(DRAFT) Haskell for the Imperative Programmer

Anamitra Ghorui

Contents

1 Introduction **4**

2 Functions in Haskell (Part 1) **6**

3 "Control Structures" **8**

4 Lists (Part 1) **9**
4.1 `head` and `tail` 9
4.2 `length` 10
4.3 `take` 11

5 Where are the loops? **11**

6 Haskell is not C **12**

7 Why Do This? **13**

8 Few More Interpreter Facilities **14**
8.1 Running Code From Files 14
8.2 Multiline Statements 15
8.3 Convenience Settings for the Interpreter 15

9 Functions in Haskell (Part 2) **16**
9.1 Operators are also functions 16
9.2 What about Operator Precedence? 17

9.3 Pattern Matching . . . 17
9.4 More Ways to Define Functions . . . 18
9.4.1 Definitions using Pattern Matching . . . 18
9.4.2 `case` Syntax . . . 20
9.4.3 Guard Syntax . . . 21
9.4.4 `let .. in ..` Syntax . . . 23
9.4.5 `where` Syntax . . . 23
9.5 Currying . . . 24
9.5.1 Currying infix operators . . . 25
9.6 Function Composition . . . 26
9.7 Anonymous or Lambda Functions . . . 26
9.8 Evaluating Functions Inside Functions . . . 27

10 Haskell's Type System **27**
10.1 Explicit Declation of the Type of an Identifier . . . 29
10.2 10.2 Defining Function Types . . . 30
10.3 Type Variables . . . 31
10.4 Typeclasses . . . 32
10.5 Functions with Functions as Arguments Revisited . . . 35

11 Lists (Part 2) **36**
11.1 "Lists" in Haskell are not Arrays . . . 36
11.2 Cons in Haskell . . . 37
11.3 Using Cons for Pattern Matching . . . 38
11.4 List Ranges . . . 38
11.5 List Comprehensions . . . 40

12 Tuples **41**
12.1 `fst` and `snd` . . . 42

13 Iteration in Haskell **43**
13.1 The `iterate` Function . . . 43
13.2 The `map` function . . . 43
13.3 The `filter` Function . . . 44
13.4 The Function Application Operator (`$`) . . . 44
13.5 The `zip` Function . . . 46
13.6 The `zipWith` Function . . . 46
13.7 The `fold` and `scan` Functions . . . 46

13.7.1 `foldl`: Fold from Left to Right 46
13.7.2 `foldr`, Fold from Right to Left 47
13.7.3 `scanl` and `scanr` 48
13.7.4 Note: Pattern Matching in these Functions 48
13.7.5 `foldl1`, `foldr1`, `scanl1` and `scanr1` 49

14 Data, Records, Types, and Typeclasses **49**
14.1 Data 49
14.1.1 Printing New Data Types 50
14.1.2 Tuple-Like Data 50
14.1.3 Type Variables in Data Constructors 51
14.1.4 Recursive Data Constructors 51
14.1.5 Records 52

Contact:

- Email: `anamitraghorui gmail`
- Github: github.com/daujerrine
- Website: visphort.net

TODO Rename “variable” to “identifier” and “definition” with “binding”

1 Introduction

I will assume that you are proificient in at least one C or C-Like language, such as Python, Java, C++, Javascript etc.

Haskell is a programming language.

The current de-facto compiler for Haskell is the Glasgow Haskell Compiler (GHC). You can find it here.

You can start an interpreter on a terminal using the `ghci` command. You will be greeted with a prompt like this:

```
$ ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /home/turpentine/.ghci
Prelude>
```

`Prelude`, here, is a Haskell module. It is loaded by default into the interpreter and contains several core functions. Any new modules you load will appear prefixed against the prompt.

For the purposes of this document, we will not show the prompt as “`Prelude>`”, but as “>”. See Section 8.3 on how to actually remove it from your interpreter.

Here are a few basic commands. Try typing them in:

Example 1.1

```
Comments:

    > -- You can put comments like this.

"Hello World" Demonstration:

    > putStrLn("Hello") -- "Put String Line"
    Hello

Variables:
```

```
> x = 5
> x
5
```

Operators:

```
> x = 5
> x + 5
10
```

Convert things to strings:

```
> show(5)
"5"
> putStrLn(show(5))
5
```

Concatenation:

```
> putStrLn("The number is: " ++ show(5))
The number is: 5
```

Unless if it is to establish precedence order, parantheses for arguments are not necessary:

Example 1.2

```
> putStrLn "Hello"
Hello
> show 5
"5"
```

Example 1.3

```
> putStrLn "Hello " ++ show 5

<interactive>:3:1: error:•
     Couldn't match expected type ‘[Char’] with actual type ‘IO ()’•
     In the first argument of ‘(++)’, namely ‘putStrLn "Hello’"
```

```
In the expression: putStrLn "Hello" ++ show 5
In an equation for ‘’it: it = putStrLn "Hello" ++ show 5
```

The error message says that there are invalid arguments to the operator ++, because ++ sees the following precedence:

```
(putStrLn "Hello ") ++ (show 5).
```

Example 1.4

```
> putStrLn "Hello " ++ (show 5)

<interactive>:4:1: error:•
     Couldn't match expected type ‘[Char’] with actual type ‘IO ()’•
     In the first argument of ‘(++)’, namely ‘putStrLn "Hello’"
      In the expression: putStrLn "Hello" ++ (show 5)
      In an equation for ‘’it: it = putStrLn "Hello" ++ (show 5)
```

Still invalid arguments to the operator ++ with the same precedence.

Example 1.5

```
> putStrLn ("Hello " ++ (show 5))
Hello 5
```

Correct arguments and precedence.

Example 1.6

```
> putStrLn ("Hello " ++ show 5)
Hello 5
```

Also correct arguments and precedence.

For now, we will ignore the detailed description of the error messages such as the above. The error messages should become intuitive to you once you are familiar with all the features in Haskell.

2 Functions in Haskell (Part 1)

In Haskell, *basic* functions behave as if they were *mathematical expressions*. That is. Functions simply contain a logical formula. The result of the formula after we

substitute any variables present and evaluate the formula is what is returned.

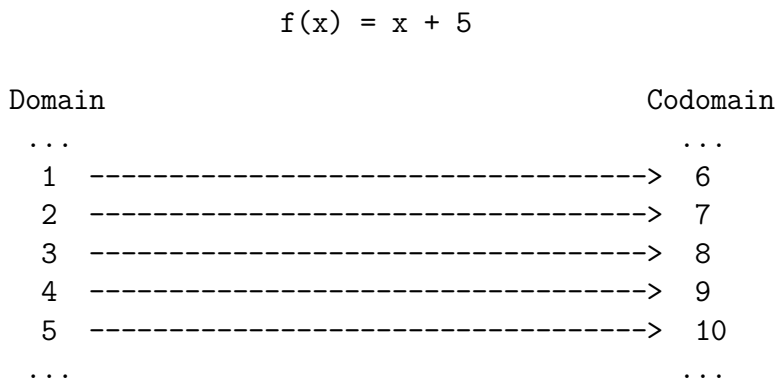
Example 2.1

```
Mathematical/Natural Language Expression:

    Let f(x) = x + 5.
    Hence, f(9) = 14.

    Let f(x, y) = x + y + 5.
    Hence, f(9, 1) = 15.

Haskell:

    > f x = x + 5
    > f 9
    14
    > f x y = x + y + 5
    > f 9 1
    15

Equivalent expression in Python:
    >>> f = lambda x: x + 5
    >>> f(9)
    14
    >>> f = lambda x, y: x + y + 5
    >>> f(9, 1)
    15
```

All of these above expressions satisfy the mathematical definition of a function:

> In mathematics, a function is a binary relation between two sets that associates every element of the first set to exactly one element of the second set.
>
> – Function (mathematics). Wikipedia, the Free Encyclopedia. Retrieved 23rd December 2020

We will refer to these two sets: the one which is the input set, and the output set to which the function *maps* each value from the input set to, as the *Domain* and the *Codomain* respectively.



