

1 Higher-Order Functions

1.1 Tail-Recursive Versions

Let's write a tail-recursive version of `sum`. In particular,

```
sumTR :: [Int] -> Int
sumTR xs = helper 0 xs
  where
    helper :: Int -> [Int] -> Int
    helper acc []      = acc
    helper acc (x:xs) = helper (acc + x) xs
```

Let us now write a tail-recursive `cat` function. Note that this is very similar to what we have above. Its implementation is

```
catTR :: [String] -> String
catTR xs = helper "" xs
  where
    helper :: String -> [String] -> String
    helper acc []      = acc
    helper acc (x:xs) = helper (acc ++ x) xs
```

Note that there is an apparent pattern here, which can be extracted to the *fold-left* pattern:

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f b xs = helper b xs
  where
    helper acc []      = acc
    helper acc (x:xs) = helper (f acc x) xs
```

In general, the pattern is:

- Use a helper function with an extra accumulator argument.
- To compute the new accumulator, combine the current accumulator with the head using some binary operation.

(Quiz.) What does this evaluate to?

```
foldl f b []      = b
foldl f b (x:xs) = foldl f (f b x) xs

quiz = foldl (:) [] [1,2,3]
```

- (a) Type error.
- (b) [1, 2, 3]
- (c) [3, 2, 1]
- (d) [[3], [2], [1]]
- (e) [[1], [2], [3]]

The answer is **A**. Note that `a` is an `Int` and `b` is an `[Int]`. So, our accumulator function is of type `[Int] -> Int -> [Int]`. But, keep in mind that `(:)` (the cons operator) takes an `Int` followed by a `[Int]`. This is a type error since the accumulator function types disagree.

What does this evaluate to?

```
foldl f b [] = b
foldl f b (x:xs) = foldl f (f b x) xs

quiz = foldl (\xs x -> x : xs) [] [1,2,3]
```

- (a) Type error.
- (b) [1,2,3]
- (c) [3,2,1]
- (d) [[3], [2], [1]]
- (e) [[1],[2],[3]]

The answer is **C**. To see why this is the case, consider the following work:

```
foldl f [] [1,2,3]
==> foldl f (1 : []) [2,3]
==> foldl f (2 : (1 : [])) [3]
==> foldl f (3 : (2 : (1 : []))) []
==> 3 : (2 : (1 : []))
= [3,2,1]
```

1.1.1 Fold Left vs. Right

To see the difference between the two fold functions, consider the following:

```
foldl f b [x1, x2, x3] ==> f (f (f b x1) x2) x3 -- Left
foldr f b [x1, x2, x3] ==> f x1 (f x2 (f x3 b)) -- Right
```

As an example, we have:

```
foldl (+) 0 [1, 2, 3] ==> ((0 + 1) + 2) + 3 -- Left
foldr (+) 0 [1, 2, 3] ==> 1 + (2 + (3 + 0)) -- Right
```

As for their types:

```
foldl :: (b -> a -> b) -> b -> [a] -> b -- Left
foldr :: (a -> b -> b) -> b -> [a] -> b -- Right
```

1.2 Useful Higher-Order Functions

Consider the function:

```
foldl (\xs x -> x : xs) [] [1,2,3]
```

This is the same thing as:

```
foldl (flip (:)) [] [1,2,3]
```

Its type signature is given by:

```
flip :: (a -> b -> c) -> (b -> a -> c)
```

There is also the *compose* function. So, instead of writing

```
map (\x -> f (g x)) ys
```

we can write

```
map (f . g) ys
```

Its type signature is given by:

```
--      f          g          f . g  
(.) :: (a -> b) -> (c -> a) -> (c -> b)
```

2 Environments and Closures

We will now begin the process of *implementing* a functional language. In this section, we will discuss how to evaluate a program given its abstract syntax tree (AST), and also prove properties about our interpreter.

We will implement the Nano programming language. Its features include

1. Arithmetic
2. Variables
3. Let-bindings
4. functions
5. Recursion

Generally, the idea is, given a string containing the program, it will be converted to its AST (abstract syntax tree) form¹. From there, it can be evaluated to the desired result.

2.1 Nano: Arithmetic

A grammar of arithmetic expressions can be represented like so:

```
e :: n
    | e1 + e2
    | e1 - e2
    | e1 * e2
```

We can represent this by the following datatype:

```
data Expr = Num Int
          | Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
```

¹This process is known as parsing.