

3. **String:**

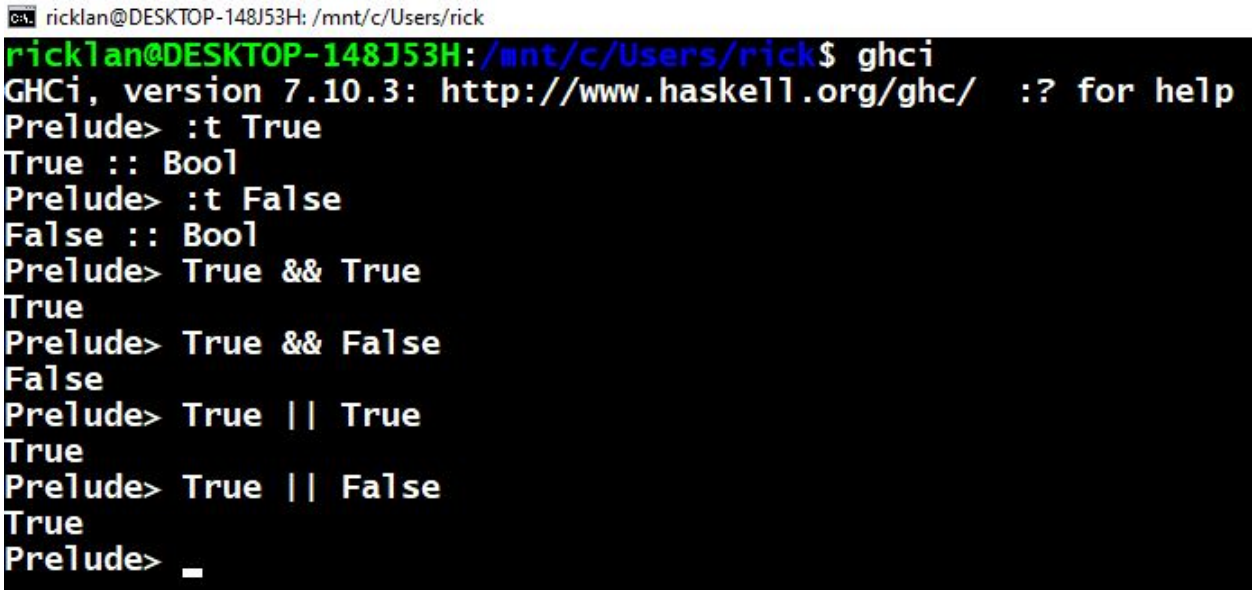
- A string is nothing but a collection of characters. There is no specific syntax for using string, but Haskell follows the conventional style of representing a string with double quotation.
- E.g. Consider the code and output below:

- **Note:** Strings are just lists of characters, as shown below:



4. **Boolean:**

- Haskell has 2 boolean values: True and False.
- E.g. Consider the code and output below:



- **Note:** True and False must have the T/F capitalized. true and false will get you errors, as shown below:

```
ricklan@DESKTOP-148J53H: /mnt/c/Users/rick
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick$ ghci
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude> :t true

<interactive>:1:1:
    Not in scope: 'true'
    Perhaps you meant data constructor 'True' (imported from Prelude)
Prelude> :t false

<interactive>:1:1:
    Not in scope: 'false'
    Perhaps you meant data constructor 'False' (imported from Prelude)
Prelude>
```

5. List:

- A **List** is a collection of the same data type separated by comma.
  E.g. [‘a’,’b’,’c’] is a list of characters.
  E.g. [1,2,3] is a list of numbers.
- Like other data types, you do not need to declare a List as a List. Haskell is intelligent enough to decode your input by looking at the syntax used in the expression.
- Lists in Haskell are homogeneous in nature, which means they won’t allow you to declare a list of different kinds of data type.
- E.g. Consider the code and output below:

```
ricklan@DESKTOP-148J53H: /mnt/c/Users/rick
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick$ ghci
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude> :t [1,2,3]
[1,2,3] :: Num t => [t]
Prelude> :t ['a','b','c']
['a','b','c'] :: [Char]
Prelude> [1,2,3]
[1,2,3]
Prelude> ['a','b','c']
"abc"
Prelude> [1,2,3,'a']

<interactive>:6:2:
    No instance for (Num Char) arising from the literal '1'
    In the expression: 1
    In the expression: [1, 2, 3, 'a']
    In an equation for 'it': it = [1, 2, 3, ....]
Prelude>
```

- To get the length of a list, L, you can do **length L**.
  E.g. Consider the code and output below:

```
main = do
    let a = [1,2,3,4,5]
    print(a)
    putStr("The length of array a is ")
    print(length a)
```

```
*Main> main
[1,2,3,4,5]
The length of array a is 5
*Main> _
```

- To get the reverse of a list, L, you can do **reverse L**.
  E.g. Consider the code and output below:

```
main = do
    let a = [1,2,3,4,5]
    print(a)
    putStr("The reverse of array a is ")
    print(reverse a)
```

```
*Main> main
[1,2,3,4,5]
The reverse of array a is [5,4,3,2,1]
*Main>
```

- To get the nth index of a list, L, you can do **L !! n**.
  Note: In Haskell, list indexes start at 0. So, **L !! 0** gets the first element, **L !! 1** gets the second element, and so on.
  E.g. Consider the code and output below:

```
main = do
    let a = [1,2,3,4,5]
    let len_a = length a - 1
    print(a)
    putStr("The first element of array a is ")
    print(a !! 0)
    putStr("The second element of array a is ")
    print(a !! 1)
    putStr("The last element of array a is ")
    print(a !! len_a)
```

```
*Main> main
[1,2,3,4,5]
The first element of array a is 1
The second element of array a is 2
The last element of array a is 5
*Main> _
```

- **Note:** **head L** returns the first element of a list while **last L** returns the last element of a list.
  E.g. Consider the code and output below:

```
main = do
  let a = [1,2,3,4,5]
  print(a)
  putStr("The first element of array a is ")
  print(head a)
  putStr("The last element of array a is ")
  print(last a)
```

```
*Main> main
[1,2,3,4,5]
The first element of array a is 1
The last element of array a is 5
```

- To add elements to the start of a list, L, you can do **element1 : element2 : … : L**. This is called **consing**. In fact, Haskell builds all lists this way by consing all elements to the empty list, []. The commas-and-brackets notation are just syntactic sugar. So [1,2,3,4,5] is exactly equivalent to 1:2:3:4:5:[].
  E.g. Consider the code and output below:

```
main = do
  let a = [1,2,3,4,5]
  print(a)

  -- Adding 1 element, 0, to the beginning of a.
  let b = 0 : a
  print(b)

  -- Adding 2 elements, 6, 7, to the beginning of b.
  let c = 6 : 7: b
  print(c)

  -- Adding 3 element, 8, 9, 10, to the beginning of c.
  let d = 8 : 9: 10: c
  print(d)
```

```
*Main> main
[1,2,3,4,5]
[0,1,2,3,4,5]
[6,7,0,1,2,3,4,5]
[8,9,10,6,7,0,1,2,3,4,5]
```

- To add elements to the end of a list, L, you can do **L ++ [element1, element2, …]**.
  E.g. Consider the code and output below:

```
main = do
  let a = [1,2,3,4,5]
  print(a)

  -- Adding 1 element, 6, to the end of a.
  let b = a ++ [6]
  print(b)

  -- Adding 2 elements, 7, 8, to the end of b.
  let c = b ++ [7, 8]
  print(c)
```

```
*Main> main
[1,2,3,4,5]
[1,2,3,4,5,6]
[1,2,3,4,5,6,7,8]
```

- To join 2 lists, L1 and L2, together, you can do **L1 ++ L2**.
  E.g. Consider the code and output below:

```
main = do
  let a = [1,2,3,4,5]
  let b = [6, 7, 8, 9, 10]
  print(a ++ b)
```

```
*Main> main
[1,2,3,4,5,6,7,8,9,10]
```

- To Delete the first N elements from a list, L, you can do **drop N L**.
  E.g. Consider the code and output below:

```
main = do
  let a = [1,2,3,4,5]
  print(drop 3 a) -- Removes the first 3 elements of a.
```

```
*Main> main
[4,5]
```

- **Note:** To remove the first element of a list, L, you can do **tail L**. To remove the last element of a list, L, you can do **init L**.
  E.g. Consider the code and output below:

```
main = do
  let a = [1,2,3,4,5]
  print(tail a) -- Removes the first element of a.
  print(init a) -- Removes the last element of a.
```

```
*Main> main
[2,3,4,5]
[1,2,3,4]
```

- To get the first N elements of a list, L, you can do **take N L**.
  Note: The output will be returned as a list.
  E.g. Consider the code and output below:

```
main = do
  let a = [1,2,3,4,5]
  print(take 2 a) -- Gets the first 2 elements of a.
```

```
*Main> main
[1,2]
```

- To split a list, L, at the Nth position, you can do **splitAt N L**.
  E.g. Consider the code and output below:

```
main = do
  let a = [1,2,3,4,5]
  print(splitAt 2 a) -- Splits a at the 2nd element.
```

```
*Main> main
([1,2],[3,4,5])
```

- To insert an element into the middle of a list, L, you have to split the list into two smaller lists, put the new element in the middle, and then join everything back together. There is no built-in function to do so.
  Syntax: **let (b,c) = splitAt n a in b ++ [new_element] ++ c**
  E.g. Consider the code and output below:

```
Prelude> let a = [1,2,3,4,5]
Prelude> let (b,c) = splitAt 4 a in b ++ [6] ++ c
[1,2,3,4,6,5]
```

- To delete an element into the middle of a list, L, you have to split the list in two, remove the element from one list, and then join them back together. There is no built-in function to do so.
  Syntax: **let (b, c) = splitAt 2 a in b ++ (tail c)**
  E.g. Consider the code and output below:

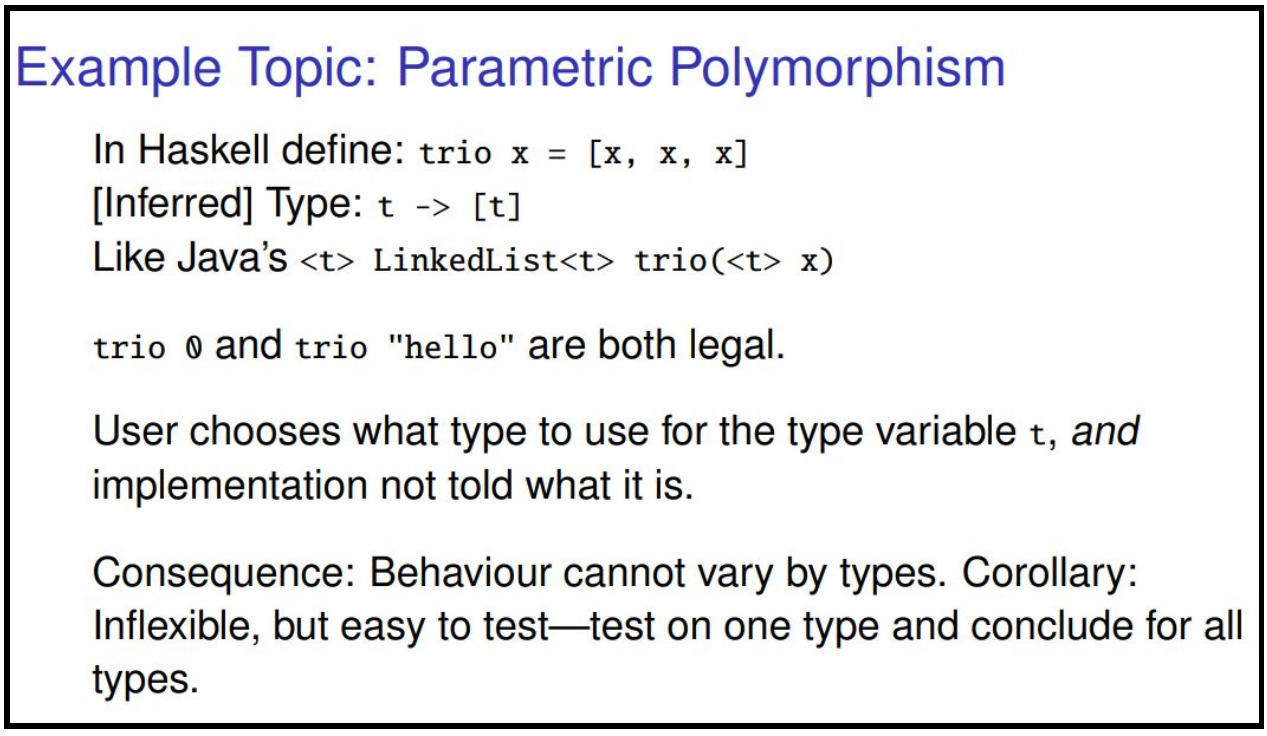
```
Prelude> let a = [1,2,3,4,5]
Prelude> let (b, c) = splitAt 2 a in b ++ (tail c)
[1,2,4,5]
```

6. **List Comprehension:**

- **List comprehension** is the process of generating a list using mathematical expression.

Parametric Polymorphism:

- A value is **polymorphic** if there is more than one type it can have. Polymorphism is widespread in Haskell and is a key feature of its type system.
- Most polymorphism in Haskell falls into one of two broad categories: **parametric polymorphism** and **ad-hoc polymorphism**.
- **Parametric polymorphism** refers to when the type of a value contains one or more (unconstrained) type variables, so that the value may adopt any type that results from substituting those variables with concrete types.
- In Haskell, this means any type in which a type variable, denoted by a name in a type beginning with a lowercase letter, appears without constraints (i.e. does not appear to the left of a =>). In Java and some similar languages, generics (roughly speaking) fill this role.
- For example, the function id :: a -> a contains an unconstrained type variable a in its type, and so can be used in a context requiring Char -> Char or Integer -> Integer or any of a literally infinite list of other possibilities. Likewise, the empty list [] :: [a] belongs to every list type, and the polymorphic function map :: (a -> b) -> [a] -> [b] may operate on any function type. Note, however, that if a single type variable appears multiple times, it must take the same type everywhere it appears, so e.g. the result type of id must be the same as the argument type, and the input and output types of the function given to map must match up with the list types.
- Since a parametrically polymorphic value does not "know" anything about the unconstrained type variables, it must behave the same regardless of its type. This is a somewhat limiting but extremely useful property known as **parametricity**.
- E.g.



Functions:

- Syntax: **function_name argument(s) = function definition**
  The **function definition** is a formula that uses the argument in context with other already defined terms.
  E.g.
  area r = pi * r ^ 2 **Note:** Here, r is an argument.

```
Prelude> area r = pi * r ^ 2
Prelude> area 5
78.53981633974483
Prelude> area 10
314.1592653589793
Prelude>
```

  increment n = n + 1 **Note:** Here, n is an argument.

```
Prelude> increment n = n + 1
Prelude> increment 2
3
Prelude> a = 3
Prelude> increment a
4
Prelude> print(a)
3
Prelude>
```

- **Note:** Call functions without parentheses.
- **Note:** Function call is left associative.
- **Note:** Function call takes precedence over operators.
- **Note:** Functions can accept more than one parameter.
- **Note:** In Haskell functions are first class values. That means they can be put in variables, passed and returned from functions, etc. You can also have **function composition**. I.e. You have a function that takes two functions and a value, applies the second function to the value and then applies the first function to the result.
- We can supply only some of the arguments to a function. If we have a function that takes N arguments and we supply K arguments, we'll get a function that takes the remaining (N - K) arguments.
  E.g. Consider the code and output below:

```
Prelude> func1 a b c = a + b + c
Prelude> func2 = func1 1 2
Prelude> func2 3
6
Prelude>
```

  Here, we have a function, func1, that takes 3 arguments and adds them up. In this case, N = 3. However, we only supply 2 arguments (1 and 2), so in this case, K = 2 and we get a new function, func2, that takes (3-2 or 1) argument. When we enter the last argument for func2, it gives the sum of the 3 arguments (The first 2 arguments were passed to func1 and the 3rd argument was passed to func2.)
- We can combine functions, too.

- E.g. Consider the code and output below:

```
Prelude> areaRectangle l w = l * w
Prelude> areaSquare s = areaRectangle s s
Prelude> areaSquare 5
25
Prelude> areaSquare 10
100
Prelude> areaTriangle b h = (areaRectangle b h)/2
Prelude> areaTriangle 5 10
25.0
Prelude> areaTriangle 10 10
50.0
Prelude>
```

Here, I created a function called areaRectangle that takes in 2 arguments, a length and width, and gives back their product. Then, I created another function called areaSquare that takes in 1 argument, a length, and gives back s^2, using areaRectangle to calculate it. Lastly, I created a third function called areaTriangle that takes 2 arguments, a base and height, and gives back the result of base*height/2, using areaRectangle to calculate base*height.

- E.g. Consider the code and output below:

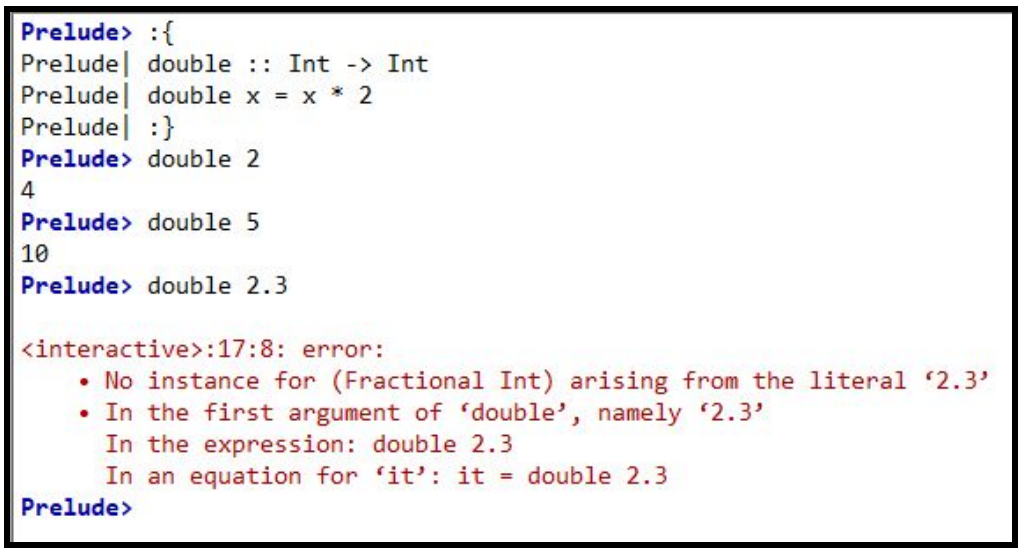
```
Prelude> double x = x * 2
Prelude> quadruple x = double double x

<interactive>:43:1: error:
    • Non type-variable argument in the constraint: Num (a -> a)
      (Use FlexibleContexts to permit this)
    • When checking the inferred type
        quadruple :: forall a. (Num a, Num (a -> a)) => a -> a
Prelude> quadruple x = double (double x)
Prelude> quadruple 4
16
Prelude> quadruple 10
40
Prelude>
```

Here, I created a function called double that takes an argument and gives back its double. Then, I created a function called quadruple that takes an argument and gives back its quadruple using the double function twice. Notice that I needed brackets for double (double x). When I tried doing double double x, it gave me an error.

- We can give values a type signature using **::**. Furthermore, we use **->** to denote the type of a function from one type to another type. **Note:** -> is right associative.

- E.g. Consider the code and output below:



In the first picture, I didn't use ::. Hence, when I did double 2.4 and double 5.6, I didn't get an error. However, in the second picture, I did double :: Int -> Int. This means that the input must be of type Int. Hence, when I do double 2.3, it gives me an error.

Addition Operator:

- The "+" operator is used for addition.
- E.g.

```
add :: Float -> Float -> Float
add x y = x + y
```

```
*Main> add 2 3
5.0
*Main> add 2.333 2.667
5.0
```

Subtraction Operator:

- The "-" operator is used for subtraction.
- E.g.

```
subtraction :: Float -> Float -> Float
subtraction x y = x - y
```

```
*Main> subtraction 6 3
3.0
*Main> subtraction 6.5 3.2
3.3
*Main> subtraction 9.5 (-3.2)
12.7
```

- **Note:** It's best to use parentheses "()" to enclose negative numbers. Otherwise, the compiler might think the "-" isn't part of the number.
  E.g.

```
*Main> subtraction 9.5 -4

<interactive>:64:1: error:
    • No instance for (Num (Float -> Float)) arising from a use of '-'
        (maybe you haven't applied a function to enough arguments?)
    • In the expression: subtraction 9.5 - 4
      In an equation for 'it': it = subtraction 9.5 - 4
*Main> subtraction 9.5 (-4)
13.5
```

Multiplication Operator:

- The "*" operator is used for multiplication.
- E.g.

```
multiply :: Float -> Float -> Float
multiply x y = x * y
```

```
*Main> multiply 2 3
6.0
*Main> multiply 4.5 2
9.0
*Main> multiply (-1.2) 5
-6.0
```

Division Operator:

- The “/” operator is used for division.
- E.g.

```
divide :: Float -> Float -> Float
divide x y = x/y
```

```
*Main> divide 6 3
2.0
*Main> divide 6 4
1.5
*Main> divide 6 (-2)
-3.0
```

Exponent Operator:

- The “^” operator is used for exponent.
- Syntax: **base^exponent**
- E.g.

```
square :: Int -> Int
square x = x^2
```

```
*Main> square 2
4
*Main> square 5
25
*Main> square (-5)
25
*Main> square (-10)
100
```

- E.g.

```
cube :: Int -> Int
cube x = x^3
```

```
*Main> cube 3
27
*Main> cube (-3)
-27
*Main> cube 10
1000
*Main> cube (-4)
-64
```

Sequence/Range Operator:

- The “..” operator is used for sequence or range.
- You can use this operator while declaring a list with a sequence of values.
- If you want to print all the values from 1 to 10, then you can use something like "**[1..10]**". Similarly, if you want to generate all the alphabets from "a" to "z", then you can just type "**[a..z]**".