We will get to how functions like `putStrLn` work later.

TODO: 1. Operators are functions 2. Function declaration methods 3. Type system

3 “Control Structures”

There is only one “control structure” in Haskell: the `if` statement.

Example 3.1

```
> x = 5
> if x == 5 then "great" else "that's okay too"
"great"
> x = 6
> if x == 5 then "great" else "that's okay too"
"that's okay too"
```

`You may notice that the if statement is returning a string.`

You can use these in functions as well:

Example 3.2

```
> isItGreat x = if x == 5 then "great" else "that's okay too"
> isItGreat 8
"that's okay too"
> putStrLn(isItGreat 8)
that's okay too
> putStrLn(isItGreat 5)
```

```
great
```

You can make a nested if-else statement like this by using the `else if` notation:

Example 3.3

```
> isItGreat x = if x == 5 then "great" else if x == 7 then "also gre
at" else "that's okay too"
> putStrLn(isItGreat 7)
also great
```

We will get to loops later.

4 Lists (Part 1)

You can declare lists in Haskell like this:

Example 4.1

```
> a = [1, 2, 3, 4]
> a
[1,2,3,4]
```

List Indexing works like this. List indices start from 0.:

Example 4.2

```
> a !! 1
> 2
```

There are a few inbuilt functions that will come in handy later:

4.1 `head` and `tail`

Example 4.3

```
(Continuing from the previous example:)

The following function returns the First element, or head of the array:

    > head a
```

```
1
```

This will return the rest of the array, excluding the array:

```
> tail a
[2,3,4]
> tail (tail a)
[3, 4]
```

- In the second expression, we use brackets to separate the arguments and establish the order of evaluation. (tail (tail a)) is also a valid and equivalent statement
- You may notice that the above two allow us to operate on arrays like linked lists.

Let's tail a few more times:

```
> tail (tail (tail (tail a)))
[]
```

We have reached the end of the list, which was of length 4. Hence ot returns an empty list. Let's keep this property in mind.

Putting in so many brackets is a bit tiresome. We can abbreviate such expressions using the "$" operator:

```
> tail $ tail $ tail $ tail a
[]
```

The "$" operator says that everything to its right is inside a new layer of parantheses. More accurately, it says that everything to its right is a parameter to the function on the left. This is also called the function application operator and a detailed description of it is given in section 9.

4.2 `length`

This one returns the length, as expected:

Example 4.4

(Continuing from the previous example:)

```
> length a
4
```

4.3 `take`

You can slice a desired number of elements from the start of a list using `take`:

Example 4.5

```
> take 2 [1, 2, 3, 4, 5]
[1,2]
```

5 Where are the loops?

There are no looping control structures in Haskell. You instead perform most computation using *recursive functions*.

Example 5.1

Summing numbers in a C-Like Language. We simply add numbers to keep track of the array index here:

```
int x[] = {1, 2, 3, 4, 5};
int sum = 0;
for (int i = 0; i < lengthOf(x); i++) {
    sum += x[i];
}
```

Summing numbers in Python. Instead of using a numerical index, we use an iterator instead (although it is still possible to use a numerical index here, the intent here is to illustrate the use of iterators in a conventional language):

```
x = [1, 2, 3, 4, 5]
sum = 0
for i in x:
    sum += i
```

Now, Summing numbers in Haskell:

```
> sumNumbers x = if x == [] then 0
      else head x + sum (tail x)
> sumNumbers [1, 2, 3, 4]
10
```

This is an implementation of a recursive function. Let's see what's happening here:

```
if x == [] then 0
```

We had previously seen that if we reach the end of a list, the tail tail function returns an empty list (since the end of the list is tailed by nothing). Hence, we say that once we reach the end of the list, we check whether x is an empty list or not. If it is, We say that the sum of the list is 0, which is true. This is our *Base Case* for the recursion.

```
else head x + sum (tail x)
```

Else, we say that the sum is the first item of the list, plus the sum of the rest of the list.

If we were to write this in a C-Like Language, it would look like this:

```
int sumNumbers(int x[]) {
    if (isEmpty(x)) {
        return 0;
    } else {
        return x[0] + sumNumbers(tail(x));
    }
}
```

Such a function would otherwise cause a Stack Overflow with very large input in other languages. However, this does not happen in Haskell.

6 Haskell is not C

By now it might be obvious to you that this is something radically different, way different than what you might have expected if you have never previously heard

about *Functional Programming Languages* of which haskell is one.

Languages such as C, C++, Java, Python all are called *Imperative Languages*. Here, we clearly specify the steps to follow in a specified order, which change the program's *current state* (such as, by changing the value of a variable) to achieve a solution. In short, *time matters most of the time*.

Functional Programming languages, however, come under *Declarative Programming Languages*. In declarative languages, we ask for a solution, but *not* how to achieve that solution. SQL and Make Files are examples of Declarative Programming languages which are *not* Functional Programming Languages.

In Functional Programming languages, we compose our programs using functions, however, there is little to no explicit definition of the order in which statements shall execute. We also avoid changing values of variables with respect to time as much as possible, that is, changing or *mutating* the state. Hence,

We attempt to minimise how much time matters.

Haskell comes under the class of *Pure* functional programming languages.

Recursion is used as a tool to define an *implicit order* in the solutions to our problems. **Haskell is built around efficient execution of such recursive functions and is the main method to solve problems in it**.

Have a look at Example 5.1 again and compare the Non-Haskell versions to the Haskell ones, and see: 1. How many variables are used in the program. 2. How many times a variable is mutated by the program in both versions. 3. How and how many times that the mutation is percievable with respect to the current scope.

7 Why Do This?

While a proper introduction of haskell's features have not yet been given, here are some of the advantages of having no mutation of state and no explicit execution order:

1. **Allow you to focus of the algorithm instead of its underlying implementation:** In languages like C, you have to implement algorithms such as binary search by explicitly defining certain variables in order to keep track of the sub array, and change their state in each step to find the solution. Thus

you have to modify the original algorithm quite a bit to get an efficient, practical program. In Haskell, however, you merely focus on the theoretical solution and you leave it to the compiler to convert it into an effecient program in most cases.

