

# Lambda Calculus: An Introduction

Adapted from Lectures by  
Professors of various universities



# Outline

- Why study lambda calculus?
- Lambda calculus
  - Syntax
  - Evaluation
  - Relationship to programming languages



# Lambda Calculus. History.

- A framework developed in 1930s by Alonzo Church to study computations with functions
- Church wanted a minimal notation
  - to expose only what is essential
- The smallest universal programming language of the world
  - Universal-Any computable function can be expressed and evaluated
- Two operations with functions are essential:
  - function creation
  - function application



# Function Creation

- Church introduced the notation

$\lambda x. E$

to denote a function with formal argument  $x$   
and with body  $E$

- Functions do not have names
  - names are not essential for the computation
- Functions have a single argument
  - Only one argument functions are discussed



# History of Notation

- Whitehead & Russel (Principia Mathematica) used the notation  $\hat{x} P$  to denote the set of  $x$ 's such that  $P$  holds
- Church borrowed the notation but moved  $\hat{\phantom{x}}$  down to create  $\wedge x E$
- Which later turned into  $\lambda x. E$  and the calculus became known as lambda calculus

# Function Application

- The only thing that we can do with a function is to apply it to an argument
- Church used the notation

$E_1 E_2$

to denote the application of function  $E_1$  to actual argument  $E_2$

$E_1$  is called (ope)rator and  $E_2$  is called (ope)rand

- All functions are applied to a single argument

# Why Study Lambda Calculus?

- $\lambda$ -calculus has had a tremendous influence on the design and analysis of programming languages
- Realistic languages are too large and complex to study from scratch as a whole
- Typical approach is to modularize the study into one feature at a time
  - E.g., recursion, looping, exceptions, objects, etc.
- Then we assemble the features together





# Why Study Lambda Calculus?

- $\lambda$ -calculus is the standard testbed for studying programming language features
  - Because of its minimality
  - Despite its syntactic simplicity the  $\lambda$ -calculus can easily encode:
    - numbers, recursive data types, modules, imperative features, exceptions, etc.
- Certain language features necessitate more substantial extensions to  $\lambda$ -calculus:
  - for distributed & parallel languages:  $\pi$ -calculus
  - for object oriented languages:  $\sigma$ -calculus





# Why Study Lambda Calculus?

*“Whatever the next 700 languages turn out to be, they will surely be variants of lambda calculus.”*

(Landin 1966)

# Syntax of Lambda Calculus

- Only three kinds of expressions

$E ::= x$

variables

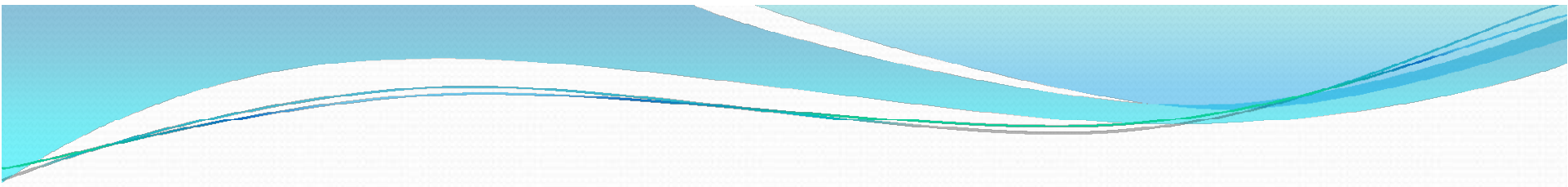
|  $E_1 E_2$

function application

|  $\lambda x. E$

function creation

- The form  $\lambda x. E$  is also called lambda abstraction, or simply abstraction
- $E$  are called  $\lambda$ -terms or  $\lambda$ -expressions



The central concept in  $\lambda$  calculus is the “expression”. A “name”, also called a “variable”, is an identifier which, for our purposes, can be any of the letters  $a, b, c, \dots$ . An expression is defined recursively as follows:

$$\begin{aligned} \langle \text{expression} \rangle &:= \langle \text{name} \rangle \mid \langle \text{function} \rangle \mid \langle \text{application} \rangle \\ \langle \text{function} \rangle &:= \lambda \langle \text{name} \rangle . \langle \text{expression} \rangle \\ \langle \text{application} \rangle &:= \langle \text{expression} \rangle \langle \text{expression} \rangle \end{aligned}$$



