THE LAMBDA CALCULUS AND THE JAVASCRIPT

orb@nostacktrace.com
Norman Richards

orb@nostacktrace.com

Norman Richards

WHY LAMBDA CALCULUS?

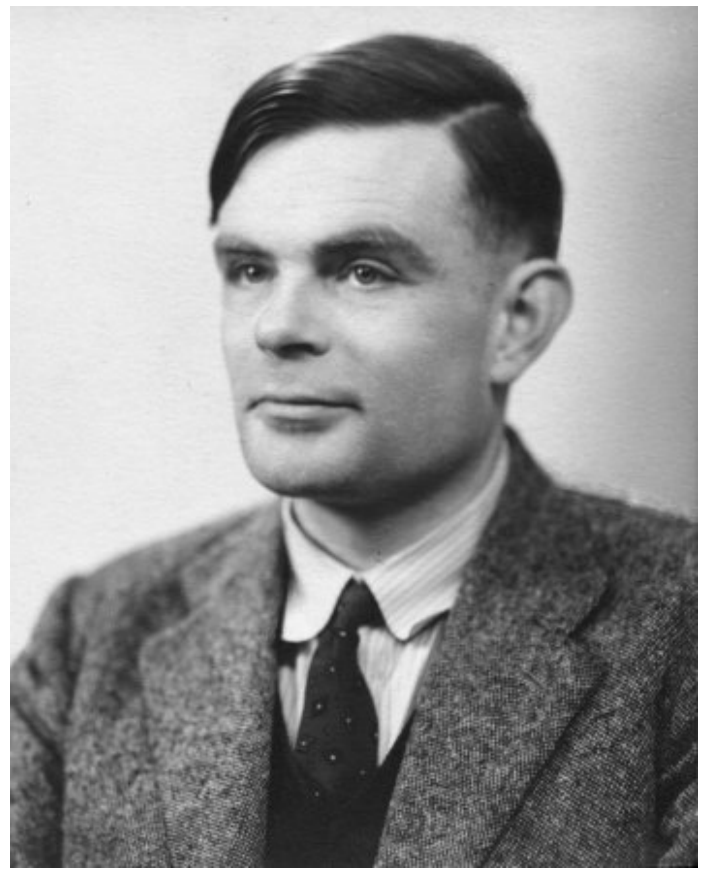
ALONZO CHURCH

Created lambda calculus, the basis for functional programming.

http://en.wikipedia.org/wiki/Alonzo_Church

ALAN TURING

Invented the turing machine, an imperative computing model



http://en.wikipedia.org/wiki/Alan_Turing

We spend all our time studying this guy's work (because he's awesome)

But if we want to think functionally, we should pay attention to this guy too (because he's also awesome)

orb@nostacktrace.com

Norman Richards

THE PREMISE

Functional programming is good

Using functional techniques will make your code better

Understanding lambda calculus will improve your functional

orb@nostacktrace.com
Norman Richards

A LAMBDA EXPRESSION

BOUND VARIABLE

This is the argument of the function

$$\lambda x . x$$

BODY

A lambda expression that uses the bound variable and represents the function value

A LAMBDA EXPRESSION

$\lambda x.x$

```
function(x) {
   return x
}
```

orb@nostacktrace.com
Norman Richards

orb@nostacktrace.com
Norman Richards

Not a lambda expression

$\lambda y.x$

Free variable

This variable isn't bound, so this expression is meaningless

orb@nostacktrace.com
Norman Richards

A LAMBDA EXPRESSION

```
λy.x
```

```
function(y) {
    return x
}
```

These are meaningless

orb@nostacktrace.com
Norman Richards

A LAMBDA EXPRESSION

PARENTHESIS

Not needed, but used for clarity here

$$\lambda x.(\lambda.y.x)$$

BOUND OR FREE?

x is bound in the outer expression and free in inner expression.

orb@nostacktrace.com
Norman Richards

A LAMBDA EXPRESSION

λx.λ.y.x

```
function(x) {
    return function(y) {
        return x
    }
}
```

orb@nostacktrace.com
Norman Richards

LAMBDA APPLICATION

UNICORNS DON'T EXIST

There are no assignments or external references. We'll use upper-case words to stand for some valid lambda expression, but it must mean something.

