# Lambda Calculus

For when functional programming isn't hard enough

By Kevin Fisher

# 0: Introduction

# Functional programming is great...

- No Side Effects
- First-Class Functions
- Higher-Order Functions
- Currying
- Extremely readable code

```
obvious = mapM ((. snd) . (,)) =<< fst
```

# ...but it's too easy

Pattern matching?
That's cheating

Names? Distracting

Numbers? Math?
We don't need those

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

# Let's make a new language

- Things we need:
  - Functions
  - Variables
  - Function applications
- That's it!

# 1: Lambda Calculus

# Why do we care about lambda calculus?

- Due to its simplicity, helpful in proofs
  - Equivalent to Turing machines
- Emphasizes the foundations of functional programming
  - Like assembly, but less (directly) useful
- Because it practices programming problem solving
  - Also, it’s fun!
    - For me

# What is lambda calculus?

Let $\Lambda$ (upper case lambda) be the set of all lambda expressions

We can describe the contents of $\Lambda$ like so:

- If `x` is a variable, then `x` is in $\Lambda$ (also written `x` $\in \Lambda$)
- If `x` is a variable and `M` $\in \Lambda$, then `(λx. M)` $\in \Lambda$
- If `M` $\in \Lambda$ and `N` $\in \Lambda$, then `(M N)` $\in \Lambda$

Note: `x` here represents any variable, in our case any one lowercase letter (`a`, `b`, `c`, `x`, `y`, `z`, etc.)

# What is lambda calculus?

Let $\Lambda$ (upper case lambda) be the set of all lambda expressions

We can describe the contents of $\Lambda$ like so:

- Variable: `x`
- Abstraction: `(λx. M)`
- Application: `(M N)`

Note: `x` here represents any variable, in our case any one lowercase letter (`a`, `b`, `c`, `x`, `y`, `z`, etc.)

# The most important operation: β-reduction

$$((\lambda x.\ M)\ E) \xrightarrow[\text{reduces to}]{\text{equivalent to}} M[x := E]$$

We won't be covering the formal definition of substitution, if you're interested check out [Wikipedia: Lambda calculus]

# Some simple functions in lambda calculus

```
ID := (λx. x)

CONST := (λc. (λx. c))
```

# Some simple functions in lambda calculus

To avoid parentheses, assume abstractions extend as far right as possible

```
ID := λx. x

CONST := λc. λx. c
```

# Some simple functions in lambda calculus

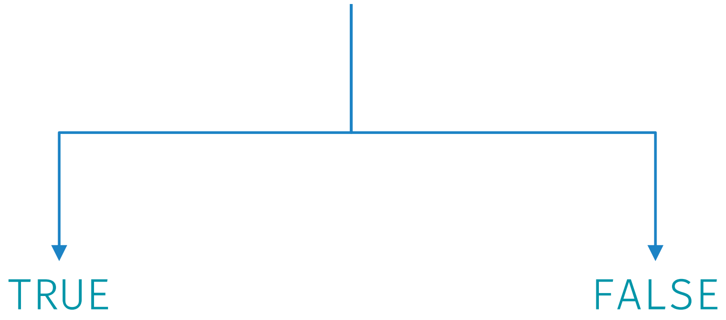
```
ID := \x. x

CONST := \c. \x. c
```

Because there's no λ key, we'll use \

# 2: Booleans

# What is a Boolean?

One way we can describe Booleans is as a binary choice



# Modeling Booleans

```
TRUE  := \x. \y. x

FALSE := \x. \y. y
```

# Modeling Booleans

Collapse “curried” parameters into one abstraction

```
TRUE  := \xy. x

FALSE := \xy. y
```

# The NOT function

| p | NOT p |
|---|---|
| TRUE | FALSE |
| FALSE | TRUE |

# The NOT function

```
NOT := \p. p FALSE TRUE
```

| p | NOT p |
|---|---|
| TRUE | FALSE |
| FALSE | TRUE |

# Check it for yourself!

- Use the template
  - https://github.com/kfish610/lambda-calc-presentation
- Open it in VSCode
- Right click "lambda-calc-toolbox-1.0.2.vsix"
- Select "Install Extension VSIX"
- Open "2-Boolean.lcs"

