Adventures in Computation

Hashan Punchihewa

"And some people prefer not to commingle the functional, lambda-calculus part of a language with the parts that do side effects. It seems they believe in the separation of Church and state." - Guy Steele

Contents

1 Preliminaries **1**
1.1 Signing Up 1
1.2 Navigating the File System 1
1.3 Editing and Programming 2
1.4 Basics 2
1.5 Arithmetic 3
1.6 Booleans 4

2 Haskell **6**
2.1 Lists 6
2.2 Pattern matching 6
2.3 Recursion 7
2.4 Typing 9
2.5 Type Classes 10
2.6 Guards 11

3 Intermediate Haskell: Part I **13**
3.1 Maps and Filters 13
3.2 Where Clauses 13
3.3 Characters and Strings 14
3.4 Functions on Characters 15
3.5 Practicum A: Caesar's Cipher 15
3.6 Folds 17
3.7 Partial Application 17
3.8 Pairs and Tuples 19
3.9 Infinite Lists 21
3.10 Practicum B: Polynomials 21

Algorithms Society 2016

1 Preliminaries

In this section, we look at the **Linux operating system**. Although, we will mostly cover programming and theoretical computer science, the **BASH shell** and Linux operating system are widely used among programmers, we will use them extensively. When using Windows or macOS, most users will interact with their computer using, what is known as a Graphical User Interface (GUI pronounced "gooey"). Although, most desktop Linux operating systems have GUIs as well, before the advent of GUIs, there existed CLIs (command-line interfaces). These are what you see on TV, when somebody is hacking a computer system. They type a command, click enter, and stuff happens. The thing they type text into is called the shell, and different shells accept slightly different commands. We will be using the BASH shell, which has become ubiquitous.

1.1 Signing Up

Go onto *asociety.xyz*, and login. The default password is "defaultpassword". To change your password type the following command:

```
passwd
```

When it asks you to type your password, it will not show the characters as you type.

1.2 Navigating the File System

To create a new folder, use the command:

```
mkdir folderName
```

To change directory:

```
cd folderName
```

To create a new file:

```
touch fileName
```

To see all the files in a folder:

```
ls
```

To go "up" a folder:

```
cd ..
```

The character `~` refers to your, home folder, so typing `cd ~`, will send you back to your home folder. `..` as you've seen above, refers to the folder, one above the folder you are currently in. `.` refers to your current folder. To remove a file, there are the commands `rm filename` and to remove a folder `rmdir foldername`. However, that will only remove an empty folder. To forcibly remove a non-empty folder, you can use `rm -rf foldername`, but be careful this is permanent.

1.3 Editing and Programming

Later we will cover, the Vim text editor. However, Vim has a steep learning curve, so for the moment we will use Nano. Nano is very easy-to-use text editor. We will be learning the **Haskell programming language**. Languages such as Java, Python, Pascal, C++ and VB.NET are known as imperative languages. These are the most prevalent. Another class of languages exists called functional languages. Haskell is one of them.

To open a file `index.txt` in nano, use the command:

```
nano index.txt
```

To save files, `Ctrl-w` is used and to exit nano, type `Ctrl-x`. We are going to split our terminal in half. To do this run:

```
tmux
```

Afterwards, press `Ctrl-b %`. This will split your terminal vertically. Then, use `Ctrl-b o` to switch, back and forth between your splits. In your first split, open a new file vim called `asociety.hs` in Nano. In your second, open the Haskell interpreter, using the command `ghci`. In the interpreter, you will be able to test your code.

1.4 Basics

In nano, write the following code:

```haskell
x = 2
y = 3
z = 4

square x = x * x
cube x   = x * x * x
mul x y = x * y
```

Here you've defined 3 variables, and 3 functions. Then in the `ghci` interpreter, type the following command to load the code you have written:

```
:l asociety.hs
```

This loads all the variables and functions you've defined in your file into the interpreter. Then you can test them in the interpreter, by typing:

```haskell
x
y
z
square 4
cube 3
mul 54 23
```

In Haskell, all variables and functions are immutable. This means they can't be changed. You might be thinking, how can we even write non-trivial programs without changing variables? However, immutability is very important. It is important to the concept of **referential transparency**. In Haskell, functions are referentially transparent. This means that if you call a function twice with the same input, it will always give the same output. In imperative languages, functions can have side effects, e.g. changing the value of a global variable. In functional programming side effects are eliminated. Here are examples in Python and Java of functions that have side effects, and are not referentially transparent:

```java
public static int globalVariable = 5;

public static int badFunction(int x) {
        globalVariable = globalVariable + 1; // Causes a side effect
        return x + globalVariable; // Not referentially transparent
        // This is because the result depends on a mutable variable
}
```

```python
globalVariable = 5

def badFunction(x):
        globalVariable = globalVariable + 1 # Causes a side effect
        return x + globalVariable # Not referentially transparent
        # This is because the result depends on a mutable variable
```

The concept of referential transparency comes from Mathematics. In Mathematics, functions e.g. sin and cos are referentially transparent. When you write $\sin 30$, this doesn't suddenly change a global variable, so that the next time you calculate $\sin 30$ you get a different result.

