Java 1995 generics lambda

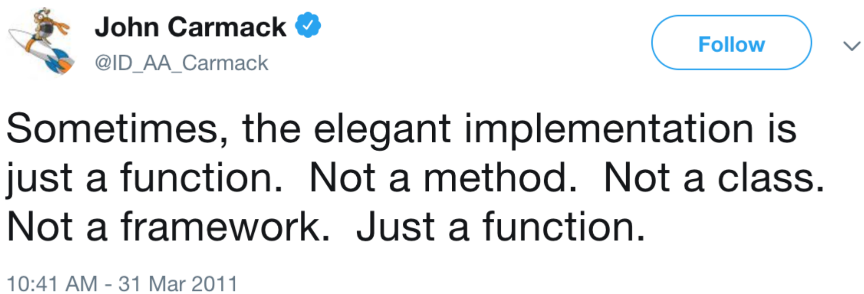
Haskell 1992

A crash course in Haskell

quake / doom

FPS

Functions and Programming



Carmack on Functions

What is Haskell?

A **typed**, **lazy**, **purely functional** programming language

Haskell = λ-calculus ++

- better syntax
- types
- built-in features
    - booleans, numbers, characters
    - records (tuples)
    - lists
    - recursion
    - ...

Why Haskell?

Haskell programs tend to be *simple* and *correct*

QuickSort in Haskell

```haskell
sort          :: (Ord a) => [a] -> [a]
sort []       = []
sort (x:xs) = sort ls ++ [x] ++ sort rs
  where
    ls      = [ l | l <- xs, l <= x ]
    rs      = [ r | r <- xs, x <  r ]
```

Goals for this week

1. Understand the code above
2. Understand what **typed**, ~~lazy~~, and **purely functional** means (and why it's cool)

Haskell vs λ-calculus: similarities

(1) Programs

$e ::= x \mid \backslash x \to e \mid e_1\ e_2$

A program is an **expression** (*not* a sequence of statements)

It **evaluates** to a value (it *does not* perform actions)

- λ:

  ```
  (\x -> x) apple     -- =~> apple
  ```

- **Haskell**:

  ```
  (\x -> x) "apple"   -- =~> "apple"
  ```

(2) Functions

Functions are *first-class values*:

- can be *passed as arguments* to other functions
- can be *returned as results* from other functions
- can be *partially applied* (arguments passed *one at a time*)

```
(\x -> (\y -> x (x y))) (\z -> z + 1) 0   -- =~> ???
```

But: unlike λ-calculus, not everything is a function!

(3) Top-level bindings

Like in Elsa, we can *name* terms to use them later

Elsa:

```
let T    = \x y -> x
let F    = \x y -> y

let PAIR = \x y -> \b -> ITE b x y
let FST  = \p -> p T
let SND  = \p -> p F

eval fst:
 FST (PAIR apple orange)
 =~> apple
```

Haskell:

```haskell
haskellIsAwesome = True

pair = \x y -> \b -> if b then x else y
fst = \p -> p haskellIsAwesome
snd = \p -> p False

-- In GHCi:
> fst (pair "apple" "orange")   -- "apple"
```

The names are called **top-level variables**

Their definitions are called **top-level bindings**

Better Syntax: Equations and Patterns

You can define function bindings using **equations**:

```
pair x y b = if b then x else y -- same as: pair = \x y b -> ...
fst p      = p True             -- same as: fst = \p -> ...
snd p      = p False            -- same as: snd = \p -> ...
```

A *single* function binding can have *multiple* equations with different **patterns** of parameters:

```
pair x y True  = x  -- If 3rd arg matches True,
                    -- use this equation;
pair x y False = y  -- Otherwise, if 3rd arg matches False,
                    -- use this equation.
```

At run time, the first equation whose pattern matches the actual arguments is chosen

For now, a **pattern** is:

- a *variable* (matches any value)
- or a *value* (matches only that value)

Same as:

```
pair x y True  = x  -- If 3rd arg matches True,
                    -- use this equation;
pair x y b     = y  -- Otherwise, use this equation.
```

Same as:

```
pair x y True  = x
pair x y _     = y
```

QUIZ

Which of the following definitions of `pair` is **incorrect**?

A. `pair x y = \b -> if b then x else y`

B. `pair x = \y b -> if b then x else y`

C.

```
pair x _ True  = x
pair _ y _     = y
```

D.

```
pair x y b     = x
pair x y False = y
```

E. all of the above

Equations with guards

An equation can have multiple guards (Boolean expressions):

```
cmpSquare x y  |  x > y*y   =  "bigger :)"
               |  x == y*y  =  "same :|"
               |  x < y*y   =  "smaller :("
```

Same as:

```
cmpSquare x y  |  x > y*y   =  "bigger :)"
               |  x == y*y  =  "same :|"
               |  otherwise =  "smaller :("
```

Recusion

Recursion is built-in, so you can write:

```
sum n = if n == 0
          then 0
          else n + sum (n - 1)
```

or you can write:

```
sum 0 = 0
sum n = n + sum (n - 1)
```

The scope of variables

Top-level variable have **global** scope, so you can write:

```haskell
message = if haskellIsAwesome          -- this var defined below
            then "I love CSE 130"
            else "I'm dropping CSE 130"

haskellIsAwesome = True
```

(f 3) evaluate to ?

(A) Type Err

(B) True

(C) False

(D) Infinite Loop!

Or you can write:

```haskell
-- What does f compute?
f 0 = True                     -- is_even
f n = g (n - 1) -- mutual recursion!

g 0 = False                    -- is_odd
g n = f (n - 1) -- mutual recursion!
```

f 3
↳ g 2
↳ f 1
↳ g 0
↳ False

Is this allowed?

```haskell
haskellIsAwesome = True

haskellIsAwesome = False -- changed my mind
```

Local variables

You can introduce a *new* (local) scope using a **`let`**-expression:

```haskell
sum 0 = 0
sum n = let n' = n - 1
        in n + sum n'  -- the scope of n' is the term after in
```

Syntactic sugar for nested **`let`**-expressions:

```haskell
sum 0 = 0
sum n = let
          n'   = n - 1
          sum' = sum n'
        in n + sum'
```

If you need a variable whose scope is an equation, use the **where** clause instead:

```haskell
cmpSquare x y  |  x > z   =  "bigger :)"
               |  x == z  =  "same :|"
               |  x < z   =  "smaller :("
  where z = y*y
```

Types

What would *Elsa* say?

```
let WEIRDO = ONE ZERO
```

What would *Python* say?

```
def weirdo():
  return 0(1)
```

What would *Java* say?

```
void weirdo() {
  int zero;
  zero(1);
}
```

In *Haskell* every expression either **has a type** or is **ill-typed** and rejected statically (at compile-time, before execution starts)

- like in Java
- unlike λ-calculus or Python

```
weirdo = 1 0     -- rejected by GHC
```