1 Haskell

1.1 What is Haskell?

To summarize, *what is* Haskell?

1.1.1 Haskell is Statically Typed

Every expression either has a type, or is *ill-typed* and rejected at compile-time. This is good because:

- We can catch errors early.
- Types are contracts (so that you don't need to handle ill-typed inputs).
- We can enable compiler optimizations.

1.1.2 Haskell is Purely Functional

- Functional: functions are *first-class values*.
- Pure: a program is an expression that evaluates to a value.
  - There are *no* side effects.
  - In other words, a function `T1 -> T1` computes a single `T1` output from a single `T1` input and does nothing else.
- Referential Transparency: The same expression always evaluates to the same value. More precisely, in a scope where `x1, ..., xn` are defined, all occurrences of `e` with $FV(\texttt{e}) = \{\texttt{x1}, \ldots, \texttt{xn}\}$ have the same values.

This is all good because it:

- is easier to reason about.
- enables compiler optimizations.
- is great for parallelization.

1.1.3 Haskell is Lazy

An expression is evaluated only when its result is needed. Most programming languages are *eager*; that is, expressions are computed when seen.

To see what is meant by *lazy*, consider the following code:

```
take :: Int -> [a] -> [a]
take 0 _        = []
take _ []       = []
take n (x:xs)   = x:(take (n - 1) xs)

take 2 (upto 1 (factorial 100))
```

Here, an eager language would compute `factorial 100` and generate a list – which would take forever – and *then* take the first 2 elements. In Haskell, this is done immediately.

This is good because:

- We can implement things like infinite lists: `[1..]`.

```
-- first n pairs of co-primes
take n [(i, j) | i <- [1..],
                 j <- [1..i],
                 gcd i j == 1]
```

- Encourages simple, general solutions.

It also has problems. Some of them are:

- Reasoning about the performance is hard – you never know what part of your program will execute.
- It makes debugging hard – unlike other non-lazy languages, there isn't a stack trace.

2 Data Types & Recursion

When talking about Haskell, we talked about some built-in data types and writing functions using pattern matching and recursion. Now, we'll talk more about recursion and user-defined data types.

2.1 Recursion

Recursion lets us define solutions for big problems from solutions for smaller problems (i.e. sub-problems). In particular, we have:

- The Base Case: What is the simplest version of this problem and how do I solve it?
- Inductive Strategy: How do I break down this problem into sub-problems?
- Inductive Case: How do I solve the problem given the solutions for subproblems?

2.1.1 Benefits of Recursion

Recurison is often – but not always – simpler and cleaner than loops. It forces you to factor code into reusable units (i.e. recursive functions).

2.1.2 Downsides of Recursion

It can be *slow*, and it can cause stack overflow. In particular, every time we call a function, we allocate a frame on the call stack, which is expensive, not to mention that the stack has a finite size.

2.2 Tail Recursion

No computations are allowed on the recursively returned value. In other words, the value returned by the recursive call is equal to the value returned by the function.

(Quiz.) Is this function tail recursive?

```
fac :: Int -> Int
fac n
    | n <= 1        = 1
    | otherwise     = n * fac(n - 1)
```

No. After making a recursive call, we do one more computation on said recursion call.

2.2.1 Tail Recursive Factorial

To convert a recursive function to a tail recursive function, you want to introduce an axillary function that will accumulate the result of the recursive calls.

Your axillary function will essentially take in one more argument than the original:

- An accumulator – the value holding the partial results of the recursive calls. Your initial accumulated value would be the base case.
- The original arguments – remember that you're still doing the recursion on your original input(s).

So, the rewritten factorial function would look like:

```
facTR :: Integer -> Integer
facTR n = loop 1 n
    where
        loop :: Integer -> Integer -> Integer
        loop acc n
            | n <= 1        = acc
            | otherwise     = loop (acc * n) (n - 1)
```

Now, the idea is that, for the base case, we can just return the result generated by the accumulation of the function calls.

At the end of the day, the idea behind tail recursion is that you're calculating the entire result as you make your way down to the bottom of the recursion tree.

2.2.2 Why Tail Recursion?

The compiler can transform your tail recursive function to a **loop**. For example, the above code would translate to something like:

```
function facTR(n) {
    let acc = 1;
    while (true) {
        if (n <= 1) { return acc; }
        else        { acc = acc * n; n = n - 1; }
    }
}
```

Thus, no stack frames are needed.