1.5 Arithmetic

Try these statements in the interpreter:

```haskell
1 + 3
1 - 2
4 * 3
4 / 3
3 + 4 * 2
(3 + 4) * 2
```

As you can see, arithmetic works much the same as in most programming languages. However, Haskell provides some additional functions such as `div` and `mod`:

```haskell
mod 12 5
div 4 3
even 3
even 4
odd 3
odd 4
max 4 5
min 4 5
min 23 (max 20 13)
```

- `mod x y` returns the remainder of `x` divided by `y`. It is similar to `%` in most languages.
- `div x y` returns the `x` divided by `y` without the remainder. This is also called the quotient.
- Notice that in the last expression, the second function call had to be put in parentheses. If we had written `min 23 max 20 13`, Haskell would read it as `min (23) (max) (20) (13)` causing an error.

1.6 Booleans

Booleans are values that are either true or false. In Haskell true is represented by `True`, and false by `False`. Try this out in the interpreter:

```
True
False
4 == 5
3 == 3
4 < 5
4 < 4
4 <= 5
4 <= 4
5 > 3 + 3
5 /= 6
True || False
True || True
False || False
True && False
True && True
False && False
5 == 5 || 6 == 5
6 == 5 || 5 == 5
5 == 5 && 6 == 5
4 > 5 || 5 > 6
```

`==`	Is equal to
`<`	Is less than
`<=`	Is less than or equal to
`>`	Is greater than
`>=`	Is greater than or equal to
`\=`	Doesn't equal
`||`	Or
`&&`	And

Most of the Boolean operators should be self-explanatory. We will look at the AND and OR operators. `||` takes two Boolean values and returns true if either is true. `&&` takes two Boolean values and returns true if both are true.

x	y	$x \,\|\|\, y$
$False$	$False$	$False$
$False$	$True$	$True$
$True$	$False$	$True$
$True$	$True$	$True$

Table 1: OR ($x \,||\, y$)

x	y	$x \,\&\&\, y$
$False$	$False$	$False$
$False$	$True$	$False$
$True$	$False$	$False$
$True$	$True$	$True$

Table 2: AND ($x \,\&\&\, y$)

Exercises

Write these in the `asociety.hs` file, and then load them into the interpreter with `:l asociety.hs`.

1. Write a function `equalFive` that takes a parameter x and returns true iff (if and only if) $x = 5$.
2. Write your own version of `even`, `even'` which takes a parameter x and returns true iff x is even.
3. Write your own version of `odd` called `odd'`.
4. Write a function called `divisibleByThreeFive` that takes a parameter x and returns true iff x is divisible by either 3 or 5.
5. Write a function called `maxDivisibleByFive` that takes parameters x and y, and returns true iff the biggest of the two is divisible by 5.
6. Write a function called `maxThree` that takes three parameters and returns the biggest of the three.

2 Haskell

2.1 Lists

In Haskell, a list is created using square brackets, with commas to separate items:

```
[]
[1,2,3,4]
[1..10]
[1,3..10]
[10,9..1]
```

As you can see, you can create lists from one number to another, using `..` notation. You can also specify the "step", that is the amount the number goes up or down by. Here are some things you can do with lists. Make sure to type the following into the interpreter:

```
1:[2,3,4] -- Prepend to a list
1:2:[3,4]
1:2:3:4:[]
head [1..5] -- First item
tail [1..5] -- Everything except the first item
null [1,2,3] -- Is the list empty?
null []
take 3 [1..10] -- Take the first 3 items
drop 3 [1..10] -- Drop the first 3 items
```

Exercises

1. Write a function `hasOneItem` that takes a list as a parameter uses `null` and `tail` to return true if the list has only 1 item.

2. Write a function `generateList` takes one parameter n and returns a list containing all the items between 1 and n.

3. Write a function `firstItemEven` that takes a list as a parameter and returns true if its first item is even.

4. Write another function `firstItemOdd` that takes a list as a parameter and returns true if its first item is odd.

5. Write another function `prepend4` that takes list as a parameter and returns the list with 4 prepended.

2.2 Pattern matching

Sometimes it is useful for a function to do different things based on its parameters. Using pattern matching, you can do different things based on the number of items, the lists you receive as parameters have. Here's an example of writing our own `head` function:

```
head' [] = error "List has no items"
head' (x:xs) = x
```

Here we've written our own `head` function. There are however two functions with the same name. Haskell, first, checks whether the arguments we've supplied match the first function, if not checks the second function. In the first function, we've specified an exact value, an empty list. Obviously, you can't return the first item of an empty list, so the `error` function is used to cause an error. The second function uses the prepend operator. Remember, the prepend operator takes two arguments: an item; and a list. The second function definition will accept any lists that can be written in that form, so a list of 1 item or more. E.g. `[1]` can be rewritten as `1:[]`, `[1,2]` can be rewritten as `1:[2]`. The first item will then be accessible under `x` and the rest of the list (the tail) will be accessible as `xs`.