- E.g.

```
main = do
    print([1..10])
    print(['a'..'z'])
    print(['A' .. 'Z'])
```

```
[1,2,3,4,5,6,7,8,9,10]
"abcdefghijklmnopqrstuvwxyz"
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

Pattern Matching:

- **Pattern matching** means that you write down a literal at the place where you're supposed to write the parameter(s).
- E.g.

```
factorial :: Int -> Int
factorial 0 = 1 -- An example of pattern matching.
factorial n = n * factorial(n - 1)
```

```
*Main> factorial 5
120
```

- E.g.

```
fibanocci :: Int -> Int
fibanocci 0 = 0 -- Example of pattern matching.
fibanocci 1 = 1 -- Example of pattern matching.
fibanocci n = fibanocci(n-1) + fibanocci(n-2)
```

```
*Main> fibanocci 5
8
```

Case Expressions:

- Allows us to write control flows on data types.
- Matches from top to bottom.
- **Note:** The pattern **_** means match anything.
- Syntax:

```
case <expr> of
  <pattern1> -> <result1>
  <pattern2> -> <result2>
  ...
  <patternN> -> <resultN>
```

- E.g.

```
fibanocci :: Int -> Int
fibanocci x = case x of
    0 -> 0
    1 -> 1
    n -> fibanocci n1 + fibanocci n2
        where
            n1 = n - 1
            n2 = n - 2
```

```
*Main> fibanocci 5
5
```

- E.g.

```
fibanocci :: Int -> Int
fibanocci x = case x of
    0 -> 0
    1 -> 1
    n -> let
        n1 = n - 1
        n2 = n - 2
        in
            fibanocci n1 + fibanocci n2
```

```
*Main> fibanocci 5
5
```

If statements:

- Syntax:

  if (condition)
  then (value)
  else if (condition)
  then (value)
  else
  (value)

- E.g.

```
factorial :: Int -> Int
factorial n =
    if n < 2
        then 1
    else
        n * factorial(n-1)
```

```
*Main> factorial 5
120
```

- **Note:** The else if is conditional, but you must have the if and the else.

type, term, value:

Those two things are called:

```
f (x*2 + 1) :: Integer
^^^^^^^^^^^    ^^^^^^^
   term          type
```

One more example:

```
h . g :: Char -> Bool
^^^^^    ^^^^^^^^^^^^
term          type
```

- **Note: term** is also widely known as **expression**.

- **Note:** 5+4 is a **term**; the result of evaluating it, 9, is a **value**.
  I.e. **term** is your code and **value** is the result of the term.

Synthesis and Evaluation:

- **Synthesis** is how you write code.
- **Evaluation** is how the computer runs your code.
- For synthesis, using induction can help you write the code.
- E.g. Consider the factorial code below:

```
factorial :: Int -> Int
factorial 0 = 1 -- An example of pattern matching.
factorial n = n * factorial(n - 1)
```

Here is the mindset of how to write it:

WTP: For all natural n: Factorial n = n!

Base case:

WTP: Factorial 0 = 0!

Notice that 0! = 1, so if I code up Factorial 0 = 1, I get Factorial 0 = 0!.

Induction step:

Let natural n ≥ 1 be given.

Induction hypothesis: Factorial (n-1) = (n-1)!

WTP: Factorial n = n!

Notice that n! = n*(n-1)!

= n * Factorial (n-1) by I.H.

So if I code up Factorial n = n * Factorial (n-1), I get Factorial n = n!.

Here is the evaluation of factorial 3:

→ 3 * factorial(3 - 1)
→ 3 * factorial(2)
→ 3 * (2 * factorial(2 - 1))
→ 3 * (2 * factorial(1))
→ 3 * (2 * (1 * factorial(1-1)))
→ 3 * (2 * (1 * (factorial(0))))
→ 3 * (2 * (1 * 1))
→ 3 * 2
→ 6

Guards:

- Denoted by "|"
- We use | to say alternatively.

- E.g.

```
absolute :: Integer -> Integer
absolute x
    | x < 0 = (-x)
    | otherwise = x
```

```
*Main> absolute (-5)
5
```

- E.g.

```
factorial :: Int -> Int
factorial n
    |n == 0 = 1
    |otherwise = n * factorial(n-1)
```

```
*Main> factorial 5
120
```

Lists:

- Some types of lists are [Integer], [Bool], [] Integer, [] Bool, etc.
- An empty list is denoted as [].
- A list literal is denoted like [4, 1, 6].
  Note: Remember that Haskell makes lists in this way:
  (4 : (1 : (6 : ([])))) or 4 : 1 : 6 : []
  The parentheses are optional.
- Formally (recursive definition as in CSCB36): a list is one of:
  - []
  - <an item here> : <a list here>
- **Note:** These are singly-linked lists. These are not arrays.
- **Note:** Lists are immutable in Haskell.
- E.g. Insertion Sort:
  Strategy: Have a helper function insert.
  Take element e and list xs. xs is assumed to have been sorted in increasing order.
  Put e into the “right place” in xs so the whole is still sorted.
  E.g. insert 4 [1,3,5,8,9,10] = [1,3,4,5,8,9,10]

Here's the code:

```haskell
insert :: Integer -> [Integer] -> [Integer]

-- Structural induction on xs.

-- Base case: xs is empty. Answer is [e] aka e:[]
insert e [] = [e]  -- e : []

-- Induction step: Suppose xs has the form x:xt (and xt is shorter than xs).
-- E.g., xs = [1,3,5,8], x = 1, xt = [3,5,8].
-- Induction hypothesis: insert e xt = put e into the right place in xt.
insert e xs@(x:xt)
    -- xs@(x:xt) is an "as-pattern", "xs as (x:xt)",
    -- so xs refers to the whole list, and it also matches x:xt
    --
    -- If e <= x, then e should be put before x, in fact all of xs, and be done.
    -- E.g., insert 1 (10 : xs) = 1 : (10 : xs)

    | e <= x = e : xs

    -- Otherwise, the answer should go like:
    -- x, followed by whatever is putting e into the right place in xt.
    -- i.e.,
    -- x, followed by insert e xt (because IH)
    -- E.g., insert 25 (10 : xt) = 10 : (insert 25 xt)

    | otherwise = x : insert e xt

insertionSort :: [Integer] -> [Integer]
insertionSort [] = []
insertionSort (x:xt) = insert x (insertionSort xt)
```

E.g.

```
*Main> insertionSort [3,2,1]
[1,2,3]
```

Introduction:

- In Haskell, lists are a **homogenous data structure**. It stores several elements of the same type. That means that we can have a list of integers or a list of characters but we can't have a list that has a few integers and then a few characters.
- Lists are denoted by square brackets and the values in the lists are separated by commas.
- E.g.

```
Prelude> let a = [1,2,3,4,5]
Prelude> print(a)
[1,2,3,4,5]
```

List Operations:

Note: I will use L1 and L2 to denote lists in the examples below, and e1 and e2 to denote elements.

1. **Adding to the beginning of a list:**

- To add to the beginning of a list, use "**:**". This is called **consing**. In fact, Haskell builds all lists this way by consing all elements to the empty list, []. The commas-and-brackets notation are just syntactic sugar. So [1,2,3,4,5] is exactly equivalent to 1:2:3:4:5:[].
- Syntax: **e1 : L1**
- E.g.

```
Prelude> let a = [1,2,3,4,5]
Prelude> let b = 0:a
Prelude> print(b)
[0,1,2,3,4,5]
```

2. **Adding to the end of a list:**

- To add to the end of a list, use "**++**".
- Syntax: **L1 ++ [e1]**
- E.g.

```
Prelude> let c = b ++ [6]
Prelude> print(c)
[0,1,2,3,4,5,6]
```

3. **Joining 2 lists:**

- To join L2 to L1, do **L1 ++ L2**.
- E.g.

```
Prelude> let c = b ++ [6]
Prelude> print(c)
[0,1,2,3,4,5,6]
Prelude> let d = [7,8,9]
Prelude> let e = c ++ d
Prelude> print(e)
[0,1,2,3,4,5,6,7,8,9]
```

4. **Comparing Lists:**

    - Lists can be compared if the stuff they contain can be compared. When using **<**, **<=**, **>**, **>=**, **==** to compare lists, they are compared in lexicographical order.

- Syntax:
  L1 > L2
  L1 >= L2
  L1 < L2
  L1 <= L2
  L1 == L2
- E.g.

```
Prelude> [1,2,3] > [1,2,4]
False
Prelude> [1,2,3] >= [1,2,4]
False
Prelude> [1,2,3] < [1,2,4]
True
Prelude> [1,2,3] <= [1,2,4]
True
Prelude> [1,2,3] == [1,2,3]
True
```

5. **Head:**
   - head takes a list and returns its first element.
   - Syntax: **head L1**
   - E.g.

```
Prelude> let a = [1,2,3,4,5]
Prelude> head a
1
```

6. **Last:**
   - last takes a list and returns its last element.
   - Syntax: **last L1**
   - E.g.

```
Prelude> let a = [1,2,3,4,5]
Prelude> last a
5
```

7. **Init:**
   - init takes a list and returns everything except its last element.
   - Syntax: **init L1**
   - E.g.

```
Prelude> let a = [1,2,3,4,5]
Prelude> init a
[1,2,3,4]
```

8. **Tail:**
   - tail takes a list and returns everything except for the first element.
   - Syntax: **tail L1**

- E.g.

```
Prelude> let a = [1,2,3,4,5]
Prelude> tail a
[2,3,4,5]
```

9. Length:

- length takes a list and returns its length.
- Syntax: **length L1**
- E.g.

```
Prelude> let a = [1,2,3,4,5]
Prelude> length a
5
```

10. Reverse:

- reverse takes a list and returns its reverse.
- Syntax: reverse L1
- E.g.

```
Prelude> let a = [1,2,3,4,5]
Prelude> reverse a
[5,4,3,2,1]
```

11. Take:

- take takes a number and a list. It extracts that many elements from the beginning of the list.
- Syntax: **take num L1**
- E.g.

```
Prelude> let a = [1,2,3,4,5]
Prelude> take (-1) a
[]
Prelude> take 0 a
[]
Prelude> take 1 a
[1]
Prelude> take 2 a
[1,2]
Prelude> take 3 a
[1,2,3]
Prelude> take 4 a
[1,2,3,4]
Prelude> take 5 a
[1,2,3,4,5]
Prelude> take 6 a
[1,2,3,4,5]
Prelude> take 10000 a
[1,2,3,4,5]
```

12. Drop:

- drop takes a number and a list and it removes that many elements from the list.

- Syntax: **drop num L1**
- E.g.

```
Prelude> let a = [1,2,3,4,5]
Prelude> drop (-1) a
[1,2,3,4,5]
Prelude> drop 0 a
[1,2,3,4,5]
Prelude> drop 1 a
[2,3,4,5]
Prelude> drop 2 a
[3,4,5]
Prelude> drop 3 a
[4,5]
Prelude> drop 4 a
[5]
Prelude> drop 5 a
[]
Prelude> drop 100 a
[]
```

13. Range/Sequence:

- Denoted by "**..**"
- Syntax: **[starting_element .. end_element]**
- E.g.

```
Prelude> print([1..10])
[1,2,3,4,5,6,7,8,9,10]
Prelude> print(['a'..'z'])
"abcdefghijklmnopqrstuvwxyz"
Prelude> print(['A' .. 'Z'])
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

14. Other facts:

- To show that a list only has 0 elements, you can do this: [].
- To show that a list only has 1 element, you can do this: [a] or (a:[]).
- To show that a list has 1 or more elements, you can do this: (a:_). The "_" means 0 or more items.
- There's also a thing called **as patterns**. Those are a handy way of breaking something up according to a pattern and binding it to names whilst still keeping a reference to the whole thing. You do that by putting a name and an **@** in front of a pattern. For instance, the pattern **xs@(x:y:ys)**. This pattern will match exactly the same thing as **x:y:ys** but you can easily get the whole list via **xs** instead of repeating yourself by typing out **x:y:ys** in the function body again.

- E.g.

```
getListLength1 :: [Integer] -> Integer
getListLength1 [] = 0
getListLength1 (a:[]) = 1 -- Another way of saying there's only 1 element. Equivalent to [a].
getListLength1 (a:b:[]) = 2 -- Another way of saying there's only 2 elements. Equivalent to [a,b].
getListLength1 (a:b:c) = 2 + x
    where x = getListLength1 c
```

```
*Main> getListLength1 []
0
*Main> getListLength1 [1]
1
*Main> getListLength1 [1,2,3,4,5,6,7,8,9]
9
```

- E.g.

```
newGetListLength1 :: [Integer] -> Integer
newGetListLength1 [] = 0
newGetListLength1 (_:a) = 1 + newGetListLength1 a
```

```
*Main> newGetListLength1 []
0
*Main> newGetListLength1 [1]
1
*Main> newGetListLength1 [1,2,3,4,5]
5
```

- E.g.

```
getListLength2 :: [Integer] -> String
getListLength2 [] = "0 elements"
getListLength2 (a:[]) = "1 elements" -- Another way of saying there's only 1 element. Equivalent to [a].
getListLength2 (a:b:_) = "2 or more elements."
```

```
*Main> getListLength2 []
"0 elements"
*Main> getListLength2 [1]
"1 elements"
*Main> getListLength2 [1,2]
"2 or more elements."
*Main> getListLength2 [1,2,3]
"2 or more elements."
```

- E.g.

```
capital :: String -> String
capital "" = "Empty string, whoops!"
capital all@(x:xs) = "The first letter of " ++ all ++ " is " ++ [x]
```

```
*Main> capital "ABCD"
"The first letter of ABCD is A"
```

Lambda Functions:

- It is an **anonymous function**, which is a function without giving it a name.
- A lambda function is denoted by the "****" character.
- Syntax: **\(var) -> (expression)**
  E.g.

```
main = do
    putStr("The next number after 5 is ")
    print((\x -> x + 1) 5)
```

```
*Main> main
The next number after 5 is 6
```

- **Note:** Lambda functions can be used as a substitute for missing parameters.
- If you intend 2 parameters, the Haskell culture is to model it as a nested function: **\x -> (\y -> 2*x - 3*y)** (those parentheses can be omitted). This creates a function that maps the 1st parameter to a function that takes the 2nd parameter. Doing this is called **currying**.
  The shorthand way of doing it is: **\x y -> 2*x - 3*y**
  E.g.

```
main2 = do
    putStr("The value of 2*3 - 3*1 is ")
    print((\x -> (\y -> 2*x - 3*y))3 1)

main3 = do
    putStr("The value of 2*3 - 3*1 is ")
    print((\x y -> 2*x - 3*y)3 1)
```

```
*Main> main2
The value of 2*3 - 3*1 is 3
*Main> main3
The value of 2*3 - 3*1 is 3
```

  Notice that both ways work and give the same result.
- Recall from earlier **diffSq x y = (x - y) * (x + y)**. This can be written as **diffSq = \x y -> (x - y) * (x + y)** or even **diffSq x = \y -> (x - y) * (x + y)**.
  E.g.

```
-- Example of diffSq using lambda functions.
-- The first example doesn't use currying while the second example does.
diffSq = \x y -> (x - y) * (x + y) -- Doesn't use currying.
diffSq2 x = \y -> (x - y) * (x + y) -- Uses currying.
```

```
*Main> diffSq 4 5
-9
*Main> diffSq2 4 5
-9
```

- When applying a function to 2 parameters, such as doing function a b, that's shorthand for (function a) b.

E.g.
diffSq 10 5 is shorthand for **(diffSq 10) 5**.

```
*Main> (diffSq 4) 5
-9
```

Compare this with the diffSq examples from above and notice that you get the same result.

- Note: It is possible to use "diffSq 10" alone. This is called a **partial application**. **Partial application** is when you decide to use a function but not give it all of the needed parameters. When it is evaluated, here is what happens:
  diffSq 10
  → (\x y -> (x - y) * (x + y)) 10
  → \y -> (10 - y) * (10 + y)
- Typewise, **X -> Y -> A** is shorthand for **X -> (Y -> A)**.

Higher Order Function:

- **Higher Order Functions** are a unique feature of Haskell where you can use a function as an input or output argument.
  Note: We use () to show that a function takes a function as an input.
- E.g.

```
four_plus_seven :: (Int -> Int) -> Int
four_plus_seven f = f 4 + f 7
```

```
*Main> four_plus_seven (\x -> x*2)
22
*Main> four_plus_seven (\x -> x^2)
65
*Main> four_plus_seven (\x -> x+2)
15
```

The first function multiplies each variable by 2. 4*2 + 7*2 = 22.
The second function squares each variable. $4^2 + 7^2 = 65$.
The third function increases each variable by 2. 4+2 + 7+2 = 15.
In the first picture above, **(Int -> Int)** shows that four_plus_seven takes in a function as an input and that function takes in an Int as an input and outputs something of type Int.

- E.g.

```
four_plus_seven :: (Int -> Int) -> Int
four_plus_seven f = f 4 + f 7

random_function :: Int -> Int
random_function x = 5*x + 12
```

```
*Main> four_plus_seven (random_function)
79
```

5*4 + 12 = 32
5*7 + 12 = 47
32 + 47 = 79

Parametric Polymorphism:

- A **polymorphic** function is a function that works for many different types.
- **Polymorphic:** Can become one of many types.

- **Monomorphic:** Stuck with being one single type.
- Also known as **generics** in other languages.
- **Type variables** always begin in lowercase whereas **concrete types** like Int or String always start with an uppercase letter.
- Just as a variable represents some value of a given type, a **type variable** represents some type. A **type variable** represents one type across the type signature and function definition in the same way a variable represents a value throughout the scope it's defined in.
- E.g.

```
add :: Int -> Int -> Int
add x y = x + y

add2 :: Num x => x -> x -> x
add2 a b = a + b
```

```
*Main> add 2 3
5
*Main> add 2.0 3.0

<interactive>:124:5: error:
    • No instance for (Fractional Int) arising from the literal ‘2.0’
    • In the first argument of ‘add’, namely ‘2.0’
      In the expression: add 2.0 3.0
      In an equation for ‘it’: it = add 2.0 3.0
*Main> add2 2 3
5
*Main> add2 2.0 3.0
5.0
*Main> add2 2.1 3.1
5.2
*Main> add2 2 3.4
5.4
*Main> add2 2 (-1.5)
0.5
```

In the add function, only Integers are allowed. Hence, when I tried doing add 2.0 3.0, I got an error. However, in add2, as long as the inputs are numbers, I can add integers, floats or a mix.

Note: **Num x =>** just means that x must be of type Num, or x must be a number. I need to put this or else I get an error. This is because if I don’t specify the type, I could, theoretically, add 2 non-numbers, which would cause an error. Hence, Haskell mandated that I put the **Num x =>** part.

- E.g.

rep2 :: a -> [a]

In **a -> [a]**, the “a” there is a **type variable** or **type parameter**. Names of type variables are up to you, doesn't have to be “a”, but does have to start with lowercase.
E.g. element, myElementType, etc

Note: Type constants/Concrete types, names of built-in types and defined types, start with uppercase.
E.g. Integer, Bool, String

- **Note:** The choice of the type is up to the user, not the provider. Furthermore, in parametric polymorphism, the "parametric" part means that the provider is not told what the user chooses. As a result, the code can be inflexible. However, it's easy to test your code.
- Generally, flexibility for the implementer is in direct conflict with predictability for the user and vice versa.

Map:

- A **map** is the name of a higher-order function that applies a given function to each element of a functor, such as a list, returning a list of results in the same order. It is often called apply-to-all when considered in functional form.
- Can be written in 2 ways:
    1. map :: (a -> b) -> [a] -> [b]
    2. map :: (a -> b) -> ([a] -> [b])
- E.g.

```
square :: Int -> Int
square x = x ^ 2
```

```
*Main> map square [1,2,3,4,5,6,7,8,9,10]
[1,4,9,16,25,36,49,64,81,100]
*Main> square [1,2,3,4,5,6,7,8,9,10]

<interactive>:13:8: error:
    • Couldn't match expected type ‘Int’ with actual type ‘[Integer]’
    • In the first argument of ‘square’, namely ‘[1, 2, 3, 4, ....]’
      In the expression: square [1, 2, 3, 4, ....]
      In an equation for ‘it’: it = square [1, 2, 3, ....]
```

By having the word "map", it allowed me to use the square function on a list.

- E.g.

```
import Data.Char
ascii_conversion x = ord x
```

```
*Main> ascii_conversion ['a', 'b']

<interactive>:16:18: error:
    • Couldn't match expected type ‘Char’ with actual type ‘[Char]’
    • In the first argument of ‘ascii_conversion’, namely ‘['a', 'b']’
      In the expression: ascii_conversion ['a', 'b']
      In an equation for ‘it’: it = ascii_conversion ['a', 'b']
*Main> map ascii_conversion ['a', 'b']
[97,98]
```

Notice that when I put the keyword "map" at the beginning, I can run ascii_conversion on a list.