# Your turn!

Write the following functions

```
AND  := \pq.
OR   := \pq.
```

| p | q | AND p q |
| --- | --- | --- |
| TRUE | TRUE | TRUE |
| TRUE | FALSE | FALSE |
| FALSE | TRUE | FALSE |
| FALSE | FALSE | FALSE |

| p | q | OR p q |
| --- | --- | --- |
| TRUE | TRUE | TRUE |
| TRUE | FALSE | TRUE |
| FALSE | TRUE | TRUE |
| FALSE | FALSE | FALSE |

# Solution: AND and OR

| p | q | AND p q |
|---|---|---|
| TRUE | TRUE | TRUE |
| TRUE | FALSE | FALSE |
| FALSE | TRUE | FALSE |
| FALSE | FALSE | FALSE |

| p | q | OR p q |
|---|---|---|
| TRUE | TRUE | TRUE |
| TRUE | FALSE | TRUE |
| FALSE | TRUE | TRUE |
| FALSE | FALSE | FALSE |

```
AND := \pq. p q p

OR  := \pq. p p q
```

# 3: Numbers

# Repeated applications

If we have some function $f$, we can...

- Apply it: $f(x)$
- Apply it twice: $f\big(f(x)\big) = (f \circ f)(x)$
- Apply it $n$ times: $(f \circ \cdots \circ f)(x)$
- Apply it $0$ times: $x$

# Function exponentiation

When we write $f^n(x)$, we mean "apply $f$ to $x$, $n$ times"

- $f^0(x) = x$
- $f^1(x) = f(x)$
- $f^2(x) = f(f(x))$
- Etc.

# Notable features of function exponentiation

$$f(f^n(x)) = f^{n+1}(x)$$

$$f^m(f^n(x)) = f^{n+m}(x)$$

$$(f^n)^m(x) = f^{n \cdot m}(x)$$

# Modeling numbers: Church numerals

```
0  := \fx. x
1  := \fx. f x
2  := \fx. f (f x)
3  := \fx. f (f (f x))
4  := \fx. f (f (f (f x)))
```

# The SUCC function

$$f(f^n(x)) = f^{n+1}(x)$$

# The SUCC function

```
SUCC := \n. \fx. f (n f x)
```

$$f(f^n(x)) = f^{n+1}(x)$$

# Aside: Weak Head Normal Form

```
(SUCC 1)

((λn. (λf. (λx. (f ((n f) x))))) 1)

(λf. (λx. (f ((1 f) x))))
```

# Aside: Normal Form

```
(SUCC 1)

((λn. (λf. (λx. (f ((n f) x))))) 1)

(λf. (λx. (f ((1 f) x))))

(λf. (λx. (f (((λf. (λx. (f x))) f) x))))

(λf. (λx. (f ((λx. (f x)) x))))

(λf. (λx. (f (f x))))
```

# Your turn!

Write the following functions:

```
ADD  := \mn.

MULT := \mn.
```

$$f^m(f^n(x)) = f^{n+m}(x)$$

$$(f^n)^m(x) = f^{n \cdot m}(x)$$

# Solution: ADD and MULT

```
ADD  := \mn. \fx. m f (n f x)
MULT := \mn. \fx. m (n f) x
```

$$f^m(f^n(x)) = f^{n+m}(x)$$

$$(f^n)^m(x) = f^{n \cdot m}(x)$$

# Solution: EXP

$$f^n(x) \longleftrightarrow$$ `n f x`

$$(\square^m)^n \longleftrightarrow$$ `n m`

```
EXP := \mn. n m
```

# 4: Pairs

# Modeling pairs

```
PAIR := \xy. \f. f x y
```

# The FIRST and SECOND functions

```
FIRST  := \p. p (\xy. x)

SECOND := \p. p (\xy. y)
```

# The FIRST and SECOND functions

```
FIRST  := \p. p TRUE
SECOND := \p. p FALSE
```

# Your turn: SHIFTEVAL

SHIFTEVAL takes in a function $f$ and a pair $(x, y)$:

$$(x, y) \longrightarrow (y, f(y))$$

This may seem arbitrary, but it will be very useful!

```
SHIFTEVAL := \fp.
```

# Solution: SHIFTEVAL

```
SHIFTEVAL := \fp. PAIR (SECOND p) (f (SECOND p))
```

# What can we do with `SHIFTEVAL`?

Since it takes in a function *first*, and all functions are implicitly curried, we can specialize this for specific functions.

```
SHIFTSUCC := SHIFTEVAL SUCC

SHIFTNOT  := SHIFTEVAL NOT
```

