COMP2022: Formal Languages and Logic

2018, Semester 2, Week 2

THE UNIVERSITY OF SYDNEY

Outline

- Revision - Lambda Calculus
-
- Encodings

- Functional Programming: LISP

Operations

- Application

Operations

- Application

- Abstraction

  - $\lambda x.M$
  - Variable x is abstracted in expressio

Rewriting

- $M[x := N]$

- d with

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

Rewriting

- $M[x := N]$

- d with

- e.g.

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

Rewriting

- $M[x := N]$

- d with

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

Rewriting

- $M[x := N]$

- d with

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

α-reduction

- Rename a λ to remove a name conflict

-

- y must be a *new* variable
  - You must not choose a symbol that is alrea

β-reduction

- Solve an abstraction

-

- Note: the free occurrences of x i
  occurrences which bound to the λx. $(\lambda x.M)$

Outline

- Revision - Lambda Calculus
-
- 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.
-

Two arguments

- Suppose we have a function $f(x, y)$ which requires two arguments.
-
- F is a function which takes one input, and ret
  F_x, which will take the *next* inpu

Two arguments

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

-

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

- The output of the second function will be $f(x, y)$.

Example

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

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

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

Example

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

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

$$(\lambda x.(\lambda y.(x + y)/2)) \quad 5 \quad 7$$
$$= (\lambda y.(5 + y)/2) \; 7$$

Example

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

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

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

Example

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

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

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

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*

- function, which returns an $(n-2$

Currying

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

- function, which returns an $(n-2$

- 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:

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

The function is *partially evaluated* at ea

Evaluation

Recall the example from earlier:

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

The function is *partially evaluated* at ea

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

Evaluation

Recall the example from earlier:

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

The function is *partially evaluated* at ea

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

Evaluation

Recall the example from earlier:

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

The function is *partially evaluated* at ea

- The first function returns $(\lambda y.(5 +$
- 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:
-

Notation

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

$$ABCDEF \equiv (((((A$$

Notation

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

-
- :

$$ABCDEF \equiv (((((A$$

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

Notation

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

Notation

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

Notation

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

$1 \quad 2 \quad 3 \quad k$

Notation

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

1 2 3 k

- This means the leftmost λ will mat
  applied to the function

Notation

- Abstraction is right associative
- Application is left associative
-

Notation

- Abstraction is right associative
- Application is left associative
-

$$= \lambda yz.((z-4) \quad y) \quad 2\ 3$$

Notation

- Abstraction is right associative
- Application is left associative
-

$$= \lambda yz.((z-4) \quad y) \quad 2\ 3$$

$$= (\lambda z.((z-4)$$

Notation

- Abstraction is right associative
- Application is left associative
-

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

Notation

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

$$\lambda xyz.((z \quad x) \quad y) \;\; 4\;2\;3$$

Notation

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

$$\lambda xyz.((z \quad x) \quad y) \;\; 4\;2\;3$$

Notation

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

$$\lambda xyz.((z \quad x) \quad y) \quad 4 \; 2 \; 3$$

$$= \quad \lambda x.\ \lambda y.(\lambda z.((z - x) \quad y)) \quad 4 \quad 2 \quad 3$$

Notation

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

$$\lambda xyz.((z \quad x) \quad y) \;\; 4 \; 2 \; 3$$

$$= \quad \lambda x. \; \lambda y.(\lambda z.((z - x) \quad y)) \quad 4 \quad 2 \quad 3$$

$$= \Big((\lambda y.(\lambda z.((z - 4) \times y))$$

$$= \big(\lambda z.((z - 4) \times 2)\big) \cdot 3$$

$$= (3 - 4) \times 2$$

$$= -2$$

Notation

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

Notation

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

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

Notation

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

- 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:

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

-

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

Currying

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

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

-

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

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

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

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

OUTLINE

- Revision - Lambda Calculus

- **Encodings**

- Functional Programming: LISP

But...

Untyped Lambda Calculus

- Lambda calculus does not have primitives
  - No numbers
  - No arithmetic operators

Untyped Lambda Calculus

- Lambda calculus does not have primitives
  - No numbers
  - No arithmetic operators
- However, I'm claiming that it is computati
  a Turing Machine!