$$\lambda x.x \text{ UNICORN}$$

APPLICATION BY SUBSTITUTION

Apply UNICORN to the expression, substituting UNICORN for every occurrence of the bound value

orb@nostacktrace.com
Norman Richards

LAMBDA APPLICATION

THE I (IDENTITY) COMBINATOR

Combinatory logic predates lambda calculus, but the models are quite similar. Many lambda expressions are referred to by their equivalent combinator names.

```
λx.x UNICORN
→ UNICORN
```

```
function(x) {
    return x
}(UNICORN);

→ UNICORN
```

orb@nostacktrace.com
Norman Richards

LAMBDA APPLICATION

THE K (CONSTANT) COMBINATOR

This function takes an input argument and returns a function that returns that same value

λx.λy.x UNICORN
→ λy.UNICORN

λy.UNICORN MITT
→ UNICORN

ALWAYS A UNICORN

No matter what argument is passed in, the result is always the same.

http://leftaction.com/action/ask-kansas-mitt-romney-unicorn

LAMBDA APPLICATION

SELF-APPLICATION

This function takes a function and applies it to itself.

$$\lambda s.(s\ s)$$

```
function(f) {
    return f(f);
}
```

orb@nostacktrace.com
Norman Richards

orb@nostacktrace.com

LAMBDA APPLICATION

Substitute λx.x for s

For clarity,
rename x as x_1

$$\lambda s.(s\ s)\ \lambda x.x$$
$$\rightarrow \lambda x.x\ \lambda x.x$$
$$\rightarrow \lambda x_1.x_1\ \lambda x.x$$
$$\rightarrow \lambda x.x$$

Substitute λx.x
for x_1.

```
function(f) {
    return f(f);
}(function (x) {
    return x;
  });
```

Norman Richards

orb@nostacktrace.com
Norman Richards

INFINITE APPLICATION

λs.(s s) λs.(s s)

→ λs.(s s) λs.(s s)

→ λs.(s s) λs.(s s)

→ ...

THE HALTING PROBLEM

Just as you'd expect, there's no algorithm to determine whether or not a given lambda expression will terminate

orb@nostacktrace.com
Norman Richards

TURING COMPLETE

```
I: λx.x
K: λx.λy.x
S: λx.λy.λz.x z (y z)
```

http://en.wikipedia.org/wiki/SKI_combinator_calculus

orb@nostacktrace.com
Norman Richards

ACTUALLY, JUST TWO LAMBDAS

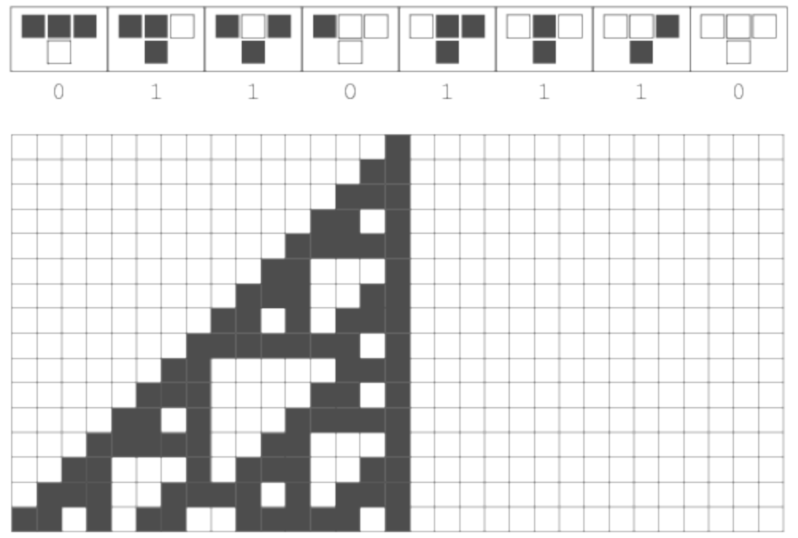
```
S K K UNICORN
→ λx.λy.λz.(x z (y z)) K K UNICORN
→ λy.λz.(K z (y z)) K UNICORN
→ λz.(K z (K z)) UNICORN
→ K UNICORN (K UNICORN)
→ λx.λy.x UNICORN (K UNICORN)
→ λy.UNICORN (K UNICORN)
→ UNICORN
```

