

CMSC 330: Organization of Programming Languages

Lambda Calculus

Programming Language Features

- Many features exist simply 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
- So what language features are really needed?

CMSC 330

2

Turing Completeness

- Computational system that can
 - Simulate a Turing machine
 - Compute every Turing-computable function
- 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

CMSC 330

3

Programming Language Theory

- Come up with a “core” language
 - That’s as small as possible
 - But still Turing complete
- Helps illustrate important
 - Language features
 - Algorithms
- One solution
 - Lambda calculus

CMSC 330

4

Lambda Calculus (λ -calculus)

- Proposed in 1930s by
 - Alonzo Church
 - Stephen Cole Kleene
- 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...



CMSC 330

5

Lambda Expressions

- A lambda calculus **expression** is defined as

$e ::= x$	variable
$ \lambda x.e$	function
$ e e$	function application

- $\lambda x.e$ is like `(fun x -> e)` in OCaml
- That’s it! Nothing but higher-order functions

CMSC 330

6

Three Conveniences

- Syntactic sugar for local declarations
 - `let x = e1 in e2` is short for $(\lambda x. e2) e1$
- Scope of λ 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

CMSC 330

7

Lambda Calculus Semantics

- All we've got are functions
 - So all we can do is call them
- To evaluate $(\lambda x. e1) e2$
 - Evaluate `e1` with `x` bound to `e2`
- This application is called **beta-reduction**
 - $(\lambda x. e1) e2 \rightarrow e1[x/e2]$
 - $e1[x/e2]$ is `e1` where occurrences of `x` are replaced by `e2`
 - Slightly different than the environments we saw for OCaml
 - Do substitutions to replace formals with actuals
 - Instead of using environment to map formals to actuals
 - We allow reductions to occur anywhere in a term

CMSC 330

8

Beta Reduction Example

- $(\lambda x. \lambda z. x z) y$
 - $\rightarrow (\lambda x. (\lambda z. (x z))) y$ // since λ extends to right
 - $\rightarrow (\lambda x. (\lambda z. (x z))) y$ // apply $(\lambda x. e1) e2 \rightarrow e1[x/e2]$
// where $e1 = \lambda z. (x z)$, $e2 = y$
 - $\rightarrow \lambda z. (y z)$ // final result
- Equivalent OCaml code
 - `(fun x -> (fun z -> (x z))) y -> fun z -> (y z)`

Parameters
• Formal
• Actual

CMSC 330

9

Lambda Calculus 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`

CMSC 330

10

Lambda Calculus 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
- $(\lambda x. \lambda y. x y) (\lambda z. zz) x \rightarrow \lambda y. ((\lambda z. zz) y) x \rightarrow (\lambda z. zz) x \rightarrow xx$

CMSC 330

11

Static Scoping & Alpha Conversion

- 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 is allowed
 - This is called **alpha-renaming** or **alpha conversion**
 - Ex. $\lambda x. x = \lambda y. y = \lambda z. z$ $\lambda y. \lambda x. y = \lambda z. \lambda x. z$

CMSC 330

12

Static Scoping (cont.)

- ▶ How about the following?
 - $(\lambda x. \lambda y. x \ y) \ y \rightarrow ?$
 - When we replace y inside, we don't want it to be **captured** by the inner binding of y
 - 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 change $(\lambda x. \lambda y. x \ y) \ y$ to $(\lambda x. \lambda z. x \ z) \ y$ first
 - > Now $(\lambda x. \lambda z. x \ z) \ y \rightarrow \lambda z. y \ z$

CMSC 330

13

Beta-Reduction, Again

- ▶ Whenever we do a step of beta reduction
 - $(\lambda x. e1) \ e2 \rightarrow e1[x/e2]$
 - We must first alpha-convert variables as necessary
 - Usually performed implicitly (w/o showing conversion)
- ▶ Examples
 - $(\lambda x. \lambda y. x \ y) \ y = (\lambda x. \lambda z. x \ z) \ y \rightarrow \lambda z. y \ z$ // $y \rightarrow z$
 - $(\lambda x. x \ (\lambda x. x)) \ z = (\lambda y. y \ (\lambda x. x)) \ z \rightarrow z \ (\lambda x. x)$ // $x \rightarrow y$
 - $(\lambda x. x \ (\lambda x. x)) \ z = (\lambda x. x \ (\lambda y. y)) \ z \rightarrow z \ (\lambda y. y)$ // $x \rightarrow y$

CMSC 330

14

Encodings

- ▶ The lambda calculus is Turing complete
- ▶ Means we can **encode** any computation we want
 - If we're sufficiently clever...
- ▶ Examples
 - Booleans
 - Pairs
 - Natural numbers & arithmetic
 - Looping

CMSC 330

15

Booleans

- ▶ Church's encoding of mathematical logic
 - $\text{true} = \lambda x. \lambda y. x$
 - $\text{false} = \lambda x. \lambda y. y$
 - if a then b else c
 - > Defined to be the λ expression: $a \ b \ c$
- ▶ Examples
 - if true then b else c $\rightarrow (\lambda x. \lambda y. x) \ b \ c \rightarrow (\lambda y. b) \ c \rightarrow b$
 - if false then b else c $\rightarrow (\lambda x. \lambda y. y) \ b \ c \rightarrow (\lambda y. y) \ c \rightarrow c$

CMSC 330

16

Booleans (cont.)

