COMP2022: Formal Languages and Logic

2018, Semester 2, Week 2

Joseph Godbehere

9th August, 2018

THE UNIVERSITY OF SYDNEY

Outline

- Revision – Lambda Calculus

- Currying
- Encodings

- Functional Programming: LISP

Operations

- Application
  - Notation: $A \cdot B$
  - Expression B is applied to expression A

Operations

- Application
  - Notation: $A \cdot B$
  - Expression B is applied to expression A

- Abstraction
  - $\lambda x.M$
  - Variable x is abstracted in expression M

Rewriting

- $M[x := N]$
- Expression M, but all free occurrences of x are replaced with N
- e.g.
  - $(xyz\lambda x.(zxz))[x := A] =$
  - $(xyz\lambda x.(zxz))[y := B] =$
  - $(xyz\lambda x.(zxz))[z := C] =$

Rewriting

- $M[x := N]$

- Expression M, but all free occurrences of x are replaced with N

- e.g.
  - $(xyz\lambda x.(zxz))[x := A] = (Ayz\lambda x.(zxz))$
  - $(xyz\lambda x.(zxz))[y := B] =$
  - $(xyz\lambda x.(zxz))[z := C] =$

Rewriting

- $M[x := N]$

- Expression M, but all free occurrences of x are replaced with N

- e.g.
  - $(xyz\lambda x.(zxz))[x := A] = (Ayz\lambda x.(zxz))$
  - $(xyz\lambda x.(zxz))[y := B] = (xBz\lambda x.(zxz))$
  - $(xyz\lambda x.(zxz))[z := C] =$

Rewriting

- $M[x := N]$

- Expression M, but all free occurrences of x are replaced with N

- e.g.
  - $(xyz\lambda x.(zxz))[x := A] = (Ayz\lambda x.(zxz))$
  - $(xyz\lambda x.(zxz))[y := B] = (xBz\lambda x.(zxz))$
  - $(xyz\lambda x.(zxz))[z := C] = (xyC\lambda x.(CxC))$

α-reduction

- Rename a λ to remove a name conflict

- $\lambda x.M \equiv \lambda y.(M[x := y])$

- y must be a *new* variable
  - You must not choose a symbol that is already in use

β-reduction

- Solve an abstraction

- $(\lambda x.M) \cdot N = M[x := N]$

- Note: the free occurrences of x in M is exactly the set of occurrences which bound to the $\lambda x.$ in $(\lambda x.M)$

Outline

- Revision - Lambda Calculus

- **Currying**

- Encodings
- Functional Programming

Two arguments

- Suppose we have a function $f(x, y)$ which requires two arguments.

Two arguments

- Suppose we have a function $f(x, y)$ which requires two arguments.

- Let $F_x = \lambda y.f(x, y)$ and $F = \lambda x.F_x$

Two arguments

- Suppose we have a function $f(x, y)$ which requires two arguments.

- Let $F_x = \lambda y.f(x, y)$ and $F = \lambda x.F_x$

- F is a function which takes one input, and returns a function F_x, which will take the *next* input.

Two arguments

- Suppose we have a function $f(x, y)$ which requires two arguments.

- Let $F_x = \lambda y.f(x, y)$ and $F = \lambda x.F_x$

- F is a function which takes one input, and returns a function F_x, which will take the *next* input.
- The output of the second function will be $f(x, y)$.

Example

Normal arithmetic: $f(x, y) = (x + y)/2$

Lambda calculus: $(\lambda x.(\lambda y.(x + y)/2))$

$$\big((\lambda x.(\lambda y.(x + y)/2)) \cdot 5\big) \cdot 7$$

Example

Normal arithmetic: $f(x, y) = (x + y)/2$

Lambda calculus: $(\lambda x.(\lambda y.(x + y)/2))$

$$\big((\lambda x.(\lambda y.(x + y)/2)) \cdot 5\big) \cdot 7$$

$$= (\lambda y.(5 + y)/2) \cdot 7$$

Example

Normal arithmetic: $f(x, y) = (x + y)/2$

Lambda calculus: $(\lambda x.(\lambda y.(x + y)/2))$

$$\begin{aligned} &\big((\lambda x.(\lambda y.(x + y)/2)) \cdot 5\big) \cdot 7 \\ &= (\lambda y.(5 + y)/2) \cdot 7 \\ &= (5 + 7)/2 \end{aligned}$$

Example

Normal arithmetic: $f(x, y) = (x + y)/2$

Lambda calculus: $(\lambda x.(\lambda y.(x + y)/2))$

$$
\begin{aligned}
&\big((\lambda x.(\lambda y.(x + y)/2)) \cdot 5\big) \cdot 7 \\
&= (\lambda y.(5 + y)/2) \cdot 7 \\
&= (5 + 7)/2 \\
&= 6
\end{aligned}
$$

Currying

- A n-ary parameter function can be represented in the lambda calculus through *Currying*

Currying

- A n-ary parameter function can be represented in the lambda calculus through *Currying*

- An n argument function returns an $(n-1)$ argument function, which returns an $(n-2)$ argument function, ...

Currying

- A n-ary parameter function can be represented in the lambda calculus through *Currying*

- An n argument function returns an $(n-1)$ argument function, which returns an $(n-2)$ argument function, ...

- e.g. $(\lambda x.(\lambda y.(\lambda z.f(x,y,z)))) \cdot 1 = (\lambda y.(\lambda z.f(1,y,z)))$

Evaluation

Recall the example from earlier:

$$
\begin{aligned}
&\big((\lambda x.(\lambda y.(x+y)/2)) \cdot 5\big) \cdot 7 \\
&= (\lambda y.(5+y)/2) \cdot 7 \\
&= (5+7)/2 \\
&= 6
\end{aligned}
$$

The function is *partially evaluated* at each step

Evaluation

Recall the example from earlier:

$$
\begin{aligned}
&\big((\lambda x.(\lambda y.(x+y)/2)) \cdot 5\big) \cdot 7 \\
&= (\lambda y.(5+y)/2) \cdot 7 \\
&= (5+7)/2 \\
&= 6
\end{aligned}
$$

The function is *partially evaluated* at each step

- The first function returns $(\lambda y.(5+y)/2)$

Evaluation

Recall the example from earlier:

$$
\begin{aligned}
& \big((\lambda x.(\lambda y.(x+y)/2)) \cdot 5\big) \cdot 7 \\
&= (\lambda y.(5+y)/2) \cdot 7 \\
&= (5+7)/2 \\
&= 6
\end{aligned}
$$

The function is *partially evaluated* at each step

- The first function returns $(\lambda y.(5+y)/2)$
- 7 is then applied to the new function

Evaluation

Recall the example from earlier:

$$
\begin{aligned}
&\big((\lambda x.(\lambda y.(x+y)/2)) \cdot 5\big) \cdot 7 \\
&= (\lambda y.(5+y)/2) \cdot 7 \\
&= (5+7)/2 \\
&= 6
\end{aligned}
$$

The function is *partially evaluated* at each step

- The first function returns $(\lambda y.(5+y)/2)$
- 7 is then applied to the new function
- $(5+7)/2$ is evaluated and returned

Notation

- Too many parentheses. Let's make it simpler:

Notation

- Too many parentheses. Let's make it simpler:

- We can write $(A \cdot B)$ as $A\ B$

Notation

- Too many parentheses. Let's make it simpler:

- We can write $(A \cdot B)$ as $A\ B$

- For function application we use association to the *left*:

$$ABCDEF \equiv (((((AB)C)D)E)F)$$

Notation

- Too many parentheses. Let's make it simpler:
- We can write $(A \cdot B)$ as $A\ B$

- For function application we use association to the *left*:

$$ABCDEF \equiv (((((AB)C)D)E)F)$$

- i.e. the leftmost application happens *first*

Notation

- For function abstraction we use association to the *right*

$$\lambda x_1 x_2 x_3 \ldots x_k . M$$

Notation

- For function abstraction we use association to the *right*

$$\lambda x_1 x_2 x_3 \ldots x_k . M$$
$$= \lambda x_1 . \lambda x_2 . \lambda x_3 . \ldots \lambda x_k . M$$

Notation

- For function abstraction we use association to the *right*

$$\begin{aligned} &\lambda x_1 x_2 x_3 \ldots x_k . M \\ &= \lambda x_1 . \lambda x_2 . \lambda x_3 . \ldots \lambda x_k . M \\ &= (\lambda x_1 . (\lambda x_2 . (\lambda x_3 . (\ldots(\lambda x_k . M) \ldots)))) \end{aligned}$$

Notation

- For function abstraction we use association to the *right*

$$
\begin{aligned}
&\lambda x_1 x_2 x_3 \ldots x_k . M \\
&= \lambda x_1 . \lambda x_2 . \lambda x_3 . \ldots \lambda x_k . M \\
&= (\lambda x_1 . (\lambda x_2 . (\lambda x_3 . (\ldots (\lambda x_k . M) \ldots))))
\end{aligned}
$$

- This means the leftmost λ will match with the first input applied to the function

Notation

- Abstraction is right associative
- Application is left associative
- The abstractions and applications match up nicely:

$$(\lambda xyz.((z-x)\times y))\ 4\ 2\ 3$$

Notation

- Abstraction is right associative
- Application is left associative
- The abstractions and applications match up nicely:

$$\begin{aligned}&\left(\lambda xyz.((z-x)\times y)\right)\ 4\ 2\ 3\\ =\ &\left(\lambda yz.((z-4)\times y)\right)\ 2\ 3\end{aligned}$$

Notation

- Abstraction is right associative
- Application is left associative
- The abstractions and applications match up nicely:

$$
\begin{aligned}
&\big(\lambda xyz.((z - x) \times y)\big)\ 4\ 2\ 3 \\
&= \big(\lambda yz.((z - 4) \times y)\big)\ 2\ 3 \\
&= \big(\lambda z.((z - 4) \times 2)\big)\ 3
\end{aligned}
$$

Notation

- Abstraction is right associative
- Application is left associative
- The abstractions and applications match up nicely:

$$
\begin{aligned}
& \left(\lambda xyz.((z - x) \times y)\right)\ 4\ 2\ 3 \\
=\ & \left(\lambda yz.((z - 4) \times y)\right)\ 2\ 3 \\
=\ & \left(\lambda z.((z - 4) \times 2)\right)\ 3 \\
=\ & (3 - 4) \times 2 \\
=\ & -2
\end{aligned}
$$

Notation

- Abstraction is right associative
- Application is left associative
- If we wrote it out in full...

$$\big(\lambda xyz.((z - x) \times y)\big)\ 4\ 2\ 3$$

Notation

- Abstraction is right associative
- Application is left associative
- If we wrote it out in full...

$$\big(\lambda xyz.((z - x) \times y)\big)\ 4\ 2\ 3$$
$$= \Big(\lambda x.(\lambda y.(\lambda z.((z - x) \times y)))\Big)\ 4\ 2\ 3$$

Notation

- Abstraction is right associative
- Application is left associative
- If we wrote it out in full...

$$
\begin{aligned}
&\big(\lambda xyz.((z-x)\times y)\big)\ 4\ 2\ 3\\
&=\Big(\lambda x.\big(\lambda y.(\lambda z.((z-x)\times y))\big)\Big)\ 4\ 2\ 3\\
&=\left(\left(\left(\lambda x.\big(\lambda y.(\lambda z.((z-x)\times y))\big)\right)\cdot 4\right)\cdot 2\right)\cdot 3
\end{aligned}
$$

Notation

- Abstraction is right associative
- Application is left associative
- If we wrote it out in full...

$$
\begin{aligned}
& \big(\lambda xyz.((z - x) \times y)\big)\ 4\ 2\ 3 \\
= & \Big(\lambda x.(\lambda y.(\lambda z.((z - x) \times y)))\Big)\ 4\ 2\ 3 \\
= & \Bigg(\bigg(\Big(\lambda x.\big(\lambda y.(\lambda z.((z - x) \times y))\big)\Big) \cdot 4\bigg) \cdot 2\Bigg) \cdot 3 \\
= & \Big(\big(\lambda y.(\lambda z.((z - 4) \times y))\big) \cdot 2\Big) \cdot 3 \\
= & \big(\lambda z.((z - 4) \times 2)\big) \cdot 3 \\
= & (3 - 4) \times 2 \\
= & -2
\end{aligned}
$$

Notation

- Question:
  1. Is $\lambda x.xy = (\lambda x.(xy))$, or
  2. is $\lambda x.xy = (\lambda x.x)y$?

Notation

- Question:
  1. Is $\lambda x.xy = (\lambda x.(xy))$, or
  2. is $\lambda x.xy = (\lambda x.x)y$?

- Answer: (1), it's $(\lambda x.(xy))$

Notation

- Question:
  1. Is $\lambda x.xy = (\lambda x.(xy))$, or
  2. is $\lambda x.xy = (\lambda x.x)y$?

- Answer: (1), it's $(\lambda x.(xy))$

- Use parentheses to limit the scope of the λ if needed

Currying

- Suppose we wanted to abstract a function with k arguments:

$$(\lambda x_1 x_2 x_3 \ldots x_k . N)$$

Currying

- Suppose we wanted to abstract a function with k arguments:

$$(\lambda x_1 x_2 x_3 ... x_k . N)$$

- If we apply k arguments, $v_1 ... v_k$, we get this:

$$(\lambda x_1 x_2 x_3 ... x_k . N) v_1 v_2 v_3 ... v_k$$

Currying

- Suppose we wanted to abstract a function with k arguments:

$$(\lambda x_1 x_2 x_3 ... x_k . N)$$

- If we apply k arguments, $v_1 ... v_k$, we get this:

$$(\lambda x_1 x_2 x_3 ... x_k . N) v_1 v_2 v_3 ... v_k$$

- Each β-reduction partially evaluates the function:
  - v_1 replaces x_1. The resulting function takes $k - 1$ arguments:

$$(\lambda x_2 x_3 ... x_k . N[x_1 : v_1]) v_2 v_3 ... v_k$$

  - ... then v_2 would replace x_2, etc.

Outline

- Revision - Lambda Calculus
- Currying
- **Encodings**
- Functional Programming: LISP

But... How do we actually *do* anything?

Untyped Lambda Calculus

- Lambda calculus does not have primitives
  - No numbers
  - No arithmetic operators
  - No aggregated data types (classes etc.)
  - No control flow (only recursion!)

Untyped Lambda Calculus

- Lambda calculus does not have primitives
  - No numbers
  - No arithmetic operators
  - No aggregated data types (classes etc.)
  - No control flow (only recursion!)

- However, I'm claiming that it is computationally equivalent to a Turing Machine!

Untyped Lambda Calculus

- Lambda calculus does not have primitives
  - No numbers
  - No arithmetic operators
  - No aggregated data types (classes etc.)
  - No control flow (only recursion!)

- However, I'm claiming that it is computationally equivalent to a Turing Machine!

- So, how can we represent data types?

Untyped Lambda Calculus

- Lambda calculus does not have primitives
  - No numbers
  - No arithmetic operators
  - No aggregated data types (classes etc.)
  - No control flow (only recursion!)

- However, I'm claiming that it is computationally equivalent to a Turing Machine!

- So, how can we represent data types?
  - They must be expressed as functions, known as *encodings*

Encodings: truth

- Boolean constants:
  - TRUE $:= \lambda xy.x$
  - FALSE $:= \lambda xy.y$

Encodings: truth

- Boolean constants:
  - TRUE := $\lambda xy.x$
  - FALSE := $\lambda xy.y$

- Now we can do conditional logic:
  - IFELSE := $\lambda fxy.fxy$ has semantics similar to:

    - `if <cond> then <x> else <y>`
    - If `<cond>` is true, return result of `<x>`, otherwise `<y>`

Encodings: truth

$$IFELSE\ TRUE\ A\ B$$

ENCODINGS: TRUTH

$$
\begin{aligned}
& \mathit{IFELSE}\ \mathit{TRUE}\ A\ B \\
&= (\lambda fxy.fxy)\ (\lambda xy.x)\ A\ B \qquad \text{(macro substitution)}
\end{aligned}
$$

Encodings: truth

$$
\begin{aligned}
& \mathit{IFELSE}\ \mathit{TRUE}\ A\ B \\
&= (\lambda fxy.fxy)\ (\lambda xy.x)\ A\ B && \text{(macro substitution)} \\
&= (\lambda fay.fay)\ (\lambda xy.x)\ A\ B && (\alpha\text{-reduction})
\end{aligned}
$$

Encodings: truth

$$
\begin{aligned}
&\mathit{IFELSE}\ \mathit{TRUE}\ A\ B \\
&= (\lambda fxy.fxy)\ (\lambda xy.x)\ A\ B && \text{(macro substitution)} \\
&= (\lambda fay.fay)\ (\lambda xy.x)\ A\ B && (\alpha\text{-reduction}) \\
&= (\lambda fab.fab)\ (\lambda xy.x)\ A\ B && (\alpha\text{-reduction})
\end{aligned}
$$

Encodings: truth

$$
\begin{aligned}
& \mathit{IFELSE}\ \mathit{TRUE}\ A\ B \\
&= (\lambda fxy.fxy)\ (\lambda xy.x)\ A\ B && \text{(macro substitution)} \\
&= (\lambda fay.fay)\ (\lambda xy.x)\ A\ B && (\alpha\text{-reduction}) \\
&= (\lambda fab.fab)\ (\lambda xy.x)\ A\ B && (\alpha\text{-reduction}) \\
&= (\lambda ab.(\lambda xy.x)ab)\ A\ B && (\beta\text{-reduction})
\end{aligned}
$$

Encodings: truth

$$
\begin{aligned}
& \mathit{IFELSE}\ \mathit{TRUE}\ A\ B \\
&= (\lambda fxy.fxy)\ (\lambda xy.x)\ A\ B && \text{(macro substitution)} \\
&= (\lambda fay.fay)\ (\lambda xy.x)\ A\ B && (\alpha\text{-reduction}) \\
&= (\lambda fab.fab)\ (\lambda xy.x)\ A\ B && (\alpha\text{-reduction}) \\
&= (\lambda ab.(\lambda xy.x)ab)\ A\ B && (\beta\text{-reduction}) \\
&= (\lambda b.(\lambda xy.x)Ab)\ B && (\beta\text{-reduction})
\end{aligned}
$$

Encodings: truth

$$
\begin{aligned}
& \mathit{IFELSE}\ \mathit{TRUE}\ A\ B \\
&= (\lambda fxy.fxy)\ (\lambda xy.x)\ A\ B && \text{(macro substitution)} \\
&= (\lambda fay.fay)\ (\lambda xy.x)\ A\ B && (\alpha\text{-reduction}) \\
&= (\lambda fab.fab)\ (\lambda xy.x)\ A\ B && (\alpha\text{-reduction}) \\
&= (\lambda ab.(\lambda xy.x)ab)\ A\ B && (\beta\text{-reduction}) \\
&= (\lambda b.(\lambda xy.x)Ab)\ B && (\beta\text{-reduction}) \\
&= (\lambda xy.x)AB && (\beta\text{-reduction})
\end{aligned}
$$

Encodings: truth

$$
\begin{aligned}
& \mathit{IFELSE}\ \mathit{TRUE}\ A\ B \\
&= (\lambda fxy.fxy)\ (\lambda xy.x)\ A\ B && \text{(macro substitution)} \\
&= (\lambda fay.fay)\ (\lambda xy.x)\ A\ B && \text{(α-reduction)} \\
&= (\lambda fab.fab)\ (\lambda xy.x)\ A\ B && \text{(α-reduction)} \\
&= (\lambda ab.(\lambda xy.x)ab)\ A\ B && \text{(β-reduction)} \\
&= (\lambda b.(\lambda xy.x)Ab)\ B && \text{(β-reduction)} \\
&= (\lambda xy.x)AB) && \text{(β-reduction)} \\
&= (\lambda y.A)B) && \text{(β-reduction)}
\end{aligned}
$$

Encodings: truth

$$
\begin{aligned}
& \mathit{IFELSE}\ \mathit{TRUE}\ A\ B \\
&= (\lambda fxy.fxy)\ (\lambda xy.x)\ A\ B && \text{(macro substitution)} \\
&= (\lambda fay.fay)\ (\lambda xy.x)\ A\ B && (\alpha\text{-reduction}) \\
&= (\lambda fab.fab)\ (\lambda xy.x)\ A\ B && (\alpha\text{-reduction}) \\
&= (\lambda ab.(\lambda xy.x)ab)\ A\ B && (\beta\text{-reduction}) \\
&= (\lambda b.(\lambda xy.x)Ab)\ B && (\beta\text{-reduction}) \\
&= (\lambda xy.x)AB) && (\beta\text{-reduction}) \\
&= (\lambda y.A)B) && (\beta\text{-reduction}) \\
&= A && (\beta\text{-reduction})
\end{aligned}
$$

Encodings: truth

$IFELSE\ FALSE\ A\ B$

Encodings: truth

$$
\begin{aligned}
&\mathit{IFELSE}\ \mathit{FALSE}\ A\ B \\
&= (\lambda fxy.fxy)\ (\lambda xy.y)\ A\ B \qquad \text{(macro substitution)}
\end{aligned}
$$

Encodings: truth

$$
\begin{aligned}
& \mathit{IFELSE}\ \mathit{FALSE}\ A\ B \\
&= (\lambda fxy.fxy)\ (\lambda xy.y)\ A\ B && \text{(macro substitution)} \\
&= (\lambda fay.fay)\ (\lambda xy.y)\ A\ B && (\alpha\text{-reduction})
\end{aligned}
$$

Encodings: truth

$$
\begin{aligned}
& \mathit{IFELSE}\ \mathit{FALSE}\ A\ B \\
&= (\lambda fxy.fxy)\ (\lambda xy.y)\ A\ B && \text{(macro substitution)} \\
&= (\lambda fay.fay)\ (\lambda xy.y)\ A\ B && \text{(α-reduction)} \\
&= (\lambda fab.fab)\ (\lambda xy.y)\ A\ B && \text{(α-reduction)}
\end{aligned}
$$

Encodings: truth

$$
\begin{aligned}
& \mathit{IFELSE}\ \mathit{FALSE}\ A\ B \\
&= (\lambda fxy.fxy)\ (\lambda xy.y)\ A\ B && \text{(macro substitution)} \\
&= (\lambda fay.fay)\ (\lambda xy.y)\ A\ B && (\alpha\text{-reduction}) \\
&= (\lambda fab.fab)\ (\lambda xy.y)\ A\ B && (\alpha\text{-reduction}) \\
&= (\lambda ab.(\lambda xy.y)ab)\ A\ B && (\beta\text{-reduction})
\end{aligned}
$$

Encodings: truth

$$
\begin{aligned}
& \mathit{IFELSE}\ \mathit{FALSE}\ A\ B \\
&= (\lambda fxy.fxy)\ (\lambda xy.y)\ A\ B && \text{(macro substitution)} \\
&= (\lambda fay.fay)\ (\lambda xy.y)\ A\ B && (\alpha\text{-reduction}) \\
&= (\lambda fab.fab)\ (\lambda xy.y)\ A\ B && (\alpha\text{-reduction}) \\
&= (\lambda ab.(\lambda xy.y)ab)\ A\ B && (\beta\text{-reduction}) \\
&= (\lambda b.(\lambda xy.y)Ab)\ B && (\beta\text{-reduction})
\end{aligned}
$$

Encodings: truth

$$
\begin{aligned}
& \mathit{IFELSE}\ \mathit{FALSE}\ A\ B \\
&= (\lambda fxy.fxy)\ (\lambda xy.y)\ A\ B && \text{(macro substitution)} \\
&= (\lambda fay.fay)\ (\lambda xy.y)\ A\ B && (\alpha\text{-reduction}) \\
&= (\lambda fab.fab)\ (\lambda xy.y)\ A\ B && (\alpha\text{-reduction}) \\
&= (\lambda ab.(\lambda xy.y)ab)\ A\ B && (\beta\text{-reduction}) \\
&= (\lambda b.(\lambda xy.y)Ab)\ B && (\beta\text{-reduction}) \\
&= (\lambda xy.y)\ A\ B && (\beta\text{-reduction})
\end{aligned}
$$

Encodings: truth

$$
\begin{aligned}
& \mathit{IFELSE}\ \mathit{FALSE}\ A\ B \\
&= (\lambda fxy.fxy)\ (\lambda xy.y)\ A\ B && \text{(macro substitution)} \\
&= (\lambda fay.fay)\ (\lambda xy.y)\ A\ B && (\alpha\text{-reduction}) \\
&= (\lambda fab.fab)\ (\lambda xy.y)\ A\ B && (\alpha\text{-reduction}) \\
&= (\lambda ab.(\lambda xy.y)ab)\ A\ B && (\beta\text{-reduction}) \\
&= (\lambda b.(\lambda xy.y)Ab)\ B && (\beta\text{-reduction}) \\
&= (\lambda xy.y)AB) && (\beta\text{-reduction}) \\
&= (\lambda y.y)B) && (\beta\text{-reduction})
\end{aligned}
$$

Encodings: truth

$$
\begin{aligned}
& IFELSE\ FALSE\ A\ B \\
&= (\lambda fxy.fxy)\ (\lambda xy.y)\ A\ B && \text{(macro substitution)} \\
&= (\lambda fay.fay)\ (\lambda xy.y)\ A\ B && (\alpha\text{-reduction}) \\
&= (\lambda fab.fab)\ (\lambda xy.y)\ A\ B && (\alpha\text{-reduction}) \\
&= (\lambda ab.(\lambda xy.y)ab)\ A\ B && (\beta\text{-reduction}) \\
&= (\lambda b.(\lambda xy.y)Ab)\ B && (\beta\text{-reduction}) \\
&= (\lambda xy.y)AB) && (\beta\text{-reduction}) \\
&= (\lambda y.y)B) && (\beta\text{-reduction}) \\
&= B && (\beta\text{-reduction})
\end{aligned}
$$

Encodings: truth

- Boolean constants:
  - TRUE := $\lambda xy.x$
  - FALSE := $\lambda xy.y$

- IFELSE := $\lambda fxy.fxy$

- Boolean operators:
  - NOT := $\lambda fxy.fyx$
  - OR := $\lambda xy.xxy$
  - AND := $\lambda xy.xyx$

Encodings: NOT

- NOT $:= \lambda fxy.fyx$

- NOT is a function which takes 3 arguments
  - Suppose f was a function which takes 2 arguments
  - x, y would be those arguments

Encodings: NOT

- NOT := $\lambda fxy.fyx$

- NOT is a function which takes 3 arguments
  - Suppose f was a function which takes 2 arguments
  - x, y would be those arguments

- i.e. NOT outputs f, except its arguments have swapped around!

Encodings: truth

NOT TRUE

Encodings: truth

$$
\begin{aligned}
& \mathit{NOT}\ \mathit{TRUE} \\
& = (\lambda fxy.fyx)(\lambda xy.x) && \text{(macro substitution)}
\end{aligned}
$$

Encodings: truth

$$
\begin{aligned}
& \mathit{NOT}\ \mathit{TRUE} \\
&= (\lambda fxy.fyx)(\lambda xy.x) && \text{(macro substitution)} \\
&= (\lambda fxy.fyx)(\lambda ay.a) && (\alpha\text{-reduction})
\end{aligned}
$$

Encodings: truth

$$
\begin{aligned}
&NOT\ TRUE \\
&= (\lambda fxy.fyx)(\lambda xy.x) && \text{(macro substitution)} \\
&= (\lambda fxy.fyx)(\lambda ay.a) && (\alpha\text{-reduction}) \\
&= (\lambda fxy.fyx)(\lambda ab.a) && (\alpha\text{-reduction})
\end{aligned}
$$

Encodings: truth

$$
\begin{aligned}
& \mathit{NOT}\ \mathit{TRUE} && \\
&= (\lambda fxy.fyx)(\lambda xy.x) && \text{(macro substitution)} \\
&= (\lambda fxy.fyx)(\lambda ay.a) && (\alpha\text{-reduction}) \\
&= (\lambda fxy.fyx)(\lambda ab.a) && (\alpha\text{-reduction}) \\
&= \lambda xy.(\lambda ab.a)yx && (\beta\text{-reduction})
\end{aligned}
$$

Encodings: truth

$$
\begin{aligned}
& \mathit{NOT}\ \mathit{TRUE} \\
&= (\lambda fxy.fyx)(\lambda xy.x) && \text{(macro substitution)} \\
&= (\lambda fxy.fyx)(\lambda ay.a) && \text{(α-reduction)} \\
&= (\lambda fxy.fyx)(\lambda ab.a) && \text{(α-reduction)} \\
&= \lambda xy.(\lambda ab.a)yx && \text{(β-reduction)} \\
&= \lambda xy.(\lambda b.y)x && \text{(β-reduction)}
\end{aligned}
$$

Encodings: truth

$$
\begin{aligned}
& \mathit{NOT}\ \mathit{TRUE} \\
&= (\lambda fxy.fyx)(\lambda xy.x) && \text{(macro substitution)} \\
&= (\lambda fxy.fyx)(\lambda ay.a) && (\alpha\text{-reduction}) \\
&= (\lambda fxy.fyx)(\lambda ab.a) && (\alpha\text{-reduction}) \\
&= \lambda xy.(\lambda ab.a)yx && (\beta\text{-reduction}) \\
&= \lambda xy.(\lambda b.y)x && (\beta\text{-reduction}) \\
&= \lambda xy.y && (\beta\text{-reduction})
\end{aligned}
$$

Encodings: truth

$$
\begin{aligned}
& \mathit{NOT}\ \mathit{TRUE} & \\
&= (\lambda fxy.fyx)(\lambda xy.x) & \text{(macro substitution)}\\
&= (\lambda fxy.fyx)(\lambda ay.a) & \text{(α-reduction)}\\
&= (\lambda fxy.fyx)(\lambda ab.a) & \text{(α-reduction)}\\
&= \lambda xy.(\lambda ab.a)yx & \text{(β-reduction)}\\
&= \lambda xy.(\lambda b.y)x & \text{(β-reduction)}\\
&= \lambda xy.y & \text{(β-reduction)}\\
&= \mathit{FALSE} & \text{(macro substitution)}
\end{aligned}
$$

Encodings: numbers

- The natural numbers can be thought of as a sequence, starting from 0, and successively increasing by one.

Encodings: numbers

- The natural numbers can be thought of as a sequence, starting from 0, and successively increasing by one.
- More formally, we can define them inductively:

Encodings: numbers

- The natural numbers can be thought of as a sequence, starting from 0, and successively increasing by one.
- More formally, we can define them inductively:
  - Basic clause: 0 is a number and is in the set

Encodings: numbers

- The natural numbers can be thought of as a sequence, starting from 0, and successively increasing by one.

- More formally, we can define them inductively:
  - Basic clause: 0 is a number and is in the set
  - Inductive clause: for any element x in the natural numbers, $x + 1$ is an element of the natural numbers

Encodings: numbers

- The natural numbers can be thought of as a sequence, starting from 0, and successively increasing by one.

- More formally, we can define them inductively:
  - Basic clause: 0 is a number and is in the set
  - Inductive clause: for any element x in the natural numbers, $x + 1$ is an element of the natural numbers
  - Extremal clause: nothing is in the set of natural numbers unless it is obtained by the inductive clause and basis clause

Church numerals

- Natural numbers in lambda calculus have two constructors:
  - ZERO $:= \lambda xy.y$
    - This represents 0
    - It is the same formula we used to encode FALSE

Church numerals

- Natural numbers in lambda calculus have two constructors:
  - ZERO $:= \lambda xy.y$
    - This represents 0
    - It is the same formula we used to encode FALSE
  - SUCCESSOR $:= \lambda xyz.y(xyz)$
    - Returns the next number in the sequence

Church numerals

- Natural numbers in lambda calculus have two constructors:
  - ZERO $:= \lambda xy.y$
    - This represents 0
    - It is the same formula we used to encode FALSE

  - SUCCESSOR $:= \lambda xyz.y(xyz)$
    - Returns the next number in the sequence

- We're now ready to start constructing the natural numbers!

Church numerals

$$
\begin{aligned}
& ONE \\
= {} & SUCCESSOR\ ZERO
\end{aligned}
$$

Church numerals

$$
\begin{aligned}
& \mathit{ONE} \\
&= \mathit{SUCCESSOR}\ \mathit{ZERO} \\
&= (\lambda xyz.y(xyz))\,(\lambda xy.y) \quad \text{(macro)}
\end{aligned}
$$

Church numerals

$$
\begin{aligned}
& ONE \\
&= SUCCESSOR\ ZERO \\
&= (\lambda xyz.y(xyz))(\lambda xy.y) && \text{(macro)} \\
&= (\lambda xyz.y(xyz))(\lambda ab.b) && (\alpha)
\end{aligned}
$$

Church numerals

$$
\begin{aligned}
& ONE \\
&= SUCCESSOR\ ZERO \\
&= (\lambda xyz.y(xyz))(\lambda xy.y) && \text{(macro)} \\
&= (\lambda xyz.y(xyz))(\lambda ab.b) && (\alpha) \\
&= \lambda yz.y((\lambda ab.b)yz) && (\beta)
\end{aligned}
$$

Church numerals

$$
\begin{aligned}
& ONE \\
&= SUCCESSOR\ ZERO \\
&= (\lambda xyz.y(xyz))(\lambda xy.y) && \text{(macro)} \\
&= (\lambda xyz.y(xyz))(\lambda ab.b) && (\alpha) \\
&= \lambda yz.y((\lambda ab.b)yz) && (\beta) \\
&= \lambda yz.y((\lambda b.b)z) && (\beta)
\end{aligned}
$$

Church numerals

$$
\begin{aligned}
& \mathit{ONE} \\
&= \mathit{SUCCESSOR}\ \mathit{ZERO} \\
&= (\lambda xyz.y(xyz))(\lambda xy.y) && \text{(macro)} \\
&= (\lambda xyz.y(xyz))(\lambda ab.b) && (\alpha) \\
&= \lambda yz.y((\lambda ab.b)yz) && (\beta) \\
&= \lambda yz.y((\lambda b.b)z) && (\beta) \\
&= \lambda yz.yz && (\beta)
\end{aligned}
$$

Church numerals

$$
\begin{aligned}
& TWO \\
= \ & SUCCESSOR\ ONE \\
= \ & (\lambda xyz.y(xyz))(\lambda yz.yz) \quad \text{(macro)}
\end{aligned}
$$

Church numerals

$$
\begin{aligned}
& TWO \\
& = SUCCESSOR\ ONE \\
& = (\lambda xyz.y(xyz))(\lambda yz.yz) && \text{(macro)} \\
& = (\lambda xyz.y(xyz))(\lambda ab.ab) && (\alpha)
\end{aligned}
$$

Church numerals

$$
\begin{aligned}
& TWO \\
&= SUCCESSOR\ ONE \\
&= (\lambda xyz.y(xyz))(\lambda yz.yz) && \text{(macro)} \\
&= (\lambda xyz.y(xyz))(\lambda ab.ab) && (\alpha) \\
&= \lambda yz.y((\lambda ab.ab)yz) && (\beta)
\end{aligned}
$$

Church numerals

$$
\begin{aligned}
& TWO \\
&= SUCCESSOR\ ONE \\
&= (\lambda xyz.y(xyz))(\lambda yz.yz) && \text{(macro)} \\
&= (\lambda xyz.y(xyz))(\lambda ab.ab) && (\alpha) \\
&= \lambda yz.y((\lambda ab.ab)yz) && (\beta) \\
&= \lambda yz.y((\lambda b.yb)z) && (\beta)
\end{aligned}
$$

Church numerals

$$
\begin{aligned}
& TWO \\
&= SUCCESSOR\ ONE \\
&= (\lambda xyz.y(xyz))(\lambda yz.yz) && \text{(macro)} \\
&= (\lambda xyz.y(xyz))(\lambda ab.ab) && (\alpha) \\
&= \lambda yz.y((\lambda ab.ab)yz) && (\beta) \\
&= \lambda yz.y((\lambda b.yb)z) && (\beta) \\
&= \lambda yz.y(yz) && (\beta)
\end{aligned}
$$

Church numerals

$$
\begin{aligned}
& THREE \\
&= SUCCESSOR\ TWO \\
&= \ldots \\
&= \lambda yz.y(y(yz))
\end{aligned}
$$

Church numerals

$$
\begin{aligned}
& \mathit{THREE} \\
&= \mathit{SUCCESSOR}\ \mathit{TWO} \\
&= \ldots \\
&= \lambda yz.y(y(yz))
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{FOUR} \\
&= \mathit{SUCCESSOR}\ \mathit{THREE} \\
&= \ldots \\
&= \lambda yz.y(y(y(yz)))
\end{aligned}
$$

Arithmetic?

- We have numbers. Do they work?

Arithmetic?

- We have numbers. Do they work?

- Arithmetic:
  - ADD $:= \lambda xypq.xp(ypq)$
  - MULT $:= \lambda xyz.x(yz)$
  - EXP $:= \lambda xy.yx$

Addition example

$$ADD\ TWO\ THREE$$

Addition example

$$\begin{aligned} & \mathit{ADD}\ \mathit{TWO}\ \mathit{THREE} \\ = & \big(\lambda xypq.xp(ypq)\big)\ (\lambda yz.y(yz))\ (\lambda yz.y(y(yz))) \end{aligned}$$

Addition example

$$
\begin{aligned}
& ADD\ TWO\ THREE \\
&= (\lambda xypq.xp(ypq))\ (\lambda yz.y(yz))\ (\lambda yz.y(y(yz))) \\
&= (\lambda xypq.xp(ypq))\ (\lambda ab.a(ab))\ (\lambda cd.c(c(cd))) && (\alpha)
\end{aligned}
$$

Addition example

$$
\begin{aligned}
& ADD\ TWO\ THREE \\
&= \big(\lambda xypq.xp(ypq)\big)\ (\lambda yz.y(yz))\ (\lambda yz.y(y(yz))) \\
&= \big(\lambda xypq.xp(ypq)\big)\ (\lambda ab.a(ab))\ (\lambda cd.c(c(cd))) && (\alpha) \\
&= \big(\lambda ypq.(\lambda ab.a(ab))p(ypq)\big)\ (\lambda cd.c(c(cd))) && (\beta)
\end{aligned}
$$

Addition example

$$
\begin{aligned}
& \mathit{ADD}\ \mathit{TWO}\ \mathit{THREE} \\
&= \big(\lambda xypq.xp(ypq)\big)\ (\lambda yz.y(yz))\ (\lambda yz.y(y(yz))) \\
&= \big(\lambda xypq.xp(ypq)\big)\ (\lambda ab.a(ab))\ (\lambda cd.c(c(cd))) && (\alpha) \\
&= \big(\lambda ypq.(\lambda ab.a(ab))p(ypq)\big)\ (\lambda cd.c(c(cd))) && (\beta) \\
&= \big(\lambda ypq.(\lambda b.p(pb))(ypq)\big)\ (\lambda cd.c(c(cd))) && (\beta)
\end{aligned}
$$

Addition example

$$
\begin{aligned}
& ADD\ TWO\ THREE \\
&= \big(\lambda xypq.xp(ypq)\big)\ (\lambda yz.y(yz))\ (\lambda yz.y(y(yz))) \\
&= \big(\lambda xypq.xp(ypq)\big)\ (\lambda ab.a(ab))\ (\lambda cd.c(c(cd))) && (\alpha) \\
&= \big(\lambda ypq.(\lambda ab.a(ab))p(ypq)\big)\ (\lambda cd.c(c(cd))) && (\beta) \\
&= \big(\lambda ypq.(\lambda b.p(pb))(ypq)\big)\ (\lambda cd.c(c(cd))) && (\beta) \\
&= \big(\lambda ypq.p(p(ypq))\big)\ (\lambda cd.c(c(cd))) && (\beta)
\end{aligned}
$$

Addition example

$$
\begin{aligned}
& ADD\ TWO\ THREE \\
&= \big(\lambda xypq.xp(ypq)\big)\ (\lambda yz.y(yz))\ (\lambda yz.y(y(yz))) \\
&= \big(\lambda xypq.xp(ypq)\big)\ (\lambda ab.a(ab))\ (\lambda cd.c(c(cd))) && (\alpha) \\
&= \big(\lambda ypq.(\lambda ab.a(ab))p(ypq)\big)\ (\lambda cd.c(c(cd))) && (\beta) \\
&= \big(\lambda ypq.(\lambda b.p(pb))(ypq)\big)\ (\lambda cd.c(c(cd))) && (\beta) \\
&= \big(\lambda ypq.p(p(ypq))\big)\ (\lambda cd.c(c(cd))) && (\beta) \\
&= \lambda pq.p(p((\lambda cd.c(c(cd)))pq)) && (\beta)
\end{aligned}
$$

Addition example

$$
\begin{aligned}
& \mathit{ADD}\ \mathit{TWO}\ \mathit{THREE} \\
&= \big(\lambda xypq.xp(ypq)\big)\ (\lambda yz.y(yz))\ (\lambda yz.y(y(yz))) \\
&= \big(\lambda xypq.xp(ypq)\big)\ (\lambda ab.a(ab))\ (\lambda cd.c(c(cd))) && (\alpha) \\
&= \big(\lambda ypq.(\lambda ab.a(ab))p(ypq)\big)\ (\lambda cd.c(c(cd))) && (\beta) \\
&= \big(\lambda ypq.(\lambda b.p(pb))(ypq)\big)\ (\lambda cd.c(c(cd))) && (\beta) \\
&= \big(\lambda ypq.p(p(ypq))\big)\ (\lambda cd.c(c(cd))) && (\beta) \\
&= \lambda pq.p(p((\lambda cd.c(c(cd)))pq)) && (\beta) \\
&= \lambda pq.p(p((\lambda d.p(p(pd)))q)) && (\beta)
\end{aligned}
$$

Addition example

$$
\begin{aligned}
&\mathit{ADD}\ \mathit{TWO}\ \mathit{THREE} \\
&= \big(\lambda xypq.xp(ypq)\big)\ (\lambda yz.y(yz))\ (\lambda yz.y(y(yz))) \\
&= \big(\lambda xypq.xp(ypq)\big)\ (\lambda ab.a(ab))\ (\lambda cd.c(c(cd))) && (\alpha) \\
&= \big(\lambda ypq.(\lambda ab.a(ab))p(ypq)\big)\ (\lambda cd.c(c(cd))) && (\beta) \\
&= \big(\lambda ypq.(\lambda b.p(pb))(ypq)\big)\ (\lambda cd.c(c(cd))) && (\beta) \\
&= \big(\lambda ypq.p(p(ypq))\big)\ (\lambda cd.c(c(cd))) && (\beta) \\
&= \lambda pq.p(p((\lambda cd.c(c(cd)))pq)) && (\beta) \\
&= \lambda pq.p(p((\lambda d.p(p(pd)))q)) && (\beta) \\
&= \lambda pq.p(p(p(p(pq)))) && (\beta)
\end{aligned}
$$

Addition example

$$
\begin{aligned}
& \mathit{ADD}\ \mathit{TWO}\ \mathit{THREE} \\
&= \big(\lambda xypq.xp(ypq)\big)\ (\lambda yz.y(yz))\ (\lambda yz.y(y(yz))) \\
&= \big(\lambda xypq.xp(ypq)\big)\ (\lambda ab.a(ab))\ (\lambda cd.c(c(cd))) && (\alpha) \\
&= \big(\lambda ypq.(\lambda ab.a(ab))p(ypq)\big)\ (\lambda cd.c(c(cd))) && (\beta) \\
&= \big(\lambda ypq.(\lambda b.p(pb))(ypq)\big)\ (\lambda cd.c(c(cd))) && (\beta) \\
&= \big(\lambda ypq.p(p(ypq))\big)\ (\lambda cd.c(c(cd))) && (\beta) \\
&= \lambda pq.p(p((\lambda cd.c(c(cd)))pq)) && (\beta) \\
&= \lambda pq.p(p((\lambda d.p(p(pd)))q)) && (\beta) \\
&= \lambda pq.p(p(p(p(pq)))) && (\beta) \\
&= \mathit{FIVE}
\end{aligned}
$$

Multiplication example

$$MULT\ EIGHT\ THIRTEEN$$

Multiplication example

$$
\begin{aligned}
& MULT\ EIGHT\ THIRTEEN \\
&= (\lambda xyz.x(yz))(\lambda fx.f(f(f(f(f(f(f(fx))))))))\\
&\quad (\lambda fx.f(f(f(f(f(f(f(f(f(f(f(f(fx)))))))))))))
\end{aligned}
$$

Multiplication example

$$
\begin{aligned}
& MULT\ EIGHT\ THIRTEEN \\
&= (\lambda xyz.x(yz))(\lambda fx.f(f(f(f(f(f(f(fx))))))))\\
&\quad (\lambda fx.f(f(f(f(f(f(f(f(f(f(f(f(fx)))))))))))))) \\
&= \ldots
\end{aligned}
$$

Just kidding

Multiplication example

$$MULT\ TWO\ THREE$$

Multiplication example

$$\begin{aligned} & \mathit{MULT}\ \mathit{TWO}\ \mathit{THREE} \\ = & (\lambda xyz.x(yz))\ \mathit{TWO}\ \mathit{THREE} \end{aligned}$$

Multiplication example

$$
\begin{aligned}
& MULT\ TWO\ THREE \\
= & (\lambda xyz.x(yz))\ TWO\ THREE \\
= & (\lambda yz.TWO\ (yz))\ THREE
\end{aligned}
$$

Multiplication example

$$
\begin{aligned}
& MULT\ TWO\ THREE \\
= & (\lambda xyz.x(yz))\ TWO\ THREE \\
= & (\lambda yz.TWO\ (yz))\ THREE \\
= & \lambda z.TWO\ (THREE\ z)
\end{aligned}
$$

Multiplication example

$$
\begin{aligned}
& \mathit{MULT}\ \mathit{TWO}\ \mathit{THREE} \\
&= (\lambda xyz.x(yz))\ \mathit{TWO}\ \mathit{THREE} \\
&= (\lambda yz.\mathit{TWO}\ (yz))\ \mathit{THREE} \\
&= \lambda z.\mathit{TWO}\ (\mathit{THREE}\ z) \\
&= \lambda z.(\lambda fx.f(fx))(\mathit{THREE}\ z)
\end{aligned}
$$

Multiplication example

$$
\begin{aligned}
& MULT\ TWO\ THREE \\
&= (\lambda xyz.x(yz))\ TWO\ THREE \\
&= (\lambda yz.TWO\ (yz))\ THREE \\
&= \lambda z.TWO\ (THREE\ z) \\
&= \lambda z.(\lambda fx.f(fx))(THREE\ z) \\
&= \lambda z.\Big(\lambda x.(THREE\ z)\big((THREE\ z)x\big)\Big)
\end{aligned}
$$

Multiplication example

$$
\begin{aligned}
& \mathit{MULT}\ \mathit{TWO}\ \mathit{THREE} \\
&= (\lambda xyz.x(yz))\ \mathit{TWO}\ \mathit{THREE} \\
&= (\lambda yz.\mathit{TWO}\ (yz))\ \mathit{THREE} \\
&= \lambda z.\mathit{TWO}\ (\mathit{THREE}\ z) \\
&= \lambda z.(\lambda fx.f(fx))(\mathit{THREE}\ z) \\
&= \lambda z.\Big(\lambda x.(\mathit{THREE}\ z)\big((\mathit{THREE}\ z)x\big)\Big) \\
&= \lambda zx.(\mathit{THREE}\ z)\big((\mathit{THREE}\ z)x\big)
\end{aligned}
$$

Multiplication example

$$
\begin{aligned}
& \mathit{MULT}\ \mathit{TWO}\ \mathit{THREE} \\
&= (\lambda xyz.x(yz))\ \mathit{TWO}\ \mathit{THREE} \\
&= (\lambda yz.\mathit{TWO}\ (yz))\ \mathit{THREE} \\
&= \lambda z.\mathit{TWO}\ (\mathit{THREE}\ z) \\
&= \lambda z.(\lambda fx.f(fx))(\mathit{THREE}\ z) \\
&= \lambda z.\Big(\lambda x.(\mathit{THREE}\ z)\big((\mathit{THREE}\ z)x\big)\Big) \\
&= \lambda zx.(\mathit{THREE}\ z)\big((\mathit{THREE}\ z)x\big) \\
&= \lambda zx.\big((\lambda fx.f(f(fx)))\ z\big)\big(((\lambda fx.f(f(fx)))\ z)x\big)
\end{aligned}
$$

Multiplication example

$$
\begin{aligned}
& \mathit{MULT}\ \mathit{TWO}\ \mathit{THREE} \\
&= (\lambda xyz.x(yz))\ \mathit{TWO}\ \mathit{THREE} \\
&= (\lambda yz.\mathit{TWO}\ (yz))\ \mathit{THREE} \\
&= \lambda z.\mathit{TWO}\ (\mathit{THREE}\ z) \\
&= \lambda z.(\lambda fx.f(fx))(\mathit{THREE}\ z) \\
&= \lambda z.\Big(\lambda x.(\mathit{THREE}\ z)\big((\mathit{THREE}\ z)x\big)\Big) \\
&= \lambda zx.(\mathit{THREE}\ z)\big((\mathit{THREE}\ z)x\big) \\
&= \lambda zx.\big((\lambda fx.f(f(fx)))\ z\big)\big(((\lambda fx.f(f(fx)))\ z)x\big) \\
&= \lambda zx.\big(\lambda x.z(z(zx))\big)\big((\lambda x.z(z(zx)))x\big)
\end{aligned}
$$

Multiplication example

$$
\begin{aligned}
& \mathit{MULT}\ \mathit{TWO}\ \mathit{THREE} \\
&= (\lambda xyz.x(yz))\ \mathit{TWO}\ \mathit{THREE} \\
&= (\lambda yz.\mathit{TWO}\ (yz))\ \mathit{THREE} \\
&= \lambda z.\mathit{TWO}\ (\mathit{THREE}\ z) \\
&= \lambda z.(\lambda fx.f(fx))(\mathit{THREE}\ z) \\
&= \lambda z.\Big(\lambda x.(\mathit{THREE}\ z)\big((\mathit{THREE}\ z)x\big)\Big) \\
&= \lambda zx.(\mathit{THREE}\ z)\big((\mathit{THREE}\ z)x\big) \\
&= \lambda zx.\big((\lambda fx.f(f(fx)))\ z\big)\big(((\lambda fx.f(f(fx)))\ z)x\big) \\
&= \lambda zx.\big(\lambda x.z(z(zx))\big)\big((\lambda x.z(z(zx)))x\big) \\
&= \lambda zx.\big(\lambda x.z(z(zx))\big)\big(z(z(zx))\big)
\end{aligned}
$$

Multiplication example

$$
\begin{aligned}
& MULT\ TWO\ THREE \\
&= (\lambda xyz.x(yz))\ TWO\ THREE \\
&= (\lambda yz.TWO\ (yz))\ THREE \\
&= \lambda z.TWO\ (THREE\ z) \\
&= \lambda z.(\lambda fx.f(fx))(THREE\ z) \\
&= \lambda z.\Big(\lambda x.(THREE\ z)\big((THREE\ z)x\big)\Big) \\
&= \lambda zx.(THREE\ z)\big((THREE\ z)x\big) \\
&= \lambda zx.\big((\lambda fx.f(f(fx)))\ z\big)\big(((\lambda fx.f(f(fx)))\ z)x\big) \\
&= \lambda zx.\big(\lambda x.z(z(zx))\big)\big((\lambda x.z(z(zx)))x\big) \\
&= \lambda zx.\big(\lambda x.z(z(zx))\big)\big(z(z(zx))\big) \\
&= \lambda zx.z(z(z(z(z(zx)))))
\end{aligned}
$$

Multiplication example

$$
\begin{aligned}
& \mathit{MULT}\ \mathit{TWO}\ \mathit{THREE} \\
&= (\lambda xyz.x(yz))\ \mathit{TWO}\ \mathit{THREE} \\
&= (\lambda yz.\mathit{TWO}\ (yz))\ \mathit{THREE} \\
&= \lambda z.\mathit{TWO}\ (\mathit{THREE}\ z) \\
&= \lambda z.(\lambda fx.f(fx))(\mathit{THREE}\ z) \\
&= \lambda z.\Big(\lambda x.(\mathit{THREE}\ z)\big((\mathit{THREE}\ z)x\big)\Big) \\
&= \lambda zx.(\mathit{THREE}\ z)\big((\mathit{THREE}\ z)x\big) \\
&= \lambda zx.\big((\lambda fx.f(f(fx)))\ z\big)\big(((\lambda fx.f(f(fx)))\ z)x\big) \\
&= \lambda zx.\big(\lambda x.z(z(zx))\big)\big((\lambda x.z(z(zx)))x\big) \\
&= \lambda zx.\big(\lambda x.z(z(zx))\big)\big(z(z(zx))\big) \\
&= \lambda zx.z(z(z(z(z(zx))))) \\
&= \mathit{SIX}
\end{aligned}
$$

Recursion

In imperative languages, we can easily write recursive code:


```
def factorial(x):
    if x == 1:
        return 1
    return x*factorial(x-1)
```


... by referencing the method itself by name.

So far, we haven't directly seen iteration or recursion in the lambda calculus.

Recursion

In the last tutorial you tried to reduce:

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

Recursion

In the last tutorial you tried to reduce:

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

… and discovered that it looped forever.

Recursion

In the last tutorial you tried to reduce:

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

... and discovered that it looped forever.

This is related to a slightly more useful construct called the Y Combinator:

$$\mathrm{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Recursion

In the last tutorial you tried to reduce:

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

… and discovered that it looped forever.

This is related to a slightly more useful construct called the Y Combinator:

$$\mathrm{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Next week, we'll use this to compute recursive functions in the lambda calculus.

Outline

- Revision - Lambda Calculus

- Currying

- Encodings
- **Functional Programming: LISP**

LISP

- LISP is the second oldest programming language in common use
- Invented in 1958 by John McCarthy

- Was very popular in the AI boom
- Is a functional programming language
- Is a practical implementation of the Lambda Calculus

- Has many dialects (e.g. Clojure, Common Lisp, Racket, Scheme, etc.)

LISP = LISt Processing

- LISP has atoms
  - Numbers, e.g. 10
  - Identifiers, e.g. Foo
  - Strings, e.g. "filename"
- LISP has lists
  - can contain other lists
  - can contain atoms
  - can contain nothing (empty)
- *very* small syntax:

```
<object> ::= <atoms> | <list>
<list> ::= "(" { <object> } ")"
```

List Examples in LISP


```
(1 2 3)
()
(+ 1 2)
(* (+ 1 2) (- 2 3))
(sq 1 2)
(setq a 100)
(defun sq (n) (* n n))
(let ((a 6)) a)
(if t 5 6)
(cons 5 6)
(cons (cons 6 7))
```

Concepts of LISP

- LISP has a data structure model
  - Lists
  - Atoms

- Even programs are written as lists.
- Even *LISP* is written as a list.
- No other data structures

Evaluation

- Prefix notation of function calls as lists
  - Operation is first element
  - Second and following elements are arguments
  - `(<operation> <arg1> ... <argn>)`

- Examples:

```
(+ 4 2)
(+ 3 (- 3 2))
(sq (* 4 2))
```

Numerical Functions

- Numerical operations:
  - Addition: `(+ 1 2)`
  - Subtraction: `(- 1 2)`
  - Multiplication: `(* 1 2)`
  - Division: `(/ 1 2)`

- Square root: `(sqrt x)`
- Base Exponent: `(expt x y)`
- Trigonometric Functions: `(sin x)`
- Absolute Value: `(abs x)`
- Modulo: `(mod x y)`
- Rounding: `(round x)`

Interaction

- Interaction with lisp is done in a *read-eval-print loop*
- Loop consists of the following steps:
  - Parse input and construct LISP object
  - Evaluate LISP object to produce output
  - Print output object
- Example:

```
>> (+ 1 2)
3
```

Variables

- Variables can be defined by:


```
(setq <var> <value>)
```

- Semantics

```
<var> := <value>
```


- Occurrence of variable symbol replaces variable symbol by the value of the variable
- Example:


```
>> (setq a (+ 5 3))
8
>> a
8
```

Quote

- If lists should not be evaluated, use function quote

```
>> (setq a (+ 1 2))
3
>> (setq a (quote (+ 1 2)))
(+ 1 2)
```


- There is a short-hand form, using a single quotation mark

```
>> (setq a '(+ 1 2))
(+ 1 2)
```

Condition Function

- Definition: `(if <cond> <true-value> <false-value>)`
- Boolean values in LISP are given by two symbols
  - Symbol `nil` (equal to the empty list) represents false
  - `T` represents true


```
>> (if nil 1 2)
2
>> (if (= 10 10) 1 2)
1
>> (if () 1 2)
2
```

Predicates

- Type checking predicates
  - `(atom x)` checks whether x is not a list
  - `(integerp x)` checks whether x is an integer
  - `(numberp x)` checks whether x is a number
  - `(stringp x)` checks whether x is a string
- Numerical predicates
  - `(oddp x)` checks whether x is integer and odd
  - `(evenp x)` checks whether x is integer and even
- Equality
  - `(equal x y)` checks structural equality
  - `(eql x y)` checks atom equality
  - `(eq x y)` checks identity
  - `(= x y)` checks numerical equality
- Logical operators
  - `(or x y)` logical OR
  - `(and x y)` logical AND

Functions

- Function declaration:

```
(defun <name> (<arg1> ... <argn>) body)
```

- Translates to:

```
(setq <name> '(lambda (<arg1> ... <argn>) body))
```

```
>> (defun factorial (x)
     (if (= x 0)
         1
         (* x (factorial (- x 1)))))
FACTORIAL
>> (factorial 4)
24
```

- Next week we'll do this in lambda calculus directly - without the impurity of defining variables

Bindings

- Definition:

```
(let ((<name1> <value1>) ... (<namen> <valuen>))
body)
```



```
>> (let
      ((a 3) (b 4) (c 5))
      (+ (* a b) c))
17
>> a
Error: variable A is unbound
```

Bindings (2)

- Let allows local bindings of variables
- Bindings might be nested – innermost variable is taken


```
>> (let
    ((a 3))
    (let
      ((a 5))
      a
    ))
5
```

List Construction

- Construction with cons: (cons <element> <list>)
- Cons returns a new list with <element> as first element, followed by elements in <list>
- Construction with list: (list <elem1> ... <elemn>)


```
>> (cons 1 nil)
(1)
>> (cons 'a '(b c))
(a b c)
>> (list 1 2 3)
(1 2 3)
```

List Access

- Access first element: `(first <list>)`
- Access all but first element: `(rest <list>)`


```
>> (first '(a b c))
a
>> (rest '(a b c))
(b c)
```

λ in LISP


```
>> ((lambda (x) (+ x 1)) 4)
5
>> ((lambda (x y z) (* (+ x x) z)) 1 3 5)
10
```

λ in Haskell


```
>> (\x -> x + 1) 4
5
>> (\x y z -> (x + x) * z) 1 3 5
10
```

λ in Python


```
>> (lambda x: x + 1) 4
5
>> (lambda x: lambda y: lambda z: (x + x) * z)(1)
10
>>> f = lambda x: lambda y: lambda z: (x + x) * z
>>> f
<function <lambda> at 0x02F66270>
>>> f(1)
<function <lambda>.<locals>.<lambda> at 0x02F6615
>>> f(1)(3)
<function <lambda>.<locals>.<lambda>.<locals>.<lar
>>> f(1)(3)(5)
10
```

λ in Python


```
>>> NOT = lambda f: lambda x: lambda y: f(y)(x)
>>> TRUE = lambda x: lambda y: x
>>> FALSE = lambda x: lambda y: y
>>> IF = lambda f: lambda x: lambda y: f(x)(y)
>>> IF (NOT(TRUE)) ("a") ("b")
'a'
>>> IF (NOT(NOT(TRUE))) ("a") ("b")
'b'
```

Review

- Lambda Calculus revision
  - Application, Abstraction
  - Rewriting
  - α and β reductions
- Currying
  - Multiple arguments
  - Associativity
- Encodings
  - Boolean logic
  - Church numerals, arithmetic
- Functional programming
  - Introduction to LISP
  - Brief look λ in other languages