2. **Allow you to write better concurrent programs:** The fact that there is no mutation of state in functional programming means that all values in a concurrent variable will be fixed. Usually we instead implement an event-based message queue, where a concurrent process waits for a value on the queue, and continues only when a value is obtained. TODO
3. **Allow you to use constructs from discrete mathematics:** There are a lot of parallels between discrete mathematics and constructs present in Haskell, and allow you to use those structures in that manner.
4. **Reduce the general complexity of a program:** This is in part due to all of the above mentioned statements.

8 Few More Interpreter Facilities

8.1 Running Code From Files

You can type all of the above examples into a file and run them in the interpreter. Try creating a file with the following code:

```
formula a b = (a * b) / (a + b)
```

And saving it with a filename, say, `code.hs`. (Haskell sourcecode uses the `.hs` extension.)

Now, open a terminal and supply the file to the interpreter. You will be greeted with the following:

```
$ ghci formula.hs
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /home/turpentine/.ghci
[1 of 1] Compiling Main             ( a.hs, interpreted )
Ok, one module loaded.
>
```

Your file now has been loaded as a module, with its identifiers now in the interpreter's namespace.

Now try using the function:

```
> formula 2 3
1.2
```

We will eventually discuss *Compiled Haskell*. However, it will come in a later section and for now this will be enough to execute the examples present in the following sections.

8.2 Multiline Statements

The examples following this section will require you to enter in multiline statements. You can start a multiline statement by typing in `:{`, and ending with `:}` in the interpreter:

```
> :{
.. a x y =
..     if x > y then 1
..     else 0
.. :}
> a 2 3
0
```

The above expression can be also written without the `:{` and `:}` commands if executed from a file. In that case, it would appear like this:

```
a x y =
    if x > y then 1
    else 0
```

Whitespace matters in Haskell, and indentation denotes a program block.

8.3 Convenience Settings for the Interpreter

You can specify certain options in the interpreter. All of these options are prefixed with a colon (`:`), like the `:{` and `:}` mentioned before.

Here are a few that will make it a bit easier to use the interpreter

- `:set +m`: Set multiline mode. If Haskell detects that a statement will require multiple lines to complete, Haskell wil automatically enter the multiline mode. This does not work in all cases, as you will see in the following sections.
- `:set prompt "> "`: This will change your prompt from `Prelude>` to `>`.
- :set prompt-cont "..": This will change the multiline block prompt from `Prelude|` to ..

You can make it so that all of these options are enabled in startup by typing these in to GHCI's startup script. Depending on your system, this file may be named `.ghci` or `ghci.conf`. Their locations cab be found here.

9 Functions in Haskell (Part 2)

9.1 Operators are also functions

All functions that have these following characters [1] allow you to supply 2 arguments to the function in *infix* order:

```
! # $ % & * + . / < = > ? @ \ ^ | - ~
```

Here's an example:

Example 9.1

```
Let's make an operator that finds the hypotenuse of a right angled
triangle:

    > a %% b = sqrt (a ^ 2 + b ^ 2)
    > 3 %% 4
    5.0

"sqrt", here, is a library function and already available to you.
```

We can use an operator in prefix operator by simply adding parantheses to the token:

Example 9.2

```
    > (%%) 3 4
    5.0
```

You can also define operators in this prefix manner:

Example 9.3

```
> (%%) a b = sqrt (a ^ 2 + b ^ 2)
> 3 %% 4
5.0
```

TODO acknowledge this is simply pattern matching

You can add as many characters as you want to the operator:

Example 9.4

```
> (%%%%%%%%%%%%%) a b = sqrt (a ^ 2 + b ^ 2)
> 3 %%%%%%%%%%%%% 4
5.0
```

Conversely, we can turn any other function into a prefix operator by enclosing its token in tildes (`):

Example 9.5

```
> hypotenuse a b = sqrt (a ^ 2 + b ^ 2)
> 3 `hypotenuse` 4
5.0
```

9.2 What about Operator Precedence?

There are certain operator/function tokens that are given a predefined precedence by Haskell. These include operations like `+`, `-`, `*`, `/`, `` `mod` `` etc. Other than this, since there are no differences between operators and functions aside from operators being in infix order by default, custom operators have the same precedence as normal functions.

9.3 Pattern Matching

Each function definition in Haskell is a "pattern". Definition of a function with arguments makes you define a pattern. Haskell will look for these series of tokens and arguments with appropriate type and *greedily* match the pattern specified by the function. By "Greedy," we mean that it will look for the earliest, and closest

match while performing the pattern matching. Mirroring the short-sighted nature of most greedy people looking for immediate satiation.

We utilise this concept in Haskell to match about any pattern consisting of any expression, and not just restricted to simple variables. We can even use constants in patterns. We will discuss this in the next section.

TODO Phrase this better maybe.

Example 9.6

```
Matching a list of 2 elements:
    > f [a, b] = a + b
    > f [2, 3]
    5

Matching a list of 2 elements and another variable:
    > f [a, b] c = a + b + c
    > f [1, 2] 3
    6
```

9.4 More Ways to Define Functions

9.4.1 Definitions using Pattern Matching

Aside from using the `if` block, we can also use pattern matching to define a function, and what should the function return at a particular value. This allows for a more natural-language or mathematical language manner of defining a function:

Example 9.7

```
Mathematical definition of the factorial function:
    Factorial(0) = 1
    Factorial(1) = 1
    Factorial(N) = N * Factorial(N - 1)

The factorial function in Haskell using pattern matching:

    > :{
    .. fac 0 = 1
    .. fac 1 = 1
```

```
    .. fac n = n * fac (n - 1)
    .. :}
    > fac 5
    120
```

```
Factorial in a C-like Language:

int fac(int n) {
    if (n == 0)
        return 1;
    else if (n == 1)
        return 1;
    else
        return n * fac(n - 1)
}
```

Note the use of `:{` and `:}` commands over here. These define a multiline block and is required for defining such a function written in multiple lines. Otherwise, the function definition is always overwritten each time we attempt to add a new pattern. This problem is avoided when executing from a file instead.

Unlike in conventional imperative languages, it completely avoids using an `if-else` block.

A more complex example is as follows:

Example 9.8

```
Pattern with a 2 element list, and matching a list that has the second
element as 1:

    > :{
    .. f [a, 1] = a + 999
    .. f [a, b] = a + b
    .. :}
    > f [1000, 1]
    1999
    > f [1000, 2]
    1002
```

Pattern with a 2 element list and a variable:

```
> :{
.. f [a, 1] 1 = a + 999 + 111
.. f [a, b] c = a + b + c
.. :}
> f [1000, 1] 1
2110
> f [1, 2] 3
6
```

9.4.2 `case` Syntax

The `case` syntax allows you to match a given expression against a set of patterns:

Example 9.9

On a single line:

```
> x = 5
> case x of 5 -> 7; 7 -> 9; 9 -> 11
7
```

In multiline mode:

```
> x = 5
> case x of
..     5 -> 7
..     7 -> 9
..     9 -> 11
..
7
```

Note how we no longer use `:{, :}` for a mutiline statement. this is due to the `:set +m` option mentioned in Section 8.3.

You can use this syntax to perform pattern matching in a function:

Example 9.10

```
> fac n = case n of
..      0 -> 1;
..      1 -> 1;
..      otherwise -> n * fac (n - 1);
..
> fac 5
120
```

Note the use of the `otherwise` keyword over here. `otherwise` contains the default case that will match any case that has not been already specified in the case block.