Thus, S K K is computationally equivalent to identity

OTHER TURING COMPLETE THINGS

MAGIC: THE GATHERING

http://www.toothycat.net/~hologram/Turing/



C++ TEMPLATES

http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.3670

THE NUMBER 110

http://mathworld.wolfram.com/Rule110.html

orb@nostacktrace.com
Norman Richards

TURING TARPIT

Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy.

Alan Perlis

http://pu.inf.uni-tuebingen.de/users/klaeren/epigrams.html

orb@nostacktrace.com
Norman Richards

orb@nostacktrace.com

Norman Richards

CURRYING

A FUNCTION THAT RETURNS A FUNCTION

.. that returns a function that returns a function. You can think of this as a function of three arguments.

$\lambda x.\lambda y.\lambda z.\mathrm{UNICORN}\ X\ Y\ Z$

```
function(x) {
  return function(y) {
    return function (z) {
      return UNICORN;
    }
  }
}(X)(Y)(Z);

function(x,y,z) {
   return UNICORN;
}(X,Y,Z);
```

CURRYING

SOMETIMES ABBREVIATED

Sometimes, you'll see lambda expressions written like this for clarity. The arguments are still curried.

λx y z.UNICORN X Y Z

→ λy z.UNICORN Y Z

→ λz.UNICORN Z

→ UNICORN

HASKELL CURRY

Curry studied combinatory logic, a variant of lambda calculus.

http://www.haskell.org/haskellwiki/Haskell_Brooks_Curry

orb@nostacktrace.com

Norman Richards

DATA STRUCTURES

CHURCH PAIR ENCODING

PAIR takes two arguments, x and y and returns a function that takes another function and applies x and y to it.

PAIR: λx.λy.λf.(f x y)

BUILDING UP STATE

PAIR takes two arguments, x and y and returns a function that takes another function and applies x and y to it.

orb@nostacktrace.com

Norman Richards

DATA STRUCTURES

```
PAIR:   λx.λy.λf.(f x y)
FIRST:  λx.λy.x
SECOND: λx.λy.y
```

ACCESSOR FUNCTIONS

FIRST and SECOND are functions that can be passed to pair. They take two arguments and return the first and second, respectively.

orb@nostacktrace.com

Norman Richards

orb@nostacktrace.com
Norman Richards

DATA STRUCTURES

```
PAIR:   λx.λy.λf.(f x y)

DOGCAT: PAIR DOG CAT
        → λx.λy.λf.(f x y) DOG CAT
        → λy.λf.(f DOG y) CAT
        → λf.(f DOG CAT)
```

orb@nostacktrace.com
Norman Richards

DATA STRUCTURES

```
DOGCAT: λf.(f DOG CAT)
FIRST:  λx.λy.x
SECOND: λx.λy.y

DOGCAT FIRST
→ λf.(f DOG CAT) λx.λy.x
→ λx.λy.x DOG CAT
→ λy.DOG CAT
→ DOG
```

orb@nostacktrace.com
Norman Richards

DATA STRUCTURES

```
DOGCAT: λf.(f DOG CAT)
FIRST:  λx.λy.x
SECOND: λx.λy.y

DOGCAT SECOND
→ λf.(f DOG CAT) λx.λy.x
→ λx.λy.x DOG CAT
→ λy.y CAT
→ CAT
```

orb@nostacktrace.com
Norman Richards

DATA STRUCTURES

```
RGB:   λr.λg.λb.λf.(f r g b)

RED:   λr.λg.λb.r
GREEN: λr.λg.λb.g
BLUE:  λr.λg.λb.b

BLACK_COLOR: RGB 255 255 255
WHITE_COLOR: RGB 0 0 0
```

NUMBERS?