# Your turn: PRED and SUB

Write the predicate function (n – 1):

```
PRED := \n.
```

And the subtract function (n – m):

```
SUB := \mn.
```

These were too hard before, but `SHIFTEVAL` will help us write `PRED`

# Solution: PRED and SUB

```
PRED := \n. FIRST (n (SHIFTEVAL SUCC) (PAIR 0 0))
```

```
SUB := \mn. n PRED m
```

# 5: Predicates

# What is a predicate?

- A predicate is just a function that takes in some value(s) and returns a Boolean.
  - `CONST_TRUE` and `CONST_FALSE` are technically predicates, though they're very boring ones

# Your turn!

Write the following predicates:

```
ISZERO := \n.

LEQ := \mn.
```

# Solution: `ISZERO` and `LEQ`

```
ISZERO := \n. n (CONST FALSE) TRUE

LEQ := \mn. ISZERO (SUB m n)
```

# 6: Lists

# Making lists from pairs

- A linked list is essentially just a pair of two things:
  a value, and the rest of the list
- So, we can model lists using composed pairs
  - This is a very common way to implement lists in functional languages

# Important definitions

```
CONS := PAIR
NIL  := \x. TRUE

HEAD := FIRST
TAIL := SECOND

# If the list is a pair, this will always give FALSE
# If the list is a NIL, NIL always returns TRUE
NULL := \l. l (\xy. FALSE)
```

# Using lists

```
LIST1 := CONS 0 (CONS 0 NIL)

LIST2 := CONS 1 (CONS 2 (CONS 3 NIL))
```

$$\text{HEAD LIST1} \longrightarrow 0$$

$$\text{HEAD (TAIL LIST2)} \longrightarrow 2$$

$$\text{HEAD (TAIL (TAIL LIST2))} \longrightarrow 3$$

# Your turn!

Write the following functions:

Get the element at n: `INDEX := \nl.`

A list with x repeated n times: `REPLICATE := \nx.`

# Solution: INDEX and REPLICATE

```
INDEX := \nl. HEAD (n TAIL l)

REPLICATE := \nx. n (CONS x) NIL
```

# 7: Recursion

# Let's write a recursive multiply

- We might think we can just use the function in itself

```
MULT_REC := \nm. (ISZERO n) 0 (ADD m (MULT_REC (PRED n) m))
```

- But there's a couple of problems with this:
  - Function names are a convenience we invented, they don't exist in "pure" lambda calculus
  - The reducer needs to do recursive calculations sometimes, which break horribly with recursive definitions

# Attempt two: passing it as a parameter

- Let's just have the function provided to us as `r`

```
MULT_REC := \rnm. (ISZERO n) 0 (ADD m (r r (PRED n) m))
```

- We would call it like this:

```
MULT_REC MULT_REC 2 3
```

- This works, but it's pretty ugly. Can we do better?

# A fixed-point of our function

- Ideally, we'd like some function Y that "provides itself":

$$Y\ f \longrightarrow f\ (Y\ f)$$

- Then we could just write our function as: (note only 1 `r`)

```
MULT_REC := \rnm. (ISZERO n) 0 (ADD m (r (PRED n) m))
```

- And call it like:

```
Y MULT_REC 2 3
```

# The Y combinator

```
Y := \f. (\x. f (x x)) (\x. f (x x))
```

# The Y combinator

```
Y := \f. (\x. f (x x)) (\x. f (x x))
```

```
Y f
 ↓
(\x. f (x x)) (\x. f (x x))
 ↓
f ((\x. f (x x)) (\x. f (x x)))
 ↕
f (Y f)
```

# Your turn!

Write the following functions:

- ▷ Standard factorial function: `FACTORIAL := \rn.`
- ▷ List that repeats `x` infinitely: `REPEAT := \rx.`
- ▷ List that looks like [`x`, `f x`, `f (f x)`, ...]: `ITERATE := \rfx.`
- ▷ Apply `f` to each element of the list: `MAP := \rfl.`

# Solutions

- `FACTORIAL := \rn. (ISZERO n) 1 (MULT n (r (PRED n)))`
- `REPEAT := \rx. CONS x (r x)`
- `ITERATE := \rfx. CONS x (r f (f x))`
- `MAP := \rfl. (NULL l) NIL (CONS (f (HEAD l)) (r f (TAIL l)))`

Any questions?