1 Representing Complex Data

1.1 Recursive Types

Let's define **natural numbers** from scratch; in particular,

```
data Nat = Zero | Succ Nat
```

Specifically, a `Nat` is either

- An empty box labeled `Zero`,
- or a box labeled `Succ` with another `Nat` in it.

Some examples of `Nat` values are

```
Zero                                   -- 0
Succ Zero                              -- 1
Succ (Succ Zero)                       -- 2
Succ (Succ (Succ Zero))                -- 3
```

1.1.1 Functions on Recursive Types

We can then write a recursive function for this type:

```
data Nat = Zero              -- Base Constructor
         | Succ Nat          -- Inductive Constructor
```

Call this function `toInt`, which convers the recursive type to its corresponding number. First, we add a pattern for each constructor.

```
toInt :: Nat -> Int
toInt Zero      = ...       -- Base Case
toInt (Succ n)  = ...       -- Inductive Case (Recursive Call Goes Here)
```

Next, we can fill in the base case.

```
toInt :: Nat -> Int
toInt Zero      = 0
toInt (Succ n)  = ...
```

After that, we can fill in the inductive case using a recursive step.

```
toInt :: Nat -> Int
toInt Zero      = 0
toInt (Succ n)  = 1 + toInt n
```

(Quiz.) What does this evaluate to?

```
let foo i = if i <= 0 then Zero else Succ (foo (i - 1))
in foo 2
```

(a) Syntax error

(b) Type error

(c) 2

(d) `Succ Zero`

(e) `Succ (Succ Zero)`

The answer is **E**. This function does the reverse of `toInt`; that is, given an `Int`, it returns the `Nat` representation.

1.1.2 Recursive Type as Result

TODO

1.1.3 Putting the Two Together

Let us now implement the `add` function.

```
add :: Nat -> Nat -> Nat
add Zero        m = m                 -- Base Case
add (Succ n')   m = Succ (add n' m) -- Inductive Case
```

The idea is that $n' = n - 1$, so we're basically doing $n - 1 + m$ for the `add` function in the inductive case. What we want is $n + m$, so we can use `Succ` to add one.

Subtraction is somewhat more difficult.

```
sub :: Nat -> Nat -> Nat
sub n           Zero        = n
sub Zero        _           = Zero
sub (Succ n')   (Succ m')  = sub n' m'
```

Here, we have that $n' = n - 1$ and $m' = m - 1$, so we're doing $n - 1 - (m - 1) = n - 1 - m + 1 = n - m$.

1.2 Lists

Note that lists are not built-in. In fact, they are an algebraic data type like any other. So,

```
data List = Nil
            | Cons Int List
```

So, `[1, 2, 3]` is represented by `Cons 1 (Cons 2 (Cons 3 Nill))`. So, we can think of the built-in list constructors `[]` and `(:)` as fancy syntax for `Nil` and `Cons`.

Functions on lists follow the same general strategy. For example,

```
length :: List -> Int
length Nil          = 0
length (Cons _ xs)  = 1 + length xs
```

What about `append`ing two lists?

```
append :: List -> List -> List
append xs ys = ??
```

One implementation is

```
append :: List -> List -> List
append Nil          ys  = ys
append (Cons x xs)  ys  = Cons x (append xs ys)
```

Now, we expect `append [1, 2] [3, 4]` to give us `[1, 2, 3, 4]`. Indeed, this is the expected result that we get.

1.3 Calculator

Suppose you want to implement an arithmetic calculator to evaluate expressions like

- `4.0 + 2.9`
- `3.78 - 5.92`
- `(4.0 + 2.9) * (3.78 - 5.92)`

What is a datatype that we can use to represent these expressions?

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

Now, how do we write said function to evaluate an expression?

```
eval :: Expr -> Float
eval (Num x)        = x
eval (Add e1 e2)    = (eval e1) + (eval e2)
eval (Sub e1 e2)    = (eval e1) - (eval e2)
eval (Mul e1 e2)    = (eval e1) * (eval e2)
```