Exercises

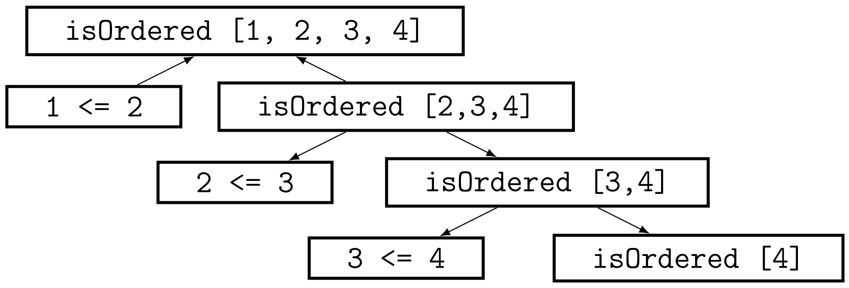
1. Write your own version of `tail` called `tail'`.
2. Write your own version of `null` called `null'`.

2.3 Recursion

Sometimes to solve problems, the problem must be broken down into smaller sub-problems. Here is an example of a function that checks whether a list is ordered:

```
isOrdered [] = True
isOrdered (x:[]) = True
isOrdered (x:y:xs) = x <= y && isOrdered y:xs
```

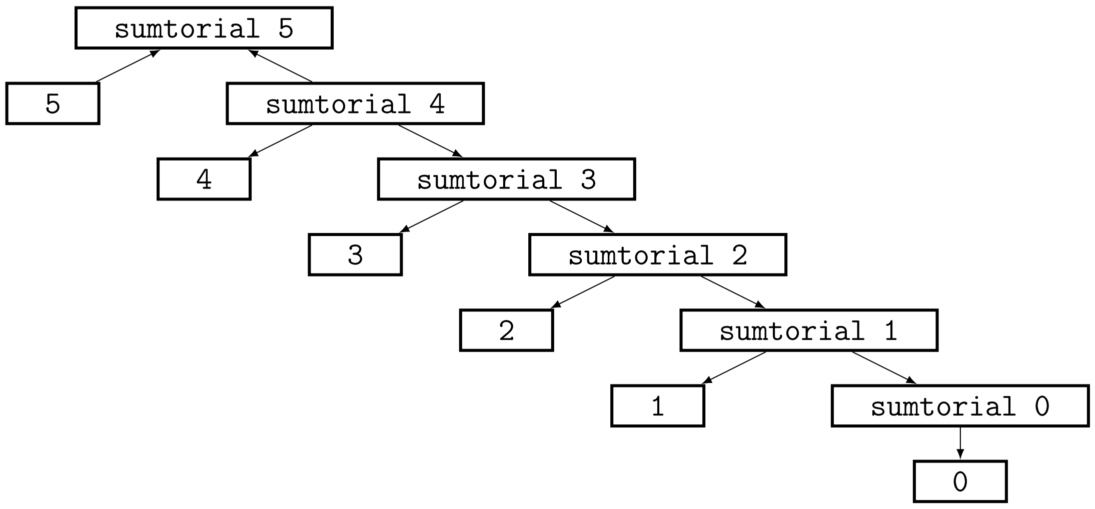
Can you see what it does? First, it states an empty list and a list containing only 1 item is ordered. This obviously has to be true. It then says, a list of 2 or more items is ordered, if the first item is smaller than or equal to the second item, and the rest of the list including the second item is sorted. Here is how the function would break down the list `[1,2,3]`:



Note that without `isOrdered (x:[])` being defined, `isOrdered` would fail here. Why? Because eventually for any non-empty list, the function will be called with a list of only 1 item and without `isOrdered (x:[])`, Haskell would be unable to find a definition of `isOrdered` whose pattern matches a list of 1 item.

The process of a function calling itself, is called **recursion** and is essential to functional programming. In functional programming there are no loops: no for-loops, no while-loops, no do-loops. The only way of performing code repeatedly is recursively calling a function. But a base case must be defined! Here is an example of recursion with numbers:[1]

```
sumtorial 0 = 0
sumtorial n = n + sumtorial (n-1)
```



Exercises

1. Describe concisely what the above function does. What is the base case? To help do it, write the function, and test it with simple values. Execute the function on paper. For instance `sumtorial 2` will become `2 + sumtorial 1` which will become `2 + 1 + sumtorial 0` which will become...