Untyped Lambda Calculus

- Lambda calculus does not have primitives
  - No numbers
  - No arithmetic operators

- However, I'm claiming that it is computati a Turing Machine!

- So, how can we represent data types?

Untyped Lambda Calculus

- Lambda calculus does not have primitives
  - No numbers
  - No arithmetic operators

- However, I'm claiming that it is computati
  a Turing Machine!

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

Encodings: truth

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

Encodings: truth

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

- Now we can do conditional logic:
  - IFELSE := $\lambda fxy.fxy$ has semanti
    - `if <cond> then <x> else <y>`
    - If `<cond>` is true, return result of `<x>` `<y>`

Encodings: truth

$$IFELSE\ TRUE\ A\ B$$

Encodings: truth

$$IFELSE\ TRUE\ A\ B$$

$$= (\lambda fxy.fxy)\ (\lambda xy.x)\ A\ B \qquad \text{(macro substitution)}$$

Encodings: truth

$$IFELSE\ TRUE\ A\ B$$

$$= (\lambda fxy.fxy)\ (\lambda xy.x)\ A\ B$$ (macro substitution)

duction)

Encodings: truth

$$IFELSE\ TRUE\ A\ B$$

$$= (\lambda fxy.fxy)\ (\lambda xy.x)\ A\ B \qquad \text{(macro substitution)}$$

duction)

duction)

Encodings: truth

$$IFELSE\ TRUE\ A\ B$$

$$= (\lambda fxy.fxy)\ (\lambda xy.x)\ A\ B \qquad \text{(macro substitution)}$$

duction)

duction)

$$= (\lambda ab.(\lambda xy.x)ab)\ A\ B \qquad \beta \text{ duction)}$$

Encodings: truth

$IFELSE\ TRUE\ A\ B$

$= (\lambda fxy.fxy)\ (\lambda xy.x)\ A\ B$ (macro substitution)

duction)

duction)

$= (\lambda ab.(\lambda xy.x)ab)\ A\ B$ β duction)

$= (\lambda b.(\lambda xy.x)Ab)\ B$ ion)

Encodings: truth

$IFELSE\ TRUE\ A\ B$

$= (\lambda fxy.fxy)\ (\lambda xy.x)\ A\ B$ (macro substitution)

duction)

duction)

$= (\lambda ab.(\lambda xy.x)ab)\ A\ B$ β duction)

$= (\lambda b.(\lambda xy.x)Ab)\ B$ ion)

$= (\lambda xy.x)\ A\ B)$ ion)

Encodings: truth

$$IFELSE\ TRUE\ A\ B$$

$$= (\lambda fxy.fxy)\ (\lambda xy.x)\ A\ B \qquad \text{(macro substitution)}$$

duction)

duction)

$$= (\lambda ab.(\lambda xy.x)ab)\ A\ B \qquad \beta \quad \text{duction)}$$

$$= (\lambda b.(\lambda xy.x)Ab)\ B \qquad \text{ion)}$$

$$= (\lambda xy.x)\ A\ B) \qquad \text{ion)}$$

$$= (\lambda y.A)B) \qquad \text{ion)}$$

Encodings: truth

$$IFELSE\ TRUE\ A\ B$$

$$= (\lambda fxy.fxy)\ (\lambda xy.x)\ A\ B \qquad \text{(macro substitution)}$$

duction)

duction)

$$= (\lambda ab.(\lambda xy.x)ab)\ A\ B \qquad \beta\ \text{duction)}$$

$$= (\lambda b.(\lambda xy.x)Ab)\ B \qquad \text{ion)}$$

$$= (\lambda xy.x)AB) \qquad \text{ion)}$$

$$= (\lambda y.A)B) \qquad \text{ion)}$$

$$= A \qquad \text{(}\beta\text{-reduction)}$$

Encodings: truth

$IFELSE\ FALSE\ A\ B$

Encodings: truth

$$\mathit{IFELSE}\ \mathit{FALSE}\ A\ B$$

$$= (\lambda fxy.fxy)\ (\lambda xy.y)\ A\ B \qquad \text{(macro substitution)}$$

Encodings: truth

$$IFELSE\ FALSE\ A\ B$$

$$= (\lambda fxy.fxy)\ (\lambda xy.y)\ A\ B$$ (macro substitution)

duction)

