CMSC 330: Organization of Programming Languages

Lambda Calculus
Turing Machine

Infinite Tape

1 0 0 0 1 1 1 1 0

Read/Write Head

Control Unit
State: Y

START

2

3

4

HALT

e; e, R

b; b, R

a; a, R

b; b, R

a; a, R

b; b, R

a; a, R

b; b, R

a; a, R

b; b, R

a; a, R
Turing Completeness

- Turing machines are the most powerful description of computation possible
  - They define the Turing-computable functions
- A programming language is Turing complete if
  - It can map every Turing machine to a program
  - A program can be written to emulate a Turing machine
  - It is a superset of a known Turing-complete language
- Most powerful programming language possible
  - Since Turing machine is most powerful automaton
Programming Language Expressiveness

- So what language features are needed to express all computable functions?
  - What’s a minimal language that is Turing Complete?
- Observe: some features exist just for convenience
  - Multi-argument functions: `foo (a, b, c)`
    - Use currying or tuples
  - Loops: `while (a < b) …`
    - Use recursion
  - Side effects: `a := 1`
    - Use functional programming pass “heap” as an argument to each function, return it when with function’s result:
      `effectful : 'a → 's → ('s * 'a)`
Programming Language Expressiveness

- It is not difficult to achieve Turing Completeness
  - Lots of things are ‘accidentally’ TC
- Some fun examples:
  - x86_64 `mov` instruction
  - Minecraft
  - Magic: The Gathering
  - Java Generics
- There’s a whole cottage industry of proving things to be TC.
- What about something a little more ‘programmable’?
You only have:
• If statement
• Plus 1
• Minus 1
• functions
Lambda Calculus (λ-calculus)

- Proposed in 1930s by
  - Alonzo Church
    (born in Washington DC!)

- Formal system
  - Designed to investigate functions & recursion
  - For exploration of foundations of mathematics

- Now used as
  - Tool for investigating computability
  - Basis of functional programming languages
    - Lisp, Scheme, ML, OCaml, Haskell…
Why Study Lambda Calculus?

- It is a “core” language
  - Very small but still Turing complete
- But with it can explore general ideas
  - Language features, semantics, proof systems, algorithms, …
- Plus, higher-order, anonymous functions (aka lambdas) are now very popular!
  - C++ (C++11), PHP (PHP 5.3.0), C# (C# v2.0), Delphi (since 2009), Objective C, Java 8, Swift, Python, Ruby (Procs), … (and functional languages like OCaml, Haskell, F#, …)
  - Excel, as of 2021!
Lambda Calculus Syntax

- A lambda calculus expression is defined as

  \[ e ::= x \text{ variable} \]
  \[ \mid \lambda x. e \text{ abstraction (fun def)} \]
  \[ \mid e \ e \text{ application (fun call)} \]

- This grammar describes ASTs; not for parsing (ambiguous!)
- Lambda expressions also known as lambda terms

- \( \lambda x. e \) is like \((\text{fun} \ x \ \rightarrow \ e)\) in OCaml

That’s it! Nothing but higher-order functions
Lambda Calculus Syntax

- How is it ambiguous?
- Let’s try: \( \lambda x.x \ x \)

\[
\begin{align*}
E & \rightarrow V \mid L \mid A \\
L & \rightarrow \lambda V.E \\
A & \rightarrow E \ E \\
V & \rightarrow v \mid \epsilon
\end{align*}
\]
Lambda Calculus Syntax

- How is it ambiguous?
- Let’s try: \( \lambda x.x \ x \)

\[
\begin{align*}
E & \rightarrow V \mid L \mid A \\
L & \rightarrow \lambda V.E \\
A & \rightarrow E \ E \\
V & \rightarrow v \mid \epsilon
\end{align*}
\]
Lambda Calculus Syntax

- While this means that our grammar is not so useful for parsing, it is still useful for write LC terms if we follow some conventions.
- Almost all literature you will find uses 2 syntactic conventions.
- We add a third convention that is very common ‘syntactic sugar’ for ease of reading larger LC terms.
Three Conventions

- Scope of $\lambda$ extends as far right as possible
  - Subject to scope delimited by parentheses
  - $\lambda x. \lambda y. x \ y$ is same as $\lambda x. (\lambda y. (x \ y))$

- Function application is left-associative
  - $x \ y \ z$ is $(x \ y) \ z$
  - Same rule as OCaml

- As a convenience, we use the following “syntactic sugar” for local declarations
  - $\text{let } x = e1 \text{ in } e2$ is short for $(\lambda x. e2) \ e1$
Quiz #1

\[ \lambda x. (y \ z) \text{ and } \lambda x. y \ z \text{ are equivalent} \]

A. True
B. False
Quiz #1

\( \lambda x. (y \ z) \) and \( \lambda x. y \ z \) are equivalent

A. True
B. False
But what does it mean

- Many ways to define the semantics of LC
- We will look at 2
  - Operational
  - Definitional Interpreter
Lambda Calculus Semantics

- Evaluation: All that’s involved are function calls 
  \((\lambda x. e_1) \ e_2\)
  - Evaluate \(e_1\) with \(x\) replaced by \(e_2\)

- This application is called \textit{beta-reduction}
  - We allow reductions to occur \textit{anywhere} in a term
    - Order reductions are applied does not affect final value! (if there is one)

- When a term \textit{cannot be reduced further} it is in \textit{beta normal form}
Operational Semantics of LC

- Because of the use of variables, we need an environment

- Recap: the environment can be thought of as a map from variable names to the term they represent.
  - Often written as $\rho : \text{Env}$
  - type $\text{Env} = \text{Variable} \rightarrow \text{Term}$

- We extend the environment by adding new associations between variables and terms
  - $\text{ext} : \text{Env} \rightarrow \text{Variable} \rightarrow \text{Term} \rightarrow \text{Env}$
Operational Semantics of LC

- Each ‘kind’ of term gets its own inference rule
- When we reach a ‘bare’ lambda, we’re done:

\[
\text{val} = \rho \, v \\
A; (\lambda x. e_1) \Rightarrow (\lambda x. e_1)
\]
Operational Semantics of LC

- The meaning of variables is based on the current environment:

\[
A(v) = t \\
A; v \Rightarrow t
\]
Operational Semantics of LC

- We didn’t say anything about the order things should happen in!
- Let’s evaluate the argument fully first, this is known as *call-by-value*

\[
\begin{align*}
A; e_2 & \Rightarrow e_3 \\
A; e_1 e_2 & \Rightarrow A; e_1 e_3 \\
\rho' &= \text{ext } p x (\lambda v. e_2) \\
A; (\lambda x. e_1)(\lambda v. e_2) & \Rightarrow A, x: (\lambda v. e_2); e_1
\end{align*}
\]
Operational Semantics of LC

Let’s avoid evaluating the argument, this is known as *call-by-name*

\[
\begin{align*}
A; e_1 & \Rightarrow e_2 \\
A; e_1 e_3 & \Rightarrow e_2 e_3
\end{align*}
\]

\[
\rho' = \text{ext } p \times e_2 \\
A; (\lambda x. e_1) e_2 & \Rightarrow A,x:e_2; e_1
\]
Operational Semantics of LC

- The rules we just showed are very common for programming languages based on LC
- You don’t **have** to choose call-by-name or call-by-value, LC as a system let’s you choose whatever order you want
- You can also reduce **under** the lambda.

\[
\begin{array}{c}
A; e_1 \Rightarrow e_2 \\
\hline
A; (\lambda x. e_1) \Rightarrow A; (\lambda x. e_2)
\end{array}
\]
Operational Semantics of LC

- Call-by-value vs. call-by-name has its tradeoffs.
- Most languages use call-by-value (e.g. Ocaml), but some use call-by-name (or a related variant known as call-by-need).
- Interestingly: more programs terminated under call-by-name. Can you think of why?
- Consider: \((\lambda x. e_2) \; e_1,\)
- What if \(e_1\) would never terminate, but \(e_2\) would?
OCaml Lambda Calc AST

define the grammar for expressions:

\[ e ::= x \mid \lambda x.e \mid e e \]

define the types:

\[ \text{type id} = \text{string} \]
\[ \text{type exp} = \text{Var of id} \mid \text{Lam of id} \ast \text{exp} \mid \text{App of exp} \ast \text{exp} \]

examples:

\[ y \rightarrow \text{Var “y”} \]
\[ \lambda x.x \rightarrow \text{Lam (“x”, Var “x”) \]
\[ \lambda x.\lambda y.x\ y \rightarrow \text{Lam (“x”, Lam (“y”, App (Var “x”, Var “y”)))} \]
\[ (\lambda x.\lambda y.x\ y)\ \lambda x.x\ x \rightarrow \text{App (Lam (“x”, Lam (“y”, App (Var “x”, Var “y”))), Lam (“x”, App (Var “x”, Var “x”)))} \]
Quiz #2

What is this term's AST?

\( \lambda x . x \ x \)

A. App (Lam ("x", Var "x"), Var "x")
B. Lam (Var "x", Var "x", Var "x")
C. Lam ("x", App (Var "x", Var "x"))
D. App (Lam ("x", App ("x", "x")))
Quiz #2

What is this term’s AST?

$$\lambda x.x \ x \ x$$

A. App (Lam ("x", Var "x"), Var "x")
B. Lam (Var "x", Var "x", Var "x")
C. Lam ("x", App (Var "x", Var "x"))
D. App (Lam ("x", App ("x", "x")))
Quiz #3

This term is equivalent to which of the following?

$$\lambda x. x\ a\ b$$

A. $$(\lambda x. x)\ (a\ b)$$
B. $$(((\lambda x. x)\ a)\ b)$$
C. $$\lambda x. (x\ (a\ b))$$
D. $$((\lambda x. ((x\ a)\ b))$$
Quiz #3

This term is equivalent to which of the following?

\[ \lambda x . x \ a \ b \]

A. \((\lambda x . x) \ (a \ b)\)
B. \(((\lambda x . x) \ a) \ b)\)
C. \(\lambda x . (x \ (a \ b))\)
D. \((\lambda x . ((x \ a) \ b))\)
Lambda Calculus on paper

- When doing things ‘by hand’ we often omit the explicit environment and think in terms of *substitutions*
- You must be careful when doing this by hand as it can get finnicky!
- Some examples will help with intuition...
Beta Reduction Examples

- $(\lambda x.x)\ z \rightarrow z$

- $(\lambda x.y)\ z \rightarrow y$

- $(\lambda x.x\ y)\ z \rightarrow z\ y$
  - A function that applies its argument to $y$
Beta Reduction Examples (cont.)

- \((\lambda x. x \, y) \, (\lambda z. z) \) → \((\lambda z. z) \, y \) → \(y\)

- \((\lambda x. \lambda y. x \, y) \, z \) → \(\lambda y. z \, y\)
  - A curried function of two arguments
  - Applies its first argument to its second

- \((\lambda x. \lambda y. x \, y) \, (\lambda z. zz) \, x \) → \((\lambda y. (\lambda z. zz) \, y) \, x \) → \((\lambda z. zz) \, x \) → \(x \) \(x\)
Beta Reduction Examples (cont.)

\((\lambda x.x (\lambda y.y)) \ (u \ r) \to\)

\((\lambda x.(\lambda w. x w)) \ (y \ z) \to\)
Beta Reduction Examples (cont.)

\[(\lambda x. x (\lambda y. y)) \ (u \ r) \rightarrow (u \ r) \ (\lambda y. y)\]

\[(\lambda x. (\lambda w. x \ w)) \ (y \ z) \rightarrow (\lambda w. (y \ z) \ w)\]
Quiz #4

\((\lambda x. y) \ z\) can be beta-reduced to

A. y
B. y \ z
C. z
D. cannot be reduced
Quiz #4

$(\lambda x. \ y) \ z$ can be beta-reduced to

A. $y$
B. $y \ z$
C. $z$
D. cannot be reduced
Quiz #5

Which of the following reduces to $\lambda z. z$?

a) $(\lambda y. \lambda z. x) z$
b) $(\lambda z. \lambda x. z) y$
c) $(\lambda y. y) (\lambda x. \lambda z. z) w$
d) $(\lambda y. \lambda x. z) z (\lambda z. z)$
Quiz #5

Which of the following reduces to $\lambda z. z$?

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

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

c) $(\lambda y. y) (\lambda x. \lambda z. z) w$

d) $(\lambda y. \lambda x. z) z (\lambda z. z)$
Lambda calculus uses static scoping.

Consider the following:

- \((\lambda x. x (\lambda x. x)) z \rightarrow ?\)
  - The rightmost “\(x\)” refers to the second binding
  - This is a function that
    - Takes its argument and applies it to the identity function

This function is “the same” as \((\lambda x. x (\lambda y. y))\)

- Renaming bound variables consistently preserves meaning
  - This is called alpha-renaming or alpha conversion
- Example: \(\lambda x. x = \lambda y. y = \lambda z. z\) \(\lambda y. \lambda x. y = \lambda z. \lambda x. z\)
Quiz #6

Which of the following expressions is alpha equivalent to (alpha-converts from)

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

a) \(\lambda y. y \ y\)
b) \(\lambda z. y \ z\)
c) \((\lambda x. \lambda z. x \ z) \ y\)
d) \((\lambda x. \lambda y. x \ y) \ z\)
Quiz #6

Which of the following expressions is \textit{alpha equivalent} to \((\alpha\text{-converts from})\)

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

a) \(\lambda y. y \ y\)

b) \(\lambda z. y \ z\)

c) \((\lambda x. \lambda z. x \ z) \ y\)

d) \((\lambda x. \lambda y. x \ y) \ z\)
Variable capture

- How about the following?
  - \((\lambda x.\lambda y.x \; y) \; y\) → ?
  - When we replace \(y\) inside, we don’t want it to be captured by the inner binding of \(y\), as this violates static scoping
  - I.e., \((\lambda x.\lambda y.x \; y) \; y \neq \lambda y.y \; y\)

- Solution
  - \((\lambda x.\lambda y.x \; y)\) is “the same” as \((\lambda x.\lambda z.x \; z)\)
    - Due to alpha conversion
  - So alpha-convert \((\lambda x.\lambda y.x \; y)\) \(y\) to \((\lambda x.\lambda z.x \; z)\) \(y\) first
    - Now \((\lambda x.\lambda z.x \; z)\) \(y\) → \(\lambda z.y \; z\)
OCaml interpreter for Call-by-value

- Now we can write our interpreter!
- First some types and utility functions:

```ocaml
type id = string

type exp =
    | Var of id
    | Lam of id * exp
    | App of exp * exp

type env = id -> exp

let emptyEnv = fun x -> failwith "Variable not in scope"

let extend (rho : env) (name : id) (term : exp) =
    fun x -> if x = name
        then term
        else rho x
```
OCaml interpreter for Call-by-value

- Now for the eval
- Return the evaluated term and the new environment:

```ocaml
let rec eval (e : exp) (rho : env) =
  match e with
  | Var i -> (rho i, rho)
  | Lam(x, e1) -> (Lam(x, e1), rho)
  | App(e1, e2) -> let arg = fst (eval e2 rho) in
    let f   = freshen e1 in
    (match (fst (eval f rho)) with
     | Lam(v, body) -> let rho2 = extend rho v arg in
       eval body rho2
     | _ -> failwith "Can't apply a non-function")
```
OCaml interpreter for Call-by-value

- We didn’t show implementation of `freshen`, which ensures that we avoid variable capture
- Fun exercise: implement `freshen`
- I used the “Barendregt Convention”: gives everything ‘fresh’ names.
  - If every name is unique, no chance of variable capture
  - Simple, but not great for performance
Quiz #7

Beta-reducing the following term produces what result?

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

A. \(y \ (\lambda z. z \ y)\)
B. \(z \ (\lambda y. y \ z)\)
C. \(y \ (\lambda y. y \ y)\)
D. \(y \ y\)
Quiz #7

Beta-reducing the following term produces what result?

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

A. \(y \ (\lambda z.z \ y)\)
B. \(z \ (\lambda y.y \ z)\)
C. \(y \ (\lambda y.y \ y)\)
D. \(y \ y\)
Quiz #8

Beta reducing the following term produces what result?

\[ \lambda x. (\lambda y. y y) \, w \, z \]

a) \( \lambda x. w \, w \, z \)
b) \( \lambda x. w \, z \)
c) \( w \, z \)
d) Does not reduce
Quiz #8

Beta reducing the following term produces what result?

\[ \lambda x. (\lambda y. y \; y) \; w \; z \]

a) \( \lambda x. \; w \; w \; z \)

b) \( \lambda x. \; w \; z \)

c) \( w \; z \)

d) Does not reduce
Let bindings

- Local variable declarations are like defining a function and applying it immediately (once):
  - \( \text{let } x = e1 \text{ in } e2 = (\lambda x. e2) \, e1 \)

- Example
  - \( \text{let } x = (\lambda y. y) \text{ in } x \, x = (\lambda x. x \, x) \, (\lambda y. y) \)

where
\[
(\lambda x. x \, x) \, (\lambda y. y) \rightarrow (\lambda x. x \, x) \, (\lambda y. y) \rightarrow (\lambda y. y) \, (\lambda y. y) \rightarrow (\lambda y. y)
\]
Booleans

- Church’s encoding of mathematical logic
  - true = λx.λy.x
  - false = λx.λy.y
  - if $a$ then $b$ else $c$
    - Defined to be the expression: $a \ b \ c$

- Examples
  - if true then $b$ else $c = (λx.λy.x) \ b \ c → (λy.b) \ c → b$
  - if false then $b$ else $c = (λx.λy.y) \ b \ c → (λy.y) \ c → c$
Booleans (cont.)

- Other Boolean operations
  - \( \text{not} = \lambda x. x \text{ false true} \)
    - \( \text{not } x = x \text{ false true } = \text{ if } x \text{ then } \text{false else } \text{true} \)
    - \( \text{not true } \rightarrow (\lambda x. x \text{ false true }) \text{ true } \rightarrow (\text{true false true }) \rightarrow \text{false} \)
  - \( \text{and} = \lambda x. \lambda y. x \text{ y false} \)
    - \( \text{and } x \text{ y } = \text{if } x \text{ then } y \text{ else } \text{false} \)
  - \( \text{or} = \lambda x. \lambda y. x \text{ true y} \)
    - \( \text{or } x \text{ y } = \text{if } x \text{ then } \text{true else } y \)

- Given these operations
  - Can build up a logical inference system
Quiz #9

What is the lambda calculus encoding of \texttt{xor} \(x\ y\)?

- \(\texttt{xor \ true \ true} = \texttt{false}\)
- \(\texttt{xor \ false \ false} = \texttt{false}\)
- \(\texttt{xor \ true \ false} = \texttt{true}\)
- \(\texttt{xor \ false \ true} = \texttt{true}\)

- \(\texttt{x \ x \ y}\)
- \(\texttt{x \ (y \ true \ false) \ y}\)
- \(\texttt{x \ (y \ false \ true) \ y}\)
- \(\texttt{y \ x \ y}\)

\[
\text{true} = \lambda x.\lambda y.x \\
\text{false} = \lambda x.\lambda y.y \\
\text{if \ a \ then \ b \ else \ c} = a \ b \ c \\
\text{not} = \lambda x.\ x \ \text{false} \ \text{true}
\]
Quiz #9

What is the lambda calculus encoding of \( \text{xor } x \ y \)?

- \( \text{xor } \text{true } \text{true } = \)  \( \text{xor } \text{false } \text{false } = \)  \( \text{false} \)
- \( \text{xor } \text{true } \text{false } = \)  \( \text{xor } \text{false } \text{true } = \)  \( \text{true} \)

- \( x \ x \ y \)
- \( x \ (y \ \text{true } \text{false} ) \ y \)
- \( x \ (y \ \text{false } \text{true} ) \ y \)
- \( y \ x \ y \)

\( \text{true } = \lambda x.\lambda y.x \)
\( \text{false } = \lambda x.\lambda y.y \)
\( \text{if } a \ \text{then } b \ \text{else } c = a \ b \ c \)
\( \text{not } = \lambda x.\lambda x.\text{false } \text{true} \)
Pairs

- Encoding of a pair \(a, b\)
  - \((a,b) = \lambda x.\text{if } x \text{ then } a \text{ else } b\)
  - \(\text{fst} = \lambda f.f \text{ true}\)
  - \(\text{snd} = \lambda f.f \text{ false}\)

- Examples
  - \(\text{fst}(a,b) = (\lambda f.f \text{ true}) (\lambda x.\text{if } x \text{ then } a \text{ else } b) \rightarrow\)
    \((\lambda x.\text{if } x \text{ then } a \text{ else } b) \text{ true} \rightarrow\)
    \(\text{if true then } a \text{ else } b \rightarrow a\)
  - \(\text{snd}(a,b) = (\lambda f.f \text{ false}) (\lambda x.\text{if } x \text{ then } a \text{ else } b) \rightarrow\)
    \((\lambda x.\text{if } x \text{ then } a \text{ else } b) \text{ false} \rightarrow\)
    \(\text{if false then } a \text{ else } b \rightarrow b\)
Natural Numbers (Church* Numerals)

- Encoding of non-negative integers
  - \( 0 = \lambda f. \lambda y. y \)
  - \( 1 = \lambda f. \lambda y. f \, y \)
  - \( 2 = \lambda f. \lambda y. f \, (f \, y) \)
  - \( 3 = \lambda f. \lambda y. f \, (f \, (f \, y)) \)
  i.e., \( n = \lambda f. \lambda y. \text{<apply } f \text{ n times to } y \text{> } \)
  - Formally: \( n+1 = \lambda f. \lambda y. f \, (n \, f \, y) \)

*(Alonzo Church, of course)*
Quiz #10

What OCaml type could you give to a Church-encoded numeral?

- (‘a -> ‘b) -> ‘a -> ‘b
- (‘a -> ‘a) -> ‘a -> ‘a
- (‘a -> ‘a) -> ‘b -> int
- (int -> int) -> int -> int
What OCaml type could you give to a Church-encoded numeral?

- (‘a -> ‘b) -> ‘a -> ‘b
- (‘a -> ‘a) -> ‘a -> ‘a
- (‘a -> ‘a) -> ‘b -> int
- (int -> int) -> int -> int
Operations On Church Numerals

- **Successor**
  - \( \text{succ} = \lambda z.\lambda f.\lambda y.f (z f y) \)
  - \( 0 = \lambda f.\lambda y.y \)
  - \( 1 = \lambda f.\lambda y.f y \)

- **Example**
  - \( \text{succ} \ 0 = \)
    - \( (\lambda z.\lambda f.\lambda y.f (z f y)) (\lambda f.\lambda y.y) \rightarrow \)
    - \( \lambda f.\lambda y.f ((\lambda f.\lambda y.y) f y) \rightarrow \)
    - \( \lambda f.\lambda y.f ((\lambda y.y) y) \rightarrow \)
    - \( \lambda f.\lambda y.f y \)
    - \( = 1 \)

Since \( (\lambda x.y) z \rightarrow y \)
Operations On Church Numerals (cont.)

- **IsZero?**

  - \( \text{iszero} = \lambda z. z \ (\lambda y. \text{false}) \ \text{true} \)
  
  This is equivalent to \( \lambda z. ((z \ (\lambda y. \text{false})) \ \text{true}) \)

- **Example**

  - \( \text{iszero} \ 0 = \)

    \[
    (\lambda z. z \ (\lambda y. \text{false}) \ \text{true}) \ (\lambda f. \lambda y. y) \rightarrow \\
    (\lambda f. \lambda y. y) \ (\lambda y. \text{false}) \ \text{true} \rightarrow \\
    (\lambda y. y) \ \text{true} \rightarrow \\ 
    \text{Since} \ (\lambda x. y) \ z \rightarrow y \\ 
    \text{true}
    \]

  - \( 0 = \lambda f. \lambda y. y \)
If M and N are numbers (as λ expressions)
  • Can also encode various arithmetic operations

**Addition**
  • M + N = \( \lambda f. \lambda y. M f (N f y) \)

  Equivalently: \(+ = \lambda M. \lambda N. \lambda f. \lambda y. M f (N f y)\)
  ➢ In prefix notation (+ M N)

**Multiplication**
  • M * N = \( \lambda f. M (N f) \)

  Equivalently: \(* = \lambda M. \lambda N. \lambda f. \lambda y. M (N f) y\)
  ➢ In prefix notation (* M N)
Arithmetic (cont.)

- Prove $1+1 = 2$
  - $1+1 = \lambda x.\lambda y. (1 \ x) \ (1 \ x \ y) =$
  - $\lambda x.\lambda y. ((\lambda f.\lambda y. f \ y) \ x) \ (1 \ x \ y) \rightarrow$
  - $\lambda x.\lambda y. (\lambda y. x \ y) \ (1 \ x \ y) \rightarrow$
  - $\lambda x.\lambda y. x \ (1 \ x \ y) \rightarrow$
  - $\lambda x.\lambda y. x \ ((\lambda f.\lambda y. f \ y) \ x \ y) \rightarrow$
  - $\lambda x.\lambda y. x \ ((\lambda y. x \ y) \ y) \rightarrow$
  - $\lambda x.\lambda y. x \ (x \ y) = 2$

- With these definitions
  - Can build a theory of arithmetic

- $1 = \lambda f.\lambda y. f \ y$
- $2 = \lambda f.\lambda y. f \ (f \ y)$
Arithmetic Using Church Numerals

- What about subtraction?
  - Easy once you have ‘predecessor’, but...
  - Predecessor is very difficult!

- Story time:
  - One of Church’s students, Kleene (of Kleene-star fame) was struggling to think of how to encode ‘predecessor’, until it came to him during a trip to the dentists office.
  - Take from this what you will

- Wikipedia has a great derivation of ‘predecessor’, not enough time today.
Looping+Recursion

- So far we have avoided self-reference, so how does recursion work?
- We can construct a lambda term that ‘replicates’ itself:
  - Define \( D = \lambda x. x x \), then
    - \( D D = (\lambda x. x x) (\lambda x. x x) \rightarrow (\lambda x. x x) (\lambda x. x x) = D D \)
    - \( D D \) is an infinite loop
- We want to generalize this, so that we can make use of looping
The Fixpoint Combinator

\( Y = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)) \)

- Then

\[
Y \ F = \\
(\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) \ F \\
(\lambda x. F (x x)) (\lambda x. F (x x)) \\
F ((\lambda x. F (x x)) (\lambda x. F (x x))) \\
= F (Y \ F)
\]

- \( Y \ F \) is a *fixed point* (aka *fixpoint*) of \( F \)

- Thus \( Y \ F = F (Y \ F) = F (F (Y \ F)) = \ldots \)
  - We can use \( Y \) to achieve recursion for \( F \)
Example

\[ \text{fact } = \lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * (f (n-1)) \]

- The second argument to fact is the integer
- The first argument is the function to call in the body
  - We’ll use Y to make this recursively call fact

\[
(Y \text{ fact}) \ 1 \ = \ (\text{fact} \ (Y \ \text{fact})) \ 1
\]

\[
\rightarrow \ \text{if } 1 = 0 \ \text{then } 1 \ \text{else } 1 * ((Y \ \text{fact}) \ 0)
\]

\[
\rightarrow 1 * ((Y \ \text{fact}) \ 0)
\]

\[
= 1 * (\text{fact} \ (Y \ \text{fact}) \ 0)
\]

\[
\rightarrow 1 * (\text{if } 0 = 0 \ \text{then } 1 \ \text{else } 0 * ((Y \ \text{fact}) \ (-1)))
\]

\[
\rightarrow 1 * 1 \rightarrow 1
\]
Factorial 4 = ?

\[(Y \ G) \ 4\]
\[G \ (Y \ G) \ 4\]
\[(\lambda n.(\text{if } n = 0 \ \text{then } 1 \ \text{else } n \times ((Y \ G) \ (n-1)))) \ (Y \ G) \ 4\]
\[(\lambda n.(\text{if } n = 0 \ \text{then } 1 \ \text{else } n \times ((Y \ G) \ (n-1)))) \ 4\]
\[\text{if } 4 = 0 \ \text{then } 1 \ \text{else } 4 \times ((Y \ G) \ (4-1))\]
\[4 \times (G \ (Y \ G) \ (4-1))\]
\[4 \times ((\lambda n.(1, \ \text{if } n = 0; \ \text{else } n \times ((Y \ G) \ (n-1)))) \ (4-1))\]
\[4 \times (1, \ \text{if } 3 = 0; \ \text{else } 3 \times ((Y \ G) \ (3-1))\]
\[4 \times (3 \times (G \ (Y \ G) \ (3-1)))\]
\[4 \times (3 \times ((\lambda n.(1, \ \text{if } n = 0; \ \text{else } n \times ((Y \ G) \ (n-1)))) \ (3-1)))\]
\[4 \times (3 \times (1, \ \text{if } 2 = 0; \ \text{else } 2 \times ((Y \ G) \ (2-1))))\]
\[4 \times (3 \times (2 \times (G \ (Y \ G) \ (2-1))))\]
\[4 \times (3 \times (2 \times ((\lambda n.(1, \ \text{if } n = 0; \ \text{else } n \times ((Y \ G) \ (n-1)))) \ (2-1))))\]
\[4 \times (3 \times (2 \times (1, \ \text{if } 1 = 0; \ \text{else } 1 \times ((Y \ G) \ (1-1))))\]
\[4 \times (3 \times (2 \times (1 \times (G \ (Y \ G) \ (1-1)))))\]
\[4 \times (3 \times (2 \times (1 \times ((\lambda n.(1, \ \text{if } n = 0; \ \text{else } n \times ((Y \ G) \ (n-1)))) \ (1-1))))\]
\[4 \times (3 \times (2 \times (1 \times (1, \ \text{if } 0 = 0; \ \text{else } 0 \times ((Y \ G) \ (0-1))))))\]
\[4 \times (3 \times (2 \times (1 \times (1))))\]
24
Most programming languages choose call-by-value:

- \((\lambda z. z) ((\lambda y. y) \, x) \rightarrow (\lambda z. z) \, x \rightarrow x\)

Call-by-name is less popular (but does exist)

- \((\lambda z. z) ((\lambda y. y) \, x) \rightarrow (\lambda y. y) \, x \rightarrow x\)

These evaluation strategies are about the relation between functions and their arguments.

What evaluating under the lambda?

- Do any programming languages do that?
Partial Evaluation

- It is also possible to evaluate within a function (without calling it):
  - $$(\lambda y. (\lambda z.z) y \ x) \rightarrow (\lambda y. y \ x)$$

- Called **partial evaluation**
  - Can combine with CBN or CBV
  - In practical languages, this evaluation strategy is employed in a limited way, as compiler optimization

```
int foo(int x) {
    return 0+x;
}

int foo(int x) {
    return x;
}
```
Discussion

- Lambda calculus is Turing-complete
  - Most powerful language possible
  - Can represent pretty much anything in “real” language
    - Using clever encodings

- But programs would be
  - Pretty slow (10000 + 1 → thousands of function calls)
  - Pretty large (10000 + 1 → hundreds of lines of code)
  - Pretty hard to understand (recognize 10000 vs. 9999)

- In practice
  - We use richer, more expressive languages
  - That include built-in primitives
The Need For Types

- Consider the **untyped** lambda calculus
  - false = \( \lambda x.\lambda y.y \)
  - 0 = \( \lambda x.\lambda y.y \)
- Since everything is encoded as a function...
  - We can easily misuse terms...
    - false 0 \( \rightarrow \) \( \lambda y.y \)
    - if 0 then ...
  ...because everything evaluates to some function
- The same thing happens in assembly language
  - Everything is a machine word (a bunch of bits)
  - All operations take machine words to machine words
Simply-Typed Lambda Calculus (STLC)

- $e ::= n | x | \lambda x : t . e | e \ e$
  - Added integers $n$ as primitives
    - Need at least two distinct types (integer & function)…
    - …to have type errors
  - Functions now include the type $t$ of their argument

- $t ::= \text{int} | t \rightarrow t$
  - int is the type of integers
  - $t1 \rightarrow t2$ is the type of a function
    - That takes arguments of type $t1$ and returns result of type $t2$
Types are limiting

- STLC will reject some terms as ill-typed, even if they will not produce a run-time error
  - Cannot type check Y in STLC
    - Or in OCaml, for that matter, at least not as written earlier.
- Surprising theorem: All (well typed) simply-typed lambda calculus terms are strongly normalizing
  - A normal form is one that cannot be reduced further
    - A value is a kind of normal form
  - Strong normalization means STLC terms always terminate
    - Proof is not by straightforward induction: Applications “increase” term size
Summary

- Lambda calculus is a core model of computation
  - We can encode familiar language constructs using only functions
    - These encodings are enlightening – make you a better (functional) programmer

- Useful for understanding how languages work
  - Ideas of types, evaluation order, termination, proof systems, etc. can be developed in lambda calculus,
    - then scaled to full languages