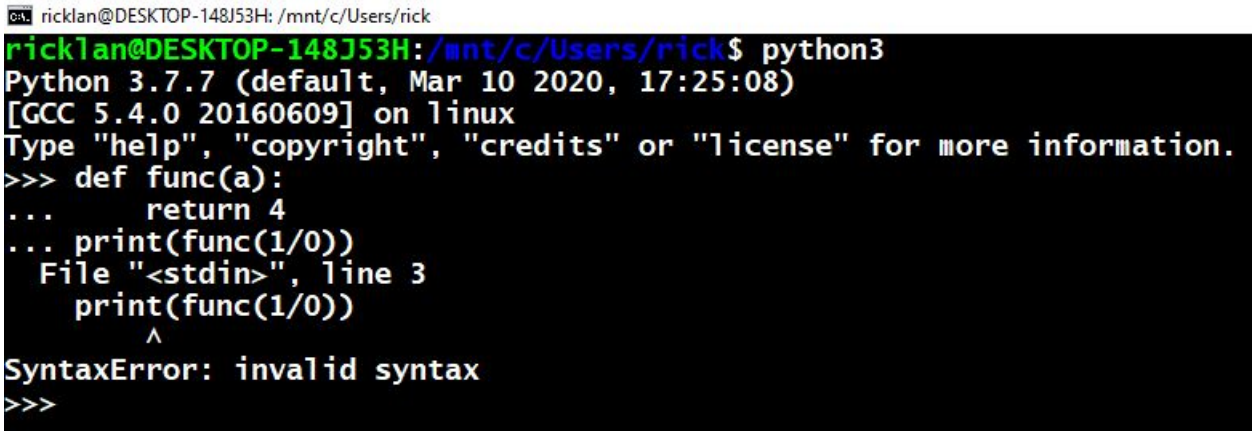
Introduction:

- 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:

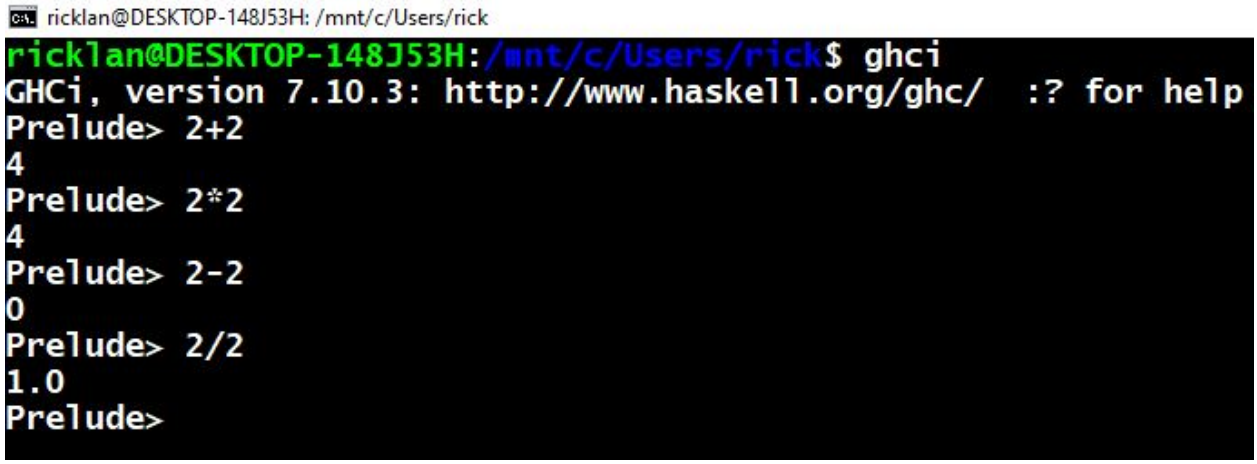
- putStr will print a string without a newline character at the end.
- putStrLn will print a string with a newline character at the end.
- print will just print whatever is in the parentheses.
- E.g. Consider the code and output below:

Basic Data Types:

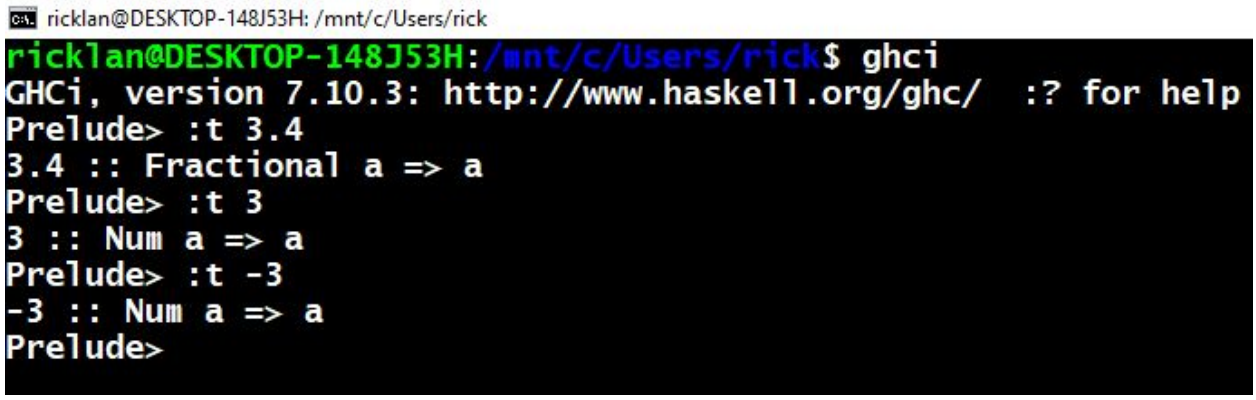
1. **Numbers:**

- 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:



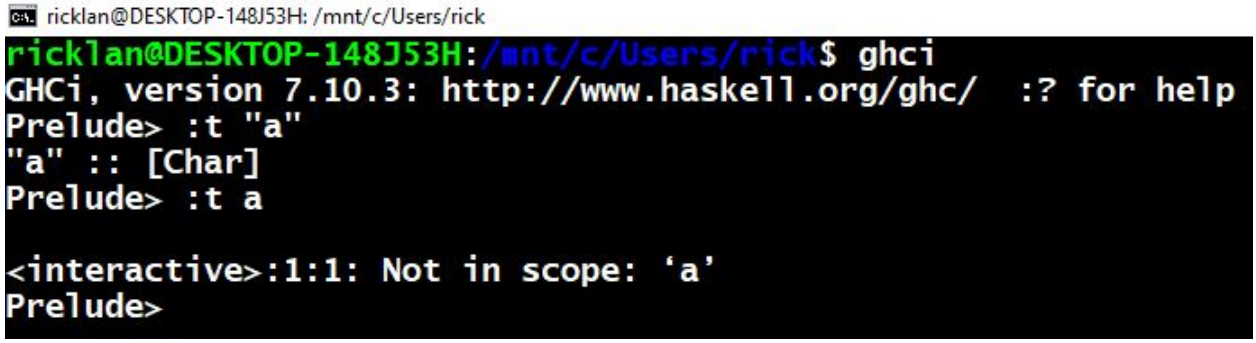
- **Note: :t** is to include the specific type related to the inputs.
- E.g. Consider the code and output below:



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:

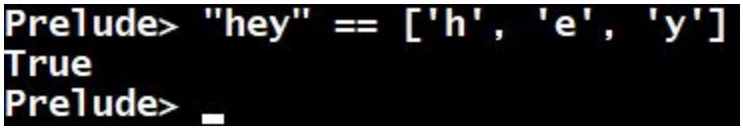


Note: The error message "<interactive>:1:1: Not in scope: `a'" means that the Haskell compiler is warning us that it is not able to recognize your input. Haskell is a type of language where everything is represented using a number.