# Examples of Lambda Expressions

- The identity function:

$$I =_{\text{def}} \lambda x. x$$

- A function that given an argument  $y$  discards it and computes the identity function:

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

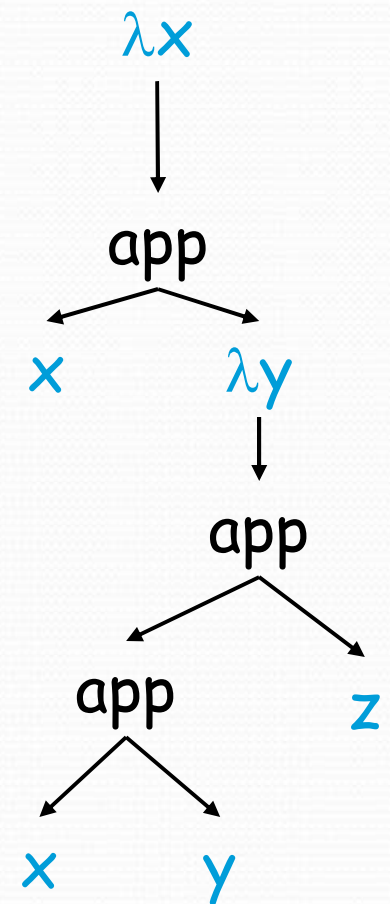
- A function that given a function  $f$  invokes it on the identity function

$$\lambda f. f (\lambda x. x)$$



# Notational Conventions

- Application associates to the left  
 $x\ y\ z$  parses as  $(x\ y)\ z$
- Abstraction extends to the right as far as possible  
 $\lambda x. x\ \lambda y. x\ y\ z$  parses as  
 $\lambda x. (x\ (\lambda y. ((x\ y)\ z)))$
- And yields the the parse tree:





# Scope of Variables

- As in all languages with variables, it is important to discuss the notion of scope
  - Recall: the **scope** of an identifier is the portion of a program where the identifier is accessible
- An abstraction  $\lambda x. E$  **binds** variable  $x$  in  $E$ 
  - $x$  is the newly introduced variable
  - $E$  is the scope of  $x$
  - we say  $x$  is **bound** in  $\lambda x. E$
  - Just like formal function arguments are bound in the function body

# Free and Bound Variables

- A variable is said to be free in  $E$  if it is not bound in  $E$
- We can define the free variables of an expression  $E$  recursively as follows:

$$\text{Free}(x) = \{ x \}$$

$$\text{Free}(E_1 E_2) = \text{Free}(E_1) \cup \text{Free}(E_2)$$

$$\text{Free}(\lambda x. E) = \text{Free}(E) - \{ x \}$$

- Example:  $\text{Free}(\lambda x. x (\lambda y. x y z)) = \{ z \}$
- Free variables are declared outside the term
- A lambda expression with no free variables is called closed.



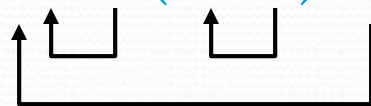
# Free and Bound Variables (Cont.)

- Just like in any language with static nested scoping, we have to worry about variable shadowing
  - An occurrence of a variable might refer to different things in different context

- E.g., in Cool: `let x ← E in x + (let x ← E' in x) + x`



- In  $\lambda$ -calculus:  `$\lambda x. x (\lambda x. x) x$`





# Renaming Bound Variables

- Two  $\lambda$ -terms that can be obtained from each other by a renaming of the bound variables are considered identical
- Example:  $\lambda x. x$  is identical to  $\lambda y. y$  and to  $\lambda z. z$
- Intuition:
  - by changing the name of a formal argument and of all its occurrences in the function body, the behavior of the function does not change
  - in  $\lambda$ -calculus such functions are considered identical



# Renaming Bound Variables (Cont.)

- Convention: we will always rename bound variables so that they are all unique
  - e.g., write  $\lambda x. x (\lambda y.y) x$  instead of  $\lambda x. x (\lambda x.x) x$
  - Variable capture or name clash problem would arise!
- This makes it easy to see the scope of bindings
- And also prevents serious confusion !