- What it does by example:
  map ord ['a', 'b', 'c']
  = [ord 'a', ord 'b', ord 'c']
  = [97, 98, 99]
- Consider this: map (map ord) [['a', 'b', 'c'], ['x', 'y', 'z']] :: [[Int]]
  Detailed type breakdown:
  - inner map :: (Char -> Int) -> ([Char] -> [Int])
    (I'm choosing a=Char, b=Int)
  - map ord :: [Char] -> [Int]
  - outer map :: ([Char] -> [Int]) -> [[Char]] -> [[Int]]
    (I'm choosing a=[Char], b=[Int])
  - map (map ord) :: [[Char]] -> [[Int]]
- What it does by example:
  map (map ord) [['a', 'b', 'c'], ['x', 'y', 'z']]
  = [map ord ['a', 'b', 'c'], map ord ['x', 'y', 'z']]
  = [[ord 'a', ord 'b', ord 'c'], [ord 'x', ord 'y', ord 'z']]
  = [[97, 98, 99], [120, 121, 122]]
- E.g.

```
Prelude Data.Char> map (map (map ord)) [[['a', 'b', 'c'], ['x', 'y', 'z']]]
[[[97,98,99],[120,121,122]]]
```

- **Note:** To use ord, you need to do import Data.Char.

Type-specific behaviour preview:

- Consider the code below:

```
square :: a -> a
square x = x^2
```

The square function takes in a number and returns the square of that number. Since any number, not just integers, can be squared, we want to use parametric polymorphism. However, there's an issue. What happens if the user enters a string or boolean? To avoid this problem, we have to do the following:

```
square :: Num a => a -> a
square x = x^2
```

By putting the Num a => part, we are saying that "a" must be a number.

User-defined types:

- Also called **algebraic data types**.
- We can define our own types using the keyword **data**.
- Each option must start with an uppercase letter.
- We use | to say alternatively.
- There needs to be at least one case, and each case can have 0 or more fields.

- E.g.

```
-- Created a new data type called Area.
data Area
    = Circle Float -- Case 1: Circle. It has one field, float.
    | Square Float Float -- Case 2: Square. It has 2 fields, float, float.
    | Triangle Float Float -- Case 3: Triangle. It has 2 fields, float, float.

surface :: Area -> Float
surface (Circle r) = pi * r ^ 2
surface (Square b h) = b * h
surface (Triangle b h) = b * h / 2
```

```
*Main Data.Char> surface $ Circle 10
314.15927
*Main Data.Char> surface $ Square 2 2
4.0
*Main Data.Char> surface $ Triangle 2 3
3.0
```

```
*Main Data.Char> :t Circle
Circle :: Float -> Area
*Main Data.Char> :t Square
Square :: Float -> Float -> Area
*Main Data.Char> :t Triangle
Triangle :: Float -> Float -> Area
```

Here, the type name is Area.
Circle, Square and Triangle are called **data constructors** or **tags**. As stated before, all these data constructors must be capitalized.
Note: These are not OOP constructors. It's only labelling, not arbitrary initialization code.
Note: These are not OOP subclasses/subtypes either. Circle is not a subtype, it's a term and value.

Recursive types:

- A recursive data type is a data definition that refers to itself.
- This lets us define even more interesting data structures such as linked lists and trees.
- The line, **deriving (Eq, Show)**, is called the **deriving clause**. It specifies that we want the compiler to automatically generate instances of the Eq and Show classes. The EQ type class is an interface which provides the functionality to test the equality of an expression. The Show type class has a functionality to print its argument as a String. Whatever may be its argument, it always prints the result as a String.

- E.g.

```haskell
data IntList = EndOfIntList | ValAndNext Integer IntList
  deriving(Show, Eq)

example = ValAndNext 0 (ValAndNext 1 (ValAndNext 2 (ValAndNext 3 (ValAndNext 4 (ValAndNext 5 EndOfIntList)))))

-- numList makes a list of numbers from 0 to n.
numList :: Integer -> IntList
numList n = make 0
    where
        make i | i > n = EndOfIntList
               | otherwise = ValAndNext i (make (i+1))

myISum :: IntList -> Integer
myISum EndOfIntList = 0
myISum (ValAndNext x xs) = x + myISum xs
```

```
*Main Data.Char> numList 5
ValAndNext 0 (ValAndNext 1 (ValAndNext 2 (ValAndNext 3 (ValAndNext 4 (ValAndNext 5 EndOfIntList)))))
*Main Data.Char> myISum example
15
```

Recursion & Lists:

- E.g. Consider the example below:

  A value of type MyIntegerList is one of:
  1. INil
  2. ICons x xs, if x::Integer and xs::MyIntegerList

```
data MyIntegerList = INil | ICons Integer MyIntegerList
  deriving (Show, Eq)

exampleMyIntegerList = ICons 4 (ICons (-10) INil)

-- `from0to n` builds a MyIntegerList from 0 to n-1
from0to :: Integer -> MyIntegerList
from0to n = make 0
 where
   make i | i >= n = INil
          | otherwise = ICons i (make (i+1))

mylSum :: MyIntegerList -> Integer
mylSum INil = 0
mylSum (ICons x xs) = x + mylSum xs
```

Recursion & Binary Trees:

- E.g. Consider the example below:

  A value of type IntegerBST is one of:
  1. IEmpty
  2. INode lt x rt, if lt::IntegerBST, x::Integer, rt::IntegerBST

```
data IntegerBST = IEmpty | INode IntegerBST Integer IntegerBST
  deriving Show

exampleIntegerBST = INode (INode IEmpty 3 IEmpty) 7 (INode IEmpty 10 IEmpty)

ibstInsert :: Integer -> IntegerBST -> IntegerBST
ibstInsert k IEmpty =
  INode IEmpty k IEmpty
ibstInsert k inp@(INode left key right)
  | k < key = INode (ibstInsert k left) key right
  | k > key = INode left key (ibstInsert k right)
  | otherwise = inp   -- INode left key right
```

Note: Since this is functional programming with immutable trees, “insert” means produce a new tree that is like the input tree but with the new key. Maybe it's better to say “the tree plus k”.

Polymorphic Types:

- Consider the example below:

```
data MyList a = Nil | Cons a (MyList a) deriving (Eq, Show)

exampleMyListI :: MyList Integer
exampleMyListI = Cons 4 (Cons (-10) Nil)

exampleMyListS :: MyList String
exampleMyListS = Cons "albert" (Cons "bart" Nil)
```

  These are homogeneous lists. They can't have different item types in the same list. For example, Cons "albert" (Cons True Nil) is illegal because what would be its type, MyList String? MyList Bool?
- Some polymorphic algebraic data types from the standard library as further examples:
- Maybe:

```
data Maybe a = Nothing | Just a
-- Great for: Sometimes there is no answer
```

- Either:

```
data Either a b = Left a | Right b
-- Great for: Having two possible types.
```

Evaluation Order

- Most languages use **call by value** for evaluation order.
  I.e. To evaluate f(x,y), evaluate x and y first (which one first depends on the language), then plug into f's body, and then evaluate the body.
- E.g. If there is a function defined as f(x, y) = x:
  f (3+4, div(4, 2)) eval a parameter, arithmetic
  → f (7, div(4, 2)) eval the other parameter, arithmetic
  → f (7, 2) ready to plug in at last
  → 7
- However, a problematic parameter can cause an error/exception even if it would be unused:
  f (3+4, div(1, 0)) eval a parameter, arithmetic
  → f (7, div(1, 0)) eval the other parameter, arithmetic
  → Error caused by div(1,0)
- Haskell uses "**lazy evaluation**." **Lazy evaluation** is also known as **call by need**.
- Lazy evaluation in Haskell (sketch):
  - To evaluate "f x y": don't evaluate x and y first. Just plug x and y into f's right hand side (RHS) and evaluate that.
    If the RHS refers to the same parameter multiple times: same shared copy, no duplication.
  - If that runs you into pattern matching: evaluate parameter(s) just enough to decide whether it's a match or non-match. If match, plug into RHS and evaluate. If it's a non-match, try the next pattern. (If it runs out of patterns, declare "undefined" aka "error".)
  - To evaluate arithmetic operations, use call-by-value.
- E.g.

```
doITerminate = take 2 (from 0)
  where
    from n = n : from (n + 1)
```

```
*Main> doITerminate
[0,1]
```

- E.g.

```
doIEvenMakeSense = take 2 zs
  where
    zs = 0 : zs
```

```
*Main> doIEvenMakeSense
[0,0]
```

Take Function in Haskell:

- The take function takes a number and a list and returns the first n elements of the list, where n is the number inputted.
- E.g. **take 3 [a,b,c,d,e] = [a,b,c]**
- E.g. **take 3 [a,b] = [a,b]**
- The implementation goes like this:
  take 0 _ = []

take _ [] = []
take n (x:xs) = x : take (n-1) xs

Single Linked List:

- Recall that lists in Haskell are linked lists.
- Singly-linked list is a very space-consuming data structure (all languages). And if you ask for “the ith item” you're doing it wrong.
- E.g.

Newton's method with lazy lists. Like in Hughes's «why FP matters». Approximately solve $x^3 - b = 0$, i.e., cube root of b.

So $f(x) = x^3 - b$, $f'(x) = 3x^2$

```
x1 = x - f(x)/f'(x)
   = x - (x^3 - b)/(3x^2)
   = x - (x - b/x^2)/3
   = (2x + b/x^2)/3
```

The local function “next” below is responsible for computing x1 from x.

```
cubeRoot b = within 0.001 (iterate next b)
    -- From the standard library:
    -- iterate f z = z : iterate f (f z)
    --             = [z, f z, f (f z), f (f (f z)), ...]
  where
    next x = (2*x + b/x^2) / 3
    within eps (x : x1 : rest)
        | abs (x - x1) <= eps = x1
        | otherwise = within eps (x1 : rest)
```

Equivalently, using the function composition operator “**.**”, we get:
cubeRoot = within 0.001 . iterate next

- With this, you really have a pipeline like Unix pipelines.
- If you use lists lazily in Haskell, it is an excellent control structure—a better for-loop than for-loops. Then list-processing functions become pipeline stages. If you do it carefully, it is even O(1)-space. If furthermore you're lucky (if the compiler can optimize your code), it can even fit entirely in registers without node allocation and GC overhead.
- Thinking in high-level pipeline stages is both more sane and more efficient—with the right languages.
- Some very notable list functions when you use lists lazily as for-loops, or when you think in terms of pipeline stages:
    - **Producers:** repeat, cycle, replicate, iterate, unfoldr, the [x..], [x..y] notation (backed by enumFrom, enumFromTo)
    - **Transducers:** map, filter, scanl, scanr, (foldr too, sometimes) take, drop, splitAt, takeWhile, dropWhile, span, break, partition, zip, zipWith, unzip
    - **Consumers:** foldr, foldl, foldl', length, sum, product, maximum, minimum, and, all, or, any
- A **producer** is some monadic action that can yield values for downstream consumption.
- A **consumer** can only await values from upstream.
- A **transducer** is like a combination of both producers and consumers.
- E.g. of iterate:

```
*Main> take 10 (iterate (\x -> x+1) 4)
[4,5,6,7,8,9,10,11,12,13]
```

When lazy evaluation hurts:

- E.g. Consider the code below:

```
mySumV2 xs = g 0 xs
  where
    g accum [] = accum
    g accum (x:xs) = g (accum + x) xs
```

It takes a number, 0, and a list of numbers and computes the sum of the numbers in the list.

```
*Main> mySumV2 [1,2,3]
6
*Main> mySumV2 [1,2,3,4,5,6]
21
*Main> mySumV2 [1..10]
55
```

Evaluation of mySumV2 [1,2,3]:

mySumV2 (1 : 2 : 3 : []) plug in
→ g 0 (1 : 2 : 3 : []) match, plug in
→ g (0 + 1) (2 : 3 : []) match, plug in
→ g ((0 + 1) + 2) (3 : []) match, plug in
→ g (((0 + 1) + 2) + 3) [] match, plug in
→ ((0 + 1) + 2) + 3 arithmetic at last
→ (1 + 2) + 3 ditto
→ 3 + 3 ditto
→ 6

This takes Ω(n) space for the postponed arithmetic.

- **Note:** If there is recursion, you bracket right to left. If there is no recursion, you bracket left to right. If you look at the example of mySumV2 [1,2,3], you'll see that it's bracketed left to right. I.e. **(((0 + 1) + 2) + 3)**
  Consider the below example:

mySum [] = 0
mySum (x:xt) = x + mySum xt

mySum [1,2,3]
→ 1 + (mySum (2 : 3 : []))
→ 1 + (2 + (mySum (3 : [])))
→ 1 + (2 + (3 + (mySum ([]))))
→ 1 + (2 + (3 + (0)))
→ 1 + (2 + (3 + 0))
→ 1 + (2 + 3)
→ 1 + 5
→ 6

Notice how because there's recursion, the brackets are right heavy.

Type Constructor vs Data Constructor:

- A **type constructor** is a function that takes 0 or more types and gives you back a new type. If it has zero arguments it is called a **nullary type constructor** or simply a **type**.
  Note: All type constructors must start with a capital letter.
- A **data constructor/tag** is a function that takes 0 or more values and gives you back a new value. If it has zero arguments is called a **nullary data constructor** or simply a **constant**.
  Note: All data constructors must start with a capital letter.
- In a data declaration, a type constructor is the thing on the left hand side of the equals sign. The data constructor(s) are the things on the right hand side of the equals sign. You use type constructors where a type is expected, and you use data constructors where a value is expected.
- Examples:
  1. This is an example of a nullary type constructor with 2 nullary data constructors:
     data Bool = True | False
  2. This is an example of a unary type constructor:
     data Tree a = Tip | Node a (Tree a) (Tree a)
     Here, “a” is a type variable.

Examples:

1. Consider the code and output below:

```
data BinaryTree = Empty | Node BinaryTree Integer BinaryTree
    deriving Show

insert :: Integer -> BinaryTree -> BinaryTree
insert k Empty = Node Empty k Empty
insert k bt@(Node left key right)
    |k < key = Node (insert k left) key right
    |k > key = Node left key (insert k right)
    |otherwise = bt

find:: Integer -> BinaryTree -> Maybe Integer
find k Empty = Nothing
find k (Node left key right)
    |k == key = Just(k)
    |k < key = find k left
    |k > key = find k right
```

```
*Main> x = insert 10 Empty
*Main> insert 9 x
Node (Node Empty 9 Empty) 10 Empty
*Main> insert 11 x
Node Empty 10 (Node Empty 11 Empty)
*Main> y = insert 9 x
*Main> insert 11 y
Node (Node Empty 9 Empty) 10 (Node Empty 11 Empty)
```

```
*Main> x = insert 10 Empty
*Main> y = insert 20 x
*Main> z = insert 14 y
*Main> z
Node Empty 10 (Node (Node Empty 14 Empty) 20 Empty)
*Main> find 14 z
Just 14
*Main> find 10 z
Just 10
*Main> find 20 z
Just 20
*Main> find 100 z
Nothing
*Main> find 0 z
Nothing
```

The line "**data BinaryTree = Empty | Node BinaryTree Integer BinaryTree**" means that BinaryTree is either Empty or Node BinaryTree Integer BinaryTree. Hence, when we output a BinaryTree for insert, we either have to output Empty or Node BinaryTree Integer BinaryTree.

2. Consider the code and output below:

```
data Boolean = T | F
    deriving(Show, Eq)

sumChecker :: (Num a, Ord a) => a -> a -> a -> Boolean
sumChecker x y z = if (x + y) == z
    then
        T
    else
        F

isBigger :: (Num a, Ord a) => a -> a -> Boolean
isBigger x y = if (x > y)
    then
        T
    else
        F

isSmaller :: (Num a, Ord a) => a -> a -> Boolean
isSmaller x y = if (x < y)
    then
        T
    else
        F

isEqual :: (Num a, Ord a) => a -> a -> Boolean
isEqual x y = if (x == y)
    then
        T
    else
        F
```

```
*Main> sumChecker 1 2 3
T
*Main> sumChecker 1 2 4
F
*Main> isBigger 1 2
F
*Main> isBigger 2 1
T
*Main> isSmaller 1 2
T
*Main> isSmaller 2 1
F
*Main> isEqual 1 2
F
*Main> isEqual 1 1
T
```

Here, Boolean is a type constructor while T and F are data constructors. As stated above, Boolean, T and F must be capitalized.

Seq:

- seq is used for killing lazy evaluation where you deem it unsuitable.
- It is a built in function.
- seq is a special function that is used to force expressions to be evaluated. seq evaluates the first parameter and passes it to the second parameter.
- To evaluate "seq x y": evaluate x to "**weak head normal form**", then continue with y.
- Weak head normal form (WHNF) means:
  - for built-in number types: until you have the number
  - for algebraic data types: until you have a data constructor
  - for functions: until you have a lambda
- Naturally, "seq x y" is most meaningful when x is something that y will need.
- E.g. mySumV2, mySumV3 and mySumV4 are used to sum a list.

```
mySumV2 xs = g 0 xs
  where
    g accum [] = accum
    g accum (x:xs) = g (accum + x) xs
```

Evaluation of mySumV2 [1,2,3]:
mySumV2 (1 : 2 : 3 : [])
→ g 0 (1 : 2 : 3 : [])
→ g (0 + 1) (2 : 3 : [])
→ g ((0 + 1) + 2) (3 : [])
→ g (((0 + 1) + 2) + 3) []
→ ((0 + 1) + 2) + 3
→ (1 + 2) + 3
→ 3 + 3
→ 6

Now, we will use seq.

```
mySumV3 xs = g 0 xs
  where
    g accum [] = accum
    g accum (x:xs) = seq accum (g (accum + x) xs)
    -- Note: accum is something that "g (accum + x) xs" will need.
```

Evaluation of mySumV3 [1,2,3]:
g 0 (1 : 2 : 3 : [])
→ seq 0 (g (0 + 1) (2 : 3 : []))
→ g (0 + 1) (2 : 3 : [])
→ seq a (g (a + 2) (3 : [])) with a = 0 + 1
→ seq a (g (a + 2) (3 : [])) with a = 1
→ g (1 + 2) (3 : [])
→ seq b (g (b + 3) []) with b = 1 + 2
→ seq b (g (b + 3) []) with b = 3

→ g (3 + 3) []
→ 3 + 3
→ 6

We can still decrease number of the iterations:

```
mySumV4 xs = g 0 xs
  where
    g accum (x:xs) = let a1 = accum + x
                     in seq a1 (g a1 xs)
    -- Note: a1=accum+x is something that "g a1 xs" will need.
```

g 0 (1 : 2 : 3 : [])
→ seq b (g b (2 : 3 : [])) with b = 0 + 1
→ seq b (g b (2 : 3 : [])) with b = 1
→ g 1 (2 : 3 : [])
→ seq c (g c (3 : [])) with c = 1 + 2
→ seq c (g c (3 : [])) with c = 3
→ g 3 (3 : [])
→ seq d (g d []) with d = 3 + 3
→ seq d (g d []) with d = 6
→ g 6 []
→ 6

Fold Functions:

- Back when we were dealing with recursion, we noticed a theme throughout many of the recursive functions that operated on lists. Usually, we'd have an edge case for the empty list. We'd introduce the x:xs pattern and then we'd do some action that involves a single element and the rest of the list. It turns out this is a very common pattern, so a couple of very useful functions were introduced to encapsulate it. These functions are called **folds**. They're sort of like the map function, only they reduce the list to some single value.
- A fold takes a binary function, a starting value, called an **accumulator**, and a list to fold up. The binary function itself takes two parameters. The binary function is called with the accumulator and the first or last element and produces a new accumulator. Then, the binary function is called again with the new accumulator and the now new first or last element, and so on. Once we've walked over the whole list, only the accumulator remains, which is what we've reduced the list to.
- Folds can be used to implement any function where you traverse a list once, element by element, and then return something based on that. Whenever you want to traverse a list to return something, chances are you want a fold. That's why folds are, along with maps and filters, one of the most useful types of functions in functional programming.
- Foldl and foldr are two fold functions.

Foldl:

- Also called the left fold.
- It folds the list up from the left side. The binary function is applied between the starting value and the head of the list. That produces a new accumulator value and the binary function is called with that value and the next element, etc.

- E.g. Consider **foldl (\acc x -> acc + x) 0 xs**, 0 is the accumulator or starting value, xs is the list, acc is the accumulator value and x is the first element of the list.
- E.g. The below functions all take in a list of numbers and sum up the numbers.

```
mySum :: (Num a)=> [a] -> a
mySum xs = foldl (\acc x -> acc + x) 0 xs

mySum2 :: (Num a)=> [a] -> a
mySum2 = foldl (+) 0
```

```
*Main> mySum [3,5,3,1]
12
*Main> mySum2 [3,5,3,1]
12
```

Here's the evaluation of mySum [3,5,3,1]:
→ **0** + **3** [3, 5, 3, 1]
→ **3** + **5** [5, 3, 1]
→ **8** + **3** [3, 1]
→ **11** + **1** [1]
→ **12**
The bolded dark green numbers represent the accumulator value.