- ▶ Other Boolean operations
 - $\text{not} = \lambda x. ((x \ \text{false}) \ \text{true})$
 - > $\text{not true} \rightarrow \lambda x. ((x \ \text{false}) \ \text{true}) \ \text{true} \rightarrow ((\text{true} \ \text{false}) \ \text{true}) \rightarrow \text{false}$
 - $\text{and} = \lambda x. \lambda y. ((x \ y) \ \text{false})$
 - $\text{or} = \lambda x. \lambda y. ((x \ \text{true}) \ y)$
- ▶ Given these operations
 - Can build up a logical inference system

CMSC 330

17

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

CMSC 330

18

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., $n = \lambda f.\lambda y.<\text{apply } f\ n\ \text{times to } y>$

*(Alonzo Church, of course)

CMSC 330

19

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

- $\text{succ } 0 =$
 $(\lambda z.\lambda f.\lambda y.f\ (z\ f\ y))\ (\lambda f.\lambda y.y) \rightarrow$
 $\lambda f.\lambda y.f\ ((\lambda f.\lambda y.y)\ f\ y) \rightarrow$
 $\lambda f.\lambda y.f\ ((\lambda y.y)\ y) \rightarrow$ Since $(\lambda x.y)\ z \rightarrow y$
 $\lambda f.\lambda y.f\ y$
 $= 1$

CMSC 330

20

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

- $\text{iszero } 0 =$ • $0 = \lambda f.\lambda y.y$
 $(\lambda z.z\ (\lambda y.\text{false})\ \text{true})\ (\lambda f.\lambda y.y) \rightarrow$
 $(\lambda f.\lambda y.y)\ (\lambda y.\text{false})\ \text{true} \rightarrow$
 $(\lambda y.y)\ \text{true} \rightarrow$ Since $(\lambda x.y)\ z \rightarrow y$
 true

CMSC 330

21

Arithmetic Using Church Numerals

If M and N are numbers (as λ expressions)

- Can also encode various arithmetic operations

Addition

- $M + N = \lambda x.\lambda y.(M\ x)((N\ x)\ y)$
Equivalently: $+ = \lambda M.\lambda N.\lambda x.\lambda y.(M\ x)((N\ x)\ y)$
➤ In prefix notation $(+ M N)$

Multiplication

- $M * N = \lambda x.(M\ (N\ x))$
Equivalently: $* = \lambda M.\lambda N.\lambda x.(M\ (N\ x))$
➤ In prefix notation $(* M N)$

CMSC 330

22

Arithmetic (cont.)

Prove $1+1 = 2$

- $1+1 = \lambda x.\lambda y.(1\ x)((1\ x)\ y) =$ • $1 = \lambda f.\lambda y.f\ y$
• $\lambda x.\lambda y.((\lambda x.\lambda y.x\ y)\ x)((\lambda x.\lambda y.x\ y)\ x)\ y) \rightarrow$ • $2 = \lambda f.\lambda y.f\ (f\ y)$
• $\lambda x.\lambda y.(\lambda y.x\ y)((\lambda x.\lambda y.x\ y)\ x)\ y) \rightarrow$
• $\lambda x.\lambda y.(\lambda y.x\ y)((\lambda y.x\ y)\ y) \rightarrow$
• $\lambda x.\lambda y.x\ ((\lambda y.x\ y)\ y) \rightarrow$
• $\lambda x.\lambda y.x\ (x\ y) = 2$ Many implicit alpha conversions

With these definitions

- Can build a theory of arithmetic

CMSC 330

23

Looping

Define $D = \lambda x.x\ x$, then

- $D\ D = (\lambda x.x\ x)\ (\lambda x.x\ x) \rightarrow (\lambda x.x\ x)\ (\lambda x.x\ x) = D\ D$

So $D\ D$ is an infinite loop

- In general, self application is how we get looping

CMSC 330

24

The “Paradoxical” Combinator

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

► Then

$Y F =$

$(\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) F \rightarrow$
 $(\lambda x. F (x x)) (\lambda x. F (x x)) \rightarrow$
 $F ((\lambda x. F (x x)) (\lambda x. F (x x)))$
 $= F (Y F)$



► Thus $Y F = F (Y F) = F (F (Y F)) = \dots$

- We can use Y to achieve recursion for F

CMSC 330

25

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

$(Y \text{ fact}) 1 = (\text{fact } (Y \text{ fact})) 1$

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

CMSC 330

26

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 ($10000 + 1 \rightarrow$ thousands of function calls)
- Pretty large ($10000 + 1 \rightarrow$ 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

CMSC 330

27

The Need For Types

► Consider the **untyped** lambda calculus

- $\text{false} = \lambda x. \lambda y. y$
- $0 = \lambda x. \lambda y. y$

► Since everything is encoded as a function...

- We can easily misuse terms...

► $\text{false } 0 \rightarrow \lambda y. y$
► $\text{if } 0 \text{ then } \dots$

...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

CMSC 330

28

Simply-Typed Lambda Calculus

► $e ::= n \mid x \mid \lambda x. t. e \mid e e$

- Added integers n as primitives
 - Need at least two distinct types (integer & function)...
 - ...to have type errors
- Functions now include the type of their argument

CMSC 330

29

Simply-Typed Lambda Calculus (cont.)

► $t ::= \text{int} \mid t \rightarrow t$

- int is the type of integers
- $t_1 \rightarrow t_2$ is the type of a function
 - That takes arguments of type t_1 and returns result of type t_2
- t_1 is the **domain** and t_2 is the **range**
- Notice this is a recursive definition
 - So we can give types to higher-order functions

► Will show how to compute types later

- Example of operational semantics

CMSC 330

30

Summary

- Lambda calculus shows issues with
 - [Scoping](#)
 - [Higher-order functions](#)
 - [Types](#)
- Useful for understanding how languages work