Case blocks are simply syntactic sugar and analogous to normal function pattern matching. Hence, we can rewrite the first definition in Example 9.3 as follows:

Example 9.11

```
> f x = case x of
..      [a, 1] -> a + 999
..      [a, b] -> a + b
..
```

Notice how we have utilised `a`, `b` as placeholder variables here.

9.4.3 Guard Syntax

Guard syntax allows you to define a function by checking against a set of conditions rather than equating patterns. This is a shorter method to write `if-else` statements present in imperative languages.

Within an interpreter you will have to use an explicit multiline block to define the function on multiple lines, unlike the previous one. Each conditional block is prefixed with an '`|`'

Example 9.12

`This function finds the max of two numbers:`

```
> :{
.. f a b
..      | a > b = a
..      | otherwise = b
```

```
..
.. :}
> f 2 3
3
```

You can also write the function in one line, albeit being less readable:

Example 9.13

```
> f a b | a > b = a | otherwise = b
> f 3 2
3
```

The `otherwise` keyword here was used in the same manner as in the previous case block method.

A more complex example is as follows:

Example 9.14

```
This is a 3 way comparator. It returns greater than, less than or equal
to depending on the two numbers. This is the "spaceship" operator you
may find in many languages like PHP and C++:
```

```
> :{
.. a <=> b
..     | a > b = GT
..     | a == b = EQ
..     | a < b  = LT
..
.. :}
> 2 <=> 2
EQ
> 2 <=> 3
LT
> 4 <=> 3
GT
```

`GT`, `EQ`, `LT` are predefined symbols. We will look into how to define our own such symbols later.

9.4.4 `let .. in ..` Syntax

`let` syntax allows you to define a set of local variables in a block, and use those variables in the final expression of the function.

Example 9.15

Inline method:

```
> hypotenuse x y = let sqrx = x^2; sqry = y^2; in sqrt (sqrx + sqry)
> hypotenuse 3 4
5.0
```

Non-Inline method:

```
> :{
.. hypotenuse x y = let
..     sqrx = x ^ 2
..     sqry = y ^ 2
..     in sqrt (sqrx + sqry)
.. :}
> hypotenuse 3 4
5.0
```

9.4.5 `where` Syntax

`where` is similar to `let`, but in this case we supply local variables after they are defined.

Example 9.16

Inline method:

```
> a = x + y where x = 4; y = 3;
..
> a
7
```

Non-Inline method:

```
> :{
.. a = x + y
..     where x = 4
..           y = 3
.. :}
> a
7
```

It can be used alongwith guards and other statements as well:

Example 9.17

`This is the 3 way comparator but we replace the numbers with their negative counterparts:`

```
> :{
.. a <=> b
..     | p > q = GT
..     | p < q = LT
..     | p == q = EQ
..     where p = -a
..           q = -b
.. :}
> 2 <=> 3
GT
> 3 <=> 2
LT
```

9.5 Currying

In Haskell, a function with multiple arguments can be turned into another function by supplying a partial number of arguments to it. This is called Currying. If the original function had `n` arguments, and we have supplied `k` arguments starting from the left of the argument list, then the new function will have `n - k` arguments.

For example, we have a function with 2 arguments over here and we supply one of them. In this case below, the parameter `x` of the function has been bound and a new function is made, with a single parameter `y`. This is then bound to a new identifier named `g`:

Example 9.18

```
> f x y = x + y
> g = f 2
> g 3
5
```

Here is a far larger example:

Example 9.19

```
> f u v w x y z = u + v + w + x + y + z
> g = f 1
> h = g 2
> i = h 3
> j = i 4
> k = l 5
> k = j 5
> l = k 6
> l
21

> m = f 1 2 3 4 5
> m 6
21
```

This is useful for, let's say, creating a function to divide all the elements in a list without having to create an explicit function to do so. We will look into how to do this later.

9.5.1 Currying infix operators

We can curry operators in an additional way to the prefix method by enclosing them in brackets and providing an operand on either side of the operator.

Example 9.20

```
> divideFour = (/4)
> fourDivide = (4/)
> divideFour 10
2.5
```

```
> fourDivide 10
0.4
```

Note: You may try currying the "-" operator, but keep in mind that unary "-" is also a separate operator and you will have to adjust for that while currying for the right hand side

9.6 Function Composition

Function composition in Haskell is exactly the same as mathematical function composion. If there are 2 functions, `f(x)` and `g(x)`, then the composed function `k(x) = f(x) . g(x)` is equivalent to `f(g(x))`.

The syntax uses the `.` operator and is used as follows:

Example 9.21

```
> f x = x - 3
> g x = 3 - x
> k = f . g
> k 0
0
> k = g . f
> k 0
6
```

This is useful for chaining the functionalities of multiple functions together.

9.7 Anonymous or Lambda Functions

Anonymous functions are functions without an explicit identifier. These are also found in languages like Python, Javascript and recent iterations of C++. These are primarily a functional programming construct.

Anonymous functions start with a backslash (\). It is followed by arguments and finally an arrow (->), after which the function defintion is written.

Here is an example usage of the syntax:

Example 9.22

```
> a = \x y z -> x + y + z
> a 1 2 3
6
```

Anonymous functions come in quite handy while designing programs, especially in iteration. We will see its usage in later chapters.

The type definition of the lambda function is inferred from context.

9.8 Evaluating Functions Inside Functions

We can use functions inside functions since we can pass functions around as simple variables. Here is a function that takes a function as an argument, evaluates it expecting a number and adds one to it:

Example 9.23

```
> evalAddOne func num = (func num) + 1
> evalAddOne (2*) 4
9
```

The function decomposes as follows:

```
1. evalAddOne (2*) 4
2. ((2*) 4) + 1
3. 8 + 1
4. 9
```

10 Haskell's Type System

Haskell is a statically-typed language. Types are determined right when a variable is compiled and retains that type for the rest of its runtime. This ensures that all type errors that will ever occur are detected at compile time.

GHCI is actually an *incremental bytecode compiler* as opposed to an actual interpreter. Thus whenever you declare a function, type checking is done as soon as you pass the definition to the interpreter, and any errors present are returned to you.

This is in contrast to what happens in a dynamically typed language such as Python. Functions forwarded to the interpreter are only checked for syntax errors, and then *type checking is deferred until the function is actually called*.