# Substitution

- The substitution of  $E'$  for  $x$  in  $E$  (written  $[E'/x]E$ )
  - **Step 1.** Rename bound variables in  $E$  and  $E'$  so they are unique ( $\alpha$ -reduction)
  - **Step 2.** Perform the textual substitution of  $E'$  for  $x$  in  $E$
- Example:  $[y (\lambda x. x) / x] \lambda y. (\lambda x. x) y x$ 
  - After **renaming**:  $[y (\lambda v. v)/x] \lambda z. (\lambda u. u) z x$
  - After **substitution**:  $\lambda z. (\lambda u. u) z (y (\lambda v. v))$





# Evaluation of $\lambda$ -terms

- There is one key evaluation step in  $\lambda$ -calculus: the function application

$(\lambda x. E) E'$  evaluates to  $[E'/x]E$

- This is called  $\beta$ -reduction
- We write  $E \rightarrow_{\beta} E'$  to say that  $E'$  is obtained from  $E$  in one  $\beta$ -reduction step
- We write  $E \rightarrow_{\beta}^* E'$  if there are zero or more steps



### Definition: $\alpha$ -reduction

If  $v$  and  $w$  are variables and  $E$  is a lambda expression,

$$\lambda v . E \Rightarrow_{\alpha} \lambda w . E[v \rightarrow w]$$

provided that  $w$  does not occur at all in  $E$ , which makes the substitution  $E[v \rightarrow w]$  safe. The equivalence of expressions under  $\alpha$ -reduction is what makes part g) of the definition of substitution correct.

### Definition: $\beta$ -reduction

If  $v$  is a variable and  $E$  and  $E_1$  are lambda expressions,

$$(\lambda v . E) E_1 \Rightarrow_{\beta} E[v \rightarrow E_1]$$

provided that the substitution  $E[v \rightarrow E_1]$  is carried out according to the rules for a safe substitution.

*$\beta$ -redex*

# Functions with Multiple Arguments

- Consider that we extend the calculus with the **add** primitive operation
- The  $\lambda$ -term  $\lambda x. \lambda y. \text{add } x y$  can be used to add two arguments  $E_1$  and  $E_2$ :

$$(\lambda x. \lambda y. \text{add } x y) E_1 E_2 \rightarrow_{\beta}$$

$$([E_1/x] \lambda y. \text{add } x y) E_2 =$$

$$(\lambda y. \text{add } E_1 y) E_2 \rightarrow_{\beta}$$

$$[E_2/y] \text{add } E_1 y = \text{add } E_1 E_2$$

- The arguments are passed one at a time



# Functions with Multiple Arguments

- What is the result of  $(\lambda x. \lambda y. \text{add } x \ y) \ E$  ?
  - It is  $\lambda y. \text{add } E \ y$   
(A function that given a value  $E'$  for  $y$  will compute  $\text{add } E \ E'$ )
- The function  $\lambda x. \lambda y. E$  when applied to one argument  $E'$  computes the function  $\lambda y. [E'/x]E$
- This is one example of higher-order computation
  - We write a function whose result is another function



# Evaluation and the Static Scope

- The definition of substitution guarantees that evaluation respects static scoping:

$$(\lambda x. (\lambda y. y x)) (y (\lambda x. x)) \rightarrow_{\beta} \lambda z. z (y (\lambda v. v))$$

(y remains free, i.e., defined externally)

- If we forget to rename the bound y:

$$(\lambda x. (\lambda y. y x)) (y (\lambda x. x)) \rightarrow_{\beta}^* \lambda y. y (y (\lambda v. v))$$

(y was free before but is bound now)



# Reduction Strategies

**Definition** : A lambda expression is in **normal form** if it contains no  $\beta$ -redexes (and no  $\delta$ -rules in an applied lambda calculus), so that it cannot be further reduced using the  $\beta$ -rule or the  $\delta$ -rule. An expression in normal form has no more function applications to evaluate. ■

- Can every lambda expression be reduced to a normal form?
- Is there more than one way to reduce a particular lambda expression?
- If there is more than one reduction strategy, does each one lead to the same normal form expression?
- Is there a reduction strategy that will guarantee that a normal form expression will be produced?



Can every lambda expression be reduced to a normal form?