We haven't seen how to construct numbers yet, but for the moment imagine that we did.

orb@nostacktrace.com

DATA STRUCTURES

```
AVG: λx.λy.???

MIX_RGB:
    λc₁.λc₂.λf.
       (f (AVG (c₁ RED)   (c₂ RED))
          (AVG (c₁ GREEN)(c₂ GREEN))
          (AVG (c₁ BLUE) (c₂ BLUE)))
```

AVERAGE?

If we did have numbers, we could probably do math like simple averaging

OO LAMBDA?

MIX_RGB takes two colors and returns a new color that is the mix of the two

Norman Richards

orb@nostacktrace.com
Norman Richards

CONDITIONALS

$$\text{COND: } \lambda e_1.\lambda e_2.\lambda c.(c\ e_1\ e_2)$$

$$\text{TRUE: } \lambda x.\lambda y.x$$

$$\text{FALSE: } \lambda x.\lambda y.y$$

LOOK FAMILIAR?

Think of a boolean condition as a PAIR of values. TRUE selects the first one and FALSE selects the second one.

orb@nostacktrace.com
Norman Richards

CONDITIONALS

COND DEAD ALIVE QUBIT

→ $\lambda e_1.\lambda e_2.\lambda c.(c\ e_1\ e_2)$ DEAD ALIVE QUBIT

→ QUBIT DEAD ALIVE

→ TRUE DEAD ALIVE

→ $\lambda x.\lambda y.x$ DEAD ALIVE

→ DEAD

→ FALSE DEAD ALIVE

→ $\lambda x.\lambda y.y$ DEAD ALIVE

→ ALIVE

https://www.coursera.org/course/qcomp

orb@nostacktrace.com
Norman Richards

BOOLEAN LOGIC

```
NOT: λx.(COND FALSE TRUE x)
AND: λx.λy.(COND y FALSE x)
 OR: λx.λy.(COND TRUE y x)
```

NUMBERS

```
ZERO: λx.x
SUCC: λn.λs.(s false n)

ONE:  SUCC ZERO
      → λs.(s false ZERO)
TWO:  SUCC ONE
      → λs.(s false ONE)
      → λs.(s false λs.(s false ZERO))
```

PEANO NUMBERS

PEANO numbers are based on a zero function and a successor function

orb@nostacktrace.com
Norman Richards

orb@nostacktrace.com
Norman Richards

NUMBERS

```
   ZERO: λx.x
ISZERO: λn.(n FIRST)
```

```
ISZERO ZERO
→ λn.(n FIRST) λx.x
→ λx.x FIRST
→ FIRST
→ TRUE
```

orb@nostacktrace.com
Norman Richards

NUMBERS

```
ISZERO: λn.(n FIRST)
   ONE: λs.(s FALSE ZERO)
```

```
ISZERO ONE
→ λn.(n FIRST) λs.(s FALSE ZERO)
→ λs.(s FALSE ZERO) FIRST
→ (FIRST FALSE ZERO)
→ FALSE
```

orb@nostacktrace.com
Norman Richards

NUMBERS

```
PRED: λn.(n SECOND)

PRED (SUCC NUM)
→ λn.(n SECOND) λs.(s FALSE NUM)
→ λs.(s FALSE ZERO) SECOND
→ SECOND FALSE NUM
→ NUM
```

BUT…

ZERO isn't SUCC of a number

orb@nostacktrace.com
Norman Richards

NUMBERS

```
PRED: λn.(ISZERO n) ZERO (n SECOND)
```

WHAT IS PRED OF ZERO?

We can at least test for zero and return another number.

orb@nostacktrace.com
Norman Richards

NUMBERS

```
ADD:λx.λy.(ISZERO Y)
    x
    (ADD (SUCC X)
         (PRED Y))
```

NOT VALID

Recursive definitions aren't possible . Remember, names are just syntactics sugar. All definitions must be finite. How do we do this?

```
function add(x,y) {
  if (y==0) {
    return x
  } else {
    return add(x+1,y-1)
  }
}
```

orb@nostacktrace.com
Norman Richards

RECURSION DIVERSION

IF WE COULD PASS THE FUNCTION IN

Now we'll make a function of three arguments, the first being the function to recurse on

```
ADD¹:λf.λx.λy.(ISZERO Y)
          X
          (f f (SUCC X) (PRED Y))
```

