Lambda Calculus - Church Boolean (6A)

Young Won Lim
8/18/23

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

(1) Church Boolean

Church Booleans are

the Church encoding of the Boolean values **true** and **false**.

Some programming languages use these

as an implementation model for **Boolean arithmetic**;

examples are **Smalltalk** and **Pico**.

https://en.wikipedia.org/wiki/Church_encoding

(2) Church encoding of **true** and **false**

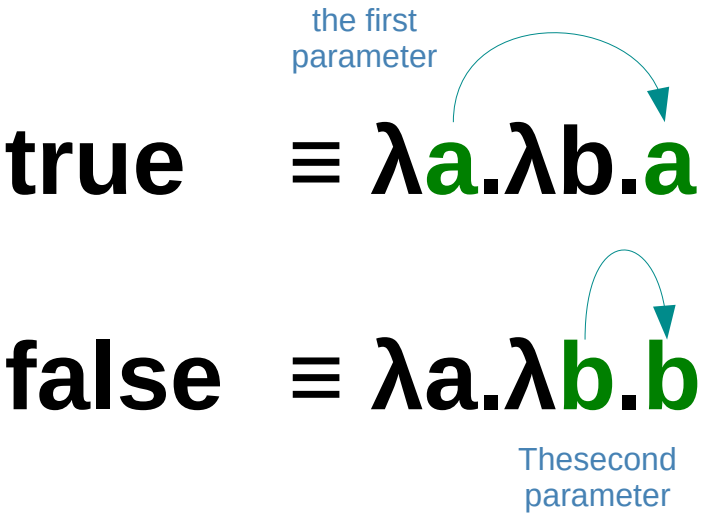
Boolean logic may be considered as a choice.

The Church encoding of **true** and **false** are
functions of two **parameters**:

true chooses the first **parameter**.
false chooses the second **parameter**.

The two definitions are known as **Church Booleans**:

true ≡ **λa.λb.a**
false ≡ **λa.λb.b**



https://en.wikipedia.org/wiki/Church_encoding

(3) Predicates act as if-clause

the **Church Booleans** definitions
allow **predicates** to directly act as **if-clauses**
predicates are **functions** returning **logical values**

A **function** returning a **Boolean**,
which is then applied to two **parameters**,
returns either the first or the second parameter:

predicate-x then-clause else-clause

evaluates to **then-clause** if **predicate-x** evaluates to **true**,
and to **else-clause** if **predicate-x** evaluates to **false**.

true ≡ λa.λb.a
false ≡ λa.λb.b

If FOO a b
FOO a b

https://en.wikipedia.org/wiki/Church_encoding

(1) TRUE, FALSE, and IF

Examples of Simple Normal-Form Expressions

λx.λy.x

λx.λy.y

Let's look at what they do when they are applied.

If we call them **TRUE** and **FALSE** respectively,

then we have replaced our **IF** construct.

(λx.λy.x) a b → a

(λx.λy.y) a b → b

IF FOO B1 B2 replaced by **FOO B1 B2**

Assuming

TRUE = λx.λy.x

FALSE = λx.λy.y

If FOO then a else b

= FOO then a else b

https://www.cs.cmu.edu/~venkatg/teaching/15252-sp20/notes/lambda-calculus-slides.pdf

(2) IF FOO a b

If FOO then a else b **(If FOO a b)**

when **FOO** is equal to **true = (λx.λy.x),**

FOO a b = a **(λx.λy.x) a b → a**

when **FOO** is equal to **= false = (λx.λy.y),**

FOO a b = b **(λx.λy.y) a b → b`**

Assuming **TRUE = λx.λy.x**

FALSE = λx.λy.y

If FOO a b ➡ **FOO a b**

(1) True, False functions and application

Roughly, "**true**" is the **function**
which takes two arguments **x**, **y**
(first takes **x**, then also takes **y**) and
returns the first one.
"**false**" returns the second one.

If-then-else is then encoded
as applying an **encoded boolean**
to the "**then**" and "**else**" branches.
In this way, the boolean chooses the right value to return.

if b then t else e = b t e

true ≡ λx.λy.x

false ≡ λx.λy.y

https://cs.stackexchange.com/questions/91759/how-to-model-conditionals-with-first-class-functions

(2) construct and destruct

the so-called **β laws**:

if true then t else e = true t e = (λx.λy.x) t e = (λy.t) e = t

if false then t else e = false t e = (λx.λy.y) t e = (λy.y) e = e

Essentially, these laws state what happens
when you "construct" a boolean (using the constants **true**,**false**),
and then "destruct" it later on (using it in an **if then else**).