Encodings: truth

$$IFELSE\ FALSE\ A\ B$$

$$= (\lambda fxy.fxy)\ (\lambda xy.y)\ A\ B \qquad \text{(macro substitution)}$$

duction)

duction)

Encodings: truth

$IFELSE\ FALSE\ A\ B$

$= (\lambda fxy.fxy)\ (\lambda xy.y)\ A\ B$ (macro substitution)

duction)

duction)

$= (\lambda ab.(\lambda xy.y)ab)\ A\ B$ β duction)

Encodings: truth

$$IFELSE\ FALSE\ A\ B$$

$$= (\lambda fxy.fxy)\ (\lambda xy.y)\ A\ B \qquad \text{(macro substitution)}$$

duction)

duction)

$$= (\lambda ab.(\lambda xy.y)ab)\ A\ B \qquad \beta \ \text{duction)}$$

$$= (\lambda b.(\lambda xy.y)Ab)\ B \qquad \text{ion)}$$

Encodings: truth

$IFELSE\ FALSE\ A\ B$

$= (\lambda fxy.fxy)\ (\lambda xy.y)\ A\ B$ (macro substitution)

duction)

duction)

$= (\lambda ab.(\lambda xy.y)ab)\ A\ B$ β duction)

$= (\lambda b.(\lambda xy.y)Ab)\ B$ ion)

$= (\lambda xy.y)\ A\ B)$ ion)

Encodings: truth

$$IFELSE\ FALSE\ A\ B$$

$$= (\lambda fxy.fxy)\ (\lambda xy.y)\ A\ B \qquad \text{(macro substitution)}$$

duction)

duction)

$$= (\lambda ab.(\lambda xy.y)ab)\ A\ B \qquad \beta\ \text{duction)}$$

$$= (\lambda b.(\lambda xy.y)Ab)\ B \qquad \text{ion)}$$

$$= (\lambda xy.y)AB) \qquad \text{ion)}$$

$$= (\lambda y.y)B) \qquad \text{ion)}$$

Encodings: truth

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

Encodings: truth

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

-

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

Encodings: NOT

- NOT := $\lambda fxy.fyx$

-
  - x, y would be those arguments

Encodings: NOT

- NOT := $\lambda fxy.fyx$

-
  - x, y would be those arguments

- i.e. NOT outputs f, except its argume around!

Encodings: truth

NOT TRUE

Encodings: truth

NOT TRUE

tution)

Encodings: truth

NOT TRUE

tution)

duction)

Encodings: truth

NOT TRUE

tution)

duction)

duction)

Encodings: truth

$NOT\ TRUE$

tution)

duction)

duction)

$= \lambda xy.(\lambda ab.a)yx$ ion)

Encodings: truth

$NOT\ TRUE$

tution)

duction)

duction)

$= \lambda xy.(\lambda ab.a)yx$ ion)

$= \lambda xy.(\lambda b.y)x$ ion)

Encodings: truth

$NOT\ TRUE$

tution)

duction)

duction)

$= \lambda xy.(\lambda ab.a)yx$ ion)

$= \lambda xy.(\lambda b.y)x$ ion)

$= \lambda xy.y$ ion)

Encodings: truth

$NOT\ TRUE$

tution)

duction)

duction)

$= \lambda xy.(\lambda ab.a)yx$ ion)

$= \lambda xy.(\lambda b.y)x$ ion)

$= \lambda xy.y$ ion)

$= FALSE$ (macro substitution)

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.
-

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.

-

  - Inductive clause: for any element
    $x + 1$ is an element of the natural numbe

Encodings: numbers

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

-
    - Inductive clause: for any element x … $x + 1$ is an element of the natural numbe

    - 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:

Church numerals

- Natural numbers in lambda calculus have two constructors:

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

Church numerals

- Natural numbers in lambda calculus have two constructors:

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

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

Church numerals

ONE

Church numerals

ONE

(macro)

Church numerals

ONE

(macro)

(α)

Church numerals

ONE

(macro)

(α)

$= \lambda yz.y((\lambda ab.b)$ (β)