2. Write a similar function `factorial n` that calculates the product of all the numbers between 1 and n.

3. Write a function, `length'` that takes a list as a parameter and returns its length.

4. Write a function, `sum'` that takes a list as a parameter and returns the sum of all its items.

5. Write a function `product'` that takes a list as a parameter and returns the product of all its items.

6. Write a function `maximum'` that takes a list of numbers as a parameter and returns the biggest number.

7. Write a function `minimum'` that takes a list of numbers as a parameter and returns the smallest number.

[1]`sumtorial` is from Penn's CIS194 Spring 13, lecture notes.

2.4 Typing

Some languages such as Python, do not check for type errors until a program is run e.g. that you have not tried passing an integer to a function that expects a string. These languages are type unsafe. Languages such as Java require types to make sense before you run your program. The problem with Java is that you have to specify what type everything is, e.g:

```
int a = 4;
String name = "Bob";

public static int multiply(int x, int y) {
        return x * y;
}
```

In Java, we have to specify the types of variables, parameters and the return values of functions. Uncool! Haskell is also a type safe language, but unlike Java we don't have to specify these things. That's before Haskell supports type inference. If, for example, you have written:

```
x = 5

triple x = x * 3
```

Haskell will figure out that x is obviously an integer, because you have assigned it to one. Similarly, Haskell will infer `triple` takes a numeric data type as a parameter, as it performs multiplication on it. It will also figure out that you are returning a numeric data type. Note that all the items of a list have to be of the same type, i.e. a list is homogenous. A Boolean list is treated as a different data type from an integer list.

However, Haskell, forever full of great ideas, allows something called polymorphism. This means you can create a functions that work on multiple similar types, with only one declaration. Here's an example:

```
head' (x:xs) = x
```

The above function is also polymorphic, it will work on lists of any types, including lists of lists. Haskell infers that the type of the list doesn't matter here.

Normally, we can rely on Haskell's type inference powers to avoid having to write types. However, it is good practice to write type definitions, for all top-level functions (all the functions you've seen so far are top-level). This is makes what we expect our functions to do explicit. It means Haskell is more likely to infer types correctly, and that our code is easier to debug. Here is how we would write a type class for `double`.

```
double :: Int -> Int
double x = x * 2
```

Note that type definitions are usually written above functions. The double colon indicates that you are making a type definition. The type definition above says that the function `double` expects an integer and returns an integer. Here is an example for a function with multiple parameters:

```haskell
mul :: Int -> Int -> Int
mul x y = x * y
```

The type definition here means expecting two integers and returning an integer. The last in the sequence of `->` is the return value.

Here is an example using lists:

```haskell
sum' :: [Int] -> [Int]
sum' [] = 0
sum' (x:xs) = x + sum' xs
```

Here we see how to denote an integer array. To denote polymorphism with lists you use any name beginning with a lowercase letter e.g.:

```haskell
head' :: [a] -> a
head' (x:xs) = x
```

The above definition means that `head` expects a list of some type *a* and returns a value of the same type *a*. One-letter names are usually chosen for convenience. Because the names of all actual types begin with an uppercase letter, lowercase letter must always begin the name used with polymorphism.

Exercises

From now on always write type definitions for functions.

1. Write a function `last'` that takes a list and returns the final item.
2. Write a function `and'` that takes a Boolean array and returns true if all the array's items are true. In Haskell the Boolean type is `Bool`.
3. Write a function `all'` that takes a Boolean array and returns true if at least one of the array's items is true.
4. Write a function `interadd'` that takes an array and an item and adds the item after every item, for instance `interadd 10 [1,2,3]` would return `[1,10,2,10,3,10]`.
5. Write a function `reverse'` that takes an array and reverses it. To do this, you will need to know how to concatenate lists. If you have two lists, `[1..3]` and `[4..6` to concatenate them, you must use the `++` operator, i.e. `[1..3] ++ [4..6]`.

2.5 Type Classes

In Haskell sometimes, you will want to create a polymorphic function, but you will want the polymorphic type to have certain capabilities. For example, the only numeric type we have looked at is the `Int` type, however in Haskell there are lots of other numeric types: `Integer`, `Float`, `Double`, etc. All of these come under one category, called a typeclass, known as

Num. Being members of Num, means that they implements certain functions, namely those for addition, subtraction, multiplication and division. The Haskell interpreter allows you to see the types of functions and variables. Let's see the type of the sum' function we created a while back, by typing:

```
:t sum'
```

You should see:

```
sum' :: (Num a) => [a] -> a
```

Whereas you probably expected something like:

```
sum' :: [Int] -> Int
```

The [a] -> a looks very much like a polymorphic type, except for the (Num a) => bit. This bit is called a type constraint. It says that the polymorphic type a must be a member of the Num typeclass. You could of course, have written an explicit type definition making it only work with integers, but Haskell's type inference chooses the most flexible possible type definition as possible by default. Let's look at the type of our equalsFive function:

```
:t equalsFive
equalsFive :: (Num a, Eq a) => a -> Bool
```

Here we have two type constraints on a. It must be a number, but also must be part of the Eq typeclass, meaning you can use == and /= on it. What about if we had a function say:

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

It's type definition would be:

```
:t greaterThanFive
greaterThanFive :: (Num a, Ord a) => a -> Bool
```

The Ord typeclass means that you can use comparison functions, such as >. So far Num, Eq and Ord are the only typeclasses you need to be aware of.