- The identity function:

$$(\lambda x. x) E \rightarrow [E / x] x = E$$

- Another example with the identity:

$$\begin{aligned} & (\lambda f. f (\lambda x. x)) (\lambda x. x) \rightarrow \\ & [\lambda x. x / f] f (\lambda x. x) = [(\lambda x. x) / f] f (\lambda y. y) = \\ & (\lambda x. x) (\lambda y. y) \rightarrow \\ & [\lambda y. y / x] x = \lambda y. y \end{aligned}$$

- A non-terminating evaluation:

$$\begin{aligned} & (\lambda x. xx)(\lambda x. xx) \rightarrow \\ & [\lambda x. xx / x]xx = [\lambda y. yy / x] xx = (\lambda y. yy)(\lambda y. yy) \rightarrow \dots \end{aligned}$$



Is there more than one way to reduce a particular lambda expression?

**Example :**  $(\lambda y . 5) ((\lambda x . x x) (\lambda x . x x))$

**Definition :** A **normal order** reduction always reduces the leftmost outermost  $\beta$ -redex (or  $\delta$ -redex) first. An **applicative or der** reduction always reduces the leftmost innermost  $\beta$ -redex (or  $\delta$ -redex) first. ■

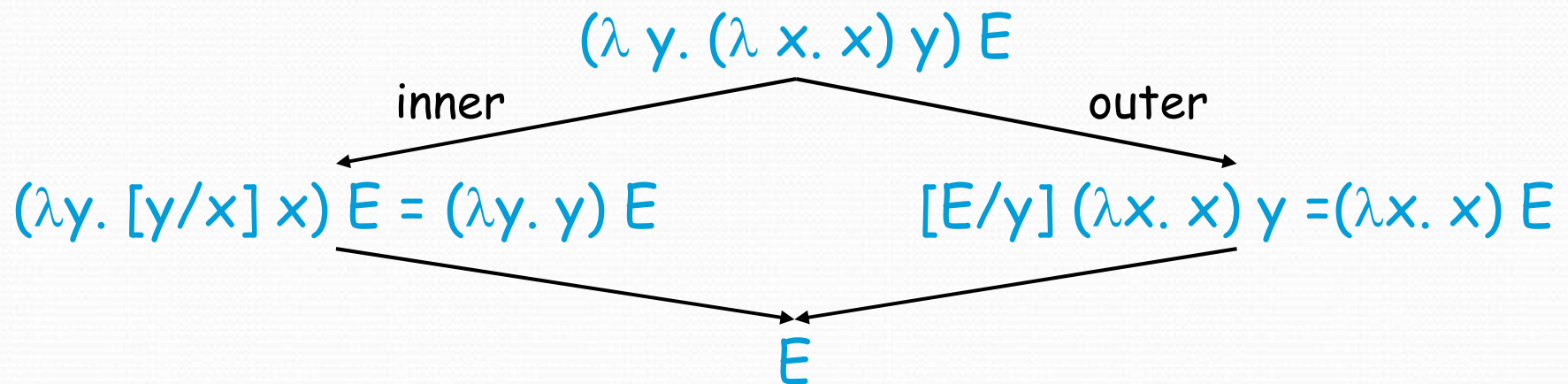


If there is more than one reduction strategy, does each one lead to the same normal form expression?

- In a  $\lambda$ -term, there could be more than one instance of  $(\lambda x. E) E'$

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

- could reduce the inner or the outer  $\lambda$
- which one should we pick?



