(A brief and incomplete)

Haskell Refresher

GHC(i)

- GHC is the Haskell compiler we will be using
- GHCi is a interactive/REPL interface to GHC
- Special GHCi commands:
  - :r – Reloads/recompiles your code
  - :t <expr> - prints the type of an expression

Function Syntax

```
fma :: Int -> Int -> Int -> Int
fma x y z = x * y + z
```

- :: is the type ascription operator
  - Specifying a type is optional, but recommended
- Function arguments are separated by only a space in both the declaration and when called
- Functions can also be written as lambdas: `(\x y z -> x * y + z)`
  - The exact rules for lambda syntax are complex, so you probably need parens when writing one

Function Currying

- Functions calls curry:
  - `mod :: Int -> Int -> Int`
  - `(mod 10) :: Int -> Int`
  - `(mod 10 10) :: Int`
- Another term for currying is partial application
- Basically, when a function is called only some of its arguments, a new function is created that automatically calls the original with the given arguments plus whatever is passed to the “partial”

Polymorphic functions

- `id :: a -> a`

- Lower case type names in a functions type are implicitly generic type variables
  - Think `T id<T>(T t) { … }` in Java

Lists

- List literal syntax: `[1, 2, 3]`

- List prepending: `1 : xs`
- List concatenating: `xs ++ ys`

- Recursively iterating a list:

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

Functions over lists

- `map :: (a -> b) -> [a] -> [b]`
  - Applies a function to each element of a list and returns a list containing the results of those function calls
- `filter :: (a -> Bool) -> [a] -> [a]`

- `foldl :: (b -> a -> b) -> b -> [a] -> b`
  `foldr :: (a -> b -> b) -> b -> [a] -> b`
  - Builds a value by applying a function with an accumulator value over a list. Can be used like a for-each loop in imperative languages.

Functions over lists

foldr is similar to the following imperative code:

```
fn foldr<A, T>(f: Fn(T, A) -> A, accum: A, ts:
[T]) -> A {
    for i in ts {
        accum = f(i, accum);
    }
    return accum;
}
```

Bindings

- let - Prefix

```
someFunc x y z = let someValue = x + y
                 in z * someValue
```

- where – Postfix

```
someFunc x y z = z * helper y
                 where helper n = (x + 1) * n
```

- Note: Bindings are NOT variables; they're immutable

Tuples

- Tuples are a way of grouping values together
- `(10, “Text”) :: (Int, String)`

- `zip :: [a] -> [b] -> [(a, b)]`
  - The zip function builds a list of tuples from two lists

Resources

- Haskell's Prelude – Documentation for the functions and data types that are imported by default

- Hoogle – Search engine for Haskell functions. Accepts either names or function types