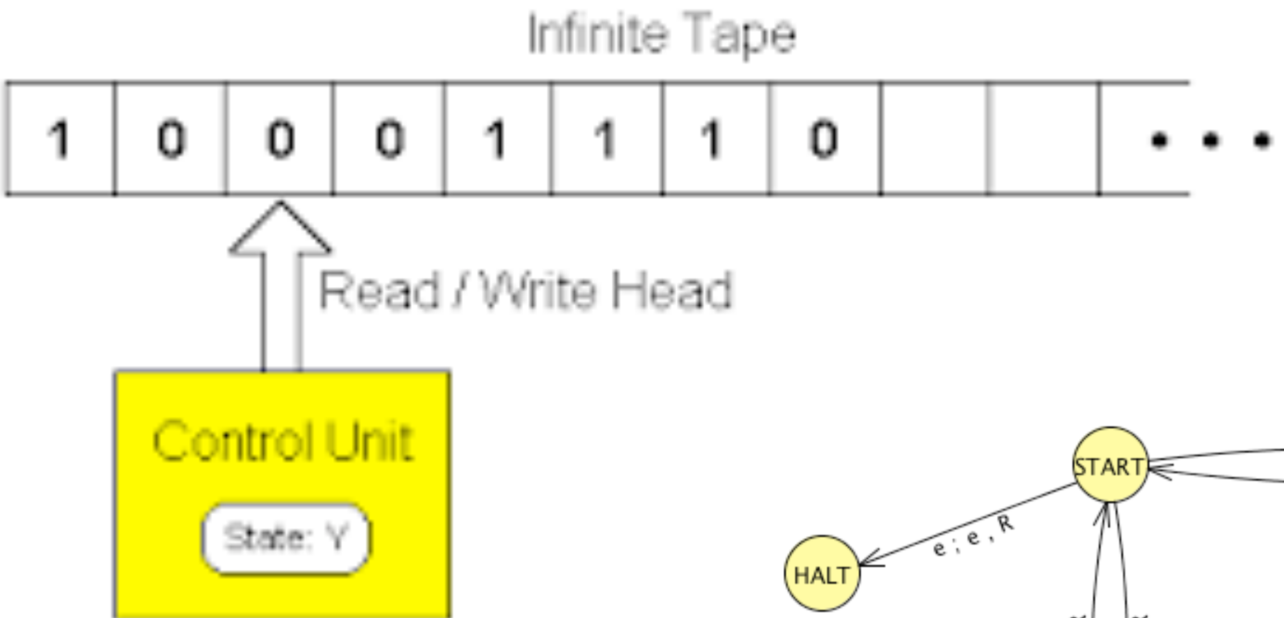
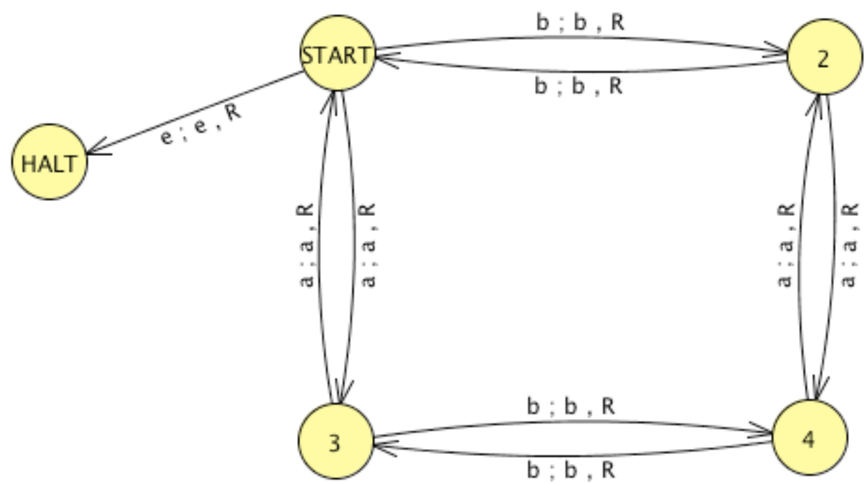
CMSC 330: Organization of Programming Languages

Lambda Calculus

Turing Machine





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
- But: What is a “core” language that is TC?

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	**variable**
| λx.e	**abstraction** (fun def)
| e e	**application** (fun call)

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

- λx.e is like `(fun x -> e)` in OCaml

That's it! Nothing but higher-order functions

Three Conventions

- Scope of λ extends as far right as possible
  - Subject to scope delimited by parentheses
  - λx. λy.x y is same as λx.(λy.(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
  - let x = e1 in e2 is short for (λx.e2) e1

Quiz #1

`λx.(y z)` and **`λx.y z`** are equivalent

A. True
B. False

Quiz #1

`λx.(y z)` and **`λx.y z`** are equivalent

A. True

B. False

Quiz #2

This term is equivalent to which of the following?

`λx.x a b`

A. **`(λx.x) (a b)`**
B. **`(((λx.x) a) b)`**
C. **`λx.(x (a b))`**
D. **`(λx.((x a) b))`**

Quiz #2

This term is equivalent to which of the following?

`λx.x a b`

A. **`(λx.x) (a b)`**
B. **`(((λx.x) a) b)`**
C. **`λx.(x (a b))`**
D. **`(λx.((x a) b))`**

But what does it mean?

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

Lambda Calculus Semantics

- Evaluation: All that's involved are function calls (λx.e1) e2
  - Evaluate e1 with x replaced by e2
- This application is called beta-reduction
  - (λx.e1) e2 → e1[x:=e2]
    - e1[x:=e2] is e1 with occurrences of x replaced by e2
    - This operation is called *substitution*
      - **Replace** formals with actuals
      - Instead of using environment to map formals to actuals
  - We allow reductions to occur *anywhere* in a term
    - Order reductions are applied does not affect final value!
- When a term cannot be reduced further it is in beta normal form

Beta Reduction Example

- (λx.λz.x z) y
  - → (λx.(λz.(x z))) y   // since λ extends to right
  - → (λx.(λz.(x z))) y   // apply (λx.e1) e2 → e1[x:=e2]
    // where e1 = λz.(x z), e2 = y
  - → λz.(y z)   // final result

Parameters
- Formal
- Actual

- Equivalent OCaml code
  - (fun x -> (fun z -> (x z))) y → fun z -> (y z)

Big-Step Operational Semantics

- Beta reduction says how to evaluate a single call
  - It doesn't say how to evaluate a term with many function calls in it
- We can use operational semantics to "fully evaluate" a term in one "big step"

$$\frac{}{(\lambda x.e1) \Downarrow (\lambda x.e1)}$$

Beta reduction, here

$$\frac{e1 \Downarrow (\lambda x.e3) \qquad e2 \Downarrow e4 \qquad e3[x:=e4] \Downarrow e5}{e1\ e2 \Downarrow e5}$$

Two Varieties

- There are two common variants of big-step semantics
  - *Eager* evaluation (aka *strict*, or *call by value*)
  - *Lazy* evaluation (aka *call by name*)

Eager

- Notice that we evaluated the argument e2 before performing the beta-reduction
  - This is the first version we saw
- Hence, *eager*

$$\frac{}{(\lambda x.e1) \Downarrow (\lambda x.e1)}$$

$$\frac{e1 \Downarrow (\lambda x.e3) \qquad e2 \Downarrow e4 \qquad e3[x:=e4] \Downarrow e5}{e1\ e2 \Downarrow e5}$$

Lazy

- Alternatively, we could have performed beta reduction *without* evaluating e2; use it as is
  - Hence, *lazy*

$$\frac{}{(\lambda x.e1) \Downarrow (\lambda x.e1)}$$

$$\frac{e1 \Downarrow (\lambda x.e3) \qquad e3[x:=e2] \Downarrow e4}{e1\ e2 \Downarrow e4}$$

Small Step Semantics

- Operational semantics rules we have seen have always been ”big step”, i.e., complete evaluation
  - e ⇓ e’ says that e will *terminate* as e’
- This is a little unsatisfying
  - It doesn’t account for nontermination
  - It doesn’t identify where a program fails to progress
- Small-step semantics addresses these problems
  - e → e’ in small-step says e **takes one step** to e’
  - We say a term e1 can be *beta-reduced* to term e2 if e1 steps to e2 after one or more steps

Small-Step Rules of LC

- Here are the "small-step" (→) rules:

$$\frac{e1 \rightarrow \mathbf{e2}}{(\lambda x.e1) \rightarrow (\lambda x.\mathbf{e2})}$$

$$\frac{e2 \rightarrow \mathbf{e3}}{e1\ e2 \rightarrow e1\ \mathbf{e3}} \qquad \frac{e1 \rightarrow \mathbf{e3}}{e1\ e2 \rightarrow \mathbf{e3}\ \ e2}$$

$$\frac{}{(\lambda x.e1)\ e2 \rightarrow e1[x{:=}e2]}$$

Evaluation Strategies

- These rules are highly flexible
  - It might be that for a given program, there are several possible rules that could apply
- Typically, a programming language will choose an *evaluation strategy* which is described by using only a subset of these rules. Examples:
  - Call by Value
  - Call by Need
  - Partial Evaluation

Call by Value

- Before doing a beta reduction, we make sure the argument cannot, itself, be further evaluated
- This is known as call-by-value (CBV)
  - This is the Eager big step approach

$$\frac{e1 \rightarrow \boldsymbol{e3}}{e1\ e2 \rightarrow \boldsymbol{e3}\ e2}$$

$$\frac{e2 \rightarrow \boldsymbol{e3}}{e1\ e2 \rightarrow e1\ \boldsymbol{e3}}$$

$$\frac{e = (\lambda x.e2) \text{ or } e = y}{(\lambda x.e1)\ e \rightarrow e1[x:=e]}$$

Beta Reductions (CBV)

- (λx.x) z → z
- (λx.y) z → y
- (λx.x y) z → z y
  - A function that applies its argument to y

Beta Reductions (CBV)

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

Quiz #3

`(λx.y) z` can be beta-reduced to

A. **`y`**
B. **`y z`**
C. **`z`**
D. cannot be reduced

Quiz #3

`(λx.y) z` can be beta-reduced to

A. **`y`**
B. **`y z`**
C. **`z`**
D. cannot be reduced

Quiz #4

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 #4

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

Evaluation Order

- The CBV rules we saw permit small-stepping either the function part or the argument part
  - If both are possible, the rules allow either one

$$\frac{e1 \rightarrow \boldsymbol{e3}}{e1\ e2 \rightarrow \boldsymbol{e3}\ e2} \qquad \frac{e2 \rightarrow \boldsymbol{e3}}{e1\ e2 \rightarrow e1\ \boldsymbol{e3}}$$

- Here's how we would require left-to-right order

$$\frac{e1 \rightarrow \boldsymbol{e3}}{e1\ e2 \rightarrow \boldsymbol{e3}\ e2} \qquad \frac{e1 = y \quad \text{or} \quad e1 = \lambda x.e \qquad e2 \rightarrow \boldsymbol{e3}}{e1\ e2 \rightarrow e1\ \boldsymbol{e3}}$$

  - The second rule prohibits evaluating e2 except when e1 cannot be evaluated further

Call by Name

- Instead of the CBV strategy, we can specifically choose to perform beta-reduction *before* we evaluate the argument
- This is known as call-by-name (CBN)
  - This is the Lazy small-step approach

$$\frac{e1 \rightarrow \boldsymbol{e3}}{e1\ e2 \rightarrow \boldsymbol{e3}\ e2}$$

$$\frac{}{(\lambda x.e1)\ e2 \rightarrow e1[x:=e2]}$$

CBN Reduction

- CBV
  - (λz.z) ((λy.y) x) → (λz.z) x → x
- CBN
  - (λz.z) ((λy.y) x) → (λy.y) x → x

Beta Reductions (CBN)

(λx.x (λy.y)) (u r) →

(λx.(λw. x w)) (y z) →

Beta Reductions (CBN)

$(\lambda x.x\ (\lambda y.y))\ (u\ r) \rightarrow (u\ r)\ (\lambda y.y)$

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

Why Does This Matter?

- The rules we just showed are very common for programming languages based on LC
  - CBV is the most common (e.g. OCaml, Java)
  - CBN does come up (Haskell uses a variant known as “call-by-need”) but is much less common
- Interestingly: more programs terminated under call-by-name. Can you think of why?
  - Consider: (λx.e2) e1,
  - What if e1 would never terminate, but e2 would?

Partial Evaluation

- That rule is useful when you have a beta-reduction *under* a lambda:
  - $(\lambda y.(\lambda z.z)\ y\ x) \rightarrow (\lambda y.y\ x)$
- Called partial evaluation
  - Can combine with CBN or CBV (just add in the rule)
  - In practical languages, this evaluation strategy is employed in a limited way, as compiler optimization

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

Static Scoping & Alpha Conversion

- Lambda calculus uses static scoping
- Consider the following
  - (λx.x (λx.x)) z → ?
    - 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 (λx.x (λy.y))
  - Renaming bound variables consistently preserves meaning
    - This is called alpha-renaming or alpha conversion
  - Ex. λx.x = λy.y = λz.z   λy.λx.y = λz.λx.z

Quiz #5

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 #5

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

Getting Serious about Substitution

- We have been thinking informally about substitution, but the details matter
- So, let’s carefully formalize it, to help us see where it can get tricky!

Defining Substitution

- Use recursion on structure of terms
  - x[x:=e] = e // Replace x by e
  - y[x:=e] = y // y is different than x, so no effect
  - (e1 e2)[x:=e] = (e1[x:=e]) (e2[x:=e])
    // Substitute both parts of application
  - (λx.e')[x:=e] = λx.e'
    - In λx.e', the x is a parameter, and thus a local variable that is different from other x's. Implements static scoping.
    - So the substitution has no effect in this case, since the x being substituted for is different from the parameter x that is in e'
  - (λy.e')[x:=e] = ?
    - The parameter y does not share the same name as x, the variable being substituted for
    - Is λy.(e'[x:=e]) correct? No…

Variable Capture

- How about the following?
  - (λx.λy.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., (λx.λy.x y) y ≠ λy.y y
- Solution
  - (λx.λy.x y) is "the same" as (λx.λz.x z)
    - Due to alpha conversion
  - So alpha-convert (λx.λy.x y) y to (λx.λz.x z) y first
    - Now (λx.λz.x z) y → λz.y z

Completing the Definition of Substitution

- Recall: we need to define (λy.e')[x:=e]
  - We want to avoid capturing (free) occurrences of y in e
  - Solution: alpha-conversion!
    - Change y to a variable w that does not appear in e' or e
      (Such a w is called fresh)
    - Replace all occurrences of y in e' by w.
    - Then replace all occurrences of x in e' by e!
- Formally:

  (λy.e')[x:=e] = λw.((e' [y:=w]) [x:=e]) (w is fresh)

Beta-Reduction, Again

- Whenever we do a step of beta reduction
  - (λx.e1) e2 → e1[x:=e2]
  - We must alpha-convert variables as necessary
  - Sometimes performed implicitly (w/o showing conversion)
- Examples
  - (λx.λy.x y) y = (λx.λz.x z) y → λz.y z // y → z
  - (λx.x (λx.x)) z = (λy.y (λx.x)) z → z (λx.x) // x → y

Quiz #6

Beta-reducing the following term produces what result?

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

A. y (λz.z y)
B. z (λy.y z)
C. y (λy.y y)
D. y y

Quiz #6

Beta-reducing the following term produces what result?

(λx.x λy.y x) y

A. **y (λz.z y)**
B. z (λy.y z)
C. y (λy.y y)
D. y y

Quiz #7

Beta reducing the following term produces what result?

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

a) λx. w w z

b) λx. w z

c) w z

d) Does not reduce

Quiz #7

Beta reducing the following term produces what result?

λx.(λy. y y) w z

a) λx. w w z

b) λx. w z

c) w z

d) Does not reduce

Lambda Calc, Impl in OCaml

- e ::= x
  | λx.e
  | e e

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

y

```
Var "y"
```

λx.x

```
Lam ("x", Var "x")
```

λx.λy.x y

```
Lam ("x",(Lam("y",App (Var "x", Var "y"))))
```

(λx.λy.x y) λx.x x

```
App
  (Lam("x",Lam("y",App(Var"x",Var"y"))),
   Lam ("x", App (Var "x", Var "x")))
```

Quiz #8

What is this term's AST?

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

λx.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 #8

What is this term's AST?

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

λx.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")))`

OCaml Implementation: Substitution

```
(* substitute e for y in m--   m[y:=e]     *)
let rec subst m y e =
  match m with
      Var x ->
        if y = x then e (* substitute *)
        else m          (* don't subst *)
    | App (e1,e2) ->
        App (subst e1 y e, subst e2 y e)
    | Lam (x,e0) -> …
```

OCaml Impl: Substitution (cont'd)

```
(* substitute e for y in m--   m[y:=e]     *)
let rec subst m y e = match m with …
    | Lam (x,e0) ->
      if y = x then m
      else if not (List.mem x (fvs e)) then
        Lam (x, subst e0 y e)
      else
        let z = newvar() in (* fresh *)
        let e0' = subst e0 x (Var z) in
        Lam (z,subst e0' y e)
```

Shadowing blocks substitution

Safe: no capture possible

Might capture; need to α-convert

CBV, L-to-R Reduction with Partial Eval

```
let rec reduce e =
  match e with
      App (Lam (x,e), e2) -> subst e x e2
    | App (e1,e2) ->
      let e1' = reduce e1 in
      if e1' != e1 then App(e1',e2)
      else App (e1,reduce e2)
    | Lam (x,e) -> Lam (x, reduce e)
    | _ -> e
```

Straight β rule

Reduce lhs of app

Reduce rhs of app

Reduce function body

nothing to do

Another Way to Avoid Capture

- Another way to avoid accidental variable capture is to use the “Barendregt Convention”: gives everything ‘fresh’ names.
  - If every name is unique, no chance of variable capture
  - Simple, but not great for performance as you have to do it after every beta-reduction!

Quick Recap on LC

- Despite its simplicity (3 AST nodes and a handful of small-step rules), LC is Turing Complete
- Any function that can be evaluated on a Turing machine can be encoded into LC (and vice-versa)
  - But we'll have to come up with the encodings!
- To *prove* that it is Turing Complete we have to map every possible Turing Machine to LC
  - We won't be doing that

The Power of Lambdas

- To give a sense of how one can encode various constructs into LC we'll be looking at some concrete examples:
  - Let bindings
  - Booleans
  - Pairs
  - Natural numbers & arithmetic
  - Looping

Let bindings

- Local variable declarations are like defining a function and applying it immediately (once):
  - let x = e1 in e2 = (λx.e2) e1

- Example
  - let x = (λy.y) in x x = (λx.x x) (λy.y)

  where

  (λx.x x) (λy.y) → (λx.x x) (λy.y) → (λy.y) (λy.y) → (λy.y)

Booleans

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

- Examples
  - if true then b else c = $(\lambda x.\lambda y.x)\ b\ c \rightarrow (\lambda y.b)\ c \rightarrow b$
  - if false then b else c = $(\lambda x.\lambda y.y)\ b\ c \rightarrow (\lambda y.y)\ c \rightarrow c$

Booleans (cont.)

- Other Boolean operations
  - not = λx.x false true
    - not *x* = *x* false true = if *x* then false else true
    - not true → (λx.x false true) true → (true false true) → false
  - and = λx.λy.x y false
    - and x y = if x then y else false
  - or = λx.λy.x true y
    - or x y = if x then true else y
- Given these operations
  - Can build up a logical inference system

Quiz #9

What is the lambda calculus encoding of xor x y?

- xor true true = xor false false = false
- xor true false = xor false true = true

- x x y
- x (y true false) y
- x (y false true) y
- y x y

true = λx.λy.x
false = λx.λy.y
if a then b else c = a b c
not = λx.x false true

Quiz #9

What is the lambda calculus encoding of xor x y?

- xor true true = xor false false = false
- xor true false = xor false true = true

- x x y
- x (y true false) y
- **x (y false true) y**
- y x y

true = λx.λy.x
false = λx.λy.y
if a then b else c = a b c
not = λx.x false true

Pairs

- Encoding of a pair a, b
  - (a,b) = λx.if x then a else b
  - fst = λf.f true
  - snd = λf.f false
- Examples
  - fst (a,b) = (λf.f true) (λx.if x then a else b) →
    (λx.if x then a else b) true →
    if true then a else b → a
  - snd (a,b) = (λf.f false) (λx.if x then a else b) →
    (λx.if x then a else b) false →
    if false then a else b → b

Natural Numbers (Church* Numerals)

- Encoding of non-negative integers
  - 0 = λf.λy.y
  - 1 = λf.λy.f y
  - 2 = λf.λy.f (f y)
  - 3 = λf.λy.f (f (f y))

    i.e., n = λf.λy.<apply f n times to y>
  - Formally: n+1 = λf.λy.f (n f y)

*(Alonzo Church, of course)

Quiz #10

n = λf.λy.<*apply* f *n times to* y>

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

Quiz #10

n = λf.λy.<*apply* f *n times to* y>

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
  - succ = λz.λf.λy.f (z f y)
    - 0 = λf.λy.y
    - 1 = λf.λy.f y
- Example
  - succ 0 =

    (λz.λf.λy.f (z f y)) (λf.λy.y) →

    λf.λy.f ((λf.λy.y) f y) →

    λf.λy.f ((λy.y) y) →

    λf.λy.f y

    = 1

Since (λx.y) z → y

Operations On Church Numerals (cont.)

- IsZero?
  - iszero = $\lambda z.z\ (\lambda y.\text{false})\ \text{true}$

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

- Example
  - iszero 0 =

    $(\lambda z.z\ (\lambda y.\text{false})\ \text{true})\ (\lambda f.\lambda y.y) \to$

    $(\lambda f.\lambda y.y)\ (\lambda y.\text{false})\ \text{true} \to$

    $(\lambda y.y)\ \text{true} \to$

    true

  - 0 = $\lambda f.\lambda y.y$

Since $(\lambda x.y)\ z \to y$

Arithmetic Using Church Numerals

- 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 = λx.λy.(1 x) (1 x y) =
  - λx.λy.((λf.λy.f y) x) (1 x y) →
  - λx.λy.(λy.x y) (1 x y) →
  - λx.λy.x (1 x y) →
  - λx.λy.x ((λf.λy.f y) x y) →
  - λx.λy.x ((λy.x y) y) →
  - λx.λy.x (x y) = 2

- 1 = λf.λy.f y
- 2 = λf.λy.f (f y)

- With these definitions
  - Can build a theory of arithmetic

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 = λf.(λx.f (x x)) (λx.f (x x))

- Then

  Y F =

  (λf.(λx.f (x x)) (λx.f (x x))) F →

  (λx.F (x x)) (λx.F (x x)) →

  F ((λx.F (x x)) (λx.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)) = ...
  - We can use **Y** to achieve recursion for F

Example

fact = λf.λn.if n = 0 then 1 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 fact) 1 = (fact (Y fact)) 1

→ if 1 = 0 then 1 else 1 * ((Y fact) 0)

→ 1 * ((Y fact) 0)

= 1 * (fact (Y fact) 0)

→ 1 * (if 0 = 0 then 1 else 0 * ((Y fact) (-1))

→ 1 * 1 → 1

Factorial 4=?

```
(Y G) 4
 G (Y G) 4
(λr.λn.(if n = 0 then 1  else n × (r (n-1)))) (Y G) 4
(λn.(if n = 0 then 1 else n × ((Y G) (n-1)))) 4
if 4 = 0 then 1 else 4 × ((Y G) (4-1))
4 × (G (Y G) (4-1))
4 × ((λn.(1, if n = 0; else n × ((Y G) (n-1)))) (4-1))
4 × (1, if 3 = 0; else 3 × ((Y G) (3-1)))
4 × (3 × (G (Y G) (3-1)))
4 × (3 × ((λn.(1, if n = 0; else n × ((Y G) (n-1)))) (3-1)))
4 × (3 × (1, if 2 = 0; else 2 × ((Y G) (2-1))))
4 × (3 × (2 × (G (Y G) (2-1))))
4 × (3 × (2 × ((λn.(1, if n = 0; else n × ((Y G) (n-1)))) (2-1))))
4 × (3 × (2 × (1, if 1 = 0; else 1 × ((Y G) (1-1)))))
4 × (3 × (2 × (1 × (G (Y G) (1-1)))))
4 × (3 × (2 × (1 × ((λn.(1, if n = 0; else n × ((Y G) (n-1)))) (1-1)))))
4 × (3 × (2 × (1 × (1, if 0 = 0; else 0 × ((Y G) (0-1))))))
4 × (3 × (2 × (1 × (1))))
24
```

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 = λx.λy.y
  - 0 = λx.λy.y
- Since everything is encoded as a function...
  - We can easily misuse terms…
    - false 0 → λy.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 | λx: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 ::= int | t → t
  - int is the type of integers
  - t1 → 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