3. **String:**

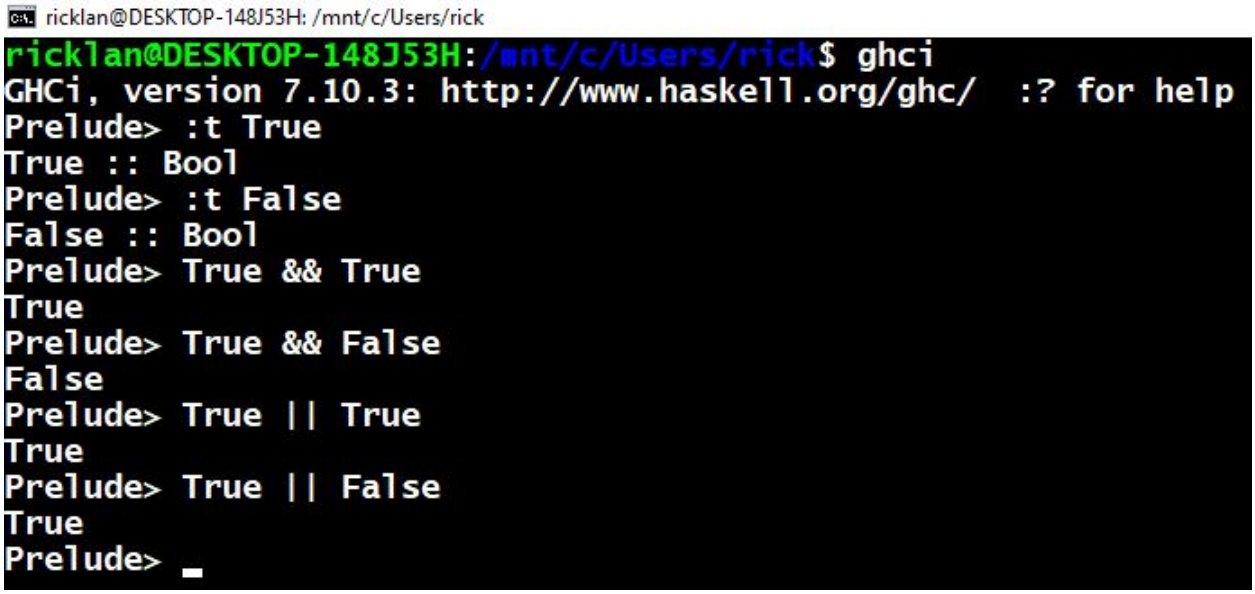
- A string is nothing but a collection of characters. There is no specific syntax for using string, but Haskell follows the conventional style of representing a string with double quotation.
- E.g. Consider the code and output below:

- **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:

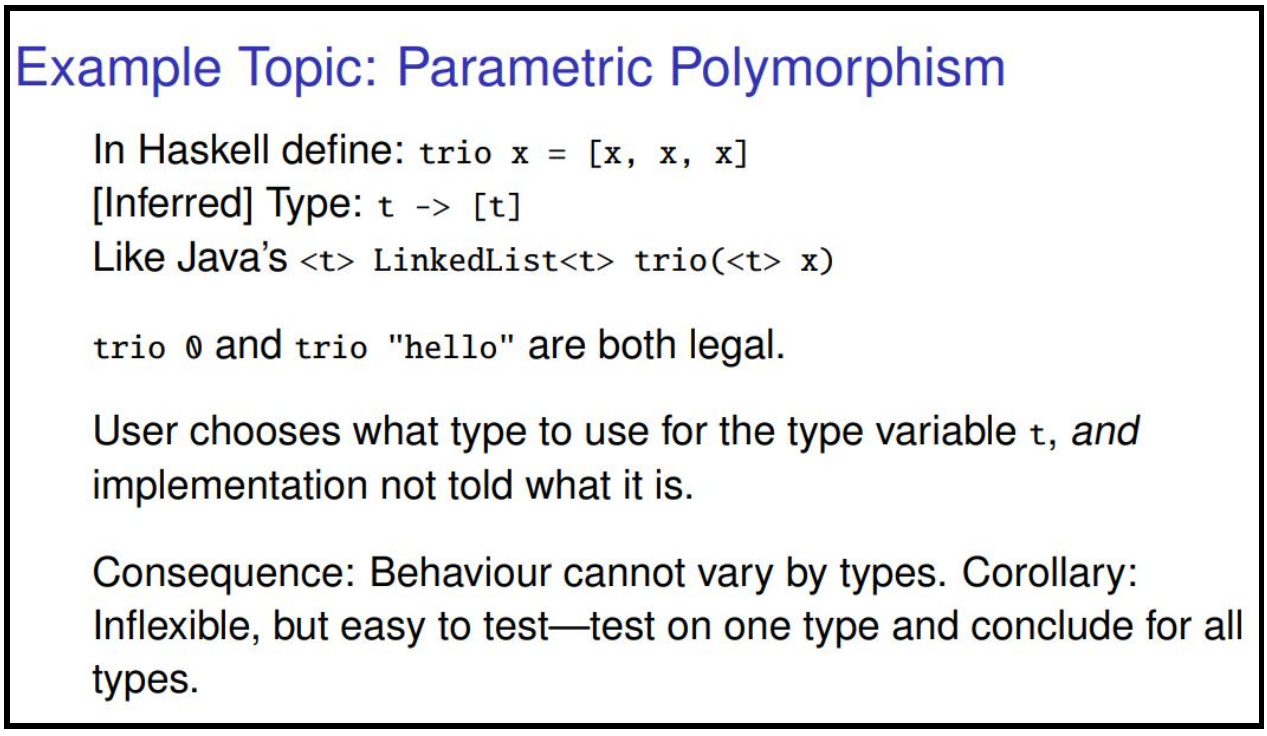
```
Prelude> let a = [1,2,3,4,5]
Prelude> let (b, c) = splitAt 2 a in b ++ (tail c)
[1,2,4,5]
```

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:

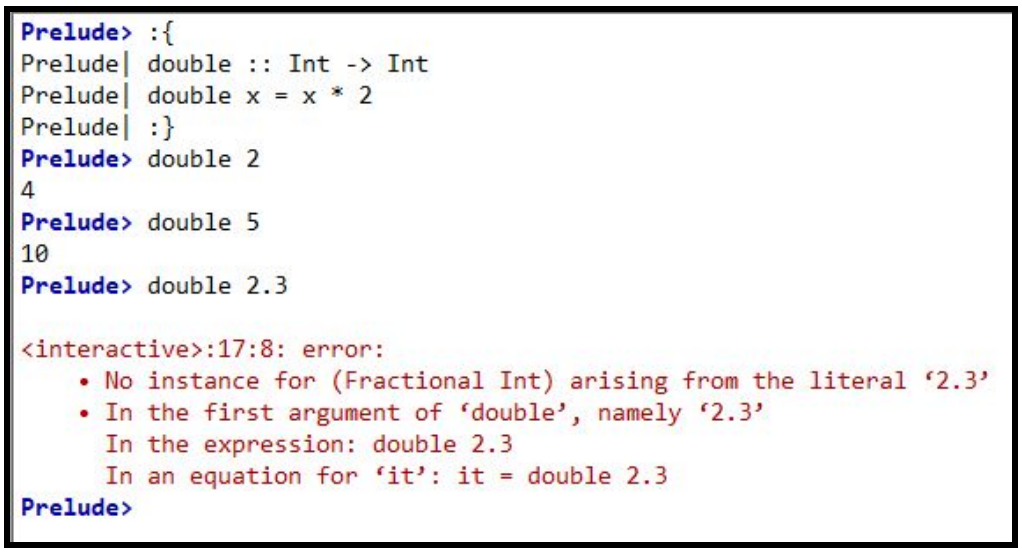
```
Prelude> double x = x * 2
Prelude> quadruple x = double double x

<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.