

Introduction to Lambda Calculus

Lecture 12-13
CS263

CS 263

1

Syntax

- The λ -calculus has three kinds of expressions (terms)
 - $e ::= x$ Variables
 - | $\lambda x.e$ Functions (abstraction)
 - | $e_1 e_2$ Application
- $\lambda x.e$ is a one-argument function with body e
- $e_1 e_2$ is a function application
- Application associates to the left
 - $x y z$ means $(x y) z$
- Abstraction extends to the right as far as possible
 - $\lambda x.\lambda y.x y z$ means $\lambda x.(x (\lambda y. ((x y) z)))$

CS 263

3

Background

- Developed in 1930's by Alonzo Church
- Subsequently studied (and still studied) by many people
- Considered the "testbed" for procedural and functional languages
 - Simple
 - Powerful
 - Easy to extend with features of interest
 - Plays similar role for PL research as Turing machine does for computability

"Whatever the next 700 languages turn out to be, they will surely be variants of lambda calculus."

CS 263

(Landin⁵66)

Plan

- Introduce lambda calculus
 - Syntax and operational semantics
 - Technicalities
 - Evaluation strategies
- Relationship to programming languages (next time)
- Study of types and type systems (later)

CS 263

2

Examples of Lambda Expressions

- The identity function:
 - $I =_{\text{def}} \lambda x. x$
- A function that given an argument y discards it and yields the identity function:
 - $\lambda y. (\lambda x. x)$
- A function that given a function f invokes it on the identity function
 - $\lambda f. f (\lambda x. x)$

CS 263

4

Scope of Variables

- As in all languages with variables it is important to discuss the notion of scope
 - Recall: the scope of an identifier is the portion of a program where the identifier is accessible
- An abstraction $\lambda x. E$ binds variable x in E
 - x is the newly introduced variable
 - E is the scope of x
 - We say x is bound in $\lambda x. E$
 - Just like formal function arguments are bound in the function body

CS 263

6

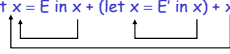

Free and Bound Variables

- A variable is said to be free in E if it has occurrences that are not bound in E
- We can define the free variables of an expression E recursively as follows:
 - $\text{Free}(x) = \{x\}$
 - $\text{Free}(E_1 E_2) = \text{Free}(E_1) \cup \text{Free}(E_2)$
 - $\text{Free}(\lambda x. E) = \text{Free}(E) - \{x\}$
- Example: $\text{Free}(\lambda x. x (\lambda y. x y z)) = \{z\}$
- Free variables are (implicitly or explicitly) declared outside the term

CS 263

7

Free and Bound Variables (Cont.)

- Just like in any language with statically nested scoping we have to worry about variable shadowing
 - An occurrence of a variable might refer to different things in different contexts
- E.g., in IMP with locals: $\text{let } x = E \text{ in } x + (\text{let } x = E' \text{ in } x) + x$

- In λ -calculus: $\lambda x. x (\lambda x. x) x$


CS 263

8

Renaming Bound Variables

- λ -terms that can be obtained from one another by renaming of the bound variables are considered identical. This is called α -equivalence.
- Example: $\lambda x. x$ is identical to $\lambda y. y$ and to $\lambda z. z$
- Intuition:
 - By changing the name of a formal argument and of all its occurrences in the function body, the behavior of the function does not change
 - In λ -calculus such functions are considered identical

CS 263

9

Renaming Bound Variables (Cont.)

- Convention: we will always try to rename bound variables so that they are all unique
 - e.g., write $\lambda x. x (\lambda y. y) x$ instead of $\lambda x. x (\lambda x. x) x$
- This makes it easy to see the scope of bindings
- And also prevents confusion!