- **Note:** Instead of **(\acc x -> acc + x)**, we can use **(+)** instead.
- **Note:** In the second example, we can omit the xs as the parameter because calling foldl (+) 0 will return a function that takes a list. Generally, if you have a function like foo a = bar b a, you can rewrite it as foo = bar b, because of currying.
- E.g.

```
reverseList :: [a] -> [a]
reverseList xs = foldl (\a x -> x:a) [] xs
```

```
*Main> reverseList [1,2,3]
[3,2,1]
*Main> reverseList [5, 4, 2, 1]
[1,2,4,5]
```

Foldr:

- The right fold, foldr, works in a similar way to the left fold, only the accumulator eats up the values from the right. Also, the left fold's binary function has the accumulator as the first parameter and the current value as the second one (so \acc x -> ...), the right fold's binary function has the current value as the first parameter and the accumulator as the second one (so \x acc -> ...). It kind of makes sense that the right fold has the accumulator on the right, because it folds from the right side.

- E.g.

```
mySum3 :: (Num a)=> [a] -> a
mySum3 xs = foldr (\x acc -> acc + x) 0 xs

mySum4 :: (Num a)=> [a] -> a
mySum4 = foldr (+) 0
```

```
*Main> mySum3 [3,5,3,1]
12
*Main> mySum4 [3,5,3,1]
12
```

Here's the evaluation of mySum3 [3,5,3,1]:
→ **0** + **1** [3, 5, 3, 1]
→ **1** + **3** [3, 5, 3]
→ **4** + **5** [3, 5]
→ **9** + **3** [3]
→ **12**
The bolded dark green numbers represent the accumulator value.

- One big difference is that right folds work on infinite lists, whereas left ones don't. To put it plainly, if you take an infinite list at some point and you fold it up from the right, you'll eventually reach the beginning of the list. However, if you take an infinite list at a point and you try to fold it up from the left, you'll never reach an end.
- **Note:** If you have **foldr op z (xs:xt)**
  where
  z = …
  op x r = …
  z is the starting value, x is the first element/value of the list and r = **foldr op z xt**.

Foldl1 and foldr1:

- The foldl1 and foldr1 functions work much like foldl and foldr, only you don't need to provide them with an explicit starting value. They assume the first or last element of the list to be the starting value and then start the fold with the element next to it. With that in mind, the sum function can be implemented like so: sum = foldl1 (+). Because they depend on the lists they fold up having at least one element, they cause runtime errors if called with empty lists. foldl and foldr, on the other hand, work fine with empty lists. When making a fold, think about how it acts on an empty list. If the function doesn't make sense when given an empty list, you can probably use a foldl1 or foldr1 to implement it.

Type Classes:

- In Haskell, every statement is considered as a mathematical expression and the category of this expression is called as a Type. You can say that "Type" is the data type of the expression used at compile time.
- In a generic way, Type can be considered as a value, whereas Type Class can be considered as a set of similar kinds of Types.
- A "type class" declares a group of overloaded operations ("methods").
- Syntax:
  class ClassName typeVar where

```
methodName :: type sig containing typeVar
-- Optional: default implementations
```

- Example: Methods == and /= are grouped under the Eq class. Its declaration in the standard library goes like this:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  -- default implementation for (==)
  x == y = not (x /= y)
  -- default implementation for (/=)
  x /= y = not (x == y)
  -- default implementations are deliberately circular so you just have to
  -- implement one of them to break the cycle
```

The role of type variable "a" is a placeholder for Integer, Char, etc.

To implement these methods for a type, e.g., the standard library has this for Bool:

```
instance Eq Bool where    -- (so a=Bool here)
  False == False    = True
  True == True      = True
  _ == _            = False

  -- default implementation for (/=) takes hold
```

We say "Bool is an instance of Eq".

- **Note:** When you do "instance Num a where …", you are making "a" a Num type.
- WARNING:
  1. A class is not a type. Eq is not a type. These are illegal:
     foo :: Eq -> Eq -> Bool
     bar :: Eq a -> Eq a -> Bool
  2. A type is not a "subclass". Bool is not a "subclass" of Eq.
- Method types outside classes look like this:
  (==) :: Eq a => a -> a -> Bool
  The additional "Eq a =>" is marked for polymorphic but the user's choice of a must be an instance of Eq. This marker also appears when you write polymorphic functions using the methods.
- Type classes use => while types use ->.
  I.e. **function :: (Type Class) => (type 1) -> … -> (type n)**
  E.g.
  sum :: (Num a) => a -> a -> a
- Built in Types and Type Classes:
  - **Int:** Int is a type representing the Integer types data. Every whole number within the range of 2147483647 to -2147483647 comes under the Int type class.
  - **Integer:** Integer can be considered as a superset of Int. This value is not bounded by any number, hence an Integer can be of any length without any limitation.
  - **Float:** Float is a floating point number with single precision at the end.
  - **Double:** Double is a floating point number with double precision at the end.
  - **Bool:** Bool is a Boolean Type. It can be either True or False.

- **Char:** Char represents Characters. Anything within a single quote is considered as a Character.
- **EQ:** EQ type class is an interface which provides the functionality to test the equality of an expression. Any Type class that wants to check the equality of an expression should be a part of this EQ Type Class. Whenever we are checking any equality using any of the types mentioned above, we are actually making a call to the EQ type class. EQ is used for == or !=.
- **Ord:** Ord is another type class which gives us the functionality of ordering. Like EQ interface, Ord interface can be called using ">", "<", "<=", ">=", "compare". An instance of Ord is also an instance of Eq. We say “Ord is a subclass of Eq”, but beware that this is unrelated to OOP subclasses.
- **Show:** Show is a type class that has a functionality to print its argument as a String. Whatever may be its argument, it always prints the result as a String.
- **Read:** Read is a type class that does the same thing as Show, but it won’t print the result in String format.
- **Enum:** Enum is another Type class which enables the sequential or ordered functionality in Haskell.
- **Bounded:** All the types having upper and lower bounds come under this Type Class.
  E.g.

```
main = do
   print (maxBound :: Int)
   print (minBound :: Int)
```

```
sh-4.3$ main
9223372036854775807
-9223372036854775808
```

- Number operations are grouped into several type classes:
  - **Num:**
    - some methods: +, -, *, abs
    - instances: all number types
  - **Integral:**
    - some methods: div, mod
    - instances: Int, Integer
  - **Fractional:**
    - some methods: /, recip
    - instances: Rational, Float, Double, Complex a
- E.g. Why is the following a type error?

```
let xs :: [Double]
   xs = [1, 2, 3]
in sum xs / length xs
```

Answer:
sum xs :: Double, but length xs :: Int
(/) wants the two operands to be of the same type.
How to fix: sum xs / fromIntegral (length xs) or use realToFrac

```
fromIntegral :: (Integral a, Num b) => a -> b
realToFrac :: (Real a, Fractional b) => a -> b
```

- Often it is straightforward but boring to write instances for these classes, so the computer offers to auto-gen for you. However, restrictions apply. You can request it at the definition of your algebraic data type like this:
  data MyType = ... deriving (Eq, Ord, Bounded, Enum, Show, Read)

Foldable:

- The Foldable type class provides a generalisation of list folding (foldr and friends) and operations derived from it to arbitrary data structures. Besides being extremely useful, Foldable is a great example of how monoids can help formulating good abstractions.
- The purpose of this section is twofold:
  1. This explains why some of the library functions for lists have types like
     length :: Foldable t => t a -> Int
     instead of the simpler
     length :: [a] -> Int
  2. This familiarizes you with things like “Foldable t” and how it is not “Foldable (t a)”.
- E.g.

```
Prelude> :type length
length :: Foldable t => t a -> Int
Prelude> :type minimum
minimum :: (Foldable t, Ord a) => t a -> a
Prelude> :type sum
sum :: (Foldable t, Num a) => t a -> a
Prelude> :type foldr
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
Prelude> :type foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
```

- A few library functions that consume a list and compute a “summary” are:
  - **length :: [a] -> Int**
  - **sum :: Num a => [a] -> a**
  - **minimum :: Ord a => [a] -> a**
    -- assumes non-empty list
  - **foldr :: (a -> b -> b) -> b -> [a] -> b**

  These make sense for other data structures representing sequences too, not just linked lists.
  E.g. sum should be used for vector and seq, too.

```
sum :: Num a => Vector a -> a
-- Vector is an array, 0-based Int index. Third-party but popular library.

sum :: Num a => Seq a -> a
-- Seq is a middle ground between array and linked list,
-- O(1) prepend and append, log time random access.
```

  Haskell supports this generalization with a type class:

```
class Foldable t where
  length :: t a -> Int    -- implicitly ∀a, similarly below
  sum :: Num a => t a -> a
  minimum :: Ord a => t a -> a
  foldr :: (a -> b -> b) -> b -> t a -> b    -- implicitly ∀a,b
```

```
-- and others
```

Note: It is not “class Foldable (t a)”. Likewise, instances go like “instance Foldable []”, not “instance Foldable ([] a)”.

Good type classes and bad type classes:

- A good type class has these traits:
  - You have multiple instances.
  - Methods satisfy useful laws or expectations, therefore can be used to build useful general algorithms.
  - Example: Ord: <= is reflexive, transitive, anti-symmetric, total. These laws are the basis of sorting algorithms and binary search tree algorithms.
- Bad type class: Created for no further benefit than:
  - Common method name.
  - Procrastinating writing actual code, procrastinating making up your mind what to do.

Fold:

- A fold takes a binary function, a starting value, called the **accumulator**, and a list to fold up. The binary function itself takes two parameters. The binary function is called with the accumulator and the first or last element and produces a new accumulator. Then, the binary function is called again with the new accumulator and the now new first or last element, and so on. Once we've walked over the whole list, only the accumulator remains, which is what we've reduced the list to.

Foldl:

- Here's the function definition and implementation of foldl:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
-- if the list is empty, the result is the initial value; else
-- we recurse immediately, making the new initial value the result
-- of combining the old initial value with the first element.
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

- **Note:** foldl consumes the list left to right and evaluates from left to right.
- With foldl, the binary function has the accumulator as the first parameter and the current value as the second one.
- E.g. foldl (-) 0 [1..10] = -55

  foldl (-) 0 [1..10]
  → foldl (-) (0 - 1) [2..10]
  → foldl (-) ((0-1) - 2) [3..10]
  → foldl (-) (((0-1) - 2) - 3) [4..10]
  → foldl (-) ((((0-1) - 2) - 3) - 4) [5..10]
  → foldl (-) (((((0-1) - 2) - 3) - 4) - 5) [6..10]
  → foldl (-) ((((((0-1) - 2) - 3) - 4) - 5) - 6) [7..10]
  → foldl (-) (((((((0-1) - 2) - 3) - 4) - 5) - 6) - 7) [8..10]
  → foldl (-) ((((((((0-1) - 2) - 3) - 4) - 5) - 6) - 7) - 8) [9..10]
  → foldl (-) (((((((((0-1) - 2) - 3) - 4) - 5) - 6) - 7) - 8) - 9) [10]
  → foldl (-) ((((((((((0-1) - 2) - 3) - 4) - 5) - 6) - 7) - 8) - 9) - 10) []
  → ((((((((((0-1) - 2) - 3) - 4) - 5) - 6) - 7) - 8) - 9) - 10)
  → (((((((((-1 - 2) - 3) - 4) - 5) - 6) - 7) - 8) - 9) - 10)
  → (((((((-3 - 3) - 4) - 5) - 6) - 7) - 8) - 9) - 10)
  → (((((((-6 - 4) - 5) - 6) - 7) - 8) - 9) - 10)
  → ((((((-10 - 5) - 6) - 7) - 8) - 9) - 10)
  → (((((-15 - 6) - 7) - 8) - 9) - 10)
  → ((((-21 - 7) - 8) - 9) - 10)
  → (((-28 - 8) - 9) - 10)
  → ((-36 - 9) - 10)
  → (-45 - 10)
  → -55

Foldr:

- Here's the function definition and implementation of foldr:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
-- if the list is empty, the result is the initial value z; else
-- apply f to the first element and the result of folding the rest
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

- **Note:** foldr consumes the list left to right but evaluates from right to left.
- With foldl, the binary function has the current value as the first parameter and the accumulator as the second one.
- E.g. foldr (-) 0 [1..10] = -5

```
foldr (-) 0 [1..10]
→ 1 - (foldr (-) 0 [2..10])
→ 1 - (2 - (foldr (-) 0 [3..10]))
→ 1 - (2 - (3 - (foldr (-) 0 [4..10])))
→ 1 - (2 - (3 - (4 - (foldr (-) 0 [5..10]))))
→ 1 - (2 - (3 - (4 - (5 - (foldr (-) 0 [6..10])))))
→ 1 - (2 - (3 - (4 - (5 - (6 - (foldr (-) 0 [7..10]))))))
→ 1 - (2 - (3 - (4 - (5 - (6 - (7 - (foldr (-) 0 [8..10])))))))
→ 1 - (2 - (3 - (4 - (5 - (6 - (7 - (8 - (foldr (-) 0 [9..10]))))))))
→ 1 - (2 - (3 - (4 - (5 - (6 - (7 - (8 - (9 - (foldr (-) 0 [10])))))))))
→ 1 - (2 - (3 - (4 - (5 - (6 - (7 - (8 - (9 - (10 - (foldr (-) 0 [])))))))))))
→ 1 - (2 - (3 - (4 - (5 - (6 - (7 - (8 - (9 - (10)))))))))
→ 1 - (2 - (3 - (4 - (5 - (6 - (7 - (8 - (-1))))))))
→ 1 - (2 - (3 - (4 - (5 - (6 - (7 - (9)))))))
→ 1 - (2 - (3 - (4 - (5 - (6 - (-2))))))
→ 1 - (2 - (3 - (4 - (5 - (8)))))
→ 1 - (2 - (3 - (4 - (-3))))
→ 1 - (2 - (3 - (7)))
→ 1 - (2 - (-4))
→ 1 - (6)
→ -5
```

Functor:

- Functor in Haskell is a kind of functional representation of different types which can be mapped over. It is a high level concept of implementing polymorphism. Types such as List, Map, Tree, etc. are instances of the Haskell Functor.
- Functor is a function which takes a function and returns another function.
- The Functor typeclass is basically for things that can be mapped over.
- A Functor is an inbuilt class with a function definition like:
  class Functor f where
  fmap :: (a -> b) -> f a -> f b → fmap takes a function and a functor and applies the function on the functor.
- Recall the map function. **map f [x, y, z] = [f x, f y, f z]**
  The map function can be applied to nothing more than a list of values (where values are of any type) whereas the fmap function can be applied to many more data types, all of which belong to the functor class (e.g. maybe, tuples, lists, etc.). Since the "list of values" data type is also a functor, because it provides an implementation for it, then fmap can be applied to is as well producing the very same result as map. In fact, map is just a fmap that works only on lists. The difference between map and fmap lies in their usage. Functor enables us to implement some more functionalists in different data types, like "Just" and "Nothing".

```
Prelude> :type map
map :: (a -> b) -> [a] -> [b]
Prelude> :type fmap
fmap :: Functor f => (a -> b) -> f a -> f b
```

- E.g. Consider the below code.

```
Prelude> map (subtract 1) [1,2,3]
[0,1,2]
Prelude> fmap (subtract 1) [1,2,3]
[0,1,2]
Prelude> map  (+7)(Just 10)

<interactive>:15:11: error:
   • Couldn't match expected type ‘[b]’
                  with actual type ‘Maybe Integer’
   • In the second argument of ‘map’, namely ‘(Just 10)’
     In the expression: map (+ 7) (Just 10)
     In an equation for ‘it’: it = map (+ 7) (Just 10)
   • Relevant bindings include it :: [b] (bound at <interactive>:15:1)
Prelude> fmap  (+7)(Just 10)
Just 17
Prelude> map  (+7) Nothing

<interactive>:17:11: error:
   • Couldn't match expected type ‘[b]’ with actual type ‘Maybe a0’
   • In the second argument of ‘map’, namely ‘Nothing’
     In the expression: map (+ 7) Nothing
     In an equation for ‘it’: it = map (+ 7) Nothing
   • Relevant bindings include it :: [b] (bound at <interactive>:17:1)
Prelude> fmap  (+7) Nothing
Nothing
```

Notice that map and fmap produce the same results on a list, but map doesn't work for types such as “Just” or “Nothing”, while fmap does.

- This is the standard library map function (**map f [x, y, z] = [f x, f y, f z]**). Here's an fmap implementation:
  fmap_List :: (a -> b) -> [] a -> [] b
  -- "[] a" means "[a]" in types.
  fmap_List f [] = []
  fmap_List f (x:xs) = f x : fmap_List f xs
- This is the definition of the Maybe type from the standard library:
  data Maybe a = Nothing | Just a
  Note: There are two perspectives for the Maybe type:
  - It's like a list of length 0 or 1.
  - It models having two possibilities: "no answer" and "here's the answer".

  Here's an fmap implementation:
  fmap_Maybe :: (a -> b) -> Maybe a -> Maybe b
  fmap_Maybe f Nothing = Nothing
  fmap_Maybe f (Just a) = Just (f a)
- This is the definition of the Either type from the standard library:
  data Either e a = Left e | Right a
  It's like Maybe, but the "no answer" case carries extra data, perhaps some kind of reason for why "no answer".
  Here's an fmap implementation:
  fmap_Either :: (a -> b) -> (Either e) a -> (Either e) b
  fmap_Either f (Left e) = Left e
  fmap_Either f (Right a) = Right (f a)
- **Note:** fmap must satisfy some axioms/laws:
  1. **Identity axiom/Functor Identity:**
     - The first functor law states that if we map the id function over a functor, the functor that we get back should be the same as the original functor. If we write that a bit more formally, it means that **fmap id = id**. So essentially, this says that if we do fmap id over a functor, it should be the same as just calling id on the functor. Id is the identity function, which just returns its parameter unmodified. It can also be written as **\x -> x**.
     - E.g.

```
Prelude> fmap id (Just 3)
Just 3
Prelude> id (Just 3)
Just 3
Prelude> fmap id [1..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude> id [1..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude> fmap id []
[]
Prelude> id []
[]
Prelude> fmap id Nothing
Nothing
Prelude> id Nothing
Nothing
```

  2. **fmap fusion/fmap is a homomorphism:**
     - The second law says that composing two functions and then mapping the resulting function over a functor should be the same as first mapping one

function over the functor and then mapping the other one. Formally written, that means that **fmap (f . g)** = **fmap f . fmap g**. Or to write it in another way, for any functor F, the following should hold:
fmap (f . g) F = **fmap f (fmap g F)**.

- Doing **fmap g (fmap f xs)** should get the same result as doing
fmap (\x -> g (f x)) xs
I.e.
fmap g . fmap f = **fmap (g . f)**

- **Note:** We can do fmap Just [1]. This is because types are functions.
E.g.

```
*Main> fmap Just [1]
[Just 1]
```

```
*Main> fmap Just (Just Nothing)
Just (Just Nothing)
*Main> fmap Just (Just 2)
Just (Just 2)
```

- Functor on its own does not have much basic practical use, apart from providing a common name “fmap”, but it is much more useful when combined with the Applicative and Monad methods. It also has an advanced practical use. On the other hand, Functor is extremely important in category theory.

Applicative:

- An Applicative Functor is a normal Functor with some extra features provided by the Applicative Type Class. It is found in the Control.Applicative module and to use it, we need to do **import Control.Applicative**.
- The class is defined like such:

```
class (Functor f) => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  liftA2 :: (a -> b -> c) -> f a -> f b -> f c
  -- And default implementations because <*> and liftA2 are equivalent.
  liftA2 f as bs = fmap f as <*> bs
  fs <*> as = liftA2 (\f a -> f a) fs as
  -- And a couple of other methods with easy default implementations.
```

pure :: a -> f a → Pure takes a value and returns a functor of that value.
(<*>) :: f (a -> b) -> f a -> f b → <*> takes a functor with a function in it and another functor and applies the function to the second functor.
liftA2 :: (a -> b -> c) -> f a -> f b -> f c → liftA2 takes a function and 2 functors and applies the function on the 2 functors.

Looking at the first line, it states the definition of the Applicative class and it also introduces a class constraint. It says that if we want to make a type constructor part of the Applicative typeclass, it has to be in Functor first. That's why if we know that if a type constructor is part of the Applicative typeclass, it's also in Functor, so we can use fmap on it.

The first method it defines is called **pure**. Its type declaration is **pure :: a -> f a**. "f" plays the role of our applicative functor instance here. pure should take a value of any type and return an applicative functor with that value inside it. We take a value and we wrap it in an applicative functor that has that value as the result inside it.

pure for [], Maybe, and Either e work as follows:

```
-- [] version
pure a = [a]

-- Maybe version
pure a = Just a

-- Either e version
pure a = Right a
```

pure plays two roles:

1. The degenerate case when you have a 0-ary function and 0 lists, kind of.
   2-ary, **liftA2 :: (t1 -> t2 -> a) -> f t1 -> f t2 -> f a**
   1-ary, **fmap :: (t1 -> a) > f t1 -> f a**
   0-ary, **pure :: a -> f a**
2. fmap can be derived from pure and <*>.
   I.e. **fmap f xs = pure f <*> xs**

The second function it defines is **<*>**. It has a type declaration of **f (a -> b) -> f a -> f b**. <*> is sort of a beefed up fmap. Whereas fmap takes a function and a functor and applies the function inside the functor, <*> takes a functor that has a function in it and another functor and sort of extracts that function from the first functor and then maps it over the second one.
Note: We can use lambda functions with <*>.
E.g.