Using functions in this way,
one can write encodings for all the usual data types,
like naturals, pairs, variants, lists, trees, etc.

true ≡ λa.λb.a

false ≡ λa.λb.b

https://cs.stackexchange.com/questions/91759/how-to-model-conditionals-with-first-class-functions

(3) if-then-else

if p then a else b = (λp.λa.λb.p a b)

if true then t else e = (λp.λa.λb.p a b) true t e
= true t e
= (λx.λy.x) t e = (λy.t) e = t

if false then t else e = (λp.λa.λb.p a b) false t e
= false t e
== (λx.λy.y) t e = (λy.y) e = e

true ≡ λa.λb.a

false ≡ λa.λb.b

https://en.wikipedia.org/wiki/Church_encoding

(1) TRUE, FALSE

By convention, the following two definitions
(known as Church booleans)
are used for the boolean values
TRUE and FALSE:

TRUE := λx.λy.x

FALSE := λx.λy.y

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

(2) Logic Operators

Then, with these two lambda terms,
we can define some logic operators
(these are just possible formulations;
other expressions are equally correct):

AND	**:= λp.λq.p q p**	**if p then q else p**
OR	**:= λp.λq.p p q**	**if p then p else q**
NOT	**:= λp.p FALSE TRUE**	**if p then false else true**
IF THEN ELSE	**:= λp.λa.λb.p a b**	**if = λp.λa.λb.p a b**

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

(3) AND

We are now able to compute some logic functions, for example:

AND TRUE FALSE

≡ (λp.λq.p q p) TRUE FALSE

→β TRUE FALSE TRUE

≡ (λx.λy.x) FALSE TRUE

→β FALSE

and we see that **AND TRUE FALSE**

is equivalent to **FALSE**.

AND := λp.λq.p q p

if p then q else p

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

(3) AND

Essentially we use our **IF-THEN-ELSE** design for **AND** operation.

AND A B = A B FALSE

= IF A then B else FALSE

Actually we can also write **AND** more simply as

AND = λa.λb. a b FALSE = λa.λb. a b (λx.λy.y)

if (**a** = **true**) then **b**

else (**a** = **false**) **false** can be written as **a**

AND = λa.λb. a b FALSE = λa.λb. a b a

Easy exercise: Work out **OR** and **NOT**.

and = if p then q else p

or = if p then p else q

not = if p then b else a

not = if p then false else true

xor = if a then (not b) else b

https://www.cs.cmu.edu/~venkatg/teaching/15252-sp20/notes/lambda-calculus-slides.pdf

(4) OR

Essentially we use our **IF-THEN-ELSE** design for **OR** operation.

OR A B = A TRUE B

= IF A then TRUE else B

Actually we can also write **OR** more simply as

OR = λa.λb. a TRUE b = λa.λb. a (λx.λy.x) b

if (**a** = **true**) then **true** can be written as **a**

else (**a** = **false**) **b**

OR = λa.λb. a TRUE b = λa.λb. a a b

and = if p then q else p

or = if p then p else q

not = if p then b else a

not = if p then false else true

xor = if a then (not b) else b

https://www.cs.cmu.edu/~venkatg/teaching/15252-sp20/notes/lambda-calculus-slides.pdf

(5) NOT

Essentially we use our **IF-THEN-ELSE** design for **NOT** operation.

NOT P A B = P B A

= IF P then B else A

Actually we can also write **NOT** more simply as

NOT = λp.λa.λb. p b a

if (**p** = **true**) then **b**

else (**p** = **false**) **a**

NOT = λp.λa.λb. a b a

and = if p then q else p

or = if p then p else q

not = if p then b else a

not = if p then false else true

xor = if a then (not b) else b

https://www.cs.cmu.edu/~venkatg/teaching/15252-sp20/notes/lambda-calculus-slides.pdf

Logic operators (1)

Because **true** and **false** choose the first or second **parameter**
they may be combined to provide **logic operators**.