CS 263

10

Substitution

- The substitution of E' for x in E (written $[E'/x]E$)
 - Step 1. Rename bound variables in E and E' so they are unique
 - Step 2. Perform the textual substitution of E' for x in E
- Example: $[y (\lambda v. v) / x] \lambda y. (\lambda x. x) y x$
 - After renaming: $[y (\lambda v. v) / x] \lambda z. (\lambda u. u) z x$
 - After substitution: $\lambda z. (\lambda u. u) z (y (\lambda v. v))$
- If we are not careful with scopes might get:
 - $\lambda y. (\lambda x. x) y (y (\lambda x. x))$

CS 263

11

The deBruijn Notation

- An alternative syntax that avoids naming of bound variables (and the subsequent confusions)
- The deBruijn index of a variable *occurrence* is the number of lambda's that separate the occurrence from its binding lambda in the abstract syntax tree
- The deBruijn notation replaces names of occurrences with their deBruijn index
- Examples:

- $\lambda x. x$	$\lambda. 0$	Identical terms have identical representations!
- $\lambda x. \lambda x. x$	$\lambda. \lambda. 0$	
- $\lambda x. \lambda y. y$	$\lambda. \lambda. 0$	
- $(\lambda x. x x) (\lambda x. x x)$	$(\lambda. 0 \ 0) (\lambda. 0 \ 0)$	
- $\lambda x. (\lambda x. \lambda y. x) x$	$\lambda. (\lambda. \lambda. 1) 0$	

CS 263

12

Combinators

- A λ -term without free variables is closed or a combinator
- Some interesting combinators:
 - $I = \lambda x. x$
 - $K = \lambda x. \lambda y. x$
 - $S = \lambda f. \lambda g. \lambda x. f (g x)$
 - $D = \lambda x. x x$
 - $Y = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$
- Theorem: Any closed term is equivalent to one written with just S, K, I
 - Example: $D =_{\beta} S I I$ (we'll discuss this form of equivalence)

CS 263

13

Informal Semantics

- We consider only closed terms
- The evaluation of $(\lambda x. e) e'$
 - Binds x to e'
 - Evaluates e with the new binding
 - Yields the result of this evaluation
- Like a function call, or like "let $x = e'$ in e "
- Example: $(\lambda f. f (f e)) g$ evaluates to $g (g e)$

CS 263

14

Operational Semantics

- There exist many operational semantics for the λ -calculus
- All are based on the equation $(\lambda x. e) e' =_{\beta} [e'/x]e$ usually oriented from left to right
- This is called the β -rule and the evaluation step a β -reduction
- The subterm $(\lambda x. e) e'$ is a β -redex
- We write $e \rightarrow_{\beta} e'$ to say that e β -reduces to e' in one step
- We write $e \rightarrow_{\beta}^* e'$ to say that e β -reduces to e' in 0 or more steps

CS 263

15

Examples of Evaluation

- The identity function: $(\lambda x. x) E \rightarrow [E/x]x = E$
- Another example with the identity: $(\lambda f. f (\lambda x. x)) (\lambda x. x) \rightarrow [\lambda x. x / f] f (\lambda x. x) = [(\lambda x. x) / f] f (\lambda y. y) = (\lambda x. x) (\lambda y. y) \rightarrow [\lambda y. y / x]x = \lambda y. y$
- A non-terminating evaluation: $(\lambda x. xx)(\lambda y. yy) \rightarrow [\lambda y. yy / x]xx = (\lambda y. yy)(\lambda y. yy) \rightarrow \dots$
- Try $T T$, where $T = \lambda x. x x x$

CS 263

16

Evaluation and the Static Scope

- The definition of substitution guarantees that evaluation respects static scoping:

$$(\lambda x. (\lambda y. y x)) (y (\lambda x. x)) \rightarrow_{\beta} \lambda z. z (y (\lambda v. v))$$

(y remains free, i.e., defined externally)

- If we forget to rename the bound y :

$$(\lambda x. (\lambda y. y x)) (y (\lambda x. x)) \rightarrow_{\beta} \lambda y. y (y (\lambda v. v))$$

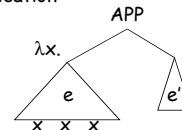
(y was free before but is bound now)

CS 263

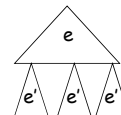
17

Another View of Reduction

- The application



- becomes:



Terms can "grow" substantially through β -reduction!