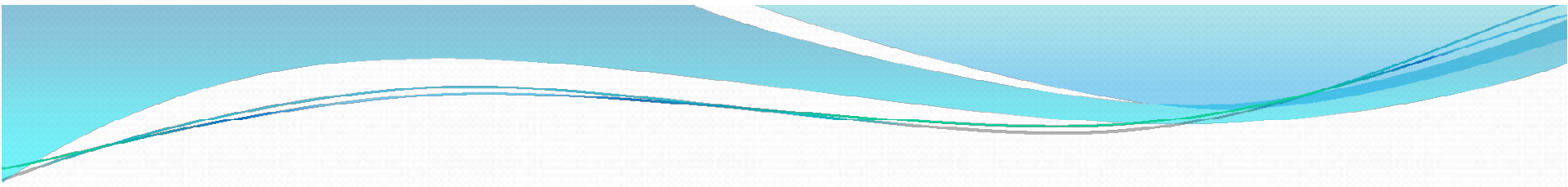


# Order of Evaluation (Cont.)

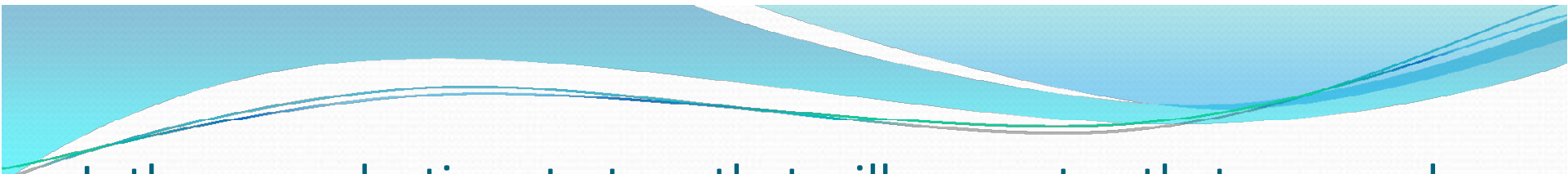
- The **Church-Rosser theorem** says that any order will compute the same result
  - A result is a  $\lambda$ -term that cannot be reduced further

**Church-Rosser Theorem I:** For any lambda expressions  $E$ ,  $F$ , and  $G$ , if  $E \Rightarrow^* F$  and  $E \Rightarrow^* G$ , there is a lambda expression  $Z$  such that  $F \Rightarrow^* Z$  and  $G \Rightarrow^* Z$ .

- Some evaluations may terminate while others may diverge. If two evaluations terminate, it will be to the same normal form.
- DIAMOND PROPERTY (confluence)

- 
- A normal order reduction can have either of the following outcomes
    - 1. It reaches a unique (up to  $\alpha$ -conversion) normal form lambda expression
    - 2. It never terminates

**Church-Rosser Theorem II:** For any lambda expressions  $E$  and  $N$ , if  $E \Rightarrow^* N$  where  $N$  is in normal form, there is a normal order reduction from  $E$  to  $N$ .



Is there a reduction strategy that will guarantee that a normal form expression will be produced?

- Turing machines are abstract machines designed in the 1930s by Alan Turing to model computable functions. It has been shown that the lambda calculus is equivalent to Turing machines in the sense that every lambda expression has an equivalent function defined by some Turing machine and vice versa.
- Alan Turing proved a fundamental result, called the undecidability of the **halting problem**, which states that there is no algorithmic way to determine whether or not an arbitrary Turing machine will ever stop running. Therefore there are lambda expressions for which it cannot be determined whether a normal order reduction will ever terminate.
- But we might want to fix the order of evaluation when we model a certain language
- In (typical) programming languages, we do not reduce the bodies of functions (under a  $\lambda$ )
  - functions are considered values



# Call by Name

- Same as Normal Order reduction
- Do not evaluate the argument prior to call
- Example:

$$(\lambda y. (\lambda x. x) y) ((\lambda u. u) (\lambda v. v)) \rightarrow_{\beta n}$$

$$(\lambda x. x) ((\lambda u. u) (\lambda v. v)) \rightarrow_{\beta n}$$

$$(\lambda u. u) (\lambda v. v) \rightarrow_{\beta n}$$

$$\lambda v. v$$



# Call by Value

- Same as Applicative order reduction
- Evaluate an argument prior to call
- Example:

$$(\lambda y. (\lambda x. x) y) ((\lambda u. u) (\lambda v. v)) \rightarrow_{\beta_v}$$

$$(\lambda y. (\lambda x. x) y) (\lambda v. v) \rightarrow_{\beta_v}$$

$$(\lambda x. x) (\lambda v. v) \rightarrow_{\beta_v}$$

$$\lambda v. v$$



# Call by Name and Call by Value

- CBN
  - difficult to implement
  - order of side effects not predictable
- CBV:
  - easy to implement efficiently
  - might not terminate even if CBN might terminate
  - Example:  $(\lambda x. \lambda z. z) ((\lambda y. yy) (\lambda u. uu))$
- Outside the functional programming language community, only CBV is used

### Definition: $\eta$ -reduction

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

$$\lambda v . (E \ v) \Rightarrow_{\eta} E.$$

$$\lambda x . (\text{sqr } x) \Rightarrow_{\eta} \text{sqr}$$

$$\lambda x . (\text{add } 5 \ x) \Rightarrow_{\eta} (\text{add } 5).$$

- $\lambda x.(5 \ x)$  is not 5

### Definition: $\delta$ -reduction

If the lambda calculus has predefined constants (that is, if it is not pure), rules associated with those predefined values and functions are called  $\delta$  rules; for example,  $(\text{add } 3 \ 5) \Rightarrow_{\delta} 8$  and  $(\text{not true}) \Rightarrow_{\delta} \text{false}$ .