2.6 Guards

Guards are similar to pattern matching, but instead of patterns, Boolean conditions are used. For example here is a custom implementation of the max function:

```
max' x y
  | x < y = y
  | otherwise = x
```

You can use guards with pattern matching as well, here is the maximum function again, that returns the biggest item in a list.

```
maximum'' (x:[]) = x
maximum'' (x:y:xs)
  | x < y = maximum'' y:xs
  | otherwise = maximum'' x:xs
```

Haskell goes through each Boolean condition in turn, until it finds one that evaluates to true. otherwise just equals true, so it acts as a catch-all case.

Exercises

Remember to continue writing type definitions for functions, using typeclasses such as `Num`, `Eq` and `Ord`, where possible.

1. Write a function `status` that takes a number and returns 0 iff the number is divisible by 2 and 3, 1 iff the number is divisible by 2 and 2 iff the number is divisible by 3.
2. Write a function `lessThanForty` that takes two numbers and return true iff the sum of the two numbers is less than 40. (You shouldn't need to use guards here.)
3. Write a version of `min` called `min'` that uses guards.
4. Write `minimum''` that takes a list and returns the smallest item in a list uses guards and pattern matching.
5. Write `elem'` that takes two parameters an item and a list and returns true iff the item is in the list using guards and pattern matching.
6. Write `merge` that takes two sorted lists and returns a new list containing the items from both of them, that is also sorted.

3 Intermediate Haskell: Part I

We've looked at the very basics of Haskell: referential transparency, recursion and types. In this section, we will dive further, looking at the bread and butter of Haskell programming: maps, filters and folds; partial function application; and tuples.

3.1 Maps and Filters

Recursion is powerful, but usually we can do better with the `map` and `filter` functions. Here is an example of the `map` function.

```haskell
double :: (Num a) => a -> a
double x = x * 2

doubleList :: (Num a) => [a] -> [a]
doubleList xs = map double xs
```

`map` is a higher-order function, because it takes a function as a parameter, in this case `double`. `map` takes a function and a list and applies the function to every element of the list. It then returns this new list. Here is an example of `filter`:

```haskell
filterEvens :: (Num a) => [a] -> [a]
filterEvens xs = filter even xs
```

`filter` also is a higher-order function. It takes a function and a list as a parameter. The function takes in an element of the list and returns true or false. If it returns true, then filter keeps the element, otherwise it discards it.

3.2 Where Clauses

In order to keep our code tidy, we can create functions and variables that are only accessible from a single function/variable we could rewrite our first example to:

```haskell
doubleList :: (Num a) => [a] -> [a]
doubleList xs = map double xs
  where double x = x * 2
```

Here's a more advanced example:

```haskell
bob = sum (filter divisible [1..1000])
  where divisible = mod n 3 == 0 || mod n 5 == 0
```

The `bob` isn't a function, it is a variable. It is the sum of all the numbers between 1 and 1000 divisible by 3 or 5.

Exercises

1. Write a function `free5` that takes a list and returns true iff it doesn't contains a multiple of 5.
2. Write a function `add5` that takes a list and returns a new list with 5 added to every element.
3. Write a function `remove6andAdd4` that takes a list, removes and 6s and adds 4 to every item.
4. Write a function `length''` that takes a list and maps every item with 1, before using `sum` to get its length.
5. Write a function `divisors` that takes a number and returns all its divisors. You will need to use arithmetic series (`[1..5]`) notation.
6. Write a function `isPrime` that takes a number and returns true iff it is a prime number using map and filter.
7. Write a function `isPerfect` that takes a number and returns true iff the number is a perfect number. A perfect number is a number that equals the sum of its proper divisors. A proper divisor is any divisor of a number, that isn't the number itself.
8. Use recursion to write your own version of the `map` function, called `map'`. It should have the following type definition:

   ```
   map' :: (a -> b) -> [a] -> [b]
   ```

   In other words, it takes two parameters, a function that turns a to b, and a list of a. It then returns a list of b.
9. Use recursion and guards, to write your own version of the `filter` called `filter'`. Try a figure out the type definition yourself.

