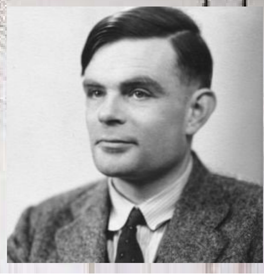
# Lambda Calculus



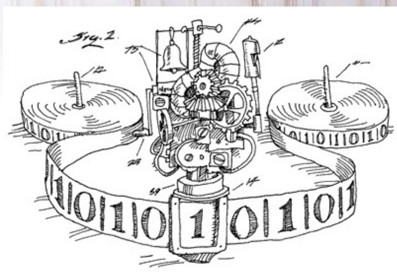
## Lambda Calculus

- Anonymous function calculus
- Functional model of computation
- **Alonzo Church** introduced **lambda calculus** in the 1930s.
  - Pure lambda calculus ('30s)
  - Applied & Simply typed lambda calculus('40s)
  - Polymorphic lambda calculus ('70s)

# Functional Model of Computation



- Express **computation** based on
  - (*anonymous*) function **abstraction** and
  - function **application** *via* **binding** and **substitution**.
- Equivalent to *state-based* **Turing Machine** by Alan Turing.
- Functional languages Lisp, ML, Haskell, F#, Clojure, Scala, etc. are derived from lambda calculus.





# Syntax

Everything in lambda calculus is an expression (**E**).

Lambda expressions are composed of:

- variables ID such as x, y, z, ...
- the abstraction symbols lambda 'λ' and dot '.'
- parentheses ( )

if M, N ∈ E, then

**rule 1**: E ::= ID

**rule 2**: E ::= λ ID . M

**rule 3**: E ::= (M N)

**rule 4**: E ::= (E)

†Pure lambda calculus does not define constants, types, operators, etc.

## Church Encodings!

Booleans, integers, and (other data structures) can be entirely replaced by functions!



## Turing Complete

- Boolean Logic
- Arithmetic
- Loops

# Boolean Logic

- TRUE = λx . λy . X
- FALSE = λx . λy . Y
- COND = λp. λq. λr. p q r
- AND := λ*a* . λ*b* . *a b FALSE*
- OR := λ*a* . λ*b* . *a TRUE b*
- NOT := λ*a* . *a* FALSE TRUE



# Church's Numerals

0 := λf . λx . x

1 := λf . λx . f x

2 := λf . λx . f (f x)

3 := λf . λx . f (f (f x))

…

succ = λn . λf . λx . f (n f x)

# Arithmetic Operators

- **add := λm . λn . λf . λx . m f (n f x)**
- **mult = λn . λm . m (add n) 0**



# Loop: Y Combinator

Y = λf .(λx . f (x x)) (λx . f (x x))

Y foo

⇒ (λf .(λx . f (x x)) (λx . f (x x)) foo (by definition of *Y*)

⇒ (λx . foo (x x)) (λx . foo (x x)) (by β-reduction of λf: applied Y to foo)

⇒ foo ((λx . foo (x x)) (λx . foo (x x))) (by β-reduction of λx)

⇒ foo (Y foo) (by second equality)

⇒ foo (foo (Y foo))

⇒ ...

⇒foo (… foo (Y foo)…)

# Examples

if M, N ∈ E, then

**rule 1**: E ::= ID

**rule 2**: E ::= λ ID . M

**rule 3**: E ::= (M N)

**rule 4**: E ::= (E)

x

λx . x

x y

λ λx . y

λx . y z

foo λ bar . (foo (bar baz))



# Disambiguation Rules

Applications are assumed to be **left associative**:

**M N P** may be written instead of **((M N) P)**

The body of an abstraction **extends as far right** as possible:

**λx . M N** means …

**λx . (M N)** and not **(λx . M) N**

**λx . λx . x** is …

**λx. ( λx . (x))**

## Semantics

Every **ID** that we see in lambda calculus is called a **variable**.

**E → λ ID . M** is called an **abstraction**

- The **ID** is the **variable** of the abstraction (also **bind variable**)
- **M** is called the **body** of the abstraction

**E → M N**

- This is called an **application**



## Semantics (Cont'd)

**λ ID . M** defines a new **anonymous function**

- This is the reason why called "***Lambda Expressions***" in Java 8 etc.
- **ID** is the **formal parameter** of the function
- **M** is the **body** of the function

**E → M N**, function **application**, is similar to calling function **M** and setting its formal parameter to be **N**

Application has higher precedence than abstraction:

- **λx . A B** means **λx . (A B)**, not **(λx . A) B**

## Computation by Rewriting

For now, think of rewriting as *replacing all occurrences of the formal parameter x in the function with the argument*:

**(λx . + x 1) 2**

**=> (+ 2 1)**

**=> 3**

This process is called **β-reduction**

*How can + function be defined if abstractions only accept 1 parameter?*



## Currying

**Translate** a function that takes *multiple* arguments ***into a sequence of functions*** that each take a **single** argument.