CS 263

18

Normal Forms

- A term without redexes is in normal form
- A reduction sequence stops at a normal form
- If e is in normal form and $e \rightarrow_{\beta}^* e'$ then e is identical to e'
- $K = \lambda x. \lambda y. x$ is in normal form
- $K I$ is not in normal form

CS 263

19

Nondeterministic Evaluation

- We define a small-step reduction relation

$$\frac{}{(\lambda x. e) e' \rightarrow [e'/x]e}$$

$$\frac{e_1 \rightarrow e_1' \quad e_2 \rightarrow e_2'}{e_1 e_2 \rightarrow e_1' e_2'}$$

$$\frac{e \rightarrow e'}{\lambda x. e \rightarrow \lambda x. e'}$$

- This is a non-deterministic semantics
- Note that we evaluate under λ

CS 263

20

Contexts

- Define contexts with one hole
 $H ::= \bullet \mid \lambda x. H \mid H e \mid e H$
- Write $H[e]$ to denote the filling of the hole in H with the expression e
- Example:
 $H = \lambda x. x \bullet \quad H[\lambda y. y] = \lambda x. x (\lambda y. y)$
- Filling the hole allows variable capture!
 $H = \lambda x. x \bullet \quad H[x] = \lambda x. x x$

CS 263

21

Context-Based Formulation of Operational Sem.

- Contexts allow concise formulations of congruence rules (application of local reduction rules on subterms)

$$\frac{}{(\lambda x. e) e' \rightarrow [e'/x]e} \quad \frac{e \rightarrow e'}{H[e] \rightarrow H[e']}$$

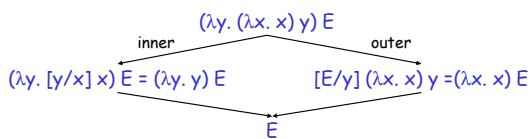
- Reduction occurs at a β -redex that can be anywhere inside the expression
- The latter rule is called a congruence or structural rule
- The above rules do not specify which redex must be reduced first

CS 263

22

The Order of Evaluation

- In a λ -term there could be more than one instance of $(\lambda x. E) E'$
 $(\lambda y. (\lambda x. x) y) E$
 - could reduce the inner or the outer λ
 - which one should we pick?

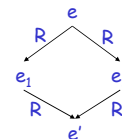


CS 263

23

The Diamond Property

- A relation R has the diamond property if whenever $e R e_1$ and $e R e_2$ then there exists e' such that $e_1 R e'$ and $e_2 R e'$



- \rightarrow_{β} does not have the diamond property
- \rightarrow_{β}^* has the diamond property
- The simplest known proof is quite technical

CS 263

24

The Diamond Property

- Languages defined by non-deterministic sets of rules are common
 - Logic programming languages
 - Expert systems
 - Constraint satisfaction systems
 - Make
- It is useful to know whether such systems have the diamond property

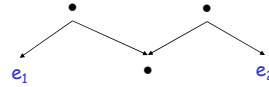
CS 263

25

Equality

- Let $=_{\beta}$ be the reflexive, transitive and symmetric closure of \rightarrow_{β}

$$=_{\beta} \text{ is } (\rightarrow_{\beta} \cup \leftarrow_{\beta})^*$$
- That is, $e_1 =_{\beta} e_2$ if e_1 converts to e_2 via a sequence of forward and backward \rightarrow_{β}

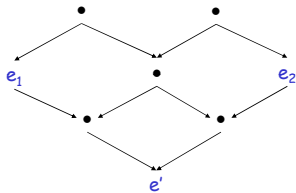


CS 263

26

The Church-Rosser Theorem

- If $e_1 =_{\beta} e_2$ then there exists e' such that $e_1 \rightarrow_{\beta}^* e'$ and $e_2 \rightarrow_{\beta}^* e'$