Note that there are multiple possible implementations of **not**.

and	**= λp.λq.p q p**	**if p then q else p**
or	**= λp.λq.p p q**	**if p then p else q**
not	**= λp.λa.λb.p b a**	**if p then b else a**
not	**= λp.p (λa.λb.b) (λa.λb.a)**	
	= λp.p false true	**if p then false else true**
xor	**= λa.λb.a (not b) b**	**if a then (not b) else b**
if	**= λp.λa.λb.p a b**	

https://en.wikipedia.org/wiki/Church_encoding

Logic operators (2)

and = λp.λq.p q p	or = λp.λq.p p q
and false false = false	or false false = false
and false true = false	or false true = true
and true false = false	or true false = true
and true true = true	or true true = true

not = λp.λa.λb.p b a

not = λp.p.(λa.λb.b)(λa.λb.a) = λp.p false true

not false = true
not true = false

https://en.wikipedia.org/wiki/Church_encoding

Logic operators (3)

xor = λa.λb.a (not b) b	if = λp.λa.λb.p a b
xor false false = false	if p then a else b
xor false true = true	
xor true false = true	if true then a else b = a
xor true true = false	if false then a else b = b

https://en.wikipedia.org/wiki/Church_encoding

and

and false false = (λp.λq.p q p) false false

= false false false = (λa.λb.b) false false = false

and false true = (λp.λq.p q p) false true

= false true false = (λa.λb.b) true false = false

and true false = (λp.λq.p q p) true false

= true false true = (λa.λb.a) false true = false

and true true = (λp.λq.p q p) true true

= true true true = (λa.λb.a) true true = true

true ≡ λa.λb.a

false ≡ λa.λb.b

https://en.wikipedia.org/wiki/Church_encoding

or

or false false = (λp.λq.p p q) false false
= false false false = (λa.λb.b) false false = false
or false true = (λp.λq.p p q) false true
= false false true = (λa.λb.b) false true = true
or true false = (λp.λq.p p q) true false
= true true false = (λa.λb.a) true false = true
or true true = (λp.λq.p p q) true true
= true true true = (λa.λb.a) true true = true

true ≡ λa.λb.a
false ≡ λa.λb.b

https://en.wikipedia.org/wiki/Church_encoding

not_1

not true = (λp.λa.λb.p b a) **true**
= (λp.λa.λb.p b a) (λa.λb.a)
= λa.λb.(λa.λb.a) b a = λa.λb.(λc.b) a
= λa.λb.b = false

not false = (λp.λa.λb.p b a) **false**
= (λp.λa.λb.p b a) (λa.λb.b)
= λa.λb.(λa.λb.b) b a = λa.λb.(λc.c) a
= λa.λb.a = true

true ≡ λa.λb.a
(λa.λb.a) b
= (λa.λc.a) b = (λc.b)
(λa.λb.a) b a
= (λc.b) a = b

false ≡ λa.λb.b
(λa.λb.b) b
= (λa.λc.c) b = (λc.c)
(λa.λb.b) b a
= (λc.c) a = a

https://en.wikipedia.org/wiki/Church_encoding

not_2

not true = (λp.p (λa.λb.b) (λa.λb.a)) (λa.λb.a)
= (λa.λb.a) (λa.λb.b) (λa.λb.a)
= (λa.λb.b) = false

not false = (λp.p (λa.λb.b) (λa.λb.a)) (λa.λb.b)
= (λa.λb.b) (λa.λb.b) (λa.λb.a)
= (λa.λb.a) = true

true ≡ λa.λb.a

false ≡ λa.λb.b

https://en.wikipedia.org/wiki/Church_encoding

xor

xor false false = (λa.λb.a (not b) b) false false
= false true false = (λa.λb.b) true false = false

xor false true = (λa.λb.a (not b) b) false true
= false false true = (λa.λb.b) false true = true

xor true false = (λa.λb.a (not b) b) true false
= true true false = (λa.λb.a) true false = true

xor true true = (λa.λb.a (not b) b) true true
= true false true = (λa.λb.a) false true = false

true ≡ λa.λb.a

false ≡ λa.λb.b

https://en.wikipedia.org/wiki/Church_encoding

ISZERO Predicate (1)

Testing a number for zero

Let **g = λx. FALSE**.