The following example illustrates this difference:

Example 10.1

Python Code:

```
>>> def half_add_str(x):
...     return x / 2 + "a";
...
>>> half_add_str(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in half_add_str
TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

Haskell Code:

```
> halfAddStr x = x / 2 + "a"

<interactive>:8:16: error:•
     No instance for (Fractional [Char]) arising from a use of ‘’/•
     In the first argument of ‘(+)’, namely ‘x / ’2
      In the expression: x / 2 + "a"
      In an equation for ‘’halfAddStr: halfAddStr x = x / 2 + "a"

<interactive>:8:16: error:•
     No instance for (Num [Char]) arising from a use of ‘’+•
     In the expression: x / 2 + "a"
      In an equation for ‘’halfAddStr: halfAddStr x = x / 2 + "a"
```

Note how the Haskell interpreter immediately notices the type error and in the Python interpreter the type checking only happens when we call the function.

Up till now, we have not once explicitly define the type of a given identifier and were depending upon Haskells inferred types to assign a type to the identifier. Now

we will learn how to define the type of an identifier and how does the type syntax work in Haskell.

10.1 Explicit Declation of the Type of an Identifier

All explicit type assignments are done using the `::` operator. Definition of the identifier must also be done on the same block as the type definition.

Example 10.2

Defintion on the same line:

```
> val :: Int; val = 5
> val
5
```

Definition in a block:

```
> :{
.. val :: Int
.. val = 5
.. :}
> val
5
```

"`Int`", here, is the integer type. For a full list of standard types, see here.

The example below illustrates how types are being enforced when we attempt to assign a floating point number to an integer:

Example 10.3

```
> :{
.. val :: Int
.. val = 5.123
.. :}

<interactive>:18:7: error:•
     No instance for (Fractional Int) arising from the literal ‘’5.123•
     In the expression: 5.123
      In an equation for ‘’val: val = 5.123
```

In the interpreter, you can check the type of an identifier by using the `:t` command.

```
> :t val
val :: Int
```

10.2 10.2 Defining Function Types

Identifier types are represented with syntax that resembles linked lists. These links are represented by arrows (->). Each element that is not the tail of the list represents an argument to a function, and the tail of the list represnents the final value the identifier will take on after all arguments are supplied, or in other words, the list is traversed.

Let us have a look at a function that takes 2 integers, and finally returns (or more accurately, transforms into) an integer variable:

Example 10.4

```
Defining the function:

    > :{
    .. addTwo :: Int -> Int -> Int
    .. addTwo a b = a + b
    .. :}

Seeing the type using the ':t' command:

    > :t addTwo
    addTwo :: Int -> Int -> Int

Using currying to get an intermediate function:

    > intermediateFunction = addTwo 3

Seeing the type. Note how we have "gone" to the next element of the
"linked list":

    > :t intermediateFunction
    intermediateFunction :: Int -> Int
```

```
Evaluating the intermeduate function:

    > finalValue = intermediateFunction 4

Seeing the final type. We have now "traversed" another element and have
reached the tail of the list. We now see the type of this final
identifier, which is "Int".

    > :t finalValue
    finalValue :: Int
    > finalValue
    7
```

We can also define list types by enclosing the type that the list will bear within square brackets (`[]`):

Example 10.5

```
> makeDoubleList :: Int -> [Int]; makeDoubleList k = [k, k * 2]
> makeDoubleList 2
[2,4]
```

10.3 Type Variables

We can use placeholder variables to describe a pattern which can be asigned as a type. A common example of this would be the type definitions of `head` and `tail` described previously:

```
> :t head
head :: [a] -> a
> :t tail
tail :: [a] -> [a]

"a" is the type variable here. It can be anything.
```

As you may infer, we know that since the function `head` will return what is present at the front of the list: 1. It will take a list with some element type `a` (hence the *list* type `[a]`), and then, 2. it will return a variable that has the type of the list element, which is `a`.

Similarly for `tail`, it takes in an array and returns an array of the same type. As expected.

Here is an example where we define such a function:

Example 10.6

```
> doubleList :: a -> [a]; doubleList a = [a, a]
> doubleList 3
[3,3]
```

These therefore allow us to generically refer to types.

10.4 Typeclasses

Let us try to devise a function that doubles a number by specifying an explicit type:

Example 10.7

```
This function will take an integer and "turn into" an integer, as
inferrable from the type declaration:

> doubleNumber :: Int -> Int; doubleNumber a = a * 2
> doubleNumber 2
4
```

Let us now attempt to put a floating point argument into `doubleNumber` now:

Example 10.8

```
> doubleNumber 2.2

<interactive>:31:14: error:•
     No instance for (Fractional Int) arising from the literal ‘’2.2•
     In the first argument of ‘’doubleNumber, namely ‘’2.2
      In the expression: doubleNumber 2.2
      In an equation for ‘’it: it = doubleNumber 2.2
```

As you can see, the function throws an error since we provide a floating point argument. What if we want our function to be usable with any numerical type available?

Let’s see how conventional languages solve this problem:

1. **Procedural languages (mainly C)**:
    - Procedural languages, such as C may provide a *preprocessor system* where the preprocessor shall simply copy paste a macro with the required functionality before compilation takes place. These can also be used to make rudimentary templating systems.
    - Otherwise, if the system permits pointers, pointers to void may be allowed which would allow access to any sort of data irrespective of its type. A function can thus take a pointer to the data and another parameter for the size of the data, and operate on it as it wishes. However such a method does not detail how the data has been actually stored, so operations on the data block containing a floating point variable as an integer will not work since they are stored differently. The function may then take a parameter for deducing the type of the data block.
    - Otherwise, we need to explicitly define separate functions for each type. No other way around it.
2. **Statically typed object-oriented languages**:
    - Languages such as Java and C++ can utilise *function polymorphism* to be able to operate multiple types together. However we have to explicitly define a new polymorphic function for each new type that is not necessarily a subclass or we require more functionality for a given subclass.
    - Java and C++ provide actual *templating systems*, where you can use placeholder types and variables. The internal system generates multiple instances of the same function with different types whenever they are called with such different types.
3. **Dynamically typed object-oriented languages**:

    Languages such as Python do not perform any type checking at all up until an operation is actually evaluated. Thus no errors are raised during definition of the operation. If bindings to an operator actually exist for all the operands with the given type, the operation succeeds, and if not an error is thrown.

    This may become a problem once we attempt to debug code for type errors and mismatches.

Haskell solves this problem by introducing polymorphism based on **Typeclasses**.

Typeclasses are, as the name suggests: classes of types. These are similar to *Interfaces* that you will find in C++ or Java.

A typeclass may include one or more *type instances*, and one or more generic member variables that the instances implement. For example, the `Num` typeclass includes the following types. These have been probed using GHCI's `:i` command:

```
> :i Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
    -- Defined in ‘GHC.’Num
instance Num Word -- Defined in ‘GHC.’Num
instance Num Integer -- Defined in ‘GHC.’Num
instance Num Int -- Defined in ‘GHC.’Num
instance Num Float -- Defined in ‘GHC.’Float
instance Num Double -- Defined in ‘GHC.’Float
```

The typeclass contains functions/operators like `(+)`, `(-)`, `abs` etc. which each instance type has implemented.

Any instance type of this typeclass `Num` can be used with any other instance type in the typeclass, provided that they interact only with functions that will maintain the type definition.

We will now rewrite Example 10.7 to allow interoperability with all the instance types:

Example 10.9

```
> doubleNumber :: Num a => a -> a; doubleNumber a = a * 2
> doubleNumber 4
8
> doubleNumber 4.0909
8.1818
```

Notice the new => symbol here. We first define all the typeclass instance variables we will be using (We separate each declaration with commas, although not shown here), put in the => symbol and then go on to define the function.

We will cover the functionality of type classes in a later chapter.

10.5 Functions with Functions as Arguments Revisited

Let us have a look at the type of `evalAddOne`, from Example 9.17:

