Fun with Functions

(in Swift)

A function in Swift

```
func f(x: A) -> B {
  // do something with x
  // return something in B
}
```

Composition

If we have

```
func f(A) -> B
func g(B) -> C
```

then we can do

```
g(f(x))
```

Multiple return values

```
func f(x: Int) -> (Int, Int) {
  return (x, 2*x)
}

f(3) // => (3, 6)
```

Composition again

```
func f(x: Int) -> (Int, Int) {
  return (x, 2*x)
}

func g(x: Int, y: Int) -> Int {
  return x*x + y*y
}

g(f(2)) // => 20
```

Better composition syntax

```
@infix func * <A,B,C> (g: B -> C, f: A -> B) -> (A -> C) {
  return {(x: A) -> C in
    return g(f(x))
  }
}

func f(A) -> B
func g(B) -> C

(g * f)(x)
```

Why do that?

- Can now define variables that are compositions:

```
let h = g * f
```

- Mathematical throwback:

```
g(f(x)) = (g ∘ f)(x)
```

- More readable:
  - `roundf(g(atanf(g(f(3)))))`
  - `(roundf * g * atanf * g * f)(3)`

Better composition syntax

```
@infix func * <A,B,C> (g: B -> C, f: A -> B) -> (A -> C) {
  return { g(f($0)) }
}
```

First class citizens

- Functions are data types.
- Functions can return functions.
  - also known as Higher Order Functions
- Functions can be defined within function bodies.

Curry

Given

```
func f(A, B) -> C
```

We can fix an x in A to get a func B -> C:

```
func f_x(y: B) -> C {
  return f(x, y)
}
```

Curry

So we have a correspondence sending x's in A to functions B `->` C, i.e.

```
func f(x: A) -> (B -> C) {
  return { f(x, $0) }
}
```

```
f(1)    // => (Function)
f(1)(2) // => 5
```

Curry

We say that

```
f: A -> (B -> C)
```

is the *curried* form of

```
f: (A, B) -> C
```

and vice-versa, *uncurried* form.

Curry

There's a correspondence:

```
(A, B) -> C
```

$$\Updownarrow$$

```
A -> (B -> C)
```

Currying in Swift

```
func f(x: A)(y: B) -> C {
  // body
}
```

```
f(x)      // => (Function) B -> C
f(x)(y) // => C
```

However, this should work too IMO:

```
f(x, y) // nope
```

Ideal

```
f: A -> B -> C -> D -> E

f(a, b, c, d) // => E
f(a, b, c)(d) // => E
f(a, b)(c)    // => D -> E
f(a)          // => B -> C -> D -> E

g: A -> B
f(g)          // => C -> D -> E
```

First class citizens

First class citizens

We've been treating functions abstractly

They have a domain, A
They have a target, B

And we just draw an arrow

```
f: A -> B
```

First class citizens

If domains and targets match-up

```
g: B -> C
```

we allow composition

```
g * f: A -> C
```

Level Up

```
let xs: [Int] = [1, 2, 3]
```

Array is a function

Given a data type

`A`

`[]` produces another data type

`[A]`

What's the domain/target?

```
[]: Type -> Type
```

Forgetting anything?

Yah

The collection `Type` of data types has more than just types.

It has functions between data types.

Surely `[]` should do something with those too?

What [ ] does with functions

Give data types

```
A, B
```

and a function

```
f: A -> B
```

is there a completely naive way for `Array` to induce a function:

```
[A] -> [B]
```

map!

```
func map(xs: [A], f: A -> B) -> [B] {
  var ys: [B] = []
  for (i, x) in xs {
    ys[i] = f(x)
  }
  return ys
}
```

map!

A little bit of currying gives an alternative signature of `map`:

```
map:(A -> B) -> ([A] -> [B])
```

How common is `map`?

Dictionaries have maps

```
Dictionary<K, A> -> Dictionary<K, B>
```

Sets have maps

```
Set<A> -> Set<B>
```

Trees have maps

```
Tree<A> -> Tree<B>
```

Stacks have maps, queues have maps, graphs have maps, …

How many more examples do we need before we stop the madness and just give this thing a name

- We'll have a language for discussing
- We'll gain clarity around current uses
- We'll see that our current uses are misleading

Functor

A **functor** is a function on the collection of types

```
F: Type -> Type
```

Given a type A, it produces a new type `F(A)`.

Given a function `f: A -> B`, it produces a new function:

```
F(f): F(A) -> F(B)
```

Fine print

```
F(id_A) = id_F(A)

F(g * f) = F(g) * F(f)
```

Examples

- `Array`
- `Dictionary`
- `Tree`
- `Set`
- `Graph`

More examples

- Tuples
- `Maybe<A>`
- `Either<A, B>`
- Monads
- `Func(A, -)`

The Math

Our collection of data types `Type` is an example of what is known in mathematics as a **Category**.

A category is a collection of **objects** and **morphisms** between those objects with a few conditions.

A **functor** is a function between categories satisfying those conditions we saw earlier.

CS is basically the study of the category `Type`. Math has more categories it considers.

Outro

Category theory re-framed all of math in terms of morphisms.

They are the thing you really want to study.

Category theory helps to do the same thing with CS.