```
*Main Control.Applicative> Just (\x -> x**2) <*> Just (5)
Just 25.0
```

The third function it defines is **liftA2**. liftA2 just applies a function between two applicatives, hiding the applicative style that we've become familiar with. The reason we're looking at it is because it clearly showcases why applicative functors are more powerful than just ordinary functors. With ordinary functors, we can just map functions over one functor. But with applicative functors, we can apply a function between several functors. It's also interesting to look at this function's type as **(a -> b -> c) -> (f a -> f b -> f c)**. When we look at it like this, we can say that liftA2 takes a normal binary function and promotes it to a function that operates on two functors.

E.g.
liftA2 (+) [1,2,3] [4,5,6]
= [1+4, 1+5, 1+6, 2+4, 2+5, 2+6, 3+4, 3+5, 3+6]
= [5,6,7,6,7,8,7,8,9]

```
Prelude Control.Applicative> liftA2 (+) [1,2,3] [4,5,6]
[5,6,7,6,7,8,7,8,9]
```

E.g.
liftA2 (-) [10,20,30] [1,2,3]
= [10-1, 10-2, 10-3, 20-1, 20-2, 20-3, 30-1, 30-2, 30-3]
= [9,8,7,19,18,17,29,28,27]

```
Prelude Control.Applicative> liftA2 (-) [10,20,30] [1,2,3]
[9,8,7,19,18,17,29,28,27]
```

Note: We can use lambda functions with liftA2.
E.g.

```
*Main Control.Applicative> liftA2 (\x y -> x**y) (Just 2) (Just 3)
Just 8.0
```

- Here is the Applicative instance implementation for Maybe.

```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  (Just f) <*> something = fmap f something
```

First off, pure. We said earlier that it's supposed to take something and wrap it in an applicative functor. We wrote **pure = Just**, because value constructors like Just are normal functions. We could have also written **pure x = Just x**.

Next up, we have the definition for <*>. We can't extract a function out of a Nothing, because it has no function inside it. So we say that if we try to extract a function from a Nothing, the result is a Nothing.

If the first parameter is not a Nothing, but a Just with some function inside it, we say that we then want to map that function over the second parameter. This also takes care of the case where the second parameter is Nothing, because doing fmap with any function over a Nothing will return a Nothing.

So for Maybe, <*> extracts the function from the left value if it's a Just and maps it over the right value. If any of the parameters is Nothing, Nothing is the result.

E.g.
Notice that if there's Nothing on either side of the <*>, the result is nothing.

```
Prelude> Just (+3) <*> Nothing
Nothing
Prelude> Nothing <*> Just (+3)
Nothing
```

Notice that the 2 statements below give the same result.

```
Prelude> Just (+3) <*> Just 9
Just 12
Prelude> pure (+3) <*> Just 9
Just 12
```

- The Applicative methods should satisfy the following axioms:
  1. **Applicative subsumes Functor**
     fmap f xs = pure f <*> xs
  2. **Applicative left-identity**
     pure id <*> xs = xs
     -- Compare with fmap identity! fmap id xs = xs
  3. **Applicative associativity, composition**
     gs <*> (fs <*> xs) = ((pure (.) <*> gs) <*> fs) <*> xs
     = (liftA2 (.) gs fs) <*> xs
     -- Analogy: g (f x) = (g . f) x
     -- It may help to elaborate the types. Assume:
     -- xs :: f a
     -- fs :: f (a -> b)
     -- gs :: f (b -> c)
     -- (.) :: (b -> c) -> (a -> b) -> (a -> c)
     -- Try to determine the types of the subexpressions
  4. **pure fusion, pure is a homomorphism**
     pure f <*> pure x = pure (f x)
     fmap f (pure x) = pure (f x)
  5. **pure interchange, almost right-identity**
     fs <*> pure x = pure (\f -> f x) <*> fs
     = fmap (\f -> f x) fs
- The first corollary is that the Applicative axioms imply the Functor axioms given fmap f xs = pure f <*> xs.
  Functor identity is immediate from Applicative left-identity. To deduce fmap fusion:

```
fmap g (fmap f xs)                    in Applicative terms
= pure g <*> (pure f <*> xs)          associativity
= ((pure (.) <*> pure g) <*> pure f) <*> xs   pure fusion
= (pure ((.) g) <*> pure f) <*> xs    pure fusion
= pure ((.) g f) <*> xs               infix notation
= pure (g . f) <*> xs                 in Functor terms
= fmap (g . f) xs
```

- The second corollary is that
  liftA3 (\x y z -> g x (f y z)) xs ys zs
  can be done by the following two equivalent ways:
  1. **(fmap (\x y z -> g x (f y z)) xs <*> ys) <*> zs**
  2. **fmap g xs <*> (fmap f ys <*> zs)**

Fmap:

- **fmap :: Functor f => (a -> b) -> f a -> f b**
- This means that fmap takes a function and a functor and applies the function over the functor.
- E.g.

```
Prelude Control.Applicative> fmap (+1) (Just 10)
Just 11
Prelude Control.Applicative> fmap (*2) (Just 10)
Just 20
Prelude Control.Applicative> fmap (\x -> (x+x)**2) (Just 10)
Just 400.0
```

- Can be thought of as liftA1. This will be explained below.

<*>:

- Also called **ap**.
- **(<*>) :: f (a -> b) -> f a -> f b**
- <*> takes a functor with a function in it and another functor and applies the function to the second functor.
- E.g.

```
Prelude Control.Applicative> (Just (+1)) <*> (Just 5)
Just 6
Prelude Control.Applicative> (Just (*2)) <*> (Just 5)
Just 10
Prelude Control.Applicative> (Just (\x -> (x+x)**2)) <*> (Just 5)
Just 100.0
```

- **Note:** You need to do **import Control.Applicative** to use ap.

LiftA2:

- **liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c**
- If you compare the above line with fmap, you'll see that they're very similar, but fmap takes 1 functor while applicative takes 2. Furthermore, the function fmap uses only takes in 1 argument, while the function liftA2 uses takes 2 arguments. This is why we can think of fmap as liftA1.
- E.g.

```
Prelude Control.Applicative> liftA2 (+) (Just 1) (Just 2)
Just 3
Prelude Control.Applicative> liftA2 (*) (Just 1) (Just 2)
Just 2
Prelude Control.Applicative> liftA2 (\x y -> (x+y)**2) (Just 1) (Just 2)
Just 9.0
```

- We can use fmap and <*> to implement liftA2.
  Here's an implementation of fmap, pure, <*> and liftA2 for the functor Maybe.

```
fmap_Maybe :: (a -> b) -> Maybe a -> Maybe b
fmap_Maybe _ Nothing = Nothing
fmap_Maybe f (Just x) = Just (f x)

pure_Maybe :: a -> Maybe a
pure_Maybe = Just

ap_Maybe :: Maybe (a -> b) -> Maybe a -> Maybe b
ap_Maybe _ Nothing = Nothing
ap_Maybe Nothing _ = Nothing
ap_Maybe (Just f) (Just x) = fmap_Maybe f (Just x)

liftA2_Maybe :: (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
liftA2_Maybe _ _ Nothing = Nothing
liftA2_Maybe _ Nothing _ = Nothing
liftA2_Maybe f (Just a) (Just b) = Just (f a b)
```

  Note that for liftA2, we're not using fmap and <*> to implement it.
  Here's how we can use fmap and <*> to implement liftA2.
  liftA2 f xs ys = (fmap f xs) <*> ys
  E.g.

```
*Main Control.Applicative> liftA2_Maybe (+) (Just 1) (Just 2) == ((fmap (+) (Just 1)) <*> (Just 2))
True
*Main Control.Applicative> liftA2_Maybe (*) (Just 1) (Just 2) == ((fmap (*) (Just 1)) <*> (Just 2))
True
*Main Control.Applicative> liftA2_Maybe (\x y -> x**y) (Just 1) (Just 2) == ((fmap (\x y -> x**y) (Just 1)) <*> (Just 2))
True
```

  The reason why **liftA2 f xs ys = (fmap f xs) <*> ys** is because fmap applies the function, f, on xs, so xs is a functor with a function in it, and then <*> applies that function onto ys.

LiftA3:

- **liftA3 :: Applicative f => (a -> b -> c -> d) -> f a -> f b -> f c -> f d**
- This is similar to liftA2, but it takes 3 arguments instead of 2.
- We can implement liftA3 using fmap and <*>.
  liftA3 f xs ys zs = (fmap f xs) <*> ys <*> zs
  E.g.

```
*Main Control.Applicative> liftA3 (\x y z -> x + y + z) (Just 1) (Just 2) (Just 3) == ((fmap (\x y z -> x + y + z) (Just 1)) <*> (Just 2) <*> (Just 3))
True
*Main Control.Applicative> liftA3 (\x y z -> x + y * z) (Just 1) (Just 2) (Just 3) == ((fmap (\x y z -> x + y * z) (Just 1)) <*> (Just 2) <*> (Just 3))
True
*Main Control.Applicative> liftA3 (\x y z -> x ** y * z) (Just 1) (Just 2) (Just 3) == ((fmap (\x y z -> x ** y * z) (Just 1)) <*> (Just 2) <*> (Just 3))
True
```

- We can implement liftA3 using liftA2 and <*>.
  liftA3 f xs ys zs = (liftA2 f xs ys) <*> zs

E.g.

```
*Main Control.Applicative> liftA3 (\x y z -> x + y + z) (Just 1) (Just 2) (Just 3) == ((liftA2 (\x y z -> x + y + z) (Just 1) (Just 2)) <*> (Just 3))
True
*Main Control.Applicative> liftA3 (\x y z -> x + y * z) (Just 1) (Just 2) (Just 3) == ((liftA2 (\x y z -> x + y * z) (Just 1) (Just 2)) <*> (Just 3))
True
*Main Control.Applicative> liftA3 (\x y z -> x ** y * z) (Just 1) (Just 2) (Just 3) == ((liftA2 (\x y z -> x ** y * z) (Just 1) (Just 2)) <*> (Just 3))
True
```

This is because **(fmap f xs) <*> ys** is equivalent to **liftA2 f xs ys**.

In general:

- If you have liftAn, where n > 1, you can implement in the following way:
    1. liftAn f a b c … z = (fmap f a) <*> b <*> c … <*> z
    2. liftAn f a b c … y z = (liftA(n-1) f a b c … y) <*> z

Parameterized "container" types as effect types:

- Think of [], Maybe, Either e as types of effects or effectful programs, not always as types of data structures.
- "Effect" is very broad and usually not given a precise definition, but if you know the phrase "function with a side-effect", that's the idea. In the Haskell culture, it refers to things a mathematical function cannot do, e.g. no answer, multiple answers, accessing state variables, performing I/O—the latter two lead to getting different answers at different times.
- **f :: Int -> String** in Haskell means that **f 4 :: String** is the same string every time, without effect. If some specific kind of effect is desired, we make a parameterized type E to Maybe or [] to represent it and change types to **f :: Int -> E String**, so **f 4 :: E String** does not have to give the same string every time, instead just the same "effectful action" every time.
- It suffices to focus on "E String" (so without "Int ->") and think of it as the type of effectful programs that give string answers. E.g., if m1, m2 :: Maybe String, think of them as two programs that give string answers (the effect is that it could fail).
- In this frame of mind, the methods of Functor, Applicative, and later Monad are connectives—combining basic effectful programs into complex ones. E.g. **liftA2 (++) m1 m2** combines m1 and m2.
- Consider "Maybe":
    - **foo :: Maybe Int** now means a program that may succeed and return an Int (conveyed by Just), or may fail (conveyed by Nothing).
    - Suppose **f :: Int -> String**.
    - **fmap f foo :: Maybe String** now means a program that runs foo, but converts the answer, if any, using f.
    - **pure 4 :: Maybe Int** now means a program that succeeds and returns 4. Note that it avoids using Maybe's effect of failure.
    - Suppose **bar :: Maybe Int**.
    - **liftA2 (+) foo bar :: Maybe Int** now means a composite program that runs foo to try to obtain a number. If running foo is successful, it runs bar. If that is successful too, the overall answer is the sum.
    - E.g. I have a recipMay function for reciprocals, but to better handle division-by-zero, I use Maybe to convey successes and failures. I then use it to write an addRecip function to add two reciprocals, again involving Maybe in anticipation of failure. This is my first version:

```
recipMay :: Double -> Maybe Double
recipMay a | a == 0 = Nothing
           | otherwise = Just (1 / a)    -- or: pure (1 / a)
```

```
addRecipV1 x y =
  case recipMay x of
    Nothing -> Nothing
    Just x_recip -> case recipMay y of
      Nothing -> Nothing
      Just y_recip -> Just (x_recip + y_recip)
```

This is my second version. It uses the new way of thinking:

```
recipMay :: Double -> Maybe Double
recipMay a | a == 0 = Nothing
           | otherwise = Just (1 / a)    -- or: pure (1 / a)

addRecipV2 :: Double -> Double -> Maybe Double
addRecipV2 x y = liftA2 (+) (recipMay x) (recipMay y)
```

Monads:

- Monads are just beefed up applicative functors.
- A monad is a way to structure computations in terms of values and sequences of computations using those values. Monads allow the programmer to build up computations using sequential building blocks, which can themselves be sequences of computations. The monad determines how combined computations form a new computation and frees the programmer from having to code the combination manually each time it is required.
- It is useful to think of a monad as a strategy for combining computations into more complex computations.
- Monads chain operations in some specific, useful way.
- Monads apply a regular function to a wrapped value and return a wrapped value. This is similar to what a functor does. The difference between a monad and a functor is that with monads, the functions aren't expecting a wrapped value.
- E.g. Consider the code and output below:

```
half x = if even x
    then Just (x `div` 2)
    else Nothing
```

```
*Main> half 4
Just 2
*Main> half 5
Nothing
*Main> half 10
Just 5
```

What if we wanted to pass in (Just 10) to half?

```
*Main> half (Just 10)

<interactive>:91:1: error:
  • Non type-variable argument in the constraint: Integral (Maybe a)
    (Use FlexibleContexts to permit this)
  • When checking the inferred type
      it :: forall a. (Integral (Maybe a), Num a) => Maybe (Maybe a)
```

It gives an error.

However, if we do (Just 10) >>= half, we get back Just 5, as shown below.

```
*Main> (Just 10) >>= half
Just 5
```

In this example, the function, half, isn't expecting a wrapped value of (Just 10). We did half x = …, not half (Just x) = …. However, by using >>=, we were able to apply (Just 10) to the half function. Here's what >>= is doing behind the scenes:

- When we do "Just 10 >>= …", the >>= takes the 10, only.
- Hence, when we did (Just 10) >>= half, it was like doing half 10.

- We can also chain monads.
  E.g.

```
*Main> (Just 8) >>= half >>= half
Just 2
*Main> (Just 4) >>= half >>= half
Just 1
*Main> (Just 4) >>= half >>= half >>= half
Nothing
```

- The monad class is defined like this:

```
class Applicative m => Monad m where
  return :: a -> m a

  (>>=) :: m a -> (a -> m b) -> m b
  x >> y = x >>= \_ -> y

  fail :: String -> m a
  fail msg = error msg
```

(>>=) :: m a -> (a -> m b) -> m b → >>= takes in a monadic value and a function that takes in a regular value but outputs a monadic value and applies the function on the monadic value.
E.g. Suppose you have something like **Just x >>= \b -> …**, here "b" is not wrapped in "Just". This is because the lambda function takes a regular value.

The first function that the Monad type class defines is **return**. It's the same as pure, only with a different name. Its type is **(Monad m) => a → m a**. It takes a value and puts it in a minimal default context that still holds that value. In other words, it takes something and wraps it in a monad. It always does the same thing as the pure function from the Applicative type class.
Note: return is nothing like the return that's in most other languages. It doesn't end function execution or anything, it just takes a normal value and puts it in a context.
It is here for historical reasons (because Applicative is more recent). Do not think of return in terms of control flow. It does not exit anything.

The next function is **>>=**, or **bind**. It's like function application, only instead of taking a normal value and feeding it to a normal function, it takes a monadic value (a value with a context) and feeds it to a function that takes a normal value but returns a monadic value.

Next up, we have **>>**. It's default implementation is **foo >> bar = foo >>= _ -> bar**. It comes in handy when you don't need foo's answer, only its effect. We call this operator "then". We use this function when we want to perform actions in a certain order, but don't care what the result of one is. Consider the example below:

```
Prelude> print "x" >>= \_ -> print "y"
"x"
"y"
Prelude> print "x" >> print "y"
"x"
"y"
```

The final function of the Monad type class is **fail**. We never use it explicitly in our code. Instead, it's used by Haskell to enable failure in a special syntactic construct for monads.

- The Monad methods satisfies these axioms:
    1. **Left Identity Law:** This law states that if we take a value, put it in a default context with return and then feed it to a function by using >>=, it's the same as just taking the value and applying the function to it. To put it formally:
       return x >>= k = k x
       E.g.

       ```
       Prelude> (\x -> Just(x+1)) 3
       Just 4
       Prelude> return 3 >>= (\x -> Just(x+1))
       Just 4
       ```

       Notice that both statements get the same result.
    2. **Right Identity Law:** This law states that if we have a monadic value and we use >>= to feed it to return, the result is our original monadic value. Formally:
       m >>= return = m
       E.g.

       ```
       Prelude> [1,2,3,4] >>= (\x -> return x)
       [1,2,3,4]
       Prelude> Just 3 >>= (\x -> return x)
       Just 3
       ```

    3. **Associativity:** This law says that when we have a chain of monadic function applications with >>=, it shouldn't matter how they're nested. Formally written: Doing **(m >>= f) >>= g** is just like doing **m >>= (\x -> f x >>= g)**

       Associative laws justify re-grouping. This is important for many refactoring and implementing long chains by recursion.
- Monad subsumes Applicative and Functor. From return and >>= we can get the methods of Applicative and Functor:
  fmap f xs = xs >>= (\x -> return (f x))
  liftA2 op xs ys = xs >>= (\x -> ys >>= (\y -> return (op x y)))

- Monads need to use both applicative and functor.
  E.g. Consider the code and output below.

```
data Maybe2 a = Nothing2 | Just2 a deriving Show

instance Monad Maybe2 where
    -- >>= :: m a -> (a -> m b) -> m b
    Nothing2 >>= f = Nothing2
    Just2 a >>= f = f a
```

```
*Main> :reload

monad_maybe.hs:3:10: error:
    • No instance for (Applicative Maybe2)
        arising from the superclasses of an instance declaration
    • In the instance declaration for 'Monad Maybe2'
  |
3 | instance Monad Maybe2 where   |          ^^^^^^^^^^
[1 of 1] Compiling Main             ( monad_maybe.hs, interpreted )
Failed, no modules loaded.
```

Now, if I add the applicative and functor instances, then I don't get that compilation error.

