Algebraic data types ====================

CIS 194 Week 2
21 January 2013

Suggested reading:

- **Real World Haskell**, chapters 2 and 3

Enumeration types

Like many programming languages, Haskell allows programmers to create their own *enumeration* types. Here's a simple example:

```
data Thing = Shoe
           | Ship
           | SealingWax
           | Cabbage
           | King
  deriving Show
```

This declares a new type called `Thing` with five *data constructors* `Shoe`, `Ship`, etc. which are the (only) values of type `Thing`. (The `deriving Show` is a magical incantation which tells GHC to automatically generate default code for converting `Thing`s to `String`s. This is what `ghci` uses when printing the value of an expression of type `Thing`.)

咒语

```
shoe :: Thing
shoe = Shoe

listO'Things :: [Thing]
listO'Things = [Shoe, SealingWax, King, Cabbage, King]
```

We can write functions on `Thing`s by *pattern-matching*.

```
isSmall :: Thing -> Bool
isSmall Shoe       = True
isSmall Ship       = False
isSmall SealingWax = True
isSmall Cabbage    = True
isSmall King       = False
```

Recalling how function clauses are tried in order from top to bottom, we could also make the definition of `isSmall` a bit shorter like so:

```
isSmall2 :: Thing -> Bool
isSmall2 Ship = False
isSmall2 King = False
isSmall2 _    = True
```

Beyond enumerations

`Thing` is an *enumeration type*, similar to those provided by other languages such as Java or C++. However, enumerations are actually only a special case of Haskell's more general *algebraic data types*. As a first example of a data type which is not just an enumeration, consider the definition of `FailableDouble`:

type constructor

```
data FailableDouble = Failure
                    | OK Double
  deriving Show
```

aka. value constructors.

This says that the `FailableDouble` type has two data constructors. The first one, `Failure`, takes no arguments, so `Failure` by itself is a value of type `FailableDouble`. The second one, `OK`, takes an argument of type `Double`. So `OK` by itself is not a value of type `FailableDouble`; we need to give it a `Double`. For example, `OK 3.4` is a value of type `FailableDouble`.

```
ex01 = Failure
ex02 = OK 3.4
```

OK :: Double → FailableDouble

Thought exercise: what is the type of `OK`?

```
safeDiv :: Double -> Double -> FailableDouble
safeDiv _ 0 = Failure
safeDiv x y = OK (x / y)
```

More pattern-matching! Notice how in the `OK` case we can give a name to the `Double` that comes along with it.

```
failureToZero :: FailableDouble -> Double
failureToZero Failure = 0
failureToZero (OK d)  = d
```

Data constructors can have more than one argument.

```
-- Store a person's name, age, and favourite Thing.
data Person = Person String Int Thing
  deriving Show

brent :: Person
brent = Person "Brent" 31 SealingWax

stan :: Person
stan  = Person "Stan" 94 Cabbage

getAge :: Person -> Int
getAge (Person _ a _) = a
```

Notice how the type constructor and data constructor are both named `Person`, but they inhabit different namespaces and are different things. This idiom (giving the type and data constructor of a one-constructor type the same name) is common, but can be confusing until you get used to it.

Algebraic data types in general

In general, an algebraic data type has one or more data constructors, and each data constructor can have zero or more arguments.

```
data AlgDataType = Constr1 Type11 Type12
                 | Constr2 Type21
                 | Constr3 Type31 Type32 Type33
                 | Constr4
```

This specifies that a value of type `AlgDataType` can be constructed in one of four ways: using `Constr1`, `Constr2`, `Constr3`, or `Constr4`. Depending on the constructor used, an `AlgDataType` value may contain some other values. For example, if it was constructed using `Constr1`, then it comes along with two values, one of type `Type11` and one of type `Type12`.

One final note: type and data constructor names must always start with a capital letter; variables (including names of functions) must always start with a lowercase letter. (Otherwise, Haskell parsers would have quite a difficult job figuring out which names represent variables and which represent constructors).

Pattern-matching

We've seen pattern-matching in a few specific cases, but let's see how pattern-matching works in general. Fundamentally, pattern-matching is about taking apart a value by *finding out which constructor* it was built with. !!! This information can be used as the basis for deciding what to do—indeed, in Haskell, this is the *only* way to make a decision.