3.3 Characters and Strings

In Python, there is no difference between using a single quote mark and using a double quote mark to represent Strings. In Haskell, as well as in Java, there is. A string is merely a list of characters. A character is something like 'a' or 'δ' or '1' or ','. In Haskell, a character is its own distinct data type. If you use a single quote marks, you are signifying a character. So for example, you can write `'a'`, but you cannot write `'Hello'`, or even `''`. If you are using double quote marks you are signifying a string, and in Haskell a string is implemented as a list of characters. So `"Hello"` is just a nicer way of saying `['h','e','l','l','o']`. Here is a function that takes a string and returns true if its first letter is 'a':

```
firstLetterA :: [Char] -> Bool
firstLetterA [] = False
firstLetterA (x:xs) = x == 'a'
```

Look! Pattern matching works just like you would expect it to. The authors of Haskell realised that people preferred to say `String` rather than `Char` for the sake of readability. In fact, there are lots of times where you want to call a complex type by a simple name. To do this they came up with a type synonym. Much like a synonym in the real world, a type synonym is just another way of referring to a type. For instance, to be able to refer to a list of integers as a sequence, one could write:

```
type Sequence = [Int]

a :: Int -> Sequence
a 0 = []
a n = n:(sequence (n-1))
```

Of course, since everybody would want to use `String`, they decided to make it part of the prelude. The prelude is loaded up by default by Haskell before anything else. It contains the functions such as `head` and `max`. It also contains the line:

```
type String = [Char]
```

Therefore, our `firstLetterA` function can be written like this, without having to write an additional type synonym:

```
firstLetterA :: String -> Bool
firstLetterA [] = False
firstLetterA (x:xs) = x == 'a'
```

3.4 Functions on Characters

We will look at the following functions that work with characters. They are not included in the prelude, instead they are available from the `Data.Char` module. To access `Data.Char`, you need this at the top of your file:

```
import Data.Char
```

`ord` returns the ASCII (numeric) value for a given character. For instance `ord 'A'` is 65. `chr` is the inverse function. It returns the character for a given number, ASCII value. So `chr 65` is `'A'`. `isUpper` and `isLower` return true iff a character is uppercase and lowercase respectively. `toUpper` and `toLower` make a character uppercase and lowercase respectively.

3.5 Practicum A: Caesar's Cipher

You are now, going to write a version of Caesar's Cipher. It will be simple and shift every alphabetic character 13 places in the alphabet. So 'A' will become 'N', 'B' will become 'O' and so on. The only code (aside from type definitions), you will be given is:

```
caesar :: String -> String
caesar str = str
```

At the moment, the string returns its input. You will change it later, to do what it should do.

Part 1

The first function you are going to write, will use `ord` and maps and take a string and return the corresponding list of ASCII values:

```haskell
ordString :: String -> [Int]
```

Part 2

We now need a function that takes an uppercase character and shifts it 13 places in the alphabet:

```haskell
shiftUpper :: Int -> Int
```

This function takes in an ASCII value and will need to shift the it 13 places. But you cannot merely add 13. Otherwise, it will not work for letters in the second half of the alphabet. There are a number of ways to go about doing this, you may want to define several additional functions in a where clause.

Part 3

You need to do the same thing for lowercase letters:

```haskell
shiftLower :: Int -> Int
```

Part 4

You need a function, that will shift any character, or not shift at all if it isn't alphabetic. This is where you would need guards, `isUpper` and `isLower`. We will also need our previously defined shift functions.

```haskell
shift :: Int -> Int
```

Part 5

Once you've shifted a list of ASCII values, you will need to convert them all back to a string, using `chr`:

```haskell
chrString :: [Int] -> String
```

Part 6

Finally, you can go to your Caesar's cipher function, and use `ordString`, `chrString` and `shift` to complete your code.

3.6 Folds

Previously, we have looked at recursive functions a lot. Recursion is great, but generally functional programmers prefer to use maps, filters and something called folds. These are higher-order functions (functions that take functions) and provide abstractions over recursion. Whereas, understanding an unfamiliar recursive function is difficult, everybody is familiar with maps, filters and folds. The first fold function we will look at is `foldl1`. It has the type:

```
foldl1 :: (a -> a -> a) -> [a] -> a
```

`foldl1` takes a function *f* that takes two a values and combines them into a single `a` value. It then takes a list of `a` values and applies *f* to items 1 and 2. It then applies *f* to the result and item 3. It then applies *f* to the result and item 4, and so on. E.g.:

```
add x y = x + y

m = foldl1 add [1,2,3,4]
```

`m` would equal 10. Why? Well `m` would equal `add(add(add(1,2),3),4)`. First it would do `add(1,2)`. This is 3. 3 is the accumulator. Then it would do, `add(add(1,2),3)`, i.e. add 3 to the value of the accumulator, yielding 6. Then it would do `add(add(add(1,2),3),4)`, i.e. add 4 to the value of accumulator, yielding 10. The fold function, takes two parameters, the accumulator and the next value. `foldl1` treats the first item, as the start value of the accumulator. There also exists `foldl` which lets you specify your own accumulator, which could be of a different type:

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

For example:

```
containsMultiplesOf5 xs = foldl f False xs
  where f acc x = acc || mod x 5 == 0
```

It would be easier to use a filter here though. There also exists, `foldr` and `foldr1`. The only difference is they start from the right and the order of the arguments to the fold function is different. The accumulator is passed second not first.

