CIS 194

TYPES

TYPES

> EVERYTHING THAT IS SYNTACTICALLY LEGAL, THAT THE COMPILER WILL ACCEPT, WILL EVENTUALLY WIND UP IN YOUR CODEBASE. AND THAT'S WHY I THINK STATIC TYPING IS SO VALUABLE BECAUSE IT CUTS DOWN ON WHAT CAN MAKE IT PAST THOSE HURDLES THERE.

John Carmack

A WORLD WITHOUT TYPES

```
def double(x):
    return 2 * x
```

Maybe `double(5) == 10`? Wrong.

How about `double(“hello”) == “hellohello”`?

Actually the correct answer is (obviously):
`double([5, “a”]) == [5, “a”, 5, “a”]`.

HW1 CONTINUED…

```
{-
Replace `undefined` with your phone number.
-}

phoneNumber = undefined
```

HW1 CONTINUED…

- `phoneNumber    = 2158985000`
- `phoneNumber'   = "2158985000"`
- `phoneNumber''  = (2,1,5,8,9,8,5,0,0,0)`
- `phoneNumber''' = [0,0,0,5,8,9,8,5,1,2]`

HW1 CONTINUED…

```
{-
Replace `undefined` with your phone number.
-}

phoneNumber :: Int
phoneNumber = undefined
```

Clearly should be `phoneNumber = 2158985000`

TYPE ANNOTATIONS

WHAT TYPES HAVE WE SEEN SO FAR?

- `Int`
- `Char`
- `String`
- `Maybe String`
- etc.

WHAT TYPES HAVE WE SEEN SO FAR?

```
favoriteNumber :: Int
favoriteNumber = 194

firstLetterOfName :: Char
firstLetterOfName = head "Palmer"

hello194 :: String
hello194 = hello "194"

githubUsername :: Maybe String
githubUsername = Just "pzp1997"
```

GENERICS

LENGTH OF A LIST

```
lengthIntList :: [Int] -> Int
lengthIntList xs =
  if null xs then
    0
  else
    1 + lengthIntList (tail xs)

lengthIntList [1,2,3] –> 3
lengthIntList "abc" –> WON’T COMPILE!
```

LENGTH OF A LIST

```
lengthString :: String -> Int
lengthString xs =
  if null xs then
    0
  else
    1 + lengthString (tail xs)

lengthString "abc"–> 3
```

LENGTH OF A LIST

```
lengthIntList :: [Int] -> Int
lengthIntList xs =
  if null xs then 0
  else 1 + lengthIntList (tail xs)

lengthString :: String -> Int
lengthString xs =
  if null xs then 0
  else 1 + lengthString (tail xs)
```

LENGTH OF A LIST

```
lengthIntList :: [Int] -> Int
lengthIntList xs =
  if null xs then 0
  else 1 + lengthIntList (tail xs)

lengthString :: String -> Int
lengthString xs =
  if null xs then 0
  else 1 + lengthString (tail xs)
```

LENGTH OF A LIST

```
length :: [???] -> Int
length xs =
  if null xs then
    0
  else
    1 + length (tail xs)
```

The ??? could be replaced by ANY type.

How can we use the type system to express that?

ANSWER: DAILY DOUBLE

```
length :: [a] -> Int
```

- `a` is a placeholder
- `a` is bound when we apply the function
- No restrictions on which types that can be bound to `a`
- b or `alexTrebek` would work too

WHAT IS BINDING ANYWAY?

Suppose we have a definition

```
twoOfAKind :: a -> a -> (a, a)
twoOfAKind x y = (x, y)
```

```
twoOfAKind 1 2 –> (1, 2)
twoOfAKind 'a' 'b' -> ('a', ‘b')
twoOfAKind 'a' 1 ->  WON’T COMPILE! WHY?
```

APPLY THE ARGUMENTS ONE AT A TIME

```
twoOfAKind :: a -> a -> (a, a)
twoOfAKind x y = (x, y)

partial = twoOfAKind 'a'

partial' :: Char -> (Char, Char)
partial' y = ('a', y)

partial 1
```

BUT 1 is not a Char!

PARAMETERIZED TYPES

MAYBE VS. MAYBE INT

```
maybeAdd mx my =
  if isJust mx && isJust my then
    Just (fromJust mx + fromJust my)
  else
    Nothing
```

What should the type of `maybeAdd` be?

How about `Maybe -> Maybe -> Maybe`?

MAYBE VS. MAYBE INT

```
maybeAdd :: Maybe -> Maybe -> Maybe
```

But then we could do

```
maybeAdd (Just 1) (Just “hello”)
```

which doesn’t make sense…

MAYBE VS. MAYBE INT

- Knowing that a value is a `Maybe` is not enough.
- We need to be able to specify the type of value stored inside of the `Maybe` too.
- In other words how can we differentiate between `Maybe` of an `Int` and a `Maybe` of a `String` at the type level?!

- **Solution:** `Maybe Int`

BREAKING DOWN MAYBE INT

- Maybe is a "type constructor"
- Maybe is parameterized by type of value stored inside it

```
myFavoriteNumber :: Maybe Int
myFavoriteNumber = Just 194
```

- In the case above `Int` is the parameter to Maybe

```
myLeastFavoriteNumber :: Maybe a
myLeastFavoriteNumber = Nothing
```

- In the case above the parameter could be anything!

SO IS MAYBE A TYPE?

- Is `Int` a type?
- Is Maybe `Int` a type?
- How about Maybe by itself?

TYPE OF A FUNCTION

WHAT IS THE TYPE OF A FUNCTION?

- Let's make a type called `Function`

```
isEven :: Function
isEven x = x `mod` 2 == 0
```

- Good first attempt
- Not a lot of type safety

WHAT IS THE TYPE OF A FUNCTION?

- We need to add more information to our type
- Let's add type of argument and return value as parameters
- Our type is now `Function arg ret`

```
isEven :: Function Int Bool
isEven x = x `mod` 2 == 0
```

DYADIC FUNCTIONS

Maybe `Function arg1 arg2 ret?`

```
repeatIt :: Function Int String String
repeatIt timesToRepeat snippet = ...
```

Actually `Function arg1 (Function arg2 ret)`

```
repeatIt :: Function Int (Function String String)
repeatIt timesToRepeat snippet = ...
```

SYNTACTIC SUGAR FTW

- `Function arg ret` is not the clearest syntax
- Haskell defines an infix type constructor `(->)` which is synonymous to `Function`
- `Function arg ret` becomes `arg -> ret`
- `isEven :: Int -> Bool`

SYNTACTIC SUGAR FTW

- `Function arg1 (Function arg2 ret)` becomes `arg1 -> (arg2 -> ret)`
- Or since (->) is right associative, `arg1 -> arg2 -> ret`
- `repeatIt :: Int -> String -> String`