```
> :t evalAddOne
evalAddOne :: Num a => (t -> a) -> t -> a
```

Let's go through the function's arguments one by one.

1. `(t -> a)`:

   The function *takes in* a function expecting an argument and will finally return a number of some sort. We can deduce from the type definition of `evalAddOne` that this argument can be of any type that the function accepts (as indicated by the type variable `t`) but must return a number.

   This function will therefore have the type definition: `Num a => (t -> a)` Where the type variable `t` is dependent on the desired function.

   In the test run we have done, we used the curried function `(*2)`. It has the following type defintition:

   ```
   > :t (*2)
   (*2) :: Num a => a -> a
   ```

   It therefore matches up with the conditions set by `evalAddOne`.

   Thus, functions as arguments will appear like this in a type defintion.

2. `t`:

   The function then takes in an argument that will be received by the argument function, and thus should match up with the type requirement of the argument function, as deductible from the type variable `t`.

3. `a`:

The function finally returns a number after all arguments have been consumed.

11 Lists (Part 2)

We will now go through a detailed description of lists and how they actually work.

11.1 "Lists" in Haskell are not Arrays

Lists are internally represented as linked lists. This therefore has a number of implications on various operations on Lists, such that random access is an O(n) operation, appending is O(1), getting the last element is O(n) and so on. All operations will have the same complexity as operating on a linked list.

These data structures come from the tradition of the Lisp family of Languages, where the basic unit of structured data are *ordered pairs*. These ordered pairs can hold, as the name suggests, two pieces of data (of course, actual contiguous arrays and other structures exist, but this is the most standard method of structural organisation in Lisp).

The `cons` function in Lisp and its dialects creates such a pair. Here is an example in scheme, a Lisp dialect which illustrates this:

```
> (cons 1 2)
(1 . 2)
```

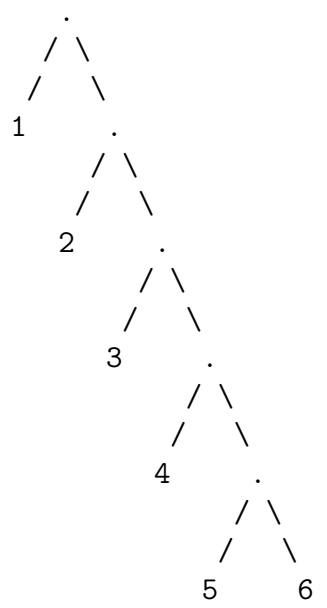
Here we bind the pair to a variable named "a":

```
> (define a (cons 1 2))
> a
(1 . 2)
```

Just as a *linked list*, we chain these ordered pairs to store multiple pieces of data:

```
> (define a (cons 1 (cons 2 (cons 3 (cons 4 (cons 5 6))))))
> a
(1 2 3 4 5 . 6)
```

A visual representation of this structure would be as follows, which resembles a linked list. Each . refering to the container element:



There are functions that are analogous to Haskell's `head` and `tail` called `car` ("current address register"), and `cdr` ("current decrement register"):

```
> (car a)
1
> (cdr a)
(2 3 4 5 . 6)
```

11.2 Cons in Haskell

Haskell also has a `cons` equivalent, an infix operator: `:`. Although uptil now we have been using the other list notation for creating these pair chains.

In the example below, we use the cons operator to extend the list:

Example 11.1

```
> k = [1, 2]
> m = 3:k
> m
[3,1,2]
```

We can also use this operator on an empty list. This allows us to construct a whole list from it.

Example 11.2

```
> k = 1:2:3:4:5:[]
> k
[1,2,3,4,5]
```

The list notation is therefore simply syntactic sugar.

11.3 Using Cons for Pattern Matching

We can use the : operator to preemptively divide a list into its head and its tail by representing the current list as a *cons* between the head and the tail of the list, and thus making it visible as two separate variables within the function.

Here we will rewrite Example 5.1 in terms of this notation:

Example 11.3

```
> :{
.. sumNumbers [] = 0 -- Sum of an empty list of numbers is 0
.. sumNumbers (x:xs) = x + sumNumbers xs -- x is the head, xs is the tail
.. :}
> sumNumbers [2,4,6,8,10]
30
```

There can be more than one elements in the cons pattern:

Example 11.4

```
> sumFive (a:b:c:d:e:[]) = a + b + c + d + e
> sumFive [1,2,3,4,5]
15
```

11.4 List Ranges

You can use the .. operator within the list notation to define a list with a certain range of numbers.

Example 11.5

We specify the starting and the ending numbers of the list.

```
> [1..10]
[1,2,3,4,5,6,7,8,9,10]
> k = [1..10]
> k
[1,2,3,4,5,6,7,8,9,10]
```

As you can see, the list elements are included in increments of 1. Even for floating point numbers, this is the case:

Example 11.6

```
> k = [1.15..10.20]
> k
[1.15,2.15,3.15,4.15,5.15,6.15,7.15,8.15,9.15,10.15]
```

Note how `10.20` is not in the list. You could say that this expression is equivalent to this for loop in a C-like language:

```
for (int i = 1.15; i <= 10.20; i += 1) {
    array.push(i)
}
```

We can specify the second element of the list to specify a custom increment value:

Example 11.7

```
> [2,4 .. 20]
[2,4,6,8,10,12,14,16,18,20]
```

We can also specify infinite lists. These may be useful in certain algorithms such as in the implementation of Eratosthenes' Sieve and other places.

Example 11.8

```
> k = [1..]
> k
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,
26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,
48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,
70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,
```

```
92,93,94,95,96,97,98,99,100,101,102...

(list does not stop printing unless interrupted by forcibly closing
the program or sending an interrupt with a key combination like
CTRL + C)
```

It is obvious to anyone that a computer will not be able to store an infinite list of numbers due to physical limitations of the computing device. Thus these numbers in ranges are generated only as it is required. This is true for non-infinite ranges as well.

11.5 List Comprehensions

List comprehensions are analogus to mathematical *set comprehensions*.

For example, the set of all odd numbers between 0 and 100 may be represented in Mathematical notation as follows:

oddSet = *{ x | x ∈ ℕ, 1 ≤ x ≤ 100, x is odd }*

This comprehension may be expressed in Haskell in a very similar manner:

Example 11.9

Haskell has a very similar notation:

```
> oddSet = [ x | x <- [1..100], x `mod` 2 /= 0 ]
> oddSet
[1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41,43,45,47,
49,51,53,55,57,59,61,63,65,67,69,71,73,75,77,79,81,83,85,87,89,91,
93,95,97,99]
```

"<-", here is the "Element of" operator. "/=" is the "not equal to" operator.

Instead of `x`, we can write any function expression that uses `x` as a parameter. We can also use multiple variables in comprehansions:

Example 11.10

```
> multiplicationTable = [(show(x)) ++ " x " ++
                         (show(y)) ++ " = " ++ (show(x*y))
                         | x <- [2..10], y <- [2..10] ]
```