3.7 Partial Application

Remember our first example of fold, well we could have rewritten it like this:

```
m = foldl1 (+) [1,2,3,4]
```

Why? Well `+` is what is called an infix function. Which means we call it like `1 + 2` instead of `+ 3 4`. But to make it a normal function, we just put parentheses around it. So this is valid:

```
(+) 1 2
```

However, obviously we wouldn't do that normally, but when using folds it is useful. We can do the opposite thing to functions that have two parameters. E.g. we could replace our second example of a fold with:

```
containsMultiplesOf5 xs = foldl f False xs
  where f acc x = acc || x `mod` 5 == 0
```

By putting backticks around `mod` we change it from a prefix function (where the function name goes at the start) to an infix function. Let's write a function called `fitler5`:

```
filter5 :: (Num a) => [a] -> [a]
filter5 xs = filter equals5 xs
  where equals5 x = x == 5
```

Well, we can rewrite that as:

```
filter5 :: (Num a) => [a] -> [a]
filter5 = filter equals5
  where equals5 x = x == 5
```

What evil sorcery is this? Well, let's think about the definition of filter:

```
filter :: (a -> Bool) -> [a] -> [a]
```

Let's add in parentheses:

```
filter :: (a -> Bool) -> ([a] -> [a])
```

`filter` doesn't take in 2 parameters it takes 1. In fact, all Haskell functions take in 1 parameter. "What???" you ask. It is true. The `filter` function takes one parameter (which also happens to be a function), and returns another function. This function takes in one parameter (a list of values to filter) and returns another parameter (a list of filtered values). In other words

```
(filter equals5 [1,2,3,4]) == ((filter equals5) [1,2,3,4])
```

In other words `filter equals5` is a partially applied function. We have applied its first argument, and now we have a new function which is waiting for a list of values. All functions in Haskell work like this, `mod` takes one argument, and returns another function that takes another value. So:

```
filter equals5 :: [Integer] -> [Integer]
```

And therefore, matches the type required of `filter5`.

We can also do this with infix functions. Let's say we want to add 5 to every element in a list, we could do:

```
add5List :: (Num a) => [a] -> [a]
add5List = map add
  where add x = x + 5
```

But we can do better!

```
add5List :: (Num a) => [a] -> [a]
add5List = map (+5)
```

Here we have partially applied the second element of an infix function. Let's write a function that takes a list of lists of integers, add prepends 5 to every element:

```
prepend5 :: (Num a) => [[a]] -> [[a]]
prepend5 = map (5:)
```

We can also do this with functions like `mod` using backticks. To write a function that takes a list of integers and takes every item to modulo 4:

```
mod4 :: (Num a) => [a] -> [a]
mod4 = map (`mod` 4)
```

Exercises

1. Write a function `sum''` that, of course, sums the input list using a fold.
2. Write a function `length'''` that uses a fold to give the length of a list.
3. Write a function `product''` that calculates the product of all the item in an input list.
4. Write a function `sumSquares` that calculates the sum of the squares on an input list, using a fold.
5. Write a function `concat'` that takes in a list of strings, and uses a fold to concatenate them.
6. Write a function `maximum''` that uses a fold to calculate the maximum item from a list.
7. Write a function `minimum''` that uses a fold to calculate the maximum item from a list.

3.8 Pairs and Tuples

A tuple is a finite ordered collection of elements, that can be of different data types. A tuple is delineated by parentheses. To play around with tuples, type the following into the interpreter:

```haskell
(1,2)
(1,2,3)
(1, "Hello")
(1, '1', "1")
(1, True, False, "Hello")
```

A 2-tuple (tuple of 2 items) is known as a pair. Pairs are extremely common, you can use them to represent co-ordinates, for instance. As shown above tuples can have items of different data types, which might seem to go against the whole type safety essence of Haskell, however it works, because tuples of different lengths and that are made of different data types are seen as, are treated as different data types. So for instance `(Integer, Integer)` (a pair of two integers) is a different type from a `(Integer, String)`, in the same way as a `[Integer]` is different from a `[Char]`. You can pattern match on pairs, for instance:

```haskell
add :: (Num a) => (a, a) -> (a, a) -> (a, a)
add (x1,y1) (x2,y2) = (x1+x2,y1+y2)
```

If you have two lists, you can zip them. This creates a new list consisting of pairs. The first component comes from the first list and the second component comes from the second list. The new list is only as long as the shortest of the input lists:

```haskell
zip :: [a] -> [b] -> [(a,b)]
```

Exercises

1. Write a function `fst'` that returns the first element in a pair.
2. Write a function `snd'` that returns the second element in a pair.
3. Write a function that takes two numbers x and y and returns a pair, where the first item is the sum of the two numbers, and the second is a string stating "Added x to y".
4. Write a function `sumPairs` that takes a list of pairs, and returns a new pair, where the first item is the sum of the first items of the list, and the second item is the sum of the second items of the list.
5. Write a function `inverse` that takes a pair `(a,b)` and returns the pair `(b,a)`.
6. Write a function `addUp` that takes a list of pairs of integers, and returns a list of only integers, with the first component and second component of each pair added together.
7. Write a function `superProduct` that takes a list of pairs of integers, and returns the product of each integer from both components of every pair.
8. Write a function `zipNat` that zips a list with the numbers from 1 to however long the other list is.

