Case expressions

Pattern matching

- **Sum types** list the different ways that a type can be constructed, each with a unique tag name (or ***constructor***) separated by the symbol `|`.
- Functions that take sum types as input**,** or that construct something of sum type during computation, almost always need to act differently on the different constructors.
- There is a Haskell expression `case` that allows us to ***pattern match*** to say how each different constructor should be dealt with (also works on values!).
- Where constructors have arguments, we can give names to each argument so that we can refer to them on the right side of the function definition.

```haskell
data Bool = True | False


myNot :: Bool -> Bool

myNot b =
  case b of
   True  ->  False
   False  ->  True
```

```haskell
data Bool = True | False


myAbs :: Int -> Int

myAbs x =
  case x >= 0 of
   True  ->  x
   False  ->  -x
```

```
data Bool = True | False


myAnd :: Bool -> Bool -> Bool

myAnd b c =
  case (b,c) of
   (True,True)  ->  True
   (True,False)  ->  False
   (False,True)  ->  False
   (False,False) ->  False
```

```
data Bool = True | False


myAnd :: Bool -> Bool -> Bool

myAnd b c =
  case (b,c) of
   (True,True)  ->  True
   _   ->  False
```

Note that the cases are run through in order from the top, so `(True,True)` will never trigger the _ case.

```
data Animal = Cat | Dog | Cow


says :: Animal -> String

says x =
  case x of
    Cat  -> "meeow"
    Dog  -> "woof"
    Cow  -> "moo"
```

user-defined

```haskell
data Animal = Cat | Dog | Cow | Parrot String


says :: Animal -> String

says x =
  case x of
   Cat            -> "meeow"
   Dog            -> "woof"
   Cow            -> "moo"
   Parrot name    -> "pretty " ++ name
```

```haskell
data Animal = Cat | Dog | Cow | Parrot String


says :: Animal -> String

says x =
  case x of
   Cat           -> "meeow"
   Dog           -> "woof"
   Cow           -> "moo"
   Parrot ""     -> "caw"
   Parrot name   -> "pretty " ++ name
```

```haskell
data Animal = Cat | Dog | Cow | Parrot String | Fox


says :: Animal -> String

says x =
  case x of
   Cat           -> "meeow"
   Dog           -> "woof"
   Cow           -> "moo"
   Parrot ""     -> "caw"
   Parrot name   -> "pretty " ++ name
   Fox           -> error "what does the fox say?"
```

Guarded expressions

- Often the type we are using `case` on is a `Bool`.
- There is special notation just for this, called **guarded expressions**, or just **guards**.
- You never *need* to use guards, but they can make programs easier to write and read, especially when you want to subject your data to lots of different tests with Boolean results.
- Commands that are not essential, but make life easier, are generally known as **syntactic sugar**.

```haskell
myAbs :: Int -> Int

myAbs x =
  case x >= 0 of
   True  ->  x
   False  ->  -x


myAbs x
  | x >= 0 = x
  | otherwise = -x
```

```
approxSize :: Int -> String

approxSize x
   | abs(x) > 10000000000000000 = "quintillions"
   | abs(x) > 1000000000000000 = "quadrillions"
   | abs(x) > 1000000000000 = "trillions"
   | abs(x) > 1000000000 = "billions"
   | abs(x) > 1000000 = "millions"
   | abs(x) > 1000 = "thousands"
   | otherwise = "small"
```

How would you write this with `case`?

Evaluated in order until the top-most guard that evaluates to `True`.

```
foo x1 x2 … xk
   | g1          = e1
   | g2          = e2
   ...
   | otherwise     = e
```

First `g1`, then `g2`, etc.

More ‘Sugar’: Piecewise definitions

- Often the type(s) we are using `case` on are inputs.
- There is then still another way to do our case expression, using **piecewise definition**.
- Unlike guards you can live without these, but some people prefer them, so you should at least see them.

```
myNot :: Bool -> Bool

myNot b =
  case b of
   True  ->  False
   False  ->  True
```

versus

```
myNot True = False
myNot False = True
```

```
myAnd :: Bool -> Bool -> Bool

myAnd b c =
  case (b,c) of
   (True,True)  ->  True
   _   ->  False
```

versus

```
myAnd True True = True
myAnd _ _ = False
```