## Lambda Calculus and Programming Languages

- Pure lambda calculus has only functions
- What if we want to compute with Booleans, numbers, lists, etc.?
- All these can be encoded in pure  $\lambda$ -calculus
- Need not encode what a value is
- For each data type, we have to describe how it can be used, as a function
  - then we write that function in  $\lambda$ -calculus



# Encoding Booleans in Lambda Calculus

- What can we do with a boolean?
  - we can make a binary choice
- A boolean is a function that given two choices selects one of them
  - $\text{true} =_{\text{def}} \lambda x. \lambda y. x$
  - $\text{false} =_{\text{def}} \lambda x. \lambda y. y$
  - $\text{if } E_1 \text{ then } E_2 \text{ else } E_3 =_{\text{def}} E_1 E_2 E_3$
- Example: if true then u else v is
$$(\lambda x. \lambda y. x) u v \rightarrow_{\beta} (\lambda y. u) v \rightarrow_{\beta} u$$

# Encoding Pairs in Lambda Calculus

The function “Pair” encapsulates two values in a given order; it is essentially the dotted pair notion (cons) in Lisp.

$$\text{define Pair} = \lambda a . \lambda b . \lambda f . f\ a\ b$$

Two selector functions “Head” and “Tail” confirm that Pair implements the cons operation.

$$\text{define Head} = \lambda g . g\ (\lambda a . \lambda b . a)$$
$$\text{define Tail} = \lambda g . g\ (\lambda a . \lambda b . b)$$

Now the correctness of the definitions is verified by reductions:

$$\text{Head (Pair } p\ q) \Rightarrow (\lambda \mathbf{g} . g\ (\lambda a . \lambda b . a)) ((\lambda \mathbf{a} . \lambda \mathbf{b} . \lambda \mathbf{f} . \mathbf{f}\ \mathbf{a}\ \mathbf{b})\ \mathbf{p}\ \mathbf{q})$$

# Encoding Natural Numbers in Lambda Calculus

- What can we do with a natural number?
  - we can iterate a number of times
- A natural number is a function that given an operation  $f$  and a starting value  $s$ , applies  $f$  a number of times to  $s$ :

$$0 =_{\text{def}} \lambda f. \lambda s. s$$

$$1 =_{\text{def}} \lambda f. \lambda s. f s$$

$$2 =_{\text{def}} \lambda f. \lambda s. f (f s)$$

and so on



# Polymorphism

- Functions that allow arguments of many types, such as this identity function, are known as **polymorphic operations**
  - $((\lambda x . x) E) = E$
- *define*  $Twice = \lambda f . \lambda x . f(f x)$ 
  - If  $D$  is any domain, the syntax (or signature) for  $Twice$  can be described as
    - $Twice : (D \rightarrow D) \rightarrow D \rightarrow D$
  - Given the square function,  $sqr : N \rightarrow N$  where  $N$  stands for the natural numbers, it follows that
  - $(Twice\ sqr) : N \rightarrow N$
  - Is twice a higher order function?
- The mechanism that allows functions to be defined to work on a number of types of data is also known as **parametric polymorphism**

# Computing with Natural Numbers

$$\begin{aligned} 1 &\equiv \lambda sz.s(z) \\ 2 &\equiv \lambda sz.s(s(z)) \\ 3 &\equiv \lambda sz.s(s(s(z))) \end{aligned}$$

- The successor function

$$\text{successor } n =_{\text{def}} \lambda wyx.y(wyx)$$

- Successor of 0 (S0) is  $_{\text{def}} (\lambda wyx.y(wyx)) \lambda sz.z$
- Successor of 1 (S1) is  $_{\text{def}} (\lambda wyx.y(wyx)) \lambda sz.s(z)$
- Addition

$$\begin{aligned} 2S3 \text{ is }_{\text{def}} & (\lambda sz.s(s(z))) (\lambda wyx.y(wyx)) (\lambda uv.u(u(u(v)))) \\ & \lambda mwyx. m y (wyx) \end{aligned}$$