3.9 Infinite Lists

To create an infinite list, starting from 1 (**Warning: Do not type in the following code into the interpreter**):

```
[1..]
```

The syntax follows the earlier syntax for arithmetic series e.g. `[1..10]`, `[1,3..10]` and `[10,8..0]`. If you typed the above code into the interpreter, it wouldn't be pretty. You can however type this into the interpreter:

```
take 10 [1..]
```

Because Haskell is a lazy language (yes, that's a technical term!), it only evaluates expressions, when it needs to, so an infinite list won't crash your computer, unless you try to print all of it. However, because I know somebody is going to disregard my earlier instruction, if you do print an entire infinite list, then type `Ctrl-C` before it is too late. Infinite lists mean that if you want to zip `[1,2,3,4]` with the even natural numbers, you can do `zip [1,2,3,4] [2,4..]` to get `[(1,2),(2,4),(3,6),(4,8)]`.

3.10 Practicum B: Polynomials

A polynomial is a mathematical expressions such as:

$$3x^4 + 9x^3 + x^2 + 15x + 29$$

We will only look at polynomials in one variable, in the above case x. Polynomials consist of terms in the form ax^n, added together. So above you have 5 such terms, $3x^4$ all the way to $29x^0$. In general a polynomial can be written in the form:

$$a_n x^n + a_{n-1}x^{n-1} + \ldots + a_0 x^0 \text{ where } n \in \mathbb{N}$$

The values of n are called the exponents. So in $3x^4$ the, 4 is the exponent. The values of a are the coefficients. So in $3x^4$, 3 is the coefficient. a_n refers to the nth coefficient, i.e. the coefficient for x^n, while a_2 would refer to the coefficient of x^2. The constraint $n \in \mathbb{N}$ means that n and by extension all exponents must be a positive integers, i.e. in the set $\{1, 2, 3, 4, \ldots\}$. These would not be valid terms in polynomials:

$$9x^{\frac{3}{4}}$$

Or:

$$9x^{-2}$$

The highest exponent is the degree of the polynomial. So this polynomial is of degree 3:

$$7x^2 + 9$$

Notice it is missing an x term; we say that the coefficient for x or a_1 is 0. We are going to write a set of functions for manipulating polynomials. We will represent polynomials as a list of integers. For example, the above polynomial could be represented as:

$$[9, 0, 7]$$

The first element is the coefficient for the highest term. The only line of code you will be given is:

```
type Polynomial = [Int]
```

Part 1

Write a function `degree` that returns the degree of the polynomial:

```
degree :: Polynomial -> Int
```

Part 2

Write a function `safeAdd` that adds two polynomials of the same degree (to add polynomials, just add each of their corresponding coefficients):

```
safeAdd :: Polynomial -> Polynomial -> Polynomial
```

Part 3

Write a function `neg` that negates every coefficient of a polynomial:

```
neg :: Polynomial -> Polynomial
```

Part 4

Write a function `add` that add two polynomials that may not be of the same degree:

```
add :: Polynomial -> Polynomial -> Polynomial
```

Part 5

Write a function `sub` that subtracts two polynomials. You can use functions you've already defined here.

```
sub :: Polynomial -> Polynomial -> Polynomial
```

Part 6

Write a function `evaluate` that takes a polynomial, and a value of x and evaluates the polynomial. To evaluate a polynomial efficiently, you can use something called Horner's rule. For example, if you have a polynomial of degree two (i.e. a quadratic):

$$a_2x^2 + a_1x + a_0$$

This can be rewritten as:

$$x(x(a_2) + a_1) + a_0$$

Use a right fold here, starting with a_2 as the initial value. Then at each iteration, multiply by x and add the next coefficient.

```haskell
evaluate :: (Num a) => Polynomial -> a -> a
```

Part 7

Write a function `differentiate` that takes a polynomial and returns the polynomial differentiated. You will have to use `zip` to get the exponents. Remember in differentiation:

$$ax^n \to anx^{n-1}$$

```haskell
differentiate :: Polynomial -> Polynomial
```

References and Further Reading

[1] Miran Lipovača. *Learn You A Haskell.* No Starch Press, April 2011.

[2] Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real World Haskell.* O'Reilly Media, December 2009.

[3] Brent Yorgey. *CIS 194, Introduction to Haskell Lecture Notes.* (`http://www.seas.upenn.edu/~cis194/spring13/lectures.html`) Spring 2013, Accessed: September 2016.