- Proof (informal): apply the diamond property as many times as necessary

CS 263

27

Corollaries

- If $e_1 =_{\beta} e_2$ and e_1 and e_2 are normal forms then e_1 is identical to e_2
 - From CR we have $\exists e'. e_1 \rightarrow_{\beta}^* e'$ and $e_2 \rightarrow_{\beta}^* e'$
 - Since e_1 and e_2 are normal forms they are identical to e'
- If $e \rightarrow_{\beta}^* e_1$ and $e \rightarrow_{\beta}^* e_2$ and e_1 and e_2 are normal forms then e_1 is identical to e_2
 - All terms have a unique normal form

CS 263

28

Evaluation Strategies

- Church-Rosser theorem says that independent of the reduction strategy we will not find more than one normal form
- But some reduction strategies might fail to find a normal form
 - $(\lambda x. y) ((\lambda y. y y) (\lambda y. y y)) \rightarrow (\lambda x. y) ((\lambda y. y y) (\lambda y. y y)) \rightarrow \dots$
 - $(\lambda x. y) ((\lambda y. y y) (\lambda y. y y)) \rightarrow y$
- We will consider three strategies
 - normal order
 - call-by-name
 - call-by-value

CS 263

29

Normal-Order Reduction

- A redex is outermost if it is not contained inside another redex.
 - Example:

$$S (K x y) (K u v)$$
 - $K x$, $K u$ and $S (K x y)$ are all redexes
 - Both $K u$ and $S (K x y)$ are outermost
 - Normal order always reduces the *leftmost outermost* redex first
- Theorem: If e has a normal form e' then normal order reduction will reduce e to e'

CS 263

30

Why Not Normal Order ?

- In most (all?) programming languages, functions are considered values (fully evaluated)
- Example:
 $\lambda x. D D = \perp$ (with normal order)
- Thus, no reduction is done under lambda
- No popular programming language uses normal order

CS 263

31

Call-by-Name

- **Don't** reduce under λ
 - **Don't** evaluate the argument to a function call
 - A value is an abstraction
- $$\frac{}{\lambda x. e \rightarrow_n^* \lambda x. e} \quad \frac{e_1 \rightarrow_n^* \lambda x. e_1' \quad [e_2/x]e_1' \rightarrow_n^* e}{e_1 e_2 \rightarrow_n^* e}$$
- Call-by-name is demand-driven: an expression is not evaluated unless needed
 - It is normalizing: converges whenever normal order converges
 - Call-by-name does not necessarily evaluate to a normal form. Example: $D D$

CS 263

32

Call by Name

- Example:
 $(\lambda y. (\lambda x. x) y) ((\lambda u. u) (\lambda v. v)) \rightarrow_{\beta n}$
 $(\lambda x. x) ((\lambda u. u) (\lambda v. v)) \rightarrow_{\beta n}$
 $(\lambda u. u) (\lambda v. v) \rightarrow_{\beta n}$
 $\lambda v. v$

CS 263

33

Call-by-Value Evaluation

- **Don't** reduce under lambda
 - **Do** evaluate the arguments to a function call
 - A value is an abstraction
- $$\frac{}{\lambda x. e \rightarrow_v^* \lambda x. e} \quad \frac{e_1 \rightarrow_v^* \lambda x. e_1' \quad e_2 \rightarrow_v^* e_2' \quad [e_2'/x]e_1' \rightarrow_v^* e}{e_1 e_2 \rightarrow_v^* e}$$
- Most languages are primarily call-by-value
 - But CBV is not normalizing: $(\lambda x. I) (D D)$
 - CBV diverges more often than normal order or even CBN

CS 263

34

Call by Value

- Example:
 $(\lambda y. (\lambda x. x) y) ((\lambda u. u) (\lambda v. v)) \rightarrow_{\beta v}$
 $(\lambda y. (\lambda x. x) y) (\lambda v. v) \rightarrow_{\beta v}$
 $(\lambda x. x) (\lambda v. v) \rightarrow_{\beta v}$
 $\lambda v. v$

CS 263

35

Considerations

- Call-by-value:
 - easy to implement
 - well-behaved (predictable) with respect to side-effects
- Call-by-name:
 - More difficult to implement (must pass unevaluated expressions)
 - The order of evaluation is harder to predict (e.g., difficulty with side-effects)
 - Has a simpler theory than call-by-value
 - Allows the natural expression of infinite data structures (e.g. streams)
 - Terminates more often than call-by-value

