

Haskell: Making Types

Principles of Programming Languages

Colorado School of Mines

<https://lambda.mines.edu>

Learning Group Activity

Review the LGA with your group. Yadda yadda yadda.

The data Keyword

In Haskell, using the `data` keyword lets us define our own types. For example, consider the `Bool` type, which could be defined as so:

```
data Bool = True | False
```

This reads, the data type `Bool` can be the value `True` or `False`.

A Simple Data Type

Suppose we wanted to define a Point type which stores two Floats:

```
data Point = Point Float Float
```

That Point on the RHS of the = is the name of the constructor. This could be a different name than the type, but typically you will find it shares the same name.

A Simple Data Type

Suppose we wanted to define a Point type which stores two Floats:

```
data Point = Point Float Float
```

That Point on the RHS of the = is the name of the constructor. This could be a different name than the type, but typically you will find it shares the same name.

To make a Point:

```
(Point 10.5 11.2)
```

Why Not Just Tuples?

While it is true you could just use a (Float, Float) tuple to represent your points, consider the downsides:

- **Type Safety:** What about (Float, Float) tuples that don't refer to points? Should these work in your methods made for points?
- **Readability:** Would another programmer reading your code understand that the 2-tuple referred to points? Maybe for points but harder for other types (what about a type Person that describes features of a person?)

Another Example

```
data Shape = Circle Point Float | Rectangle Point Point
```

We can refer to our type in our functions by pattern matching:

```
area :: Shape -> Float
```

```
area (Circle _ r) = pi * r^2
```

```
area (Rectangle (Point x1 y1) (Point x2 y2)) = (x2 - x1) * (y2 - y1)
```

Activity

With your learning group, write a perimeter function. Then, extend the Shape data type to allow for a triangle (how you store this is up to you, right triangles are OK) and extend your perimeter function.

Record Syntax

You could imagine writing functions to help you get the center or radius of a `Circle`:

```
center (Circle c _) = c  
radius (Circle _ r) = r
```

Fortunately, *you don't have to do this*. Haskell provides a **Record Syntax** that defines these functions for you, and makes your code a bit more readable:

```
data Circle = Circle { center :: Point, radius :: Float }
```

This also gives us access to a new constructor syntax:

```
Circle {center=(Point 1 1), radius=10}
```

Type Parameters

Type parameters allow your type to take another data type. A simple example is the Maybe type builtin to Haskell:

```
data Maybe a = Nothing | Just a
```

How does this read? Maybe is either Nothing or the type specified by **a**.

As another example, suppose you wanted to define a 3D Vector data type that could work on any type:

```
data Vector a = Vector a a a
```

Deriving Typeclasses

Suppose we want to be able to print our circles to the screen:

```
data Point = Point Float Float deriving (Show)
```

```
data Circle = Circle {  
    center :: Point,  
    radius :: Float  
} deriving (Show)
```

Now...

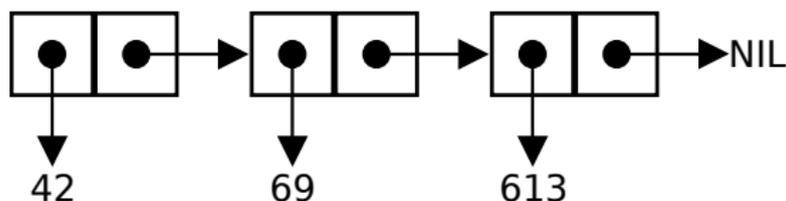
```
GHCi> Circle (Point 1 1) 1  
Circle (Point 1 1) 1
```

We can use this to derive other attributes:

```
data Point = Point Float Float deriving (Show, Eq)
```

How are lists really defined?

One of the most fundamental types in programming languages is the **cons cell**. A cons cell is simply an ordered pair where either element may be a pointer to another.



Haskell uses cons cells to make lists as shown in the image above. In fact, `[42,69,113]` is really just a fancy syntax for `42:69:113:[]`!

Recursive Types

Suppose we wanted to make our own list type:

```
data List a = Empty | Cons a (List a)
```

Then, we can build "lists" using this notation:

```
-- Equivalent to [5]  
(Cons 5 Empty)
```

```
-- Equivalent to [4,5]  
(Cons 4 (Cons 5 Empty))
```

```
-- Equivalent to [3,4,5]  
(Cons 3 (Cons 4 (Cons 5 Empty)))
```

Recursive Types

Suppose we wanted to make our own list type:

```
data List a = Empty | Cons a (List a)
```

Then, we can build "lists" using this notation:

```
-- Equivalent to [5]  
(Cons 5 Empty)
```

```
-- Equivalent to [4,5]  
(Cons 4 (Cons 5 Empty))
```

```
-- Equivalent to [3,4,5]  
(Cons 3 (Cons 4 (Cons 5 Empty)))
```

What do we get if we rename Cons to : and Empty to []?

Activity: Tree Type

With your learning group, create a `Tree` type that defines a binary tree.

Hint: Go back to our `cons` cell definition.

Once done, think about how you could extend the type to support n -ary trees.