CHAPTER 7: MORE FUNCTIONAL PATTERNS

A value that can be used as an argument to a function is a *first-class* value.

Some ERROR cases!!!

The first one, not in scope because *y* was *inside* the `let` expression.

```haskell
bindExp :: Integer -> String
bindExp x = let z = y + x in -- Here y is not in scope!!!
            let y = 5 in "the integer was: "
            ++ show x ++ " and y was: "
            ++ show y ++ " and z was: " ++ show z
```

This time, `x = 10` shadows the `x` argument.

```haskell
bindExp :: Integer -> String
bindExp x = let x = 10; y = 5 in
            "the integer was: " ++ show x
            ++ " and y was: " ++ show y
```

The reason is that Haskell has **lexical/static scoping** (depends on the location in the code and the lexical context - like `let` or `where` clauses).

Anonymous functions

```haskell
Prelude> (\x -> x * 3) 5
15
```

It doesn't need a name, since it's called only once.

Pattern matching

If `x` is not matched yet, `f x =` *bottom* (a non-value - the program can't return a value/result - like infinite loop) and throw an exception.

```
*** Exception: :50:33-48:
  Non-exhaustive patterns
      in function isItTwo
```

Pattern matching against data constructors

Some data constructors have **parameters**, and pattern matching helps expose the data in their arguments.

- `newtype` is `data` but only 1 field and 1 constructor.

```haskell
newtype Username = Username String -- 1 field and 1 constructor
```

Higher-order function

Higher-order functions are functions that accept functions as arguments. Functions are values.

Guards evaluate **sequentially**, order it from the most common case to the least.

Function composition and "pointfree" style

```haskell
(f . g) x = f (g x)
Prelude> negate . sum $ [1, 2, 3, 4, 5] -- or (negate . sum) [1, 2, 3, 4, 5]
-15
Prelude> let f x = take 5 . enumFrom $ x
Prelude> f 3
[3,4,5,6,7]
```

Btw, using `$` makes the *applications* happen **after** functions are composed.

Currying

Currying is the process of transforming a function that takes multiple arguments into **a series of functions** which each take 1 argument and return one result.

And don't use error.