CS 263

36

CBV vs. CBN

- The debate about whether languages should be strict (CBV) or lazy (CBN) is nearly 20 years old
- This debate is confined to the functional programming community (where it is sometimes intense)
- Outside the functional community CBN is rarely considered (but remember TeX)

CS 263

37

Caveats

- The terms lazy and strict are not used consistently in the literature
- Call-by-value and call-by-name are well defined
- There are parameter passing mechanisms besides call-by-value and call-by-name that cannot be naturally expressed in the lambda calculus
 - by reference
 - value-result
- These additional mechanisms deal with side-effects
- Many languages have a mixture of parameter passing mechanisms

CS 263

38

Lambda-calculus and Functional Programming

CS 263

39

Review

- λ -calculus is a calculus of functions
 $e ::= x \mid \lambda x. e \mid e_1 e_2$
- Several evaluation strategies exist based on β -reduction
 $(\lambda x. e) e' \rightarrow_{\beta} [e'/x] e$
- How does this simple calculus relate to real programming languages?

CS 263

40

Functional Programming

- The λ -calculus is a prototypical functional language with:
 - no side effects
 - several evaluation strategies
 - lots of functions
 - nothing but functions (pure λ -calculus does not have any other data type)
- How can we program with functions?
- How can we program with only functions?

CS 263

41

Programming With Functions

- Functional programming style is a programming style that relies on lots of functions
- A typical functional paradigm is using functions as arguments or results of other functions
 - Higher-order programming
- Some "impure" functional languages permit side-effects (e.g., Lisp, ML)
 - references (pointers), in-place update, arrays, exceptions

CS 263

42

Variables in Functional Languages

- We can introduce new variables:
 $\text{let } x = e_1 \text{ in } e_2$
 - x is bound by `let`
 - x is statically scoped in e_2
- This is pretty much like $(\lambda x. e_2) e_1$
- In a functional language, variables are never updated
 - they are just names for expressions or values
 - E.g., x is a name for the value denoted by e_1 in e_2
- This models the meaning of "let" in math

CS 263

43

Referential Transparency

- In "pure" functional programs, we can reason equationally, by substitution
 $\text{let } x = e_1 \text{ in } e_2 \equiv [e_1/x]e_2$
- In an imperative language a "side-effect" in e_1 might invalidate the above equation
- The behavior of a function in a "pure" functional language depends only on the actual arguments
 - Just like a function in math
 - This makes it easier to understand and to reason about functional programs

CS 263

44

Expressiveness of λ -Calculus

- The λ -calculus is a minimal system but can express
 - data types (integers, booleans, lists, trees, etc.)
 - branching
 - recursion
- This is enough to encode Turing machines
- Corollary: $e =_{\beta} e'$ is undecidable
- Still, how do we encode all these constructs using only functions?
- Idea: encode the "behavior" of values and not their structure

CS 263

45

Encoding Booleans in Lambda Calculus

- What can we do with a boolean?
 - we can make a binary choice
- A boolean is a function that given two choices selects one of them
 - $\text{true} =_{\text{def}} \lambda x. \lambda y. x$
 - $\text{false} =_{\text{def}} \lambda x. \lambda y. y$
 - if E_1 then E_2 else $E_3 =_{\text{def}} E_1 E_2 E_3$
- Example: "if true then u else v " is
 $(\lambda x. \lambda y. x) u v \rightarrow_{\beta} (\lambda y. u) v \rightarrow_{\beta} u$

CS 263

46

Encoding Pairs in Lambda Calculus

- What can we do with a pair?
 - we can select one of its elements
- A pair is a function that given a boolean returns the left or the right element
 $\text{mkpair } x y =_{\text{def}} \lambda b. b x y$
 $\text{fst } p =_{\text{def}} p \text{ true}$
 $\text{snd } p =_{\text{def}} p \text{ false}$
- Example:
 $\text{fst } (\text{mkpair } x y) \rightarrow (\text{mkpair } x y) \text{ true} \rightarrow \text{true } x y \rightarrow x$