```
data Maybe2 a = Nothing2 | Just2 a deriving Show

instance Functor Maybe2 where
    -- fmap :: (a -> b) -> fa -> fb
    fmap f Nothing2 = Nothing2
    fmap f (Just2 a) = Just2 (f a)

instance Applicative Maybe2 where
    -- pure :: a -> f a
    -- <*> :: f(a -> b) -> f a -> f b
    pure = Just2
    Nothing2 <*> _ = Nothing2
    _ <*> Nothing2 = Nothing2
    Just2 f <*> Just2 a = Just2 (f a)

instance Monad Maybe2 where
    -- >>= :: m a -> (a -> m b) -> m b
    Nothing2 >>= f = Nothing2
    Just2 a >>= f = f a
```

```
Prelude> :reload
[1 of 1] Compiling Main             ( monad_maybe.hs, interpreted )
Ok, one module loaded.
```

- Examples of Maybe2:

```haskell
data Maybe2 a = Nothing2 | Just2 a deriving Show

instance Functor Maybe2 where
    -- fmap :: (a -> b) -> fa -> fb
    fmap f Nothing2 = Nothing2
    fmap f (Just2 a) = Just2 (f a)

instance Applicative Maybe2 where
    -- pure :: a -> f a
    -- <*> :: f(a -> b) -> f a -> f b
    pure = Just2
    Nothing2 <*> _ = Nothing2
    _ <*> Nothing2 = Nothing2
    Just2 f <*> Just2 a = Just2 (f a)

instance Monad Maybe2 where
    -- >>= :: m a -> (a -> m b) -> m b
    Nothing2 >>= f = Nothing2
    Just2 a >>= f = f a


half x = if even x
    then Just2 (x `div` 2)
    else Nothing2
```

```
*Main> (Just2 10) >>= half
Just2 5
*Main> Nothing2 >>= half
Nothing2
*Main> (Just2 100) >>= half >>= half >>= half
Nothing2
*Main> (Just2 100) >>= half >>= half
Just2 25
```

State Monad:

- The **state monad** is a built-in monad in Haskell that allows for chaining of a state variable through a series of function calls, to simulate stateful code. It is defined as:
  newtype State s a = State { runState :: (s -> (a,s)) }
- The Haskell type State describes functions that consume a state and produce both a result and an updated state, which are given back in a tuple.
- A few basic operations are provided below. Furthemore, "State s" is an instance of Functor, Applicative, and Monad, so you have connectives to chain up basic operations too.

1. **-- "get" reads and returns the current value of the state variable.**
   get :: State s s
   get = State (\s -> (s,s))

2. **-- "put s1" sets the state variable to s1. It returns the 0-tuple because there**
   -- is no information to return.
   put :: s -> State s ()
   put newState = State (\s -> (newState, ()))

3. **-- functionize prog s0 runs prog starting with initial state value s0 and gives**
   -- you the final answer.
   functionize :: State s a -> s -> a

- E.g. We want to build a binary tree out of given elements, inorder, balanced.
  I.e. buildTree [a, b, c, d, e, f, g] means at d, which is the root, recursively create a,b,c in left subtree, and e,f,g in right subtree. We need to count length for halving. A non-obvious but linear-time strategy is:
  - Count length just once.
  - State variable holds unused elements (initially all).
  - Recursive helper takes parameter n, uses the first n elements from state var to build tree. Algorithm:
    - Split n into n = m1 + 1 + m2, m1 is left subtree size, m2 is right subtree size, and 1 element for the middle node.
    - Recursive call: build tree of m1 elements, this will be the left subtree.
    - Take out one element from state var, this will be for the middle node.
    - Recursive call: build tree of m2 elements, this will be the right subtree.
    - Compose the middle node, it's my answer

  Here's the code:

```
-- Recall: data BinTree a = BTNil | BTNode a (BinTree a) (BinTree a)

-- buildTreeHelper n: Use n elements from [a] state var to build tree.
-- Precondition: n <= length of state var.
buildTreeHelper :: Int -> State [a] (BinTree a)
buildTreeHelper 0 = pure BTNil
buildTreeHelper n =
  buildTreeHelper m1        -- Make left subtree, m1 elements, call it lt.
  >>= \lt -> get
  >>= \(x:xt) -> put xt     -- Which elements remaining? Take one for myself.
  >> buildTreeHelper m2     -- Make right subtree, m2 elements, call it rt.
  >>= \rt -> pure (BTNode x lt rt) -- Put it together, this is my answer.
```

```
  where
    n' = n - 1
    m1 = div n' 2
    m2 = n' - m1

buildTree :: [a] -> BinTree a
buildTree xs = functionize (buildTreeHelper (length xs)) xs
-- Whole list for initial state. Use all to build tree.
```

In the example above,
```
>>= \lt -> get
>>= \(x:xt) -> put xt
```
you need to use bind, >>=, to get the return value of get. In this case, (x:xt) is the return value of get.

- The basic idea of state monad is to have a state transition function instead, like s→s, and have some starter function, functionize, that feeds it the initial value. However, we also want it to give an answer and not a state. So s→(s,a) is a function from the old-state to a pair of the new-state and answer.
- E.g.

```
data State s a = MkState (s -> (s, a))

-- Unwrap MkState.
deState :: State s a -> s -> (s, a)
deState (MkState stf) = stf

functionize :: State s a -> s -> a
functionize prog s0 = snd (deState prog s0)

get :: State s s
get = MkState (\s0 -> (s0, s0))
-- old state = s0, new state = old state = s0, answer s0 too.

put :: s -> State s ()
put s = MkState (\s0 -> (s , ()))
-- ignore old state, new state = s, answer the 0-tuple ().

instance Functor (State s) where
   -- fmap :: (a -> b) -> State s a -> State s b
   fmap f (MkState stf) = MkState
      (\s0 ->
        -- Goal: Like stf but use f to convert a to b
        -- old state = s0, give to stf for new state s1 and answer a
        case stf s0 of (s1, a) ->
        -- overall new state is also s1, but change answer to f a
                    (s1, f a))
```

```
testStateFunctor = deState (fmap length program) 10
  where
    program :: State Integer String
    program = MkState (\s0 -> (s0+2, "hello"))
-- should give (12, 5)

instance Applicative (State s) where
    -- pure :: a -> State s a
    -- Goal: Give the answer a and try not to have an effect.
    -- "effect" for State means state change.
    pure a = MkState (\s0 -> (s0, a))
    -- so new state = old state

    -- liftA2 :: (a -> b -> c) -> State s a -> State s b -> State s c
    --
    -- State transition goal:
    --                overall old state
    -- --1st-program--> intermediate state
    -- --2nd-program--> overall new state
    --
    -- (Why not the other order? Actually would be legitimate, but we usually
    -- desire liftA2's order to be consistent with >>='s order.)
    liftA2 op (MkState stf1) (MkState stf2) = MkState
        (\s0 ->
          -- overall old state = s0, give to stf1
          case stf1 s0 of { (s1, a) ->
          -- intermediate state = s1, give to stf2
          case stf2 s1 of { (s2, b) ->
          -- overall new state = s2
          -- overall answer = op a b
          (s2, op a b) }} )

testStateApplicative = deState (liftA2 (:) prog1 prog2) 10
  where
    prog1 :: State Integer Char
    prog1 = MkState (\s0 -> (s0+2, 'h'))
    prog2 :: State Integer String
    prog2 = MkState (\s0 -> (s0*2, "ello"))
-- should give (24, "hello").  24 = (10+2)*2.
```

```
instance Monad (State s) where
  return = pure

  -- (>>=) :: State s a -> (a -> State s b) -> State s b
  -- Goal:
  -- 1. overall old state --1st-program--> (intermediate state, a)
  -- 2. give a and intermediate state to the 2nd program.
  MkState stf1 >>= k = MkState
    (\s0 ->
      -- overall old state = s0, give to stf1
      case stf1 s0 of { (s1, a) ->
      -- k is waiting for the answer a
      --     and also the intermediate state s1
      -- technicality: "(k a) s1" is conceptually right but nominally a
      -- type error because (k a) :: State s b, not s -> (s, b)
      -- Ah but deState can unwrap! (Or use pattern matching.)
      deState (k a) s1 } )
```

Dependency injection, Template method, Mock testing:

- Here is the first version of my file format checker for my toy file format. The first three characters should be A, L, and newline.

```
toyCheckV1 :: IO Bool
toyCheckV1 =
  getChar
  >>= \c1 -> getChar
  >>= \c2 -> getChar
  >>= \c3 -> return ([c1, c2, c3] == "AL\n")
```

- We can also use dependency injection to test it. One way of doing this is to define our own type class for the relevant, permitted operations.

```
class Monad f => MonadToyCheck f where
  toyGetChar :: f Char
-- Simplifying assumptions: Enough characters, no failure.  A practical version
-- should add methods for raising and catching EOF exceptions.
```

  The checker logic should be polymorphic in that type class.

```
toyCheckV2 :: MonadToyCheck f => f Bool
toyCheckV2 =
  toyGetChar
  >>= \c1 -> toyGetChar
  >>= \c2 -> toyGetChar
  >>= \c3 -> return ([c1, c2, c3] == "AL\n")
```

  Only things toyCheckV2 can do: toyGetChar, monad methods, purely functional programming. Because the user chooses f. And toyCheckV2 doesn't even know what it is. All it knows is it can call toyGetChar.

- Now we can instantiate in two different ways, one way for production code, another way for mock testing.

- For production code:

```
instance MonadToyCheck IO where
   toyGetChar = getChar

realProgram :: IO Bool
realProgram = toyCheckV2
```

- For purely functional mock testing:

```
data Feeder a = MkFeeder (String -> (String, a))
-- Again, simplifying assumptions etc.  But basically like the state monad, with
-- the state being what's not yet consumed in the string.

-- Unwrap MkFeeder.
unFeeder :: Feeder a -> String -> (String, a)
unFeeder (MkFeeder sf) = sf

instance Monad Feeder where
   return a = MkFeeder (\s -> (s, a))
   prog1 >>= k = MkFeeder (\s0 -> case unFeeder prog1 s0 of
                              (s1, a) -> unFeeder (k a) s1)

instance MonadToyCheck Feeder where
   -- toyGetChar :: Feeder Char
   toyGetChar = MkFeeder (\(c:cs) -> (cs, c))

instance Functor Feeder where
   fmap f p = p >>= \a -> return (f a)

instance Applicative Feeder where
   pure a = MkFeeder (\s -> (s, a))
   pf <*> pa = pf >>= \f -> pa >>= \a -> return (f a)

testToyChecker2 :: String -> Bool
testToyChecker2 str = snd (unFeeder toyCheckV2 str)

toyTest1 = testToyChecker2 "ALhello"   -- should be False
toyTest2 = testToyChecker2 "AL\nhello" -- should be True
```

- Here's the code in its entirety:

```
class Monad f => MonadToyCheck f where
   toyGetChar :: f Char
-- Simplifying assumptions: Enough characters, no failure.  A practical version
-- should add methods for raising and catching EOF exceptions.

toyCheckV2 :: MonadToyCheck f => f Bool
toyCheckV2 =
   toyGetChar
   >>= \c1 -> toyGetChar
   >>= \c2 -> toyGetChar
   >>= \c3 -> return ([c1, c2, c3] == "AL\n")
```

```
data Feeder a = MkFeeder (String -> (String, a))
-- Again, simplifying assumptions etc.  But basically like the state monad, with
-- the state being what's not yet consumed in the string.

-- Unwrap MkFeeder.
unFeeder :: Feeder a -> String -> (String, a)
unFeeder (MkFeeder sf) = sf

instance Monad Feeder where
   return a = MkFeeder (\s -> (s, a))
   prog1 >>= k = MkFeeder (\s0 -> case unFeeder prog1 s0 of
                           (s1, a) -> unFeeder (k a) s1)

instance MonadToyCheck Feeder where
   -- toyGetChar :: Feeder Char
   toyGetChar = MkFeeder (\(c:cs) -> (cs, c))

instance Functor Feeder where
   fmap f p = p >>= \a -> return (f a)

instance Applicative Feeder where
   pure a = MkFeeder (\s -> (s, a))
   pf <*> pa = pf >>= \f -> pa >>= \a -> return (f a)

testToyChecker2 :: String -> Bool
testToyChecker2 str = snd (unFeeder toyCheckV2 str)

toyTest1 = testToyChecker2 "ALhello"   -- should be False
toyTest2 = testToyChecker2 "AL\nhello" -- should be True
```

- E.g.

```
*Main> unFeeder toyCheckV2 "ALBERT"
("ERT",False)
*Main> unFeeder toyCheckV2 "AL\nBERT"
("BERT",True)
```

Context Free Grammar:

- A context-free grammar is a set of recursive rules used to generate patterns of strings.
- Parser programs in compilers can be generated automatically from context-free grammars.
- Context-free grammars have the following components:
  - A **set of terminal symbols** which are the characters that appear in the language/strings generated by the grammar. Terminal symbols never appear on the left-hand side of the production rule and are always on the right-hand side.
  - A **set of nonterminal symbols (or variables)** which are placeholders for patterns of terminal symbols that can be generated by the nonterminal symbols. These are the symbols that will always appear on the left-hand side of the

production rules, though they can be included on the right-hand side. The strings that a CFG produces will contain only symbols from the set of nonterminal symbols.
    - A **set of production rules** which are the rules for replacing nonterminal symbols. Production rules have the following form: **variable → string of variables and terminals**.
    - A **start symbol** which is a special nonterminal symbol that appears in the initial string generated by the grammar.
- To create a string from a context-free grammar, follow these steps:
    - Begin the string with a start symbol.
    - Apply one of the production rules to the start symbol on the left-hand side by replacing the start symbol with the right-hand side of the production.
    - Repeat the process of selecting nonterminal symbols in the string, and replacing them with the right-hand side of some corresponding production, until all nonterminals have been replaced by terminal symbols. Note, it could be that not all production rules are used.
- E.g. A context-free grammar looks like this bunch of rules:
  Rule 1. E → E + E
  Rule 2. E → M
  Rule 3. M → M × M
  Rule 4. M → A
  Rule 5. A → 0
  Rule 6. A → 1
  Rule 7. A → (E)
  E, M, A are **non-terminal symbols** or **variables**. When you see them, you apply rules to expand. One of them is designated as the **start symbol**. You always start from it. Here, E is the start symbol.
  +, ×, 0, 1, (,) are **terminal symbols**. They are the characters you want in your language.
- **Derivation/generation** is a finite sequence of applying the rules until all non-terminal symbols are gone. We often aim for a specific final string.
- E.g.

E → M	(By Rule 2)
→ M × M	(By Rule 3)
→ A × M	(By Rule 4)
→ 1 × M	(By Rule 6)
→ 1 × A	(By Rule 4)
→ 1 × (E)	(By Rule 7)
→ 1 × (E + E)	(By Rule 1)
→ 1 × (M + E)	(By Rule 2)
→ 1 × (A + E)	(By Rule 4)
→ 1 × (0 + E)	(By Rule 5)
→ 1 × (0 + M)	(By Rule 2)
→ 1 × (0 + M × M)	(By Rule 3)
→ 1 × (0 + A × M)	(By Rule 4)
→ 1 × (0 + 1 × A)	(By Rule 6)
→ 1 × (0 + 1 × 1)	(By Rule 6)

- Context-free grammars can support matching parentheses and unlimited nesting.

Backus-Naur Form (BNF):

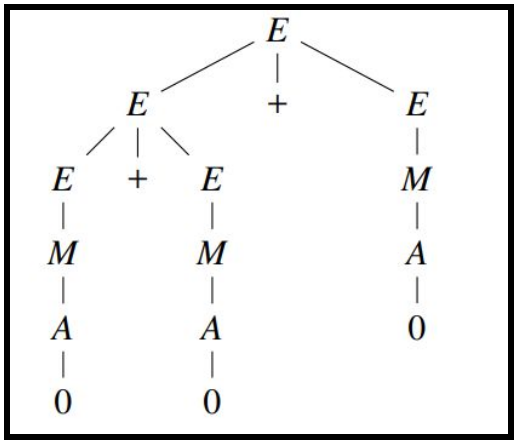
- **Backus-Naur Form** is a computerized, practical notation for CFGs.
- Surround non-terminal symbols by **<>**.
- Allow multi-letter names.
  Note: In some versions, we don't need <> around non-terminal symbols.
- Merge rules with the same LHS.
- In some versions, we surround terminal strings by single or double quotes.
- Use **::=** for →.
- Our example grammar in BNF:
  <expr> ::= <expr> "+" <expr> | <mul>
  <mul> ::= <mul> "*" <mul> | <atom>
  <atom> ::= "0" | "1" | "(" <expr> ")"

Extended Backus-Naur Form (EBNF):

- Use **{...}** for 0 or more occurrences.
- Use **[...]** for 0 or 1 occurrences.
- In some versions, no <> is needed around non-terminal symbols.

Parse Tree/Derivation Tree:

- A **parse tree/derivation tree** presents a derivation with more structure (tree), less repetition.
- E.g. This example generates 0 + 0 + 0.

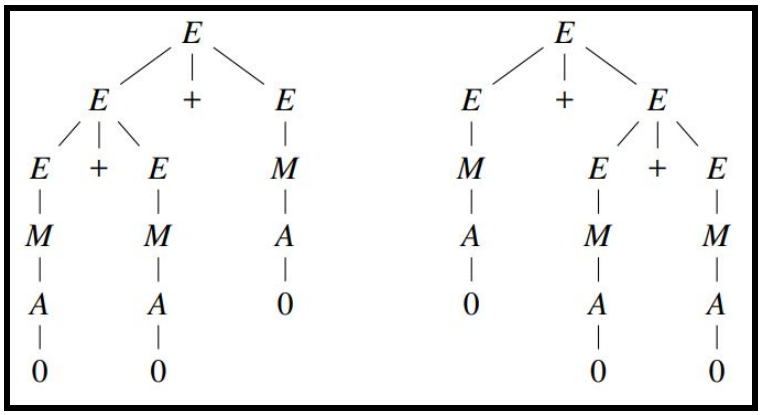


This is how we would write the example using derivation:

E → E + E (By Rule 1)
→ E + E + E (By Rule 1)
→ M + E + E (By Rule 2)
→ M + M + E (By Rule 2)
→ M + M + M (By Rule 2)
→ A + M + M (By Rule 4)
→ A + A + M (By Rule 4)
→ A + A + A (By Rule 4)
→ 0 + A + A (By Rule 5)
→ 0 + 0 + A (By Rule 5)
→ 0 + 0 + 0 (By Rule 5)

- In parse trees:
  - Internal nodes are non-terminal symbols.
  - Both operators and operands are terminal symbols at leaves.
  - The whole string is recorded, just scattered.

  - The purpose is to help visualize derivation and grammar as well as making writing the derivations easy and simple.
- When 2 or more different trees generate the same output, we say that the grammar is **ambiguous**.
  E.g. Two different trees generate the same 0 + 0 + 0:



  We try to design unambiguous grammars.
  CFG ambiguity is undecidable.
  Equivalence of two CFGs is also undecidable.
- **Note:** Generally, the reason we have ambiguity in languages is because there are more than 1 calls to the same item in 1 line.
  E.g. In the above language, we have
  <expr> ::= <expr> + <expr>
  <mul> ::= <mul> * <mul>
  Hence, if we have 0+0+0 or 0*0*0, we can use either <expr> or <mul> for expansion.
- Here is an unambiguous grammar that generates the same language as our ambiguous grammar example from above.
  <expr> ::= <expr> “+” <mul> | <mul>
  <mul> ::= <mul> “*” <atom> | <atom>
  <atom> ::= “0” | “1” | “(“ <expr> “)”

Left Recursive vs Right Recursive:

- **<expr> ::= <expr> “+” <mul>**
  That is a left recursive rule. The recursion is at the beginning (left).
- **<expr> ::= <mul> “+” <expr>**
  That is a right recursive rule. The recursion is at the end (right).
- Sometimes they convey intentions of left association or right association, but not always.
- They affect some parsing algorithms.
- Recursive descent parsing is a simple strategy for writing a parser.
- For each non-terminal symbol, we create a procedure based on RHS:
  - Non-terminal Symbol: Procedure call, possibly mutual recursion. (Thus “recursive descent”, also “top-down”.)
  - Left recursion needs special treatment to avoid infinite loops.
  - Terminal Symbol: Consume input and check.
  - Alternatives: Look ahead to choose, or try and backtrack.
- Some options for handling left recursion:
  - Redesign grammar to not have left recursion.

  - Many left recursive rules just express left-associating operators. Can be done without left recursive code.

- E.g.

<sub> ::= <atom> "-" <sub> | <atom>
<atom> ::= "0" | "1" | "(" <sub> ")"

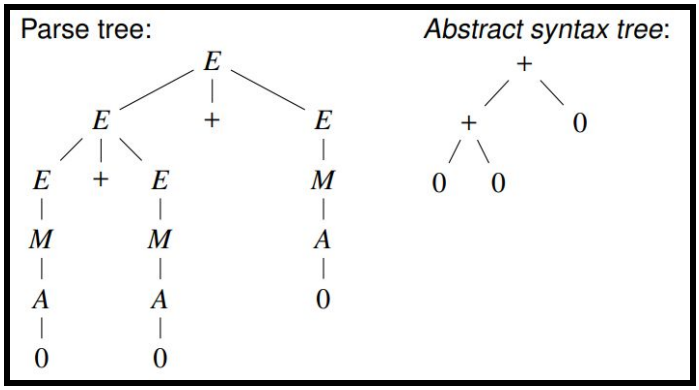
Starting with <sub>, we look at its RHS, which is <atom> "-" <sub> | <atom>.
We see <atom>, which is a non-terminal symbol. Hence, we make a procedure call to <atom>.
Next, we see a terminal symbol. We check if that terminal symbol is “-”. If it is, we continue to <sub>. Otherwise, we go to <atom>.
Since the terminal symbol is “-”, we see <sub>, so we make a procedure call to <sub>.
Pseudo-code of recursive descent parser:

```
sub:
        try (atom;
            read; if not "-" then fail;
            sub ;)
        if that failed: atom;

atom:
        read;
        if "0" or "1": success;
        if "(": sub;
                read; if not ")" then fail;
        else: fail;
```

Abstract Syntax Tree (AST) (vs Parse Tree):

- Abstract Syntax Tree General Points:
  - Internal nodes are operators/constructs.
    An example of a construct is if-then-else.
  - Non-terminal symbols are gone or replaced by constructs.
  - Many terminal symbols are gone too if they play no role other than nice syntax.
    E.g. spaces, parentheses, punctuations
    Those bearing content are replaced by appropriate representations and do not stay as characters.
    E.g. The character '+' is replaced by a data constructor.
    E.g. The character '0' is replaced by the number 0.
  - The purpose is to present only the essential structure and content, ready for interpreting, compiling, analyses.
  - Parsers usually output abstract syntax trees when successful.
- Comparison of Parse Tree and Abstract Syntax Tree:



Lexical Analysis/Tokenization:

- In principle, grammar and parser can work on characters directly, but it is usually messy.
- In practice, we have 2 stages:
  1. We chop character streams into chunks and classify into **lexemes/tokens** and we discard spaces. Furthermore, we typically use objects or data representations instead of the actual strings.
     E.g.
     " (xa * xb)**25 " →
     [Open, Var "xa", Op Mul, Var "xb", Close, Op Exp, NumLiteral 25]
     Here, we use the object or data representation "Open" to denote the open parentheses, "(". Furthermore, notice the space between " and (. In the array, the space isn't shown.
     Note: We can use regular expression to determine what category, variable, number, operator, etc, an item is.
     This step is called **lexical analysis/tokenization**.
  2. Parsing is based on CFG. Terminal symbols are tokens, not characters.

Recursive Descent Parsing:

- Recursive descent parsers can be nicely expressed in an embedded domain-specific language, built from a few primitives and composed using the connectives from Functor, Applicative, and Monad. There is one more relevant connective type class, Alternative, for failures and choices.
- The Alternative type class is used for backtracking.

Parser representation:

- Each parser can be represented as a function taking an input string, consuming a prefix of it, and giving one of:
    1. failure
    2. success, with unconsumed suffix and answer
- When the result is "success", it is important to give the unconsumed suffix rather than losing it. When liftA2 and >>= chain up two parsers, the second parser needs to see the leftover from the first parser. You can also think of a state variable for the current string to parse. Overall we are combining two effects, failure and state.
- So we use the below function type to define our parser type:

```
data Parser a = MkParser (String -> Maybe (String, a))

unParser :: Parser a -> String -> Maybe (String, a)
unParser (MkParser sf1) = sf1
```

- Each parser, there could be multiple, takes in an input string and consumes the first few characters of that input string. Then, the parser can either:
    1. Declare that this is not what I'm looking for. (Declare failure)
    2. Declare that this is what I'm looking for. The parser will give the unconsumed suffix to the next parser and give an answer. (Declare success)

  The parser is a function that takes in a string and returns the rest of the string and an answer upon success or nothing upon failure.
- We can use unParser to use our parser.
- I use Maybe because I anticipate at most one valid answer, and for simplicity I don't include error information for failures.
- It is also possible to use [] to anticipate ambiguous grammars and multiple valid answers, with the empty list for failure.
- Here is the function for using a parser. You give it an input string and it gives you an overall final answer (success or failure). It discards the final leftover as usually we aren't interested in it. If you're interested, use unParser above.

