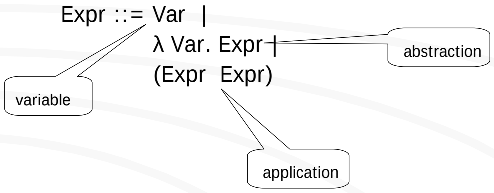
CSE 505

Lecture #7

September 24, 2012

Lambda Calculus

A higher-order functional language, where functions are used as input amd output, and also to encode data.



Expr ::= Var |
λ Var. Expr |
(Expr Expr)

Examples of lambda terms

- λ x. x
- λ x. λ y. x
- λ f. λ x. (f (f x))
- λ f. λ g. λ x. (f (g x))
- …

Sometimes called “anonymous functions”

Informal Meaning

- λx. x
  ➔ identity function
- λx. λy. x
  ➔ a function of two parameters that returns the first parameter
- λf. λg. λx. (f (g x))
  ➔ the composition of two functions, f ◦ g

Bound and Free Occurrences

$\lambda x.(\underline{x}\ \ \underline{y})$

$\lambda f.\ \lambda x.\ (\underline{f}\ \ (\underline{f}\ \ \underline{x}))$

$\lambda x.\ (\lambda y.\ (\lambda x.(\underline{z}\ \ \underline{y})\ \ \underline{x})\ \ \underline{x})$

Free Occurrence of Variable

V occurs_free_in W iff V = W

V occurs_free_in λW.T iff V ǂ W and
V occurs_free_in T

V occurs_free_in (T_1 T_2) iff V occurs_free_in T_1
or
V occurs_free_in T_2

Substitution

(will be used for parameter passing)

Substitution of all free occurrences of a variable V by term T1 in term T2:

T2 [V ← T1]

e.g. λx. ($\underline{f}$ ($\underline{f}$ $\underline{x}$)) [f ← λy.y]

= λx. (λy.y (λy.y $\underline{x}$))

Note: λx. ($\underline{f}$ ($\underline{f}$ $\underline{x}$)) [x ← y]

ǂ λx. ($\underline{f}$ ($\underline{f}$ $\underline{y}$)) -- since x is bound

Substitution (cont'd)

λx. ($\underline{f}$ ($\underline{f}$ $\underline{x}$)) [f ← λy.(y x)]

ǂ λx. (λy.(y x) (λy.(y x) $\underline{x}$))

This is called the "variable capture" problem.

Correct way to do the substitution:

λx'. (λy.(y x) (λy.(y x) $\underline{x'}$))

Substitution Rule (definition)

$$V \ [V \leftarrow T] \ = \ T$$

$$V_1 \ [V \leftarrow T] \ = \ V_1, \ \textbf{if} \ V \neq V_1$$

$$(T_1 \ T_2) \ [V \leftarrow T] \ = \ (T_1[V \leftarrow T] \ \ T_2[V \leftarrow T])$$

$$\lambda \enclose{circle}{V}.T_1 \ [\enclose{circle}{V} \leftarrow T] \ = \ \lambda V.T_1$$

Substitution Rule

(definition continued)

$$\lambda V_1.T_1 \ [V \leftarrow T] \ = \ \lambda V_1.T_1[V \leftarrow T] \quad \text{if} \ \ V \neq V_1 \ \wedge \ \neg(V_1 \ \mathit{occurs\_free\_in} \ T)$$

$$\lambda V_1.T_1 \ [V \leftarrow T] \ = \ \lambda V_2.T_1[V_1 \leftarrow V_2][V \leftarrow T],$$

$$\begin{aligned} \textbf{if} \ V \neq V_1 \ &\wedge \ (V_1 \ \mathit{occurs\_free\_in} \ T) \\ &\wedge \neg(V_2 \ \mathit{occurs\_free\_in} \ T) \\ &\wedge \neg(V_2 \ \mathit{occurs\_free\_in} \ T_1) \end{aligned}$$

Renaming Bound Variables

Renaming the binder variables is always permissible – similar to renaming the formal parameters of a