CS 263

47

Encoding Natural Numbers in Lambda Calculus

- What can we do with a natural number?
 - we can iterate a number of times over some function
- A natural number is a function that given an operation f and a starting value s , applies f a number of times to s :
 $0 =_{\text{def}} \lambda f. \lambda s. s$
 $1 =_{\text{def}} \lambda f. \lambda s. f s$
 $2 =_{\text{def}} \lambda f. \lambda s. f (f s)$
and so on
- These are numerals in unary representation
- Also called Church numerals

CS 263

48

Computing with Natural Numbers

- The successor function
 - $\text{succ } n =_{\text{def}} \lambda f. \lambda s. f (n f s)$
 - or $\text{succ } n =_{\text{def}} \lambda f. \lambda s. n f (f s)$
- Addition
 - $\text{add } n_1 n_2 =_{\text{def}} n_1 \text{ succ } n_2$
- Multiplication
 - $\text{mult } n_1 n_2 =_{\text{def}} n_1 (\text{add } n_2) 0$
- Testing equality with 0
 - $\text{iszero } n =_{\text{def}} n (\lambda b. \text{false}) \text{true}$
- Subtraction
 - homework exercise

CS 263

49

Computing with Natural Numbers. Example

```
mult 2 2 →
2 (add 2) 0 →
(add 2) ((add 2) 0) →
2 succ (add 2 0) →
2 succ (2 succ 0) →
succ (succ (succ (succ 0))) →
succ (succ (succ (λf. λs. f (0 f s)))) →
succ (succ (succ (λf. λs. f s))) →
succ (succ (λg. λy. g ((λf. λs. f s) g y)))
succ (succ (λg. λy. g (g y))) →* λg. λy. g (g (g y)) = 4
```

CS 263

50

Computing with Natural Numbers. Example

- What is the result of the application $\text{add } 0$?
 - $(\lambda n_1. \lambda n_2. n_1 \text{ succ } n_2) 0 \rightarrow_{\beta}$
 - $\lambda n_2. 0 \text{ succ } n_2 =$
 - $\lambda n_2. (\lambda f. \lambda s. s) \text{ succ } n_2 \rightarrow_{\beta}$
 - $\lambda n_2. n_2 =$
 - $\lambda x. x$
- By computing with functions we can express some optimizations
 - But we need to reduce under the lambda

CS 263

51

Encoding Recursion

- Given a predicate P encode the function "find" such that "find P n " is the smallest natural number which is larger than n and satisfies P
 - with find we can encode all recursion
- "find" satisfies the equation
 - $\text{find } p \ n = \text{if } p \ n \ \text{then } n \ \text{else } \text{find } p \ (\text{succ } n)$
- Define
 - $F = \lambda f. \lambda p. \lambda n. (p \ n) \ n \ (f \ p \ (\text{succ } n))$
- We need a fixed point of F
 - $\text{find} = F \ \text{find}$
- or
 - $\text{find } p \ n = F \ \text{find } p \ n$

CS 263

52

The Fixed-Point Combinator

- Let $Y = \lambda F. (\lambda y. F(y y)) (\lambda x. F(x x))$
 - This is called the fixed-point combinator
 - Verify that $Y F$ is a fixed point of F
 - $Y F \rightarrow_{\beta} (\lambda y. F(y y)) (\lambda x. F(x x)) \rightarrow_{\beta} F(Y F)$
 - Thus $Y F =_{\beta} F(Y F)$
- Given any function in λ -calculus we can compute its fixed-point
- Thus we can define "find" as the fixed-point of the function from the previous slide
- The essence of recursion is the self-application " $y y$ "

CS 263

53

Expressiveness of Lambda Calculus

- Encodings are fun
- But programming in pure λ -calculus is painful
- We will add constants (0, 1, 2, ..., true, false, if-then-else, etc.)
- And we will add types

CS 263

54