For example, to decide what to do with a value of type `AlgDataType` (the made-up type defined in the previous section), we could write something like

```
foo (Constr1 a b)   = ...
foo (Constr2 a)     = ...
foo (Constr3 a b c) = ...
foo Constr4         = ...
```

name will bound to value

Note how we also get to give names to the values that come along with each constructor. Note also that parentheses are required around patterns consisting of more than just a single constructor.

This is the main idea behind patterns, but there are a few more things to note.

1. An underscore _ can be used as a "wildcard pattern" which matches anything.

2. A pattern of the form `x@pat` can be used to match a value against the pattern `pat`, but *also* give the name `x` to the entire value being matched. For example:

!!!

```
baz :: Person -> String
baz p@(Person n _ _) = "The name field of (" ++ show p ++ ") is " ++ n

*Main> baz brent
"The name field of (Person \"Brent\" 31 SealingWax) is Brent"
```

3. Patterns can be *nested*. For example:

```
checkFav :: Person -> String
checkFav (Person n _ SealingWax) = n ++ ", you're my kind of person!"
checkFav (Person n _ _)          = n ++ ", your favorite thing is lame."

*Main> checkFav brent
"Brent, you're my kind of person!"
*Main> checkFav stan
"Stan, your favorite thing is lame."
```

   Note how we nest the pattern `SealingWax` inside the pattern for `Person`.

In general, the following grammar defines what can be used as a pattern:

```
pat ::= _
     |  var
     |  var @ ( pat )
     |  ( Constructor pat1 pat2 ... patn )
```

The first line says that an underscore is a pattern. The second line says that a variable by itself is a pattern: such a pattern matches anything, and "binds" the given variable name to the matched value. The third line specifies `@`-patterns. The last line says that a constructor name followed by a sequence of patterns is itself a pattern: such a pattern matches a value if that value was constructed using the given constructor, *and* `pat1` through `patn` all match the values contained by the constructor, recursively.

(In actual fact, the full grammar of patterns includes yet more features still, but the rest would take us too far afield for now.)

Note that literal values like `2` or `'c'` can be thought of as constructors with no arguments. It is as if the types `Int` and `Char` were defined like

```
data Int  = 0 | 1 | -1 | 2 | -2 | ...
data Char = 'a' | 'b' | 'c' | ...
```

which means that we can pattern-match against literal values. (Of course, `Int` and `Char` are not *actually* defined this way.)

Case expressions

The fundamental construct for doing pattern-matching in Haskell is the `case` expression. In general, a `case` expression looks like

```
case exp of
  pat1 -> exp1
  pat2 -> exp2
  ...
```

When evaluated, the expression `exp` is matched against each of the patterns `pat1`, `pat2`, … in turn. The first matching pattern is chosen, and the entire `case` expression evaluates to the expression corresponding to the matching pattern. For example,

```
ex03 = case "Hello" of
           []      -> 3
```

```
            ('H':s) -> length s
            _       -> 7
```

evaluates to `4` (the second pattern is chosen; the third pattern matches too, of course, but it is never reached).

In fact, the syntax for defining functions we have seen is really just convenient syntax sugar for defining a `case` expression. For example, the definition of `failureToZero` given previously can equivalently be written as

```haskell
failureToZero' :: FailableDouble -> Double
failureToZero' x = case x of
                     Failure -> 0
                     OK d    -> d
```

Recursive data types

Data types can be *recursive*, that is, defined in terms of themselves. In fact, we have already seen a recursive type—the type of lists. A list is either empty, or a single element followed by a remaining list. We could define our own list type like so:

```haskell
data IntList = Empty | Cons Int IntList
```

Haskell's own built-in lists are quite similar; they just get to use special built-in syntax (`[]` and `:`). (Of course, they also work for any type of elements instead of just `Int`s; more on this next week.)

type variable

We often use recursive functions to process recursive data types:

```haskell
intListProd :: IntList -> Int
intListProd Empty      = 1
intListProd (Cons x l) = x * intListProd l
```

As another simple example, we can define a type of binary trees with an `Int` value stored at each internal node, and a `Char` stored at each leaf:

```haskell
data Tree = Leaf Char
          | Node Tree Int Tree
  deriving Show
```

(Don't ask me what you would use such a tree for; it's an example, OK?) For example,

```haskell
tree :: Tree
tree = Node (Leaf 'x') 1 (Node (Leaf 'y') 2 (Leaf 'z'))
```

`Generated 2013-03-14 14:39:58.373475`