1 Data and Codata

Data is defined by its introduction rules.
We can define a data type with

```
data Either a b
  = Left a
  + Right b
```

This definition will effectively introduce two functions:

```
Left : a -> Either a b
Right : b -> Either a b
```

In order to destruct data we have to use patterns which let you match on constructors.

```
case e of
   Left x -> e1
   Right y -> e2
```

Here, e represents the value being destructed, and each branch represents a constructor with which it might have been constructed. We are effectively dispatching on possible pasts of e.

Codata is defined by its elimination rules.
We can define a codata type with

```
data Both a b
  = first a
  * second b
```

This definition will effectively introduce two functions:

```
first : Both a b -> a
second : Both a b -> b
```

In order to construct codata we have to use copatterns which let you match on destructors.

```
merge x from
   first x <- e1
   second x <- e2
```

Here, x represents the value being constructed, and each branch represents a destructor with which it might eventually be destructed. We are effectively dispatching on possible futures of x.

2 Codata, Records, and Copatterns

In the same way that named sums are a natural way to represent data, records are a natural way to represent codata. In lieu of the above syntax, one often sees codata represented as something more like

```
record Both a b = { .first : a, .second : b }

x : Both Int Bool
x = { .first = 2, .second = true }

assert x.first == 2
assert x.second == true
```

The `merge` syntax is used here for conceptual symmetry with `case`. Additionally, the use of copatterns is nicely dual with the extra expressivity that patterns offer. For example, we can use nested patterns with constructors of various types, as in this function which processes a list of `Either Int Bool` values by summing the integers in the list until it reaches a `false` value or the end of the list:

```
f : List (Either Int Bool) -> Int
f lst = case lst of
  Cons (Left i) xs -> i + f xs
  Cons (Right b) xs -> if b then f xs else 0
  Nil            -> 0
```

Similarly, we can define an infinite stream of pairs using nested copatterns as so:

```
s : Int -> Stream (Both Int Bool)
s n = merge x from
  first (head x) <- n
  second (head x) <- n > 0
  tail x      <- x
```

Copatterns are also practically expressive, as in this concise and readable definition of the fibonacci numbers in terms of the `merge` expression:

```
data Stream a
  = head a
  * tail (Stream a)

zipWith : (a -> b -> c) -> Stream a -> Stream b -> Stream c
zipWith f s1 s2 = merge x from
  head x <- f (head s1) (head s2)
  tail x <- zipWith f (tail s1) (tail s2)

fibs : Stream Int
fibs = merge x from
  head x     <- 0
  head (tail x) <- 1
  tail (tail x) <- zipWith (+ ) x (tail x)
```

3 Row-Typed Codata

It is possible to build an open sum by a small modification of the datatype mechanism. Instead of naming types and listing their associated constructors, we represent a type as a list of constructors and types.

```
{- .. -} : [ Left: a + Right: b ]
```

Use of a constructor no longer specifies a specific type, but rather any type that can be constructed with that constructor.

```
Left : a -> [ Left: a + ... ]
Right : b -> [ Right: b + ... ]
```

If we want to destruct an open value like this, we can use a `case` construct, as before.

```
f : [ Left Int + Right Bool ] -> Int
f e = case e of
        Left  i -> i
        Right b -> if b then 1 else 0
```

It is possible to build an open product by a small modification of the datatype mechanism. Instead of naming types and listing their associated destructors, we represent a type as a list of destructors and types.

```
{- .. -} : { first: a * second: b }
```

Use of a destructor no longer specifies a specific type, but rather any type that can be destructed with that destructor.

```
first : { first: a * ... } -> a
second : { second: b * ... } -> b
```

If we want to construct an open value like this, we can use a `merge` construct, as before.

```
f : Int -> { First Int * Second Bool }
f i = merge x from
        first  x <- i
        second x <- i == 0
```