Introduction to Haskell and Fuctional Programming

Reading

- Sections 1.1-1.5
- Practice exercises (1.7): 1, 2, 3, 4, 5

Functional Programming and Haskell

Functional Programming is a philosophy and approach to programming that espouses functions as the primary elements of the program. It is considerably different from the "iterative programming" that most students are used to. Here are some of its key aspects:

- Functions are a driving force. We program by providing input to functions and further processing their output.
- We can use functions as the input/output values of functions. *A function can return a function as its result.* Functions that take other functions as input are important means of abstraction.
- Recursive functions become the primary means of performing loops and iterations.
- We rely on the results of functions and not on their side-effects (*pure functions*).

While there are many functional programming languages out there, and in fact many "mainstream" languages include functional-programming elements, we will focus on a specific language called Haskell, in honor of the logicial Haskell Brooks Curry[1] who developed many of the early ideas behind functional programming.

Haskell differs from most other languages you may have seen in many ways:

- Its syntax and the high-level nature of functional programming lead to very **concise programs**. Most Haskell are less than 4 lines long, some of them being only one line.
- Haskell utilizes a very **powerful and expressive type system**. This system allows for a large number of errors to be detected at compile time. At the same time, because of an awesome process called *type inference*, we get all these benefits without almost ever having to specify the types of elements and functions.
- **Lists** of elements are the core data structure for working with a collection of elements, and **List Comprehensions** are a powerful means of expressing the processing of such a list.

[1] https://en.wikipedia.org/wiki/Haskell_Curry

- Essentially all functions in Haskell are **pure**: For a given set of inputs they produce a corresponding output, with **no side-effects**. Calling the function a second time would produce the same output. This will take some getting used to, but it is also very useful. There is *no hidden state* that the functions consult during their operation.
- There is **no mutation** in Haskell. You cannot change the value of a variable as you go through a loop for example. In fact you cannot do a normal "for" loop, as that would require a variable `i` that keeps changing values. This is another feature that will take some getting used to and will be frustrating at times. But knowing that values cannot change can be very beneficial too once you get used to it, and it leads to *safer* programs.
- Haskell implements something called **lazy evaluation**. Expressions are not evaluated until they absolutely have to. As a result, you can provide Haskell with an "infinite list" and it will be OK, because Haskell will only read as much of the list as it needs to.

Example 1: sumUpTo

As an example of the differences between Haskell-style functional programming and iterative style, let us consider a simple function, called `sumUpTo`. `sumUpTo` takes as input one integer, `n`, and is supposed to return the sum of all the numbers from `1` to `n`, with `0` if `n < 1`.

In an iterative language like Python, we might do something like this:

```
# A function sumUpTo in Python
def sumUpTo(n):
    total = 0
    for i in range (1, n + 1):
        total = total + i
    return total
```

Or a C version would look like this:

```
// A function sumUpTo in C
int function sumUpTo(int n) {
    int total = 0;
    for (int i = 1; i <= n; i += 1) {
        total = total + i;
    }
    return total;
}
```

In both cases the logic goes as follows:

- Initialize a `total` value to 0.
- Iterate over each number `i` from `1` to `n`.
- For each such number `i`, add that number to the `total`.

- Return the final value of total

This is a standard approach to iterative programming. We take steps, and on each step we instruct the computer to adjust the values of some variables.

Functional programming is very different. It is more **declarative**. There are two approaches to the problem. One would be as follows, essentially a recursive approach:

- The "sumUpTo" for an n < 1 is 0.
- The "sumUpTo" for another n is *the outcome of* adding n to the result of calling sum on n−1.

```
-- A sumUpTo function in Haskell
let sumUpTo n | n < 1      = 0
              | otherwise = n + sumUpTo (n - 1)
```

This is the approach closest to iterative programming. We have effectively defined a recursive function.

Before we move to other approaches though, notice one important feature of Haskell: Functions do not need parentheses around their arguments, unless the parentheses are needed to resolve ambiguities. So sumUpTo n did not need any parentheses, but sumUpTo (n − 1) needed them so that it is not mistaken for (sumUpTo n) − 1.