# Addition

- $(\text{int} \times \text{int}) \rightarrow \text{int}$  ---not pure lambda calculus
- $(\text{int} \rightarrow \text{int}) \rightarrow \text{int}$  ----pure form (using currying)
- $\lambda x. \lambda y. x + y$
- Evaluate  $((\lambda x. \lambda y. y \ x) \ 3)$



# Predecessor function

- $\text{PRED} := \lambda nfx.n (\lambda gh.h (g f)) (\lambda u.x) (\lambda u.u)$
- $\text{PRED } 1 \iff$
- $(\lambda nfx.n (\lambda gh.h (g f)) (\lambda u.x) (\lambda u.u)) (\lambda hy. h y) \iff$
- $\lambda fx. (\lambda hy. h y) (\lambda gh.h (g f)) (\lambda u.x) (\lambda u.u) \iff$
- $\lambda fx. (\lambda gh.h (g f)) (\lambda u.x) (\lambda u.u) \iff$
- $\lambda fx. (\lambda u.u) ((\lambda u.x) f) \iff$
- $\lambda fx. (\lambda u.u) x \iff$
- $\lambda fx. x \iff$
- $0$

# Y Combinator

- Y combinator can be defined as
  - $Y = \lambda t. (\lambda x. t (x x)) (\lambda x. t (x x))$
- $Yt = t(Yt) = t(t(Yt)) = \dots$

```
define factorial =  $\lambda n. \text{if } (= n 1) 1$   
                   $(* n (\text{factorial } (- n 1)))$ 
```

```
define factorial =  $\underline{T}$  factorial  
define  $\underline{T}$  =  $\lambda f. \lambda n. \text{if } (= n 1) 1$   
             $(* n (f (- n 1)))$ 
```

```
 $(Y \underline{T}) 1$  =  $\underline{T} (Y \underline{T}) 1$   
          =  $\text{if } (= 1 1) 1 (* 1 (Y \underline{T} (- 1 1)))$   $\beta$ -reduction  
          = 1 calculating arithmetic
```

- $(Y \underline{T}) 2 = 2$

$$Y \equiv (\lambda y. (\lambda x. y(xx)) (\lambda x. y(xx)))$$

This function applied to a function R yields:

$$YR = (\lambda x. R(xx)) (\lambda x. R(xx))$$

which further reduced yields:

$$R((\lambda x. R(xx)) (\lambda x. R(xx)))$$

but this means that  $YR = R(YR)$ , that is, the function R is evaluated using the recursive call YR as the first argument.

*To compute sum of natural numbers from 0 to n*

$$R \equiv (\lambda r n. Z n 0 (n S (r (P n))))$$

•  $Z n 0$  : if  $n == 0$  then the result of the sum is 0 else the successor function is applied n times through the recursive call (r)

To compute the sum of 0 to 3,  $YR3 = R(YR)3 = Z30(3S(YR(P3)))$

$\Rightarrow 3S(YR2) \Rightarrow 3S2S1S0$



# Objects in lambda calculus

- Self application is used very fundamentally in implementing object-oriented programming languages. Suppose we have an object  $x$  with a method  $m$ .
- We might invoke this method by writing something like  $x.m(y)$ .
- Inside the method  $m$ , there would be references to keywords like “self” or “this” which are supposed to represent the object  $x$  itself.
- But how does  $m$  know what the object  $x$  is?
- One way of solving the problem is to translate the method  $m$  into a function  $m'$  that takes two arguments: in addition to the proper argument  $y$ , the object on which the method is being invoked. So, the definition of  $m'$  looks like:
- $m' = \lambda \text{self}. \lambda y. \dots \text{the body of } m \dots$
- The object  $x$  has a collection of such functions encoding the methods.
- The method call  $x.m(y)$  is then translated as  $x.m'(x)(y)$ .
- This is a form of self application.
- The function  $m'$ , which is a part of the structure  $x$ , is applied to the structure  $x$  itself.



# Expressiveness of Lambda Calculus

- The  $\lambda$ -calculus can express
  - data types (integers, booleans, lists, trees, etc.)
  - branching (using booleans)
  - recursion
- This is enough to encode Turing machines
- Encodings can be done
- But programming in pure  $\lambda$ -calculus is painful
  - add constants (0, 1, 2, ..., true, false, if-then-else, etc.)
  - add types