THEN WE COULD CALL IT RECURSIVELY

Just remember that the function takes three arguments, so we should pass the function to itself.

orb@nostacktrace.com

Norman Richards

RECURSION DIVERSION

```
ADD¹:λf.λx.λy.(ISZERO Y)
           X
          (f f (SUCC X) (PRED Y))

ADD: ADD¹ ADD¹
→ λx.λy.(ISZERO Y)
      X
      (ADD¹ ADD¹ (SUCC X) (PRED Y))
```

FINITE DEFINITION

This definition of ADD is a bit repetitive, but it doesn't recurse infinitely. If only we could clean this up a bit...

orb@nostacktrace.com

Norman Richards

BEHOLD, THE Y COMBINATOR

Y: λf.(λs.(f (s s)) (λs.(f (s s)))

ADD^1: λf.λx.λy.(ISZERO Y) X
(F (SUCC X) (PRED Y))

ADD: Y ADD^1

→ λs.(ADD^1 (s s)) (λs.(ADD^1 (s s))

→ ADD^1 (λs.(ADD^1 (s s)) λs.(ADD^1 (s s)))

→ ADD^1 (Y ADD^1)

→ ADD^1 ADD

Y SELF REPLICATES

This is similar to the hand prior call, except that the replication state is in the passed function.

orb@nostacktrace.com
Norman Richards

CHURCH NUMBERS

```
ZERO:  λf.λx.x
ONE:   λf.λx.f x
TWO:   λf.λx.f (f x)
THREE: λf.λx.f (f (f x))
FOUR:  λf.λx.f (f (f (f x)))
```

REPEATED FUNCTION APPLICATION

Church numbers take a function and an argument and apply the function to the argument n times. So N is defined by doing something N times.

orb@nostacktrace.com
Norman Richards

CHURCH NUMBERS

```
ZERO: λn.(n λx.FALSE TRUE)
SUCC: λn.λf.λx.f (n f x)
ADD:  λm.λn.λf.λx.m f (n f x)
MUL:  λm.λn.λf.m (n f)
POW:  λm.λn.m n
PRED: λn.λf.λx.n (λg.λh. h (g f))
                  λu.x λu.u
```