```
runParser :: Parser a -> String -> Maybe a
runParser (MkParser sf) inp = case sf inp of
                                Nothing -> Nothing
                                Just (_, a) -> Just a
                              -- OR: fmap (\(_,a) -> a) (sf inp)
```

  In the case of Nothing, we get Nothing back.
  In the case of any string and an answer, a, we get the answer, a, back.

Parsing primitives (character level):

- In this example, a basic parser reads a character and gives it to you. It fails when/if there's no character to read.

Here's the code:

```
anyChar :: Parser Char
anyChar = MkParser sf
  where
    sf "" = Nothing
    sf (c:cs) = Just (cs, c)
```

E.g.

```
*ParserLib> unParser anyChar "xyz"
Just ("yz",'x')
*ParserLib> unParser anyChar ""
Nothing
```

```
*ParserLib> runParser anyChar "xyz"
Just 'x'
*ParserLib> runParser anyChar ""
Nothing
```

- In this example, the parser is expecting a specific character and wants to read and check it. It fails if the character it's reading is not the expected character or if there's no character to read.

Here's the code:

```
char :: Char -> Parser Char
char wanted = MkParser sf
  where
    sf (c:cs) | c == wanted = Just (cs, c)
    sf _ = Nothing
```

E.g.

```
*ParserLib> unParser (char 'A') "xyz"
Nothing
*ParserLib> unParser (char 'A') "xyza"
Nothing
*ParserLib> unParser (char 'A') "xyzA"
Nothing
*ParserLib> unParser (char 'A') "Axyz"
Just ("xyz",'A')
*ParserLib> runParser (char 'A') "xyz"
Nothing
*ParserLib> runParser (char 'A') "xyzA"
Nothing
*ParserLib> runParser (char 'A') "Axyz"
Just 'A'
```

- In this example, the parser is expecting a character that satisfies a specific condition or predicate.

Here's the code:

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy pred = MkParser sf
  where
    sf (c:cs) | pred c = Just (cs, c)
    sf _ = Nothing
```

If you expect a letter, you say “satisfy isAlpha” (isAlpha is from Data.Char.).
E.g.

```
*ParserLib> unParser (satisfy isAlpha) "Axyz"
Just ("xyz",'A')
*ParserLib> unParser (satisfy isAlpha) "1xyz"
Nothing
*ParserLib> unParser (satisfy isAlpha) "xyz"
Just ("yz",'x')
*ParserLib> runParser (satisfy isAlpha) "xyz"
Just 'x'
*ParserLib> runParser (satisfy isAlpha) ""
Nothing
*ParserLib> runParser (satisfy isAlpha) "1xyz"
Nothing
```

- In this example, the parser is checking that the input string is empty. So its failure/success criterion is the opposite of char's.

Here’s the code:

```
eof :: Parser ()
eof = MkParser sf
  where
    sf "" = Just ("", ())
    sf _ = Nothing
```

E.g.

```
*ParserLib> unParser eof ""
Just ("",())
*ParserLib> unParser eof " xyz"
Nothing
*ParserLib> unParser eof "xyz"
Nothing
*ParserLib> runParser eof ""
Just ()
*ParserLib> runParser eof " xyz"
Nothing
*ParserLib> runParser eof "xyz"
Nothing
```

Functor, Applicative, Monad, Alternative connectives:

- The effects of the Parser type are a combination of failure and state. Accordingly, the implementation of the Functor, Applicative, and Monad methods also combine those of Maybe and State.
  I.e. Checking for Nothing vs Just, and plumbing for state values (input strings and leftovers).

- Fmap, Applicative and Monad implementations:

```
instance Functor Parser where
  -- fmap :: (a -> b) -> Parser a -> Parser b
  fmap f (MkParser sf) = MkParser sfb
    where
      sfb inp = case sf inp of
                  Nothing -> Nothing
                  Just (rest, a) -> Just (rest, f a)
                -- OR: fmap (\(rest, a) -> (rest, f a)) (sf inp)

instance Applicative Parser where
  -- pure :: a -> Parser a
  pure a = MkParser (\inp -> Just (inp, a))

  -- liftA2 :: (a -> b -> c) -> Parser a -> Parser b -> Parser c
  -- Consider the 1st parser to be stage 1, 2nd parser stage 2.
  liftA2 op (MkParser sf1) p2 = MkParser g
    where
      g inp = case sf1 inp of
                Nothing -> Nothing
                Just (middle, a) ->
                  case unParser p2 middle of
                    Nothing -> Nothing
                    Just (rest, b) -> Just (rest, op a b)

instance Monad Parser where
  -- return :: a -> Parser a
  return = pure

  -- (>>=) :: Parser a -> (a -> Parser b) -> Parser b
  MkParser sf1 >>= k = MkParser g
    where
      g inp = case sf1 inp of
                Nothing -> Nothing
                Just (rest, a) -> unParser (k a) rest
```

- In Control.Applicative there are more utility connectives, two of which are useful for parsing.

```
(*>) :: Applicative f => f a -> f b -> f b
p *> q = liftA2 (\a b -> b) p q
-- Drop p's answer, give only q's answer.  Like (>>) but Applicative.

(<*) :: Applicative f => f a -> f b -> f a
p <* q = liftA2 (\a b -> a) p q
-- Drop q's answer, give only p's answer.
```

Example of chaining up several primitive parsers sequentially: I want a letter, then a digit, then '!'; the answer is the letter and the digit in a string, and drop the '!'.
I can use the following code to do it:

```
Ide :: Parser String
Ide = liftA2 (\x y -> [x,y]) (satisfy isAlpha) (satisfy isDigit) <* (char '!')
```

```
*ParserLib> unParser Ide "B6!A"
Just ("A","B6")
*ParserLib> unParser Ide "B36!A"
Nothing
*ParserLib> unParser Ide "2B6!A"
Nothing
*ParserLib> unParser Ide "B6a!"
Nothing
```

```
*ParserLib> unParser Ide "B6!"
Just ("","B6")
```

- There is one more type class "Alternative" in the standard library containing methods for failure and choice.
  choice is an associative binary operator, and failure is the identity element.
  Note: You need to import it from Control.Applicative.

```
class Applicative f => Alternative f where
   empty :: f a
   (<|>) :: f a -> f a -> f a
   many :: f a -> f [a]  -- has default implementation
   some :: f a -> f [a]  -- has default implementation
```

This type class was actually inspired by parsing.
The <|> operator came from the "|" in BNF, and many and some came from "0 or more times" and "1 or more times".

Here's the implementation for our Parser:

```
instance Alternative Parser where
   -- empty :: Parser a
   -- Always fail.
   -- Putting empty under if-then-else or some conditional branching makes it
     useful
   empty = MkParser (\_ -> Nothing)

   -- (<|>) :: Parser a -> Parser a -> Parser a
   -- Try the 1st one. If success, done; if failure, do the 2nd one
   MkParser sf1 <|> p2 = MkParser g
    where
      g inp = case sf1 inp of
              Nothing -> unParser p2 inp
              j -> j       -- the Just case
```

```
-- many :: Parser a -> Parser [a]
-- 0 or more times, maximum munch, collect the answers into a list.
-- Can use default implementation. And it goes as:
many p = some p <|> pure []
-- How to make sense of it: To repeat 0 or more times, first try 1 or more
-- times!  If that fails, then we know it's 0 times, and the answer is the
-- empty list.

-- some :: Parser a -> Parser [a]
-- 1 or more times, maximum munch, collect the answers into a list.
-- Can use default implementation. And it goes as:
some p = liftA2 (:) p (many p)
-- How to make sense of it: To repeat 1 or more times, do 1 time, then 0 or
-- more times!  Use liftA2 to chain up and collect answers.
```

E.g.

```
*ParserLib> unParser (many (satisfy isAlpha)) "abc012"
Just ("012","abc")
*ParserLib> unParser (many (satisfy isAlpha)) "a012"
Just ("012","a")
*ParserLib> unParser (many (satisfy isAlpha)) "012"
Just ("012","")
*ParserLib> unParser (some (satisfy isAlpha)) "abc012"
Just ("012","abc")
*ParserLib> unParser (some (satisfy isAlpha)) "a012"
Just ("012","a")
*ParserLib> unParser (some (satisfy isAlpha)) "012"
Nothing
*ParserLib> unParser (some (satisfy isAlpha)) "abc012xyz"
Just ("012xyz","abc")
*ParserLib> unParser (many (satisfy isAlpha)) "abc012xyz"
Just ("012xyz","abc")
```

Example use of <|>: I want ‘A’ or ‘B’, followed by ‘0’ or ‘1’:

```
ab01 :: Parser String
ab01 = liftA2 (\x y -> [x,y]) (char 'A' <|> char 'B') (char '0' <|> char '1')
```

E.g.

```
*ParserLib> ab01 = liftA2 (\x y -> [x,y]) (char 'A' <|> char 'B') (char '0' <|> char '1')
*ParserLib> unParser ab01 "x1"
Nothing
*ParserLib> unParser ab01 "A1"
Just ("","A1")
*ParserLib> unParser ab01 "B0"
Just ("","B0")
*ParserLib> unParser ab01 "B1"
Just ("","B1")
*ParserLib> unParser ab01 "A0"
Just ("","A0")
```

- In Control.Applicative there is also a utility connective based on Alternative. It's very handy when an ENBF rule says “0 or 1 time”.

optional :: Alternative f => f a -> f (Maybe a)
optional p = fmap Just p <|> pure Nothing

E.g.

```
*ParserLib> optional p = fmap Just p <|> pure Nothing
*ParserLib> unParser (optional (char '0')) "xyz"
Just ("xyz",Nothing)
*ParserLib> unParser (optional (char '0')) "xyz0"
Just ("xyz0",Nothing)
*ParserLib> unParser (optional (char '0')) "0"
Just ("",Just '0')
*ParserLib> unParser (optional (char '0')) "0000"
Just ("000",Just '0')
*ParserLib> unParser (optional (char '0')) "0000xyz"
Just ("000xyz",Just '0')
```

Furthermore, if we do **unParser (char '0') "0xyz"**, we get back **Just ("xyz",'0')**.
If we do **unParser (optional (char '0')) "0xyz"**, we get back **Just ("xyz",Just '0')**.

```
*ParserLib> unParser (char '0') "0xyz"
Just ("xyz",'0')
```

```
*ParserLib> unParser (optional (char '0')) "0xyz"
Just ("xyz",Just '0')
```

Parsing Primitives (lexeme/token level):

- We won't actually use the character-level primitives directly. A reason is that spaces will get into the way. Another is that we think at the token level. We only use character-level primitives to implement token-level primitives such as the ones below. Then we use connectives and token-level primitives for the grammar.
- Whitespace handling convention: Token-level primitives assume there are no leading spaces, and skip trailing spaces, so the next token primitive may assume no leading

spaces. Something else at the outermost level will have to skip initial leading spaces. This will be discussed later.

```
-- | Space or tab or newline (unix and windows).
whitespace :: Parser Char
whitespace = satisfy (\c -> c `elem` ['\t', '\n', '\r', ' '])

-- | Consume zero or more whitespaces, maximum munch.
whitespaces :: Parser String
whitespaces = many whitespace

-- | Read a natural number (non-negative integer), then skip trailing spaces.
natural :: Parser Integer
natural = fmap read (some (satisfy isDigit)) <* whitespaces
-- read :: Read a => String -> a
-- For converting string to your data type, assuming valid string.  Integer
-- is an instance of Read, and our string is valid, so we can use read.

-- | Read an identifier, then skip trailing spaces.  Disallow the listed keywords.
identifier :: [String] -> Parser String
identifier keywords =
   satisfy isAlpha
   >>= \c -> many (satisfy isAlphaNum)
   >>= \cs -> whitespaces
   >> let str = c:cs
   in if str `elem` keywords then empty else return str

-- | Read the wanted keyword, then skip trailing spaces.
keyword :: String -> Parser String
keyword wanted =
   satisfy isAlpha
   >>= \c -> many (satisfy isAlphaNum)
   >>= \cs -> whitespaces
   *> if c:cs == wanted then return wanted else empty
```

CFG Parsing:

- A parser for a context-free grammar can mostly look like the grammar rules. There are however a few things to watch out for, some tricks, and that lingering issue of initial leading spaces.
- The parsers here will produce abstract syntax trees of this type:

```
data Expr
   = Num Integer
   | Var String
   | Prim2 Op2 Expr Expr       -- Prim2 op operand operand
   | Let [(String, Expr)] Expr -- Let [(name, rhs), ...] body

data Op2 = Add | Mul
```

Right-associating Operator:

- Take this simple rule, and suppose we intend the operator to associate to the right:
  muls ::= natural { "*" natural } OR **muls ::= natural ["*" muls]**.
  The second form uses right recursion to convey right association.
  This is perfect for recursive descent parsing.

```
mulsRv1 :: Parser Expr
mulsRv1 = liftA2 link
        (fmap Num natural)
        (optional (liftA2 (,)
                (operator "*" *> pure (Prim2 Mul))
                mulsRv1))
 where
   link x Nothing = x
   link x (Just (op,y)) = op x y
```

Note:

```
*ParserLib> (,) True 'x'
(True,'x')
*ParserLib> (,) True False
(True,False)
```

```
Prelude Control.Applicative> liftA2 (,) [1..3] [1..4]
[(1,1),(1,2),(1,3),(1,4),(2,1),(2,2),(2,3),(2,4),(3,1),(3,2),(3,3),(3,4)]
```

We have the line **fmap Num natural** because we want the return type to be something in Expr. If we get an integer from natural, we want to add the Num tag to it.
E.g.

```
*ParserLib> unParser (fmap Num natural) "33"
Just ("",Num 33)
*ParserLib> runParser (fmap Num natural) "33"
Just (Num 33)
```

We have the line **(operator "*" *> pure (Prim2 Mul))** because if we see a "*", we ignore it and use pure(Prim2 Mul) to represent it.

Lastly, we have the line **mulsRv1** because we want to make a recursive call.

E.g. of running mulsRv1:

```
*ParserLib> runParser mulsRv1 "2*5"
Just (Prim2 Mul (Num 2) (Num 5))
*ParserLib> runParser mulsRv1 "2*5*7"
Just (Prim2 Mul (Num 2) (Prim2 Mul (Num 5) (Num 7)))
*ParserLib> runParser mulsRv1 "2*5*7*9"
Just (Prim2 Mul (Num 2) (Prim2 Mul (Num 5) (Prim2 Mul (Num 7) (Num 9))))
*ParserLib> runParser mulsRv1 "2"
Just (Num 2)
```

- Instead of writing this recursion by hand again for every right-associative operator, we can call a re-factored function and specify just your operand parser and operator parser.

Here is the re-factored general function for right-associative operators.

```
chainr1 :: Parser a                 -- ^ operand parser
        -> Parser (a -> a -> a)     -- ^ operator parser
        -> Parser a                 -- ^ whole answer
chainr1 getArg getOp = liftA2 link getArg
                           (optional
                               (liftA2 (,) getOp (chainr1 getArg getOp)))
 where
   link x Nothing = x
   link x (Just (op,y)) = op x y
```

So here is how we will implement the rule in practice:

```
mulsRv2 :: Parser Expr
mulsRv2 = chainr1 (fmap Num natural) (operator "*" *> pure (Prim2 Mul))
```

E.g.

```
Prelude ParserLib> runParser (chainr1 (fmap Var (identifier [])) (operator "*" *> pure (Prim2 Mul))) "x*y*z"
Just (Prim2 Mul (Var "x") (Prim2 Mul (Var "y") (Var "z")))
```

Left-associating operator:

- Suppose we want the operator to associate to the left instead. We cannot code up left recursion directly, but the trick is to implement the other form of the rule.

  Still imagine that the grammar rule is of this form: muls ::= natural { "*" natural }.
  Use many for the "{ "*" natural }" part to get a list of tuples of (operator, number).

  For example if the input string is "2 * 5 * 3 * 7", my plan is to:
  1. read "2" and get Num 2
  2. read "* 5 * 3 * 7" with the help of many and get [(Prim2 Mul, Num 5), (Prim2 Mul, Num 3), (Prim2 Mul, Num 7)]
  3. Then using foldl on the list, starting with Num 2 as the initial accumulator, will build the left-leaning tree

  I.e. The parser still does right-associating recursion, but we will use foldl on the return value to make it left-associating.

  Here's the code:

```
mulsLv1 :: Parser Expr
mulsLv1 = liftA2 link
        (fmap Num natural)
        (many (liftA2 (,)
              (operator "*" *> pure (Prim2 Mul))
              (fmap Num natural)))

  where
    link x opys = foldl (\accum (op,y) -> op accum y) x opys
```

  fmap Num natural gets us "Num 2".

```
(many (liftA2 (,)
              (operator "*" *> pure (Prim2 Mul))
              (fmap Num natural)))
```

  gets us [(Prim2 Mul, Num 5), (Prim2 Mul, Num 3), (Prim2 Mul, Num 7)]

  link x opys = foldl (\accum (op,y) -> op accum y) x opys combines "Num 2" with [(Prim2 Mul, Num 5), (Prim2 Mul, Num 3), (Prim2 Mul, Num 7)].
  The argument, x, is "Num 2."
  The argument, opys, is [(Prim2 Mul, Num 5), (Prim2 Mul, Num 3), (Prim2 Mul, Num 7)].
  In **foldl (\accum (op,y) -> op accum y) x opys**, accum is "Num 2", op is "Prim2 Mul" and y is "Num _". It's taking the value of **fmap Num natural** and putting "Prim2 Mul" over it and the first "Num _" in the list.

  Note: The recursive call is in "many".

E.g.

```
*ParserLib> runParser mulsLv1 "2"
Just (Num 2)
*ParserLib> runParser mulsLv1 "2*5"
Just (Prim2 Mul (Num 2) (Num 5))
*ParserLib> runParser mulsLv1 "2*5*7"
Just (Prim2 Mul (Prim2 Mul (Num 2) (Num 5)) (Num 7))
*ParserLib> runParser mulsLv1 "2*5*7*9"
Just (Prim2 Mul (Prim2 Mul (Prim2 Mul (Num 2) (Num 5)) (Num 7)) (Num 9))
```

- Again in practice we don't write this code again, we re-factor this into a general function:

```
chainl1 :: Parser a                 -- ^ operand parser
        -> Parser (a -> a -> a)     -- ^ operator parser
        -> Parser a                 -- ^ whole answer
chainl1 getArg getOp = liftA2 link
                getArg
                (many (liftA2 (,) getOp getArg))
 where
  link x opys = foldl (\accum (op,y) -> op accum y) x opys
```

Then we use it like:

```
mulsLv2 :: Parser Expr
mulsLv2 = chainl1 (fmap Num natural) (operator "*" *> pure (Prim2 Mul))
```

E.g.

```
Prelude ParserLib> runParser (chainl1 (fmap Var (identifier [])) (operator "*" *> pure (Prim2 Mul))) "x*y*z"
Just (Prim2 Mul (Prim2 Mul (Var "x") (Var "y")) (Var "z"))
```

Comparing between mulsLv1 and mulsRv1:

- E.g.

```
Prelude ParserLib> runParser mulsRv1 "2"
Just (Num 2)
Prelude ParserLib> runParser mulsLv1 "2"
Just (Num 2)
```

```
Prelude ParserLib> runParser mulsRv1 "2*5"
Just (Prim2 Mul (Num 2) (Num 5))
Prelude ParserLib> runParser mulsLv1 "2*5"
Just (Prim2 Mul (Num 2) (Num 5))
```

```
Prelude ParserLib> runParser mulsRv1 "2*5*7"
Just (Prim2 Mul (Num 2) (Prim2 Mul (Num 5) (Num 7)))
Prelude ParserLib> runParser mulsLv1 "2*5*7"
Just (Prim2 Mul (Prim2 Mul (Num 2) (Num 5)) (Num 7))
```

```
Prelude ParserLib> runParser mulsRv1 "2*5*7*9"
Just (Prim2 Mul (Num 2) (Prim2 Mul (Num 5) (Prim2 Mul (Num 7) (Num 9))))
Prelude ParserLib> runParser mulsLv1 "2*5*7*9"
Just (Prim2 Mul (Prim2 Mul (Prim2 Mul (Num 2) (Num 5)) (Num 7)) (Num 9))
```

Initial space, final junk:

- Token-level parsers assume no leading spaces.
  Notice how if we have spaces in front, natural doesn't work.

```
Prelude ParserLib> unParser natural "      45"
Nothing
Prelude ParserLib> unParser natural "5      "
Just ("",5)
```

  This is because natural is expecting to read numbers and it's reading spaces instead.
- Another problem is that a small parser for a part of the grammar can leave non-space stuff unconsumed, since we anticipate that later a small parser for another part may need it. But the overall combined parser for the whole grammar cannot leave any non-space stuff unconsumed. By the time you're done with the whole grammar, any non-space leftover means the original input string is actually erroneous.
  E.g. We don't consider "2*3*" to be a legal arithmetic expression because our muls parsers can make sense of the prefix "2*3" but leaves the last "*" unconsumed.

```
Prelude ParserLib> unParser mulsLv1 "2*3*"
Just ("*",Prim2 Mul (Num 2) (Num 3))
```

- The trick for solving both is to have a “main” parser whose job is simply to clear initial leading spaces, call the parser for the start symbol, then use eof to check that there is nothing left.