which returns always **FALSE** for any input

Then for any numeral **n**, therefore we have the function

n g TRUE = TRUE ***iff*** **n=0**

ISZERO = λn. n (λx. FALSE) TRUE

λf.λx x	**0**
λf.λx f x	**1**
λf.λx f (f x)	**2**
λf.λx f (f (f x))	**3**
λf.λx f (f (f (f x)))	**4**

https://www.cs.cmu.edu/~venkatg/teaching/15252-sp20/notes/lambda-calculus-slides.pdf

ISZERO Predicate (2)

n g TRUE = TRUE iff n=0

ISZERO = λn. n (λx. FALSE) TRUE

	n	(λx. FALSE)	TRUE		
0	λf.λx x	(λx. FALSE)	TRUE	→	true
1	λf.λx f x	(λx. FALSE)	TRUE	→	false
2	λf.λx f (f x)	(λx. FALSE)	TRUE	→	false
3	λf.λx f (f (f x))	(λx. FALSE)	TRUE	→	false
4	λf.λx f (f (f (f x)))	(λx. FALSE)	TRUE	→	false

f x = (λx. FALSE) TRUE = false

f (f x) = (λx. FALSE) FALSE = false

f (f (f x)) = (λx. FALSE) FALSE = false

f (f (f (f x))) = (λx. FALSE) FALSE = false

https://www.cs.cmu.edu/~venkatg/teaching/15252-sp20/notes/lambda-calculus-slides.pdf

ISZERO Predicate (3)

A **predicate** is a function that returns a boolean value.

the **ISZERO** predicate

returns **TRUE**

if its argument is the Church numeral **0**,

returns **FALSE**

if its argument is *any other* Church numeral:

ISZERO := λn.n (λx.FALSE) TRUE

n=0: **λf.λx x (λx. FALSE) TRUE → TRUE**

n=1: **λf.λx f x (λx. FALSE) TRUE → FALSE**

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

ISZERO Predicate (4)

Another approach:

T = λx.λy.x

F = λx.λy.y

af = λx.F … **af** : always false

zp = λn.((n af) T) … **zp** : zero predicate

= λn.((n λx.F) T)

Here we take a function "**af**" that always returns "**false**".

af T = (λx.F) T = F

af F = (λx.F) F = F

n g TRUE = TRUE iff n=0

ISZERO = λn. n (λx. FALSE) TRUE

= λn. ((n af) T)

https://www.cs.unc.edu/~stotts/723/Lambda/church.html

ISZERO Predicate (5)

zp = λn.((n af) T) … **zp** : zero predicate

since a church numeral is
- a **function** of two arguments, … **f x** or **g y**

we apply the number to be tested … **two**
- to "**af**" as the *first* argument … **f** or **g**
- and to "**T**" as the *second* argument. … **x** or **y**

consider **two = λf.λx.(f (f x))**
- **af** – avmpplied as the *first* argument **f**
- **T** – applied as the *second* argument **x**

two = **λf.λx.(f (f x))**
= **λg.λy.(g (g y))**

https://www.cs.unc.edu/~stotts/723/Lambda/church.html

ISZERO Predicate (6)

Let's try this for a test of zero:

(zp zero) **=> (λn ((n af) T) zero)**

=> ((zero af) T)

=> ((λg.λy. y) af) T) **… af** cannot be applied

=> ((λy. y) T) … return **T**

=> T

If the number being tested is *zero* (which is the same as "**F**"),

λf.λx.x = λx.λy.y = F

then it selects the second argument and so returns "**T**".

λx.λy.y af T = T

T = λx.λy.x

F = λx.λy.y

af = λx.F

zp = λn.((n af) T)

zero = λf.λx.x

= λg.λy.y

= λx.λy.y

= F

https://www.cs.unc.edu/~stotts/723/Lambda/church.html

ISZERO Predicate (7)

Let's try this for a test of two:

(zp two) => (λn ((n af) T) two)

=> ((two af) T)

=> ((λg.λy.(g (g y))) af) T)

=> af (af T) … always returns F

T = λx.λy.x

F = λx.λy.y

af = λx.F

zp = λn.((n af) T)

two = λf.λx.(f (f x))

= λg.λy.(g (g y))

https://www.cs.unc.edu/~stotts/723/Lambda/church.html

ISZERO Predicate (8)

Let's try this for a test of two:

(zp two) => (λn ((n af) T) two)
=> ((two af) T)
=> ((λg.λy.(g (g y))) af) T)
=> (λy (af (af y)) T)
=> (af (af T))
=> (λx F (λy F T))
=> (λx F F)
=> F

T = λx.λy.x
F = λx.λy.y
af = λx.F
zp = λn.((n af) T)

two = λf.λx.(f (f x))
= λg.λy.(g (g y))

https://www.cs.unc.edu/~stotts/723/Lambda/church.html

ISZERO Predicate (9)

If the **number** is not *zero*

then it will behave as a church numeral and

apply the **function** some number of times...

and the **function** it applies is

a **function** that returns "**F**" in all circumstances...

no matter how many times it's applications are nested.