```
> multiplicationTable
["2 x 2 = 4","2 x 3 = 6","2 x 4 = 8","2 x 5 = 10","2 x 6 = 12",
"2 x 7 = 14","2 x 8 = 16","2 x 9 = 18","2 x 10 = 20","3 x 2 = 6",
"3 x 3 = 9","3 x 4 = 12","3 x 5 (...)
```

Note that in order to split the list comprehension across multiple lines, the square bracket must be ended at the same or greater level of indentation as the rest of the block,

Here is a larger example. This program implements the popular "fizz-buzz" problem:

Example 11.11

```
> fizzBuzz = [
..     if (x `mod` 3) == 0 && (x `mod` 5) == 0 then "FizzBuzz"
..     else if (x `mod` 3) == 0 then "Fizz"
..     else "Buzz"
..     | x <- [1..100],
..       ((x `mod` 3) == 0) || ((x `mod` 5) == 0) ]
> fizzBuzz
["Fizz","Buzz","Fizz","Fizz","Buzz","Fizz","FizzBuzz","Fizz","Buzz",
"Fizz","Fizz","Buzz","Fizz","FizzBuzz","Fizz","Buzz","Fizz","Fizz",
"Buzz","Fizz","FizzBuzz","Fizz","Buzz","Fizz","Fizz","Buzz","Fizz",
"FizzBuzz","Fizz","Buzz","Fizz","Fizz","Buzz","Fizz","FizzBuzz",
"Fizz","Buzz","Fizz","Fizz","Buzz","Fizz","FizzBuzz","Fizz","Buzz",
"Fizz","Fizz","Buzz"]
```

We will learn a method to print all of these strings later.

12 Tuples

Tuples are arrays of elements that are immutable, and cannot be changed in size. These can be useful to store objects such as database rows, coordinates, vectors etc.

```
> t = (1, 2)
> t
(1,2)
```

```
> :t t
t :: (Num a, Num b) => (a, b)
```

As you can see, tuple types are represented using rounded brackets.

Here is an example of a list which stores tuples of fruit names and their prices:

Example 12.1

```
> fruitList :: [([Char], Float)]; fruitList = [("apple", 12.3),
("banana", 4.2)]
> fruitList
[("apple",12.3),("banana",4.2)]
```

Now, attempting to insert an illegal element:

```
> newList = (132, 123) : fruitList

<interactive>:75:12: error:•
     No instance for (Num [Char]) arising from the literal ‘’132•
     In the expression: 132
      In the first argument of ‘(:)’, namely ‘(132, 123)’
      In the expression: (132, 123) : fruitList
```

12.1 `fst` and `snd`

Haskell by default includes functions for accessing the first and second elements of a 2 element tuple:

```
> k = (4, 5)
> fst k
4
> snd k
5
```

If we want more custom functions for accessing data from n-tuples, we can define them as follows. Since tuples contain fixed size, immutable, heterogenous data, there is no way to “traverse” them.

```
> tuple3Fst (a, b, c) = a
```

```
> tuple3Snd (a, b, c) = b
> tuple3Thrd (a, b, c) = c
> tuple3Thrd (3, 4, 1)
1
```

13 Iteration in Haskell

We have several utility functions that act on lists. There are a lot of familliar functions that you will notice here if you have used languages such as Python.

13.1 The `iterate` Function

`iterate` is similar to an infinite list comprehension, but one can use any desired iteration function to generate successive numbers in the list.

Example 13.1

```
> q = (iterate (1+) 1)
> q
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,...

> q = (iterate (2*) 1)
>
> q
[1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768,
65536,131072,...
```

13.2 The `map` function

The `map` function applies a function to all the elements on a list.

In this example we increment all numbers present in a list:

Example 13.2

```
> map (1+) [1,2,3,4,5]
[2,3,4,5,6]
```

The map function can also be used to create new curried functions which then can be later applied:

Example 13.3

```
> addCurryList = map (+) [1..20]
> (head addCurryList) 10
11
```

13.3 The `filter` Function

The filter function, as the name may suggest, filters things from a list. It takes a function that returns a truth value and operates it on each element of a list. If the function returns true for an element, then that element is accepted into the new list that `filter` will return.

Example 13.4

This statement will get odd numbers between 1 and 20:

```
> filter (\x -> (x `mod` 2) == 0) [1..20]
[2,4,6,8,10,12,14,16,18,20]
```

Note how we were able to avoid assigning a new identifier to a function with the same functionality by using a lambda function instead.

13.4 The Function Application Operator (`$`)

This operator takes in a function argument on the left hand side and an argument for the function argument on the right hand side, then evaluates the argument function and returns the result of the evaluation.

Let us have a look at the type of this function:

```
> :t ($)
($) :: (a -> b) -> a -> b
```

You may see that this is more or less equivalent to the type definition of the function `evalAddOne` presented in Example 9.17 (and discussed further in section 10.5), only that no type classes are used here, thus making the function entirely generic.

Let us see an example usage of this:

Example 13.5

```
> sqrt $ 16
4.0
> ($) sqrt 16
4.0
```

This function comes in very handy when we attempt to use `map` on all the functions created in the list in example 13.2. It prevents us entirely from defining an explicit lambda function to evaluate it.

Example 13.6

```
Evaluating list without the $ operator:

    > map (\func -> func 100) addCurryList
    [101,102,103,104,105,106,107,108,109,110,111,112,113,114,115,116,
    117,118,119,120]

Now evaluating the list with the $ operator:

    > map ($ 100) addCurryList
    [101,102,103,104,105,106,107,108,109,110,111,112,113,114,115,116,
    117,118,119,120]
```

We can also use it to evaluate a function on a list of operands by putting the function on the left hand side:

Example 13.7

```
> map (sqrt $) [1..20]
[1.0,1.4142135623730951,1.7320508075688772,2.0,2.23606797749979,
2.449489742783178,2.6457513110645907,2.8284271247461903,3.0,
3.1622776601683795,3.3166247903554,3.4641016151377544,
3.605551275463989,3.7416573867739413,3.872983346207417,4.0,
4.123105625617661,4.242640687119285,4.358898943540674,
4.47213595499958]
```

13.5 The `zip` Function

This function takes two lists, and combines each pair of their elements them into tuples in successive order:

Example 13.8

```
> zip [1, 2, 3] [4, 5, 6]
[(1,4),(2,5),(3,6)]

> zip [1, 2] [1, 2, 3, 4]
[(1,1),(2,2)]
```

13.6 The `zipWith` Function

This function takes two lists, and a function that will operate on each successive pair of elements, and generate a new list from all the values returned from this function:

Example 13.9

```
> zipWith (/) [1, 2, 3] [4, 5, 6]
[0.25,0.4,0.5]
```

13.7 The `fold` and `scan` Functions

The `fold` functions are what you use to update the state or keep "persistent variables" while performing iteration. Essentially, this is your `for` loop in Haskell.

13.7.1 `foldl`: Fold from Left to Right

`foldl` has the following arguments:

```
foldl func accumulator list
```

The argument `func` has the following arguments:

```
func accumulator currentVariable
```

- `accumulator` is the persistent variable. It has the initial value `initialValue`.
- `currentVariable` is the current list element we are on.

- It will return the new value of the accumulator

`foldl` recursively operates in the following manner:

```
foldl func accumulator [] = accumulator
foldl func accumulator list = foldl func (func accumulator (head list))
                              (tail list)
```

- If the list is empty, the accumulator value will be the initial value. This is the base case.
- Else, the new accumulator value will be the return value of func. We will supply it with the initial accumulator value, and the current index value of the list.

Thus, we are operating the list in a "folding" manner.

Let us have a look at an example. This expression generates a number with the elements as the digits in the list:

Example 13.10

```
> foldl (\x y -> x*10 + y) 5 [4, 3, 2, 1]
54321
```

The traceback of this function will be as follows;

```
1. foldl func 5     [4, 3, 2, 1]
2. foldl func 54    [3, 2, 1]
3. foldl func 543   [2, 1]
4. foldl func 5432  [1]
5, foldl func 54321 []
6. 54321
```

13.7.2 `foldr`, Fold from Right to Left

`foldr` is implemented like so:

```
> :{
..     foldr func i [] = i
..     foldr func i (x:xs) = func x (foldr func i xs)
.. :}
```

This time, as you might be able to notice, the current variable and accumulator variables are swapped.