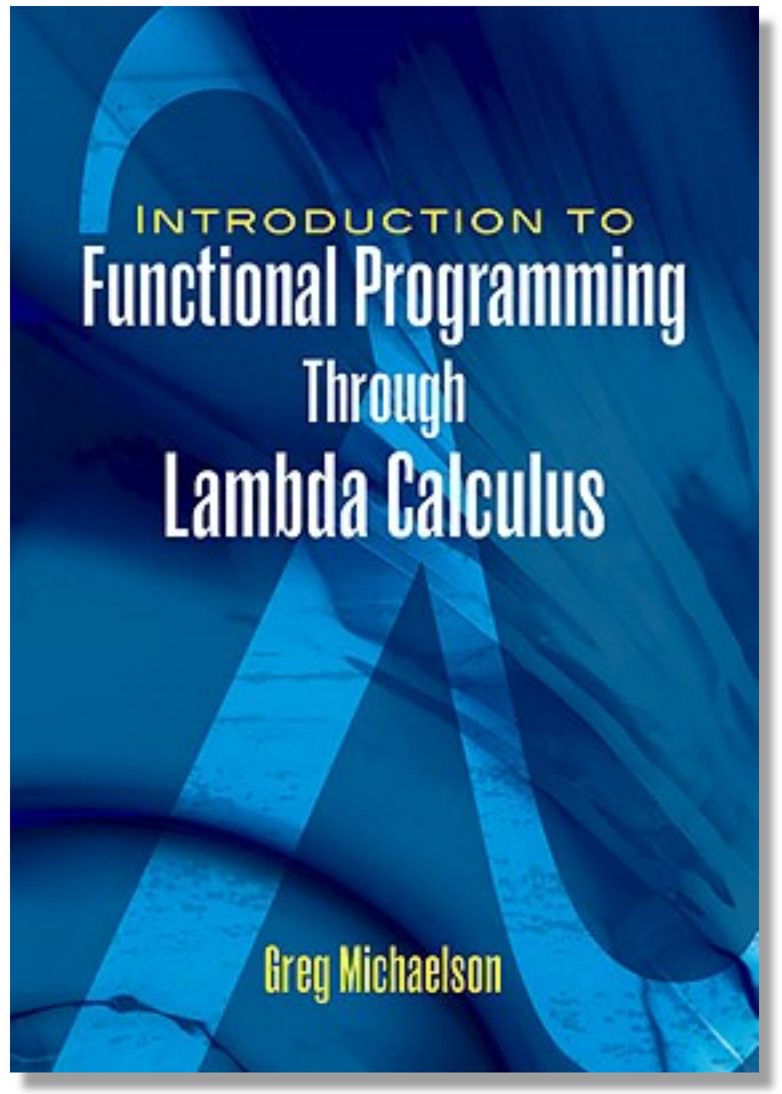
NO RECURSION FOR SIMPLE MATH

Church numbers have some very simple operations, Operations like subtraction are more complex than Peano numbers.

orb@nostacktrace.com

Norman Richards

And the rest is functional



http://www.amazon.com/dp/0486478831

An introduction to
Functional Programming
Through Lambda Calculus

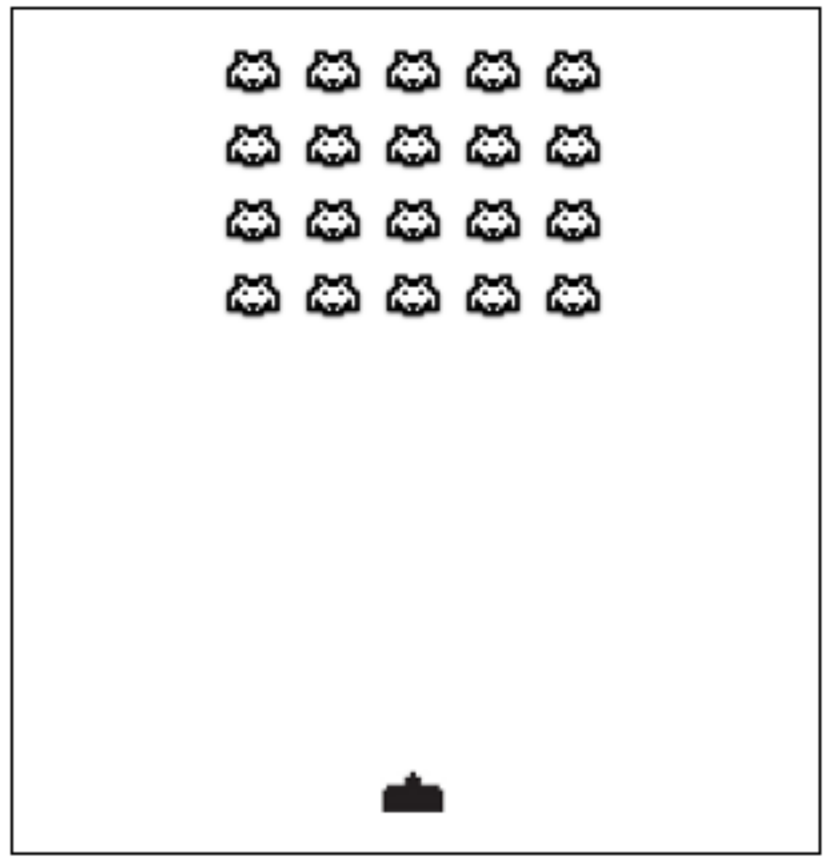
Greg Michaelson

orb@nostacktrace.com
Norman Richards

LEISURE

https://github.com/zot/Leisure

- Lambda Calculus engine written in Javascript
- Intended for programming
- REPL environment with node.js
- Runs in the browser



http://tinyconcepts.com/invaders.html

orb@nostacktrace.com
Norman Richards

λ THANK YOU

http://www.slideshare.net/normanrichards/the-lambda-calculus-and-the-javascript