Church numerals

$$ONE$$

$$\text{(macro)}$$

$$(\alpha)$$

$$= \lambda yz.y((\lambda ab.b) \qquad (\beta)$$

$$= \lambda yz.y((\lambda b.b)z \qquad (\beta)$$

Church numerals

ONE

(macro)

(α)

$$= \lambda yz.y((\lambda ab.b) \qquad (\beta)$$

$$= \lambda yz.y((\lambda b.b)z \qquad (\beta)$$

$$= \lambda yz.yz \qquad (\beta)$$

Church numerals

TWO

(macro)

Church numerals

TWO

(macro)

(α)

Church numerals

$$TWO$$

(macro)

(α)

$$= \lambda yz.y((\lambda ab.ab$$ (β)

Church numerals

TWO

(macro)

(α)

$= \lambda yz.y((\lambda ab.ab$ (β)

$= \lambda yz.y((\lambda b.yb)$ (β)

Church numerals

$$TWO$$

(macro)

(α)

$$= \lambda yz.y((\lambda ab.ab \qquad (\beta)$$

$$= \lambda yz.y((\lambda b.yb) \qquad (\beta)$$

$$= \lambda yz.y(yz) \qquad (\beta)$$

Church numerals

$$
\begin{aligned}
&\mathit{THREE} \\
&= \mathit{SUCCESSOR}\ \mathit{TWO}
\end{aligned}
$$

Church numerals

$$
\begin{aligned}
THREE \\
&= SUCCESSOR\ TWO
\end{aligned}
$$

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

Arithmetic?

- We have numbers. Do they work?

ARITHMETIC?

- We have numbers. Do they work?

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

Addition example

$$ADD\ TWO\ THREE$$

Addition example

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

Addition example

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

(α)

Addition example

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

Addition example

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

Addition example

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

Addition example

$$\textit{ADD TWO THREE}$$

$$= \lambda xypq.xp(ypq) \;\; (\lambda yz.y(yz)) \; (\lambda yz.y(y(yz)))$$

$$(\alpha)$$

$$(\beta)$$

$$= \lambda ypq.(\lambda b.p(pb))(ypq) \;\; (\lambda cd.c(c(cd))) \qquad (\beta)$$

$$= (\lambda ypq.p(p(ypq))) \; (\lambda cd.c(\qquad (\beta)$$

$$= \lambda pq.p(p((\lambda cd.c(c(cd)))p \qquad (\beta)$$

Addition example

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

Addition example

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

Addition example

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

Multiplication example

Multiplication example

Multiplication example

$= \ldots$

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

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

Multiplication example

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

Multiplication example

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

Multiplication example

$$MULT\ TWO\ THREE$$

$$= (\lambda xyz.x(yz))\ TWO\ THREE$$

$$= \lambda z.\ \lambda x.(THREE\ z)\big((T$$

Multiplication example

$$MULT\ TWO\ THREE$$

$$= (\lambda xyz.x(yz))\ TWO\ THREE$$

$$= \lambda z.\ \lambda x.(THREE\ z)\big((T$$

$$= \lambda zx.(THREE\ z)\big((THRE$$

Multiplication example

$$MULT\ TWO\ THREE$$

$$= (\lambda xyz.x(yz))\ TWO\ THREE$$

$$= \lambda z.\ \lambda x.(THREE\ z)\big((T$$

$$= \lambda zx.(THREE\ z)\big((THRE$$

$$= \lambda zx.\big((\lambda fx.f(f(fx)))\ z\big)\big(((\!$$

Multiplication example

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

$$= \lambda z.\ \lambda x.(\mathit{THREE}\ z)\big((T$$

$$
\begin{aligned}
&= \lambda zx.(\mathit{THREE}\ z)\big((\mathit{THRE} \\
&= \lambda zx.\big((\lambda fx.f(f(fx)))\ z\big)\big(((\\
&= \lambda zx.\big(\lambda x.z(z(zx))\big)\big((\lambda x.z(z(zx)))x\big)
\end{aligned}
$$