**λ (x, y) . (+ x y)** *// invalid λ-expression*

➔ **λx . λy . ((+ x) y)**

(λx . λy . ((+ x) y)) 10 20

=> (λy . ((+ 10) y)) 20

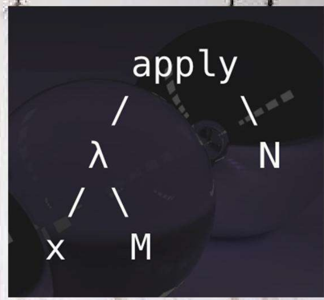
=> ((+ 10) 20) = 30

# Problems with the naive rewriting rule

Simple rule for rewriting **(λx . M)N** was

*"Replace all occurrences of **x** in **M** with **N**".*

However, there are two problems with this rule.




**Problem #1**:

"*scope escape*" problem

=> don't want to replace all occurrences of x

**Problem #2**:

"*capture*" or "*name clash*" problem



# **Problem #1** (Scope Escape)

Consider the following:

(λx . (x + ((λx . x + 1) 3))) 2

Let's rewrite the ***inner*** expression first.

(λx . (x + ((λx . x + 1) 3))) 2

=> (λx . (x + (3 + 1))) 2

=> (λx . (x + 4)) 2

=> (2 + 4)

=> 6

## **Problem # 1** (Scope Escape) (Cont'd)

(λx . (x + ((λx . x + 1) 3))) 2

This time, let's rewrite the ***outer*** expression first.

(λx . (x + ((λx . x + 1) 3))) 2

=> (2 + ((λx . 2 + 1) 3))

=> (2 + (2 + 1))

=> (2 + 3)

=> 5



## **Problem #2**

Consider the following:

((λx . λy . x) y) z

=> (λy . y) z

=> z

*What should be the correct answer?*

# The Issue behind Problem #1 is Scoping

To understand how to fix the first problem, we need to understand **scoping**, which involves the following:

- **Bound Variable**: a variable that is associated with some lambda.
- **Free Variable**: a variable that is *not* associated with any lambda.
- Intuitively, in lambda-expression **M**, variable **x** is **bound** if, in the AST, **x** is in the subtree of a lambda with left child **x**:

```
     λ
    / \
   x   /\
      /  \
     /    \
    /..x...\
       |
       this x is bound
```



# Free & Bound Variables

The same variable can appear many times in different contexts. Some instances may be bound, others free.

```
(λx.y)(λy.yx)
    |      ||
    |      |free
  free     |
         bound
```

# Free Variables?

x free in λ x . x y z?

y free in λ x . x y z?

x free in (λ x . (+ x 1)) x?

z free in λ x . λ y . λ z . z y x?

x free in (λ x . z foo) (λ y . y x)?

x free in x λ x . x ?

x free in (λ x . x y) x ?

x free in λ x . y x ?



# Combinators

An expression is a **combinator** if it does not have any free variables. The expression is also said to be **closed**.

λx . λy . x y x combinator?

λx . x combinator?

λz . λx . x y z combinator?

# Revised Rewriting Rule

To solve problem #1 above, given lambda expression **(λx . M) N**

"**Replace all occurrences of x in M with N**."



# Revised Rewriting Rule

To solve problem #1 above, given lambda expression **(λx . M) N**

"**Replace all occurrences of x <u>that are *free*</u> in M with N**."

For example:

```
     +----- M ---------+
     |                 |
(λx.  x + ((λx.x + 1)3)) 2
      |        |
      |        |
     free     bound
     in M     in M

 => 2 + ((λx.x + 1)3)
```

## The Issue behind Problem #2 is Name Clash

The variable **y** that is ***free* in the argument** to a λ-expression ***becomes bound*** after rewriting, because it is put into the scope of a λ-expression with a formal parameter named **y**:

```
((λx.λy.x)y)z
          |
          |
          free, but gets bound after application
```

***free* argument becomes bound after application!**



## Equivalence

What does it mean for two functions to be equivalent?

- λ y . y = λ x . x ?
- λ x . x y = λ y . y x ?
- λ x . x = λ x . x ?

***Two expressions that are α-equivalent must have the same set of free variables.***

# α-equivalence

**α-equivalence** is when two functions vary only by the names of the bound variables: $\mathbf{M =_\alpha N}$

To solve Problem #2, we need a way to rename variables in an expression:

- Simple find and replace?
- λx. x λy. x y z
  - Can we rename x to foo?
  - Can we rename y to bar?
  - Can we rename y to x?
  - Can we rename x to z?



# α-conversion

*as long as there is no name conflict with other variables*

The basic idea is that formal parameter names are unimportant; so rename them as needed to avoid capture.

**α-conversion** modifies expressions of the form **λx . M** to **λz . M′ .**

Renames all the occurrences of **x** that are ***free*** in **M** to some other variable **z** that does not occur in **M** (and then **λx** is changed to **λz**).

For example,

**λx . λy . x + y** *alpha-reduces* to **λz . λy . z + y**

# Substitution

Renaming allows us to replace one variable name with another.

However, our goal is to reduce **(λ x . + x 1) 2** to **(+ 2 1)**, which replaces **x** with the expression **2**.

We need another operator, called **substitution**, to replace a variable by a lambda expression.

- **E[x→N]**, where **E** and **N** are lambda expressions and **x** is a name



# Substitution

(λx . + x 1) 2

=> (+ x 1) [x→2]

=> (+ 2 1)

(λx . (λx . + x 1)) 2

=> (λx . + x 1) [x→2]

=> (λx . + x 1)

(λy . λx . y x) (λz . x z)

=> (λx . y x) [y→ λ z . x z]

=> (λx . (λz . **x** z) x) => trouble!

=> (λw . (λz . x z) w)

// substitution after alpha-reduction!

## Substitution Rule

E [x→N]

1. x [x→N] = N
2. y [x→N] = y, if x ≠ y
3. ($E_1$ $E_2$) [x→N] = ($E_1$ [x→N]) ($E_2$ [x→N])
4. (λ x . E) [x→N] = (λ x . E)
5. (λ y . E) [x→N] = (λ y . E [x→N]) when y ≠x and y $\notin FV(N)$
6. (λ y . E) [x→N] = (λ z . E {z/y} [x→N]) when y ≠x and y ∈ $FV(N)$, and z ≠x and z $\notin FV(E,N)$



## Precise Meaning of Rewriting: β-reduction

Defined using **substitution** (which in turn uses **α-reduction**).

Denoted by **(λx . M) N $\rightarrow_\beta$ M [x -> N]**

The left-hand side **(λx . M) N** is called the **redex**.

The notation means **M** with all ***free*** occurrences of **x** replaced with **N** in a way that avoids capture.

We say that **(λx . M) N** beta-reduces to **M** with **N** substituted for **x**.

# Normal Form

Computing with λ-expressions involves rewriting them using β-reduction.

A computation is finished when there are **no more redexes**.

**A λ-expression without redexes is in normal form,**

A λ-expression has a normal form *iff* there is some sequence of β-reduction (and/or expansions) that that leads to a normal form.

$$E_1 \Rightarrow^* E_2$$



# *η*-Reduction

If **v** is a variable, **E** is a lambda expression (denoting a function), and **v** has no free occurrence in **E**,

$$\lambda v \,.\, (E\ v) \Rightarrow_\eta E$$

$\lambda x \,.\, (sqr\ x) \Rightarrow_\eta sqr$

$\lambda x \,.\, (add\ 5\ x) \Rightarrow_\eta (add\ 5)$.

# Interesting Questions

**Q1**: Does every λ-expression have a normal form?

**Q2**: If a λ-expression does have a normal form, can we get there using only β-reductions?

**Q3**: If a λ-expression does have a normal form, do all choices of reduction sequences get there?

**Q4**: Is there a strategy for choosing β-reductions that is guaranteed to result in a normal form if one exists?



# Q1: Does every λ-expression have a normal form?

- No!

(λ x . x x) (λ x . x x)

=> (x x) [x→(λ x . x x)]

=> (λ x . x x) (λ x . x x)

=> (x x) [x→(λ x . x x)]

=> (λ x . x x) (λ x . x x)

…

## Q2: If a λ-expression does have a normal form, can we get there using only beta-reductions?

- Yes!
- See the "**Church-Rosser Theorem**".



## Q3: If a λ-expression does have a normal form, do all choices of reduction sequences get there?

Consider the following lambda expression:

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

The sequence of choices that we make **can** determine whether or not we get to a normal form.

## Q4: Is there a strategy for choosing β-reductions that is guaranteed to result in a normal form if one exists?

- Yes!
- **leftmost-outermost** *aka* **normal-order-reduction** **(NOR)**



## Outermost and Innermost Redexes

**Definition:** An ***outermost redex*** is a redex that is not contained inside another one. (Similarly, an ***innermost* redex** is one that has no redexes inside it.)

# Normal Order Reduction (NOR)

To do a normal-order reduction, always choose the **leftmost** of the **outermost** redexes.

NOR is like **call-by-name** parameter passing, where you evaluate an actual parameter only when the corresponding formal is used.



# Applicative-Order Reduction (AOR)

Choose the **leftmost** of the **innermost** redexes.

AOR corresponds to **call-by-value** parameter passing: the arguments are reduced before applying the function.

The advantage of AOR is **efficiency**.

The disadvantage is that AOR **may fail to terminate** on a lambda expression that has a normal form.

## Call-by-Need: Best of Both World

- **Call-by-need** is like call-by-name in that **an actual parameter is only evaluated when the corresponding formal is used** …
- However, the difference is that when using call-by-need, **the result of the evaluation is saved and is then reused** for each subsequent use of the formal.