In fact in Haskell multiple arguments to a function are simply written next to each other: Instead of f(x, y) we would write f x y.

There are other approaches to writing the sumUpTo function. Another approach breaks the problem in two steps, and creates a helper function along the way:

- Given an integer n, build the list of numbers from 1 to n.
- Given a list of numbers, add them all up. For that we would define a function sumList.

```
-- A sumUpTo function in Haskell
let sumUpTo n = sum [1..n]
        where sum [] = 0
              sum (x:xs) = x + sum xs
```

You may think this is inefficient, as it has to create the list first, but remember that Haskell uses "lazy evaluation": It only computes entries as it needs them, it never has to build the whole list at once.

Finally, a third approach uses a higher-order function called foldl which goes through the elements in a list and uses a function to combine them two at a time:

```
-- A sumUpTo function in Haskell using foldl
let sumUpTo n = foldl (+) 0 [1:n]
```

This is probably the hardest one to read, but it is also more idiomatic of Haskell. By the end of this course you will feel comfortable writing such functions. What this does is the following:

- Create the list of numbers from 1 to n (`[1:n]`).
- Give that list as input to the `foldl` function along with two other inputs:
  - The initial value `0`, analogous to setting our total to 0 to begin with.
  - The function to use to combine values, in this case `(+)` which stands for the addition function.

Example 2: Quicksort

Quicksort[2] is a popular sorting algorithm. It is a divide-and-conquer algorithm (you will learn more about them in CS225), which sorts the values in an array. It operates as follows:

- Start with the first element, the "pivot". (Some variants pick a random element instead.)
- Partition the elements in two groups: Those that are less than or equal to the pivot, and those that are greater than the pivot. Arrange them on either side of the pivot.
- Recursively sort each of these two groups separately.

Let's take a look at how this may look in C code. It typically uses a separate `partition` function. Don't worry if this doesn't make sense right away.

```
// Quicksort in C
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

int partition (int arr[], int low, int high) {
    int pivot = arr[high];     // pivot
    int i = (low - 1);         // Index of smaller element

    for (int j = low; j <= high- 1; j++) {
        if (arr[j] <= pivot) {  // Current element must go to left side
            i++;                // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
```

[2] https://en.wikipedia.org/wiki/Quicksort

```
    swap(&arr[i + 1], &arr[high]);    // Put pivot in its place
    return (i + 1);
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

The Haskell approach is in essence the same, except it focuses on a more high-level description of the process: *The result of performing quicksort on a list is the result of taking all elements less than its first element, followed by that first element, followed by all elements greater than the first element.*

This is not the most efficient implementation of this algorithm in Haskell, but it is illustrative of the language's expressiveness.

```
-- Quicksort in Haskell
let qsort []     = []
    qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
                   where
                        smaller = [a | a <- xs, a <= x]
                        larger  = [b | b <- xs, b > x]
```

And here is an implementation that is a bit closer to the C version:

```
-- Quicksort in Haskell with helper function
--
-- partition returns a pair of the values that are up to
-- the pivot and those that are above.
let partition pivot [] = ([], [])
    partition pivot (x:xs)
                | x <= pivot     = (x:less, more)
                | otherwise      = (less, x:more)
                where (less, more) = partition pivot xs

let qsort [] = []
    qsort (pivot:rest) = qsort less ++ [pivot] ++ qsort more
            where (less, more) = partition pivot rest
```

Practice

To start an interactive session with Haskell, type `ghci` in the terminal. This will bring up the Haskell prompt `Prelude>`. The `Prelude` part there means that a standard library of functions is loaded.

In order to make writing these multiline functions possible, you must run the following each time you start a session:

```
:set +m
```

Start by going to the sumUpTo functions we saw earlier and pasting them in, followed by pressing Return for an extra blank line. Then you can tests these functions by typing something like:

```
sumUpTo 10          -- Should be 55
sumUpTo (-3)        -- Should be 0
```

After you have done this, use these functions as models for creating three versions of productUpTo, which multiplies the numbers up to n (with 1 being the default if n < 1). So the following should work with these functions:

```
productUpTo 4       -- Should be 24
productUpTo (-3)    -- Should be 1
```