Multiplication example

$$MULT\ TWO\ THREE$$
$$= (\lambda xyz.x(yz))\ TWO\ THREE$$

$$= \lambda z.\ \lambda x.(THREE\ z)\big((T$$
$$= \lambda zx.(THREE\ z)\big((THRE$$

$$= \lambda zx.\big((\lambda fx.f(f(fx)))\ z\big)\big(((($$
$$= \lambda zx.\big(\lambda x.z(z(zx))\big)\big((\lambda x.z(z(zx)))x\big)$$
$$= \lambda zx.\big(\lambda x.z(z(zx))\big)\big(z(z(zx))\big)$$

Multiplication example

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

$$
\begin{aligned}
&= \lambda z.\ \lambda x.(\mathit{THREE}\ z)\big((T \\
&= \lambda zx.(\mathit{THREE}\ z)\big((\mathit{THRE} \\
&= \lambda zx.\big((\lambda fx.f(f(fx)))\ z\big)\big(((\\
&= \lambda zx.\big(\lambda x.z(z(zx))\big)\big((\lambda x.z(z(zx)))x\big) \\
&= \lambda zx.\big(\lambda x.z(z(zx))\big)\big(z(z(zx))\big) \\
&= \lambda zx.z(z(z(z(z(zx)))))
\end{aligned}
$$

Multiplication example

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

$$
\begin{aligned}
&= \lambda z.\ \lambda x.(\mathit{THREE}\ z)\big((T
\end{aligned}
$$

$$
\begin{aligned}
&= \lambda zx.(\mathit{THREE}\ z)\big((\mathit{THRE} \\
&= \lambda zx.\big((\lambda fx.f(f(fx)))\ z\big)\big(((\\
&= \lambda zx.\big(\lambda x.z(z(zx))\big)\big((\lambda x.z(z(zx)))x\big) \\
&= \lambda zx.\big(\lambda x.z(z(zx))\big)\big(z(z(zx))\big) \\
&= \lambda zx.z(z(z(z(z(zx))))) \\
&= \mathit{SIX}
\end{aligned}
$$

Recursion

In imperative languages, we can easily write recursive code:

d

... 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 d

Recursion

In the last tutorial you tried to reduce:

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

... and d

This is

Combinator:

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

Recursion

In the last tutorial you tried to reduce:

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

... and d

This is

Combinator:

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

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

OUTLINE

- Revision - Lambda Calculus
-
- Encodings

- **Functional Programming: LISP**

LISP

- LISP is the second oldest programming language in common
-
-

- Is a functional programming language
- Is a practical implementation of the Lambd
- Has many dialects (e.g. Clojure, Common L Scheme, etc.)

LISP = LISt Processing

- LISP has atoms
  - Numbers, e.g. 10
  - Identifiers, e.g. Foo
-
  - 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) (
(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

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

Evaluation

- Prefix notation of function calls as lists

-

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

Numerical Functions

- Numerical operations:
  - Addition: `(+ 1 2)`
  - Subtraction: `(- 1 2)`

- 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*
-

- Example:

  $>>$ $(+\ 1\ 2)$
  3

Variables

- Variables can be defined by:

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


-
-

  value of the variable
- Example:


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

Quote

- If lists should not be evaluated, use function quote

- There is a short-hand form, using a single quo


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

Condition Function

- Definition: `(if <cond> <true-value> <false-value>)`
- Boolean values in LISP are given by two symbols


```
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
-
- Equality
  - `(equal x y)` checks structural eq
  - `(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:

```
(* x (factorial (

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>))
```



```
    (+ (* 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


```
               ((a  5))
                 a
            ))
    5
```

List Construction

- Construction with cons: `(cons <element> <list>)`
- Cons returns a new list with `<element>` as first element,
-

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

List Access

- Access first element: `(first <list>)`
-

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

λ in LISP


```
>
5
>
10
```

λ in Haskell

$>$
5

$>$
10

λ in Python


```
>> (lambda x: x + 1) 4
5
>> (lambda x: lambda y: lambda z: (x + x) * z)(1)
1
>
>
<f
>>> f(1)
<function <lambda>.<locals>.<lambda> at 0x02
>>> 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)
>>
>>
>>
>>
'a'
>>> IF (NOT(NOT(TRUE)))("a")("b")
'b'
```

Review

- Lambda Calculus revision
  - Application, Abstraction
  - Rewriting
-

- Encodings
  - Boolean logic
  - Church numerals, arithmetic

- Functional programming
  - Introduction to LISP
  - Brief look λ in other languages