```
func currentVariable accumulator
```

The rationale for such an arrangement comes from the fact that we are “folding” the list leftwards, and the “folds” “accumulate” on the right or the left side of the list. Thus allowing for a mnemonic representation of the accumulation.

Example 13.11

```
> foldr (-) 5 [1, 2, 3, 4]
3
```

If we were to expand this whole operation into a single expression. It would look like this:

```
(5 - (4 - (3 - (2 - 1))))
```

13.7.3 `scanl` and `scanr`

These two functions are same as the `fold` functions, but instead of simply returning the final accumulator value at the end, the whole list of each successive accumulator values are returned.

Example 13.12

```
> scanl (\x y -> x*10 + y) 5 [4, 3, 2, 1]
[5,54,543,5432,54321]
>  scanr (-) 5 [1, 2, 3, 4]
[3,-2,4,-1,5]
```

13.7.4 Note: Pattern Matching in these Functions

One can simply set the initial accumulator value and then update it through the updation function. This allows us to update multiple values in each iteration step.

Here is an example of how we would write a function that would calculate the average of a series of numbers:

Example 13.13

```
> :{
.. average l = (fst k) / (snd k)
..     where k = foldl (\(s, c) y -> (s + y, c + 1)) (0, 0) l
.. :}
> average [1, 2, 3, 4, 5]
3.0
```

13.7.5 `foldl1`, `foldr1`, `scanl1` and `scanr1`

These functions suffixed with 1 do not require the initial accumulator variable as an argument. The accumulator variable is therefore of the type of the initial variable of the list:

Example 13.14

```
> foldl1 (\x y -> x + y) [1, 2, 3, 4, 5]
15
```

14 Data, Records, Types, and Typeclasses

14.1 Data

"`data`" definitions are a method of making a new set of symbols that act as values for variables. These are quite similar to enumerations.

In this example we define a data type enumerating the several stages of human life and make a variable with that data type:

Example 14.1

```
> :{
..     data HumanStage = Baby | Toddler | Kid
..         | Teenager | YoungAdult | MiddleAgedAdult | Old | Senile
.. :}
> a = Senile
> :t a
a :: HumanStage
```

14.1.1 Printing New Data Types

On attempting to simply pass the variable to the interpreter, which as we have seen before simply evaluates the expression and attempts to print it we get the following error:

```
> a

<interactive>:11:1: error:•
     No instance for (Show HumanStage) arising from a use of ‘’print•
     In a stmt of an interactive GHCi command: print it
```

This is because of how GHCI attempts to print the datatype after the successful evaluation. It invokes the `print` function which in turn invokes `show` to convert it into a string.

In order to tell Haskell to use a verbatim string representation of the data type symbols, we have to make it derive from the `Show` typeclass:

Example 14.2

```
> :{
.. data HumanStage =
..     Baby | Toddler | Kid | Teenager | YoungAdult |
..     MiddleAgedAdult | Old | Senile
..     deriving (Show)
.. :}
> print Baby
Baby
> a = Teenager
> print a
Teenager
> putStrLn $ (show Teenager) ++ " " ++ (show Baby)
Teenager Baby
```

14.1.2 Tuple-Like Data

We can add parameters to each of our symbols in our data types to make them hold data.

Example 14.3

```
> :{
.. data Coordinates =
..     TwoDim Float Float |
..     ThreeDim Float Float Float
.. :}
> (TwoDim 4 5)
TwoDim 4.0 5.0
> (ThreeDim 4 5 6)
ThreeDim 4.0 5.0 6.0
> a = (TwoDim 6 7)
> a
(TwoDim 6.0 7.0)
```

`TwoDim` and `ThreeDim` here are called Data Constructors.

We can pattern match such tuples in functions like this:

```
> twoDimSquare (TwoDim a b) = (TwoDim (a * 2) (b * 2))
> twoDimSquare (TwoDim 4 5)
TwoDim 8.0 10.0
```

In fact, these data constructors are functions and can be used as such. The bottom value of the function will be the data "tuple".

14.1.3 Type Variables in Data Constructors

We can add arguments beside our data type identifier to add in type variables, hence allow us to generalise the data type:

Example 14.4

```
> data WeightedValue a = IntWeightedValue a Int | FloatWeightedValue a Float
   deriving (Show)
> q :: WeightedValue [Char]; q = (IntWeightedValue "Apple" 5)
> q
IntWeightedValue "Apple" 5
```

14.1.4 Recursive Data Constructors

You can define recursive types, much like the way one would use self referrential struct pointers in C/C++. This allows you to define data structures like trees:

Example 14.5

```
> :{
.. data TreeNode =
..       Node Integer TreeNode TreeNode
..     | BottomNode deriving (Show)
.. treeleft (Node v l r) = l
.. treeright (Node v l r) = r
.. treeval (Node v l r) = v
.. :}
> a = (Node 1
..          (BottomNode)
..          (Node 3
..              (Node 4
..                  (BottomNode)
..                  (BottomNode))
..              (BottomNode)))
> a
Node 1 BottomNode (Node 3 (Node 4 BottomNode BottomNode) BottomNode)
> treeright a
Node 3 (Node 4 BottomNode BottomNode) BottomNode
> treeleft $ treeright $ a
Node 4 BottomNode BottomNode
> treeval $ treeleft $ treeright $ a
4
```

14.1.5 Records

Records are like the previously mentioned Tuple-Like Data Types, except each member of the tuple has an explicit identifier. Quite a lot like a struct in C or class member access in object oriented languages.

Here we reimplement our previous tree definition using records:

Example 14.6

```
data Tree a =
      Node {
          value :: a,
          left  :: Tree a,
```

```
      right :: Tree a
  }
| BottomNode;
```

We can define a variable record as follows:

```
> a :: Tree Int; a = Node {value = 5, left = BottomNode, right = BottomNode}
```

We can pattern-match records in functions as follows:

```
> treeleft (Node {value = v, left = l, right = r}) = l
> treeleft a
BottomNode
```

It is not necessary that we match all the variables all the time for a function. Matching only those which we require in the function is enough:

```
> treeleft (Node {left = l}) = l
> treeleft a
BottomNode
```