  Here’s the code:

```
lesson2 :: Parser Expr
lesson2 = whitespaces *> muls <* eof
  where
    muls = chainl1 (fmap Num natural) (operator "*" *> pure (Prim2 Mul))
```

```
Prelude ParserLib> unParser lesson2 "2*3*"
Nothing
Prelude ParserLib> unParser lesson2 "2*3"
Just ("",Prim2 Mul (Num 2) (Num 3))
Prelude ParserLib> unParser lesson2 "2*3      "
Just ("",Prim2 Mul (Num 2) (Num 3))
Prelude ParserLib> unParser lesson2 "      2*3"
Just ("",Prim2 Mul (Num 2) (Num 3))
Prelude ParserLib> unParser lesson2 "      2*3      "
Just ("",Prim2 Mul (Num 2) (Num 3))
```

- **Note:** We do *> and <* outside of the parser because if we do it inside, there may be recursive calls to it in the middle of your grammar which may give back the wrong result.

Operator precedence and parentheses:

- Suppose I have two operators “*” and “+”, with “+” having lower precedence, and I also support parentheses for overriding precedence.
- In other words, from lowest precedence (binding most loosely) to highest (binding most tightly) is “+”, then “*”, then individual numbers and parentheses (same level without ambiguity).
- The trick is to have lower (looser) rules call higher (tighter) rules, and have the parentheses rule call the lowest rule for recursion. The start symbol is from the lowest rule. This is also how you can write your grammar to convey precedence.

- E.g.

  So my grammar goes like (start symbol is adds):

```
adds ::= muls { "+" muls }
muls ::= atom { "*" atom }
atom ::= natural | "(" adds ")"
```

  And my parser goes like (let's say left-associating operators):

```
lesson3 :: Parser Expr
lesson3 = whitespaces *> adds <* eof
 where
   adds = chainl1 muls (operator "+" *> pure (Prim2 Add))
   muls = chainl1 atom (operator "*" *> pure (Prim2 Mul))
   atom = fmap Num natural <|> (openParen *> adds <* closeParen)
```

  E.g.

```
Prelude ParserLib> unParser lesson3 "2+3*4"
Just ("",Prim2 Add (Num 2) (Prim2 Mul (Num 3) (Num 4)))
Prelude ParserLib> unParser lesson3 "2+3*4+5"
Just ("",Prim2 Add (Prim2 Add (Num 2) (Prim2 Mul (Num 3) (Num 4))) (Num 5))
```

Keywords and variables:

- Here is the whole grammar and the start symbol is expr:

```
expr ::= local | adds
local ::= "let" { var "=" expr ";" } "in" expr
adds ::= muls { "+" muls }
muls ::= atom { "*" atom }
atom ::= natural | var | "(" expr ")"
```

  A problem is “let inn+4” should be a syntax error, but a naïve parser implementation sees “let”, “in”, “n”, “+”, “4”.

  One solution is to use a parser for a reserved word should first read as many alphanums as possible, not just the expected letters, and then check that the whole string equals the keyword. This is what keyword does in an earlier section.

  Conversely, the parser for identifiers should read likewise, but then check that the string doesn't clash with reserved words. This is why identifier from earlier takes a parameter for reserved words to avoid.

- Here is the whole parser:

```
lesson4 :: Parser Expr
lesson4 = whitespaces *> expr <* eof
 where
  expr = local <|> adds

  local = pure (\_ eqns _ e -> Let eqns e)
        <*> keyword "let"
        <*> many equation
        <*> keyword "in"
        <*> expr
  -- Basically a liftA4.
  -- Could also be implemented in monadic style, like equation below.

  equation = var
          >>= \v -> operator "="
          >> expr
          >>= \e -> semicolon
          >> return (v, e)
  -- Basically a liftA4.
  -- Recall that liftA4 f a b c d = pure f <*> a <*> b <*> c <*> d
  -- Could also be implemented in applicative style, like local above.

  semicolon = char ';' *> whitespaces
  adds = chainl1 muls (operator "+" *> pure (Prim2 Add))
  muls = chainl1 atom (operator "*" *> pure (Prim2 Mul))
  atom = fmap Num natural
       <|> fmap Var var
       <|> (openParen *> expr <* closeParen)
  var = identifier ["let", "in"]
```

E.g.

```
*ParserLib> unParser lesson4 "let x=5; in x+5"
Just ("",Let [("x",Num 5)] (Prim2 Add (Var "x") (Num 5)))
*ParserLib> unParser lesson4 "let x=5; y=1; in x+5"
Just ("",Let [("x",Num 5),("y",Num 1)] (Prim2 Add (Var "x") (Num 5)))
*ParserLib> unParser lesson4 "let in x+5"
Just ("",Let [] (Prim2 Add (Var "x") (Num 5)))
```

Semantics I: Expressions, Bindings, Functions:

- Is used to implement interpreters.
- **Definition: static X** means doing X by analysing the code, without needing to run it and wait, while **dynamic X** means doing X while running the code.
- For a basic setup, I will start with dynamic checking of types and dynamic checking that variables exist when they are used.

Setup:

- I will have several kinds of run-time errors such Type errors, variable not found, and if you support division, you will also have division by zero.
- I use the Either monad to represent this possibility.
- We should use an algebraic data type for error messages, especially if later you support catching and handling errors.
- Here's the data type for my interpreter:

```
mainInterp :: Expr -> Either String Value
```

  Note: For simplicity, the professor used String for error messages. We should use an algebraic data type.
  If we get an error, we get something of type String.
  Otherwise, we get something of type Value.
- We can use a monad to model the fact that my language has the effect of errors/exceptions.
- I have a function, raise, for raising errors. It is defined to be simply Left in this lecture but it won't be that easy in a more featureful interpreter for a more complex language, such as stateful languages.
  Here's the code for raise:

```
raise :: String -> Either String a
raise = Left
```

- I will also be passing around a dictionary that maps variables to values. So I need this:

```
mainInterp expr = interp expr Map.empty
interp :: Expr -> Map String Value -> Either String Value
```

  The recursion happens in interp, which is a helper interpreter.
  Map String Value is used to map variable names to Values.
  Now I will implement interp for each construct.

Basic constructs:

- This language has number literals, boolean literals, and binary operators:

```
data Expr = Num Integer
          | Bln Bool
          | Prim2 Op2 Expr Expr        -- Prim2 op operand operand
          | ...

data Op2 = Eq | Plus | Mul
```

- I will be evaluating them and more to number values, boolean values, and later another kind of values.
- A clean habit is not to re-use the abstract syntax tree type, but to define a separate type, since the values have much fewer possibilities, and some possibilities will not correspond well to any abstract syntax tree.

E.g.

```
data Value = VN Integer
           | VB Bool
```

- Here is how I evaluate a number literal. Boolean literal is similar.

```
interp (Num i) _ = pure (VN i)
```

Arithmetic (all operands evaluated):

- Here is how I evaluate addition; most other arithmetic operators are similar. The insight is to use structural recursion to evaluate the operands, then you will have number values to add. The annoying part is to check that the values are actually numbers, so I re-factor out the checking to a helper function.
- Here's the code:

```
interp (Prim2 Plus e1 e2) env =
  interp e1 env
  >>= \a -> intOrDie a
  >>= \i -> interp e2 env
  >>= \b -> intOrDie b
  >>= \j -> return (VN (i+j))

intOrDie :: Value -> Either String Integer
intOrDie (VN i) = pure i
intOrDie _ = raise "type error"
```

Note: env is a dictionary.

Note: When checking division, we have to check for division by 0 by checking if the denominator is 0.

Short-circuiting, conditionals (operands selectively evaluated):

- I have an if-then-else:

```
data Expr = ...
          | Cond Expr Expr Expr        -- Cond test then-branch else-branch
```

If test is true, then we evaluate the "then-branch" only.
Otherwise, we evaluate the "else-branch" only.

This is a short-circuiting operator: some operands are selectively evaluated, others skipped.
Here is the code:

```
interp (Cond test eThen eElse) env =
  interp test env
  >>= \a -> case a of
    VB True -> interp eThen env
    VB False -> interp eElse env
    _ -> raise "type error"
```

You can add short-circuiting logical operators, and, or, and their semantics will be similar.

Variables, local bindings, environments (scopes):

- Here's the data type:

```
data Expr = ...
          | Var String
```

  However, where do we get the contents of variables from?
  A nice solution is to maintain a dictionary that maps variables to contents, so we can just look up.
  This dictionary is called **environment** and mapping a variable to its content is called **binding**.
  Also, the nature of the contents depend on the evaluation strategy of the language. For example, call by value just needs values, while lazy evaluation needs something more complex to cater for partly evaluated, partly unevaluated expressions.
- We will use Data.Map for dictionaries. Practical interpreters use hash tables and compilers use an array because they compile variable names to addresses.
- To evaluate a variable, we just look it up.
  If we find it, we return it.
  Otherwise, we raise an exception.
- Here's the code:

```
interp (Var v) env = case Map.lookup v env of
  Just a -> pure a
  Nothing -> raise "variable not found"
```

- E.g.

```
*SemanticsFunctions> interp (Var "x") (Map.fromList [("x", VN 4)])
Right (VN 4)
```

```
*SemanticsFunctions> interp (Var "x") (Map.fromList [("y", VN 4)])
Left "variable not found"
```

```
*SemanticsFunctions> interp (Prim2 Plus (Var "x") (Num 7)) (Map.fromList [("x", VN 4)])
Right (VN 11)
```

```
*SemanticsFunctions> interp (Prim2 Plus (Var "x") (Num 7)) (Map.fromList [("x", VB False)])
Left "type error"
```

- My local binding construct wraps an expression inside a new local context of 0 or more "name = expr" bindings.
  Here's the data type:

```
data Expr = ...
          | Let [(String, Expr)] Expr   -- Let [(name, rhs), ...] eval-me
```

  E.g.
  let { x=1; y=0 } in x+y is represented as
  Let [("x", Num 1), ("y", Num 0)] (Prim2 Plus (Var "x") (Var "y")).

- There are a number of decisions to make about the semantics of the local binding construct. It is also possible to offer many different local binding constructs, one for each way of making these decisions.
    1. **Scoping and recursion:**
       Suppose you have let { x=2+3; y=x+4; } in ….
        a. Here, it is equivalent to let { x=2+3 } in let { y=x+4 } in …
           So for y=x+4, we use the x in x=2+3.
           This is called **sequential binding**.
           If you choose sequential binding, right after you process one equation, you have to extend the environment to include its new binding, under which you process the remaining equations and eventually the wrapped expression.
        b. An alternative is that y=x+4 uses an outer x.
           This is called **parallel binding**.
           Note: This does not always imply parallel computing. It only implies semantic independence.

       Note: Those two choices don't support mutual recursion. The third alternative supports mutual recursion, so every equation may use every variable defined in the same group. If you do call by value, then you also need to place restrictions on the RHSes.
    2. **Evaluation strategy:**
       The choices are call by value, lazy evaluation, and call by name.
       With call by value, first go through the equations in the given order, evaluate the RHS of each one right away, and lastly evaluate the wrapped expression.
       E.g.
       I will evaluate 2+3 and store the result in x.
       Then, I will evaluate x+4, where x=5, and store the result in y.
       Then, I will evaluate the stuff after “in”.
- Here’s the code. This is for call by value.

```
interp (Let eqns evalMe) env =
   extend eqns env
   >>= \env' -> interp evalMe env'
   -- Example:
   --    let x=2+3; y=x+4 in x+y
   -- -> x+y   (with x=5, y=9 in the larger environment env')
   -- "extend env eqns" builds env'
  where
   extend [] env = return env
   extend ((v,rhs) : eqns) env =
      interp rhs env
      >>= \a -> let env' = Map.insert v a env
            in extend eqns env'
```

For extend, if the list is empty, we give back the environment
Otherwise, for each tuple in the list, “v” is the variable and “rhs” is what the variable is set to.
Since this is call by value, right away, we evaluate RHS under the given environment.
This is the line “interp rhs env”.

"interp rhs env" gives back some value. We bind that value with the lambda function. The lambda function puts the variable and the evaluated rhs into the original environment and then it makes a recursive call to process the rest of the equation.

E.g. Working on let { x=2+3; y=x+4; } in x+y, we get:

extend's nth iteration	v	rhs	env	env'
1	x	2+3	{}	{ x = 5 }
2	y	x+4	{ x = 5 }	{ x = 5, y = 9 }

Now we're ready to evaluate x+y under {x = 5, y = 9}.
However these are local variables unknown to the outside.
Suppose I have (let {x=2+3; y=x+4;} in x+y) * (1+1).
I make a recursive call to handle the "let-in". Inside that recursive call the new environment {x = 5, y = 9} is built for internal use, but not returned or passed back to the outside. The outside still uses the outside environment for "1+1".

- E.g.

```
*SemanticsFunctions> mainInterp (Let [("x", Num 1), ("y", Num 0)] (Prim2 Plus (Var "x") (Var "y")))
Right (VN 1)
```

```
*SemanticsFunctions> mainInterp (Let [("x", Num 1), ("y", (Prim2 Plus (Var "x") (Num 4)))] (Prim2 Plus (Var "x") (Var "y")))
Right (VN 6)
```

Here:
x = 1
y = x + 4 = 1 + 4 = 5
x + y = 1 + 5 = 6

```
*SemanticsFunctions> mainInterp (Let [("x", (Prim2 Plus (Num 3) (Num 4))), ("y", (Prim2 Plus (Var "x") (Num 4)))] (Prim2 Plus (Var "x") (Var "y")))
Right (VN 18)
```

Here:
x = 3 + 4 = 7
y = x + 4 = 7 + 4 = 11
x + y = 7 + 11 = 18.

```
*SemanticsFunctions> e1 = Let [("x", Num 5), ("y", Num 4)] (Prim2 Plus (Var "x") (Var "y"))
*SemanticsFunctions> e2 = Prim2 Plus (Num 1) (Num 1)
*SemanticsFunctions> mainInterp (Prim2 Mul e1 e2)
Right (VN 18)
*SemanticsFunctions> e2 = Prim2 Plus (Var "x") (Var "y")
*SemanticsFunctions> mainInterp (Prim2 Mul e1 e2)
Left "variable not found"
```

In the first example, e1 = 5+4 = 9 and e2 = 1+1 = 2. e1 * e2 = 18.
In the second example, e2 = Prim2 Plus (Var "x") (Var "y"). However, when we do mainInterp (Prim2 Mul e1 e2), the x and y in e1 are not shown to e2. Hence, we get the error message "variable not found".

Function construction (lambda), closures:

- For simplicity, I just have a lambda construct for anonymous functions.
  If you want to define a function with a name, use lambda together with let.
- Here's the data type:
  data Expr = ...
  | Lambda String Expr -- Lambda var body
- E.g. \x -> … is represented as Lambda "x" (...).
- Suppose your lambda is **\y->x+y**.
  y is a **bound variable** and x is a **free variable**.
  This also carries to let. In **let y=x+1 in x*y**, y is a bound variable and x is a free variable.
  Bound variable of/in an expression: You can see where the variable is introduced or declared or defined.
  Free variable of/in an expression: You can't see where the variable is introduced or declared or defined. It has to come from the outside.
  E.g. A free variable like "x" is probably bound in an outer context.
- When the interpreter runs into the lambda, and if it already knows x=10 from the outer context, it needs to attach "x=10" to the lambda so it is not forgotten.
- The value after evaluating a lambda needs to remember 3 things
  1. parameter name
  2. function body
  3. the environment in scope for this lambda.
- **Definition:** The combination of "\y->x+y" plus "x=10 from an outer context" is called a **closure**. A **closure** is a record or data structure that stores an expression together with the environment for all of its free variables.
- In this lecture, my only use of closures is for lambdas, so my closure representation is specialized for that purpose only.
- Here's the data type and code:
  data Value = ...
  | VClosure (Map String Value) String Expr
  interp (Lambda v body) env = pure (VClosure env v body)

  (Map String Value) is the environment.
  String is the parameter name(s).
  Expr is the function body/expression.
- E.g.

```
*SemanticsFunctions> mainInterp (Lambda "y" (Prim2 Plus (Var "y") (Num 5)))
Right (VClosure (fromList []) "y" (Prim2 Plus (Var "y") (Num 5)))
```

```
*SemanticsFunctions> mainInterp (Let [("x", Num 10)] (Lambda "y" (Prim2 Plus (Var "x") (Var "y"))))
Right (VClosure (fromList [("x",VN 10)]) "y" (Prim2 Plus (Var "x") (Var "y")))
```

  This is equivalent to let {x = 10;} in \y -> x + y

Function application:

- Here's the data type:

```
data Expr = ...
          | App Expr Expr          -- App func param
```

- There is a decision to make about the semantics of function application, namely which evaluation strategy should be used: call by value, lazy evaluation, or call by name. Here, I will do call by value.
  Note: All of them require you to evaluate the function until you get a function closure, sooner or later.
- Since I do call by value, I evaluate the parameter until I get a value. Then, I plug the value in. To plug the value in, just like evaluating let, I can first extend the environment to bind the parameter name to the parameter value, then it makes sense to evaluate the function body under that environment.
- Here's the code:

```
interp (App f e) env =
   interp f env
   >>= \c -> case c of
     VClosure fEnv v body ->
        interp e env
        >>= \eVal -> let bEnv = Map.insert v eVal fEnv  -- fEnv, not env
                     in interp body bEnv
        -- E.g.
        --    (\y -> 10+y) 17
        -- -> 10 + y      (but with y=17 in environment)
        --
```

E.g. for (let x=7 in \y -> x+y) 10:

1. interp on the function gives: VClosure {x = VN 7} "y" (x+y)
2. interp on the "10" gives: VN 10
3. Now do: interp (x+y) {x = VN 7, y = VN 10}

E.g.

```
*SemanticsFunctions> e = Num 10
*SemanticsFunctions> f = Let [("x", Num 7)] (Lambda "y" (Prim2 Plus (Var "x") (Var "y")))
*SemanticsFunctions> mainInterp (App f e)
Right (VN 17)
```

- **Dynamic scoping** is when a variable name refers to whoever has that name at the time of evaluation. We shouldn't do this.
  E.g.
  let {x=10; f = \y->x+y;} in
  let {x=5;} in
  f0 → 5+0 (Here, x=5 is used instead of x=10)
- **Lexical/Static scoping** is when a variable name refers to whoever has that name in the code location. We should do this.

Recursion:

- The trick is to take an extra parameter for a function to be called. Then, call a function with itself as a parameter.
- Here's an example of doing factorial recursively.
  Here's a trace of **let mkFac = \f -> \n -> if n=0 then 1 else n * (f f) (n-1) in mkFac mkFac 2**

```
  mkFac mkFac 2
→ (\f -> \n -> if n=0 then 1 else n * (f f) (n-1)) mkFac 2
→ (\n -> if n=0 then 1 else n * (mkFac mkFac) (n-1)) 2
→ if 2=0 then 1 else 2 * (mkFac mkFac) (2-1)
→ 2 * (mkFac mkFac) (2-1)
→ 2 * (\f -> \n -> ...) mkFac (2-1)
→ 2 * (\n -> if n=0 then 1 else n * (mkFac mkFac) (n-1)) (2-1)
→ 2 * (\n -> if n=0 then 1 else n * (mkFac mkFac) (n-1)) 1
→ 2 * if 1=0 then 1 else 1 * (mkFac mkFac) (1-1)
→ 2 * 1 * mkFac mkFac (1-1)
→ 2 * 1 * (\n -> if n=0 then 1 else n * mkFac mkFac (n-1)) (1-1)
→ 2 * 1 * (\n -> if n=0 then 1 else n * mkFac mkFac (n-1)) 0
→ 2 * 1 * if 0=0 then 1 else 0 * ...
→ 2 * 1 * 1
```