(zp three) => (λn ((n af) T) three) => ((three af) T)

=> ((λg.λy.(g (g (g y)))) af) T)

=> (λy (af (af (af y))) T) => (af (af (af T)))

=> (λx F (λx F (λy F T)))

=> (λx F (λx F F)) => (λx F F) => F

af T = (λx.F) T = F

= (λx.F) F = F

three = λf.λx.(f (f (f x)))

= λg.λy.(g (g (g y)))

https://www.cs.unc.edu/~stotts/723/Lambda/church.html

LEQ Predicate

The **LEQ** predicate tests

whether the first argument is less-than-or-equal-to the second:

LEQ := λm.λn.ISZERO (SUB m n)

since **m = n**, if **LEQ m n** and **LEQ n m**,
it is straightforward to build a predicate for numerical equality.

The availability of predicates and
the above definition of **TRUE** and **FALSE**
make it convenient to write "**if-then-else**" expressions
in lambda calculus.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Church Pairs (1)

Church pairs are the Church encoding of the **pair** (**two-tuple**) type.

The pair is represented as a function that takes a function argument.

the argument is applied to the two **components** of the **pair**.

The definition in lambda calculus is,

pair = λx.λy.λz. z x y
first = λp. p (λx. λy. x)
second = λp. p (λx. λy. y)

https://en.wikipedia.org/wiki/Church_encoding#Church_pairs

Church Pairs (2)

first (pair a b)
= (λp. p (λx. λy. x))((λx.λy.λz. z x y) a b)
= (λp. p (λx. λy. x))(λz. z a b)
= (λz. z a b) (λx. λy. x)
= (λx. λy. x) a b = a

second (pair a b)
= (λp. p (λx. λy. y))((λx.λy.λz. z x y) a b)
= (λp. p (λx. λy. y))(λz. z a b)
= (λz. z a b) (λx. λy. y)
= (λx. λy. y) a b = b

pair = λx.λy.λz. z x y
first = λp. p (λx. λy. x)
second = λp. p (λx. λy. y)

PAIR	:= λx.λy.λf.f x y
FIRST	:= λp.p TRUE
SECOND	:= λp.p FALSE
NIL	:= λx.TRUE
NULL	:= λp.p (λx.λy.FALSE)

https://en.wikipedia.org/wiki/Church_encoding#Church_pairs

Pairs (1)

A pair (2-tuple) can be defined in terms of **TRUE** and **FALSE**,
by using the Church encoding for pairs.
For example, **PAIR** encapsulates the pair (x,y),
FIRST returns the first element of the pair,
and **SECOND** returns the second.

PAIR := **λx.λy.λf.f x y**
FIRST := **λp.p TRUE**
SECOND := **λp.p FALSE**
NIL := **λx.TRUE**
NULL := **λp.p (λx.λy.FALSE)**

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Pairs (2)

A linked list can be defined as either **NIL** for the empty list,
or the **PAIR** of an element and a smaller list.

The predicate **NULL** tests for the value **NIL**.

Alternatively, with **NIL := FALSE**, the construct
obviates the need for an explicit **NULL** test

l (λh.λt.λz.deal_with_head_h_and_tail_t) (deal_with_nil)

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Pairs (1-1)

If we bind variables **a** and **b** to values **9** and **5**
we can see how a pair is constructed:

pair 9 5 = λ9.λ5.λf.((f a) b)=λf.((f 9) 5)

These functions take two variables and
simply return one of them.

pair = λx.λy.λz. z x y
first = λp. p (λx. λy. x)
second = λp. p (λx. λy. y)

https://sookocheff.com/post/fp/representing-pairs-and-lists-in-lambda-calculus/

Pairs (1-2)

p = pair 9 5 = λ9.λ5.λf.((f a) b) = λf.((f 9) 5) = (λf. f 9 5)

assume that **p** is our previously defined **pair 9 5**

We can apply this to our pair by first creating a pair, and then applying first or second to that pair.

first p = λp. p (λx. λy. x) p
= p (λx. λy. x)
= (λf. f 9 5) (λx. λy. x)
= (λx. λy. x) 9 5 = 9

pair = λx.λy.λz. z x y
first = λp. p (λx. λy. x)
second = λp. p (λx. λy. y)

https://sookocheff.com/post/fp/representing-pairs-and-lists-in-lambda-calculus/

Pairs (1)

we'll need

pair: a → b → pair a b

first: pair a b → a

secnd: pair a b → b

Define:

pair = λx.λy.λk. k x y

first = λp. p (λx. λy. x) **= λp. p T**

second = λp . p (λx. λy. y) **= λp. p F**

pair = λx.λy.λz. z x y

first = λp. p (λx. λy. x)

second = λp. p (λx. λy. y)

http://web.cecs.pdx.edu/~black/CS311/Lecture%20Notes/Lambda%20Calculus.pdf

Pairs (1-1)

1. So far we are limited to functions of one variable:
our syntax does not allow anything else.

We have been stacking function calls to implement
multiple arguments.

Curry proved that m-ary functions can be
computed by m-deep stacked functions and vice-versa,
and thus we shall soon introduce
multi-argument functions into our syntax.

https://www.cs.rochester.edu/u/brown/173/lectures/functional/lambda/LambdaCalculusPairs.pdf

Pairs (1-2)

Meanwhile it may help to impose our intentions
on the interpretation of functions. Recall:

def make-pair = λ first. λ second. λ func. ((func first) second)
= λx. λy. λz. z x y

Does this mean make-pair is a 3-argument function?
Not really.
Here's a real 3-arg function,
which returns the "middle" (second) argument.

def middle = λ first. λ second. λ third. (second)

pair = λx.λy.λz. z x y
first = λp. p (λx. λy. x)
second = λp. p (λx. λy. y)

https://www.cs.rochester.edu/u/brown/173/lectures/functional/lambda/LambdaCalculusPairs.pdf

Pairs (2-1)

What's the difference?

Not the names (**func** not same as **third**).

In fact nothing differs but how we want to think about it!

Consider applying **make-pair** and **middle** to **arguments A**, **B**, **C**.

First consider middle.

(((λ first. λ second. λ third. (second) A)B)C) -- first ← A

= ((λ second. λ third. (second) B)C) -- second ← B

= (λ third. (B) C) -- third ← C

= B

https://www.cs.rochester.edu/u/brown/173/lectures/functional/lambda/LambdaCalculusPairs.pdf

Pairs (2-2)

def middle = λ first. λ second. λ third. (second)

Now at this point we do NOT think “oh yeah,

middle has created a **function** that requires another argument”

(though that’s true).

No, we think “middle is still looking for its third argument”.

We think of it as a 3-ary function not completely invoked.

But…

https://www.cs.rochester.edu/u/brown/173/lectures/functional/lambda/LambdaCalculusPairs.pdf

Pairs (3-1)

With a pair, analogously, after sucking up two arguments
we have
(λ func. (func first) second)
λz. z x y

And this IS interesting!
If **first** and **second** have some sort of semantics
like first-element and rest-of-list, or
then-value and else-value,
then it's like at this point we have
a little 'data structure' or 'function structure'
that's ready and waiting to have a function applied to it.

https://www.cs.rochester.edu/u/brown/173/lectures/functional/lambda/LambdaCalculusPairs.pdf

Pairs (3-2)

So usually a **pair** has its first two **arguments**

'frozen in', 'sucked up' 'already bound',

and is now sitting there with its tongue hanging out saying

"what do I do with these?"

Answer: apply yourself to some function!

https://www.cs.rochester.edu/u/brown/173/lectures/functional/lambda/LambdaCalculusPairs.pdf

Pairs (4)

Attention: Sometimes our descriptive language *obscures*
what's going on.

For instance: if a **pair** like the last example "*is applied to*" a **function**,
Then on evaluation the **function**
is "*applied to*" the two **pair components**l

So when we apply the **pairs** to their relevant operations,,
It sounds backwards but upon evaluation
the operations are *applied* to the **components** of the **pairs**.

https://www.cs.rochester.edu/u/brown/173/lectures/functional/lambda/LambdaCalculusPairs.pdf

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf

[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf