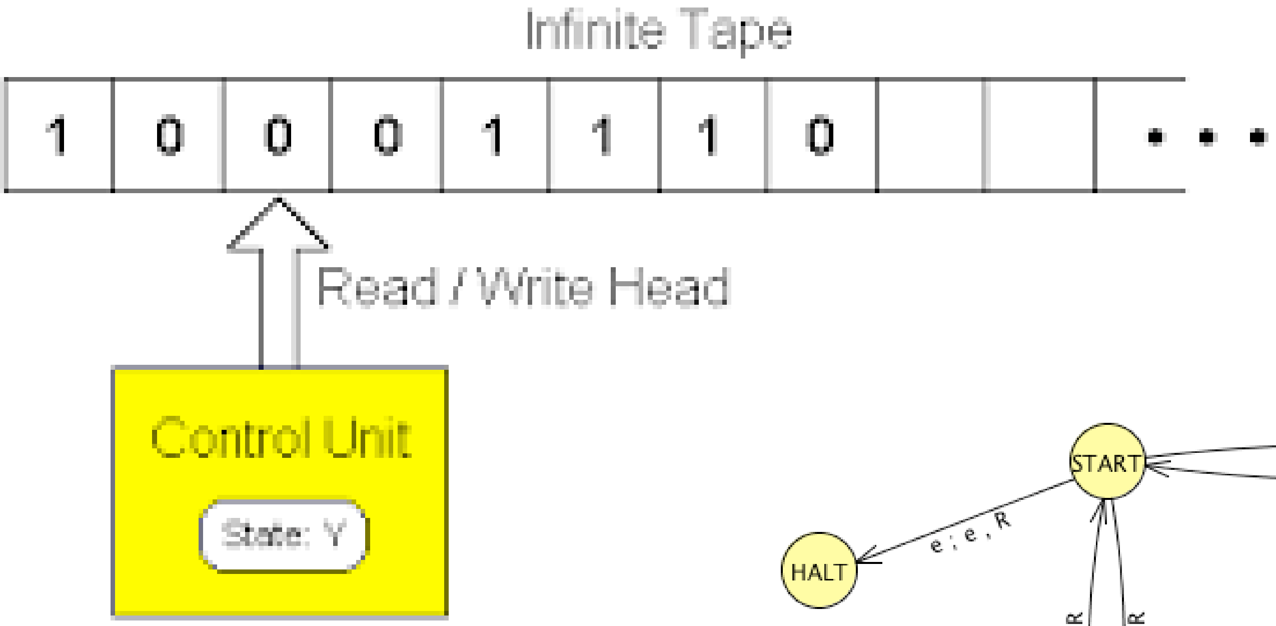
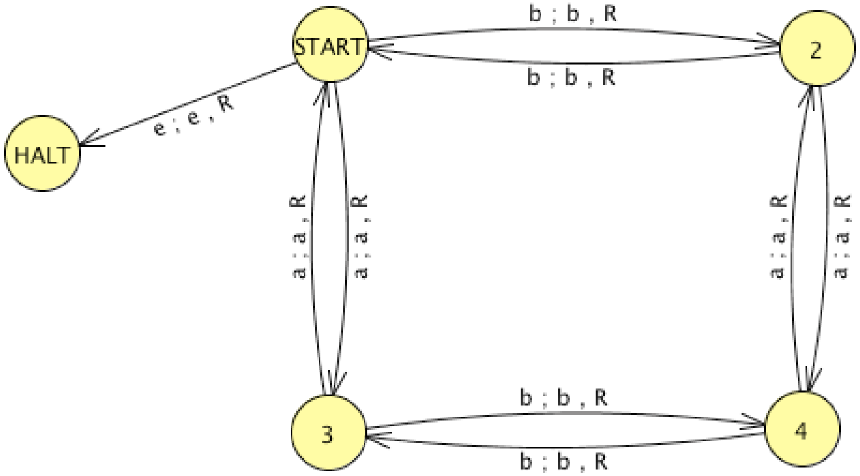
CMSC 330: Organization of Programming Languages

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.
- What about something a little more ‘programmable’?

Mini C

You only have:

- If statement
- Plus 1
- Minus 1
- functions

sum n = 1+2+3+4+5…n in Mini C

```
int add1(int n){return n+1;}
int sub1(int n){return n-1;}
int add(int a,int b){
    if(b == 0) return a;
    else return add( add1(a),sub1(b));
}
int sum(int n){
    if(n == 1) return 1;
    else return add(n, sum(sub1(n)));
}
int main(){
    printf("%d\n",sum(5));
}
```

Lambda Calculus (λ-calculus)

- Proposed in 1930s by
  - Alonzo Church
    (born in Washingon 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**

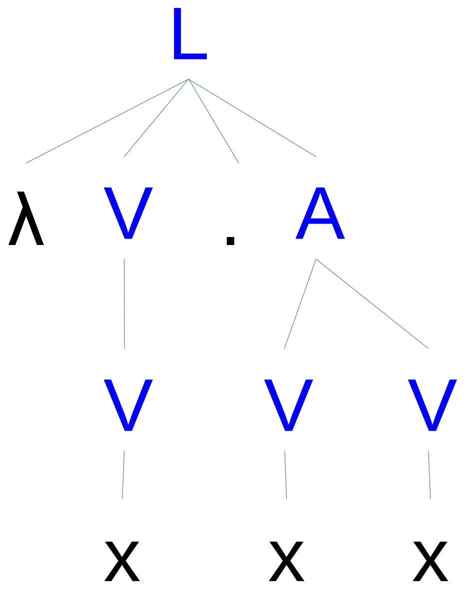
- λx.e is like `(fun x -> e)` in OCaml

That’s it! Nothing but higher-order functions

Lambda Calculus Syntax

- How is it ambiguous?
- Let’s try: λx.x x

E → V | L | A
L → λV.E
A → E E
V → v | ε



Lambda Calculus Syntax

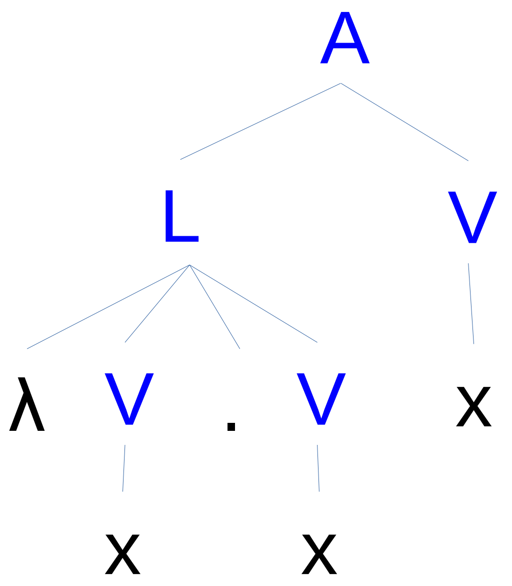
- How is it ambiguous?
- Let’s try: λx.x x

E → V | L | A

L → λV.E

A → E E

V → v | ε



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 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

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 (λx.e1) e2
  - Evaluate e1 with x replaced by e2
- This application is called beta-reduction
  - We allow reductions to occur *anywhere* in a term
    - Order reductions are applied does not affect final value! (if there is one)
- When a term cannot be reduced further it is in 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 ρ : Env
  - type Env = Variable → Term
- We *extend* the environment by adding new associations between variables and terms
  - ext : Env → Variable → Term → Env

Operational Semantics of LC

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

$$\frac{\texttt{val} = \rho\ v}{A;\ (\lambda x.e1) \Rightarrow (\lambda x.e1)}$$

Operational Semantics of LC

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

$$\frac{A(v) = t}{A;\ v \Rightarrow t}$$

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*

$$\frac{A;\ e2 \Rightarrow \boldsymbol{e3}}{A;\ e1\ e2 \Rightarrow A;\ e1\ \boldsymbol{e3}}$$

$$\frac{A;\ e1 \Rightarrow \boldsymbol{e2}}{A;\ e1\ (\lambda v.e3) \Rightarrow A;\ \boldsymbol{e2}\ \ (\lambda v.e3)}$$

$$\frac{\rho' = \text{ext p x}\ (\lambda v.e2)}{A;\ (\lambda x.e1)\ (\lambda v.e2) \Rightarrow A,x{:}(\lambda v.e2);\ e1}$$

Operational Semantics of LC

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

$$\frac{A;\ e1 \Rightarrow \boldsymbol{e2}}{A;\ e1\ e3 \Rightarrow \boldsymbol{e2}\ \ e3}$$

$$\frac{\rho' = \text{ ext } p\ x\ e2}{A;\ (\lambda x.e1)\ e2 \Rightarrow A,x{:}e2;\ e1}$$

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.

$$\frac{A;\ e1 \Rightarrow \boldsymbol{e2}}{A;\ (\lambda x.e1) \Rightarrow A;\ (\lambda x.\boldsymbol{e2})}$$

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: (λx.e2) e1,
- What if e1 would never terminate, but e2 would?

OCaml Lambda Calc AST

- 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 #2

What is this term's AST?

```
type id = string
type exp =
      Var of id
    | Lam of id * exp
    | App of exp * exp
type env = id -> 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 #2

What is this term's AST?

λx.x x

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

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?

`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 #3

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))`

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

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

Beta Reduction Examples (cont.)

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

Beta Reduction Examples (cont.)

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

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

Beta Reduction Examples (cont.)

(λx.x (λy.y)) (u r) → (u r) (λy.y)

(λx.(λw. x w)) (y z) → (λw. (y z) w)

Quiz #4

`(λx.y) z` can be beta-reduced to

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

Quiz #4

`(λx.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 λz. z?

a) (λy. λz. x) z
b) (λz. λx. z) y
c) (λy. y) (λx. λz. z) w
d) (λy. λx. z) z (λz. z)

Quiz #5

Which of the following reduces to λz. z?

a) (λy. λz. x) z
b) (λz. λx. z) y
c) (λy. y) (λx. λz. z) w
d) (λy. λx. z) z (λz. z)

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 #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 alpha equivalent to (alpha-converts from)

(λx. λy. x y) y

a) λy. y y

b) λz. y z

c) (λx. λz. x z) y

d) (λx. λy. x y) z

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

OCaml interpreter for Call-by-value

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

```
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:

```
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?

(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 #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) λx. w w z

b) λx. w z

c) w z

d) Does not reduce

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 = 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
  - 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 = λz.z (λy.false) true

    This is equivalent to λz.((z (λy.false)) true)

- Example
  - iszero 0 =

    (λz.z (λy.false) true) (λf.λy.y) →

    (λf.λy.y) (λy.false) true →

    (λy.y) true →

    true

  - 0 = λf.λy.y

Since (λx.y) z → y

Arithmetic Using Church Numerals

- If M and N are numbers (as λ expressions)
  - Can also encode various arithmetic operations
- Addition
  - M + N = λf.λy.M f (N f y)

    Equivalently: + = λM.λN.λf.λy.M f (N f y)
    - In prefix notation (+ M N)
- Multiplication
  - M * N = λf.M (N f)

    Equivalently: * = λM.λN.λf.λy.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 = λx.x x, then
    - D D = (λx.x x) (λx.x x) → (λx.x x) (λx.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
```

Call-by-name vs. Call-by-value redux

- Most programming languages choose call-by-value:
  - (λz.z) ((λy.y) x) → (λz.z) x → x
- Call-by-name is less popular (but does exist)
  - (λz.z) ((λy.y) x) → (λy.y) x → 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):
  - (λy.(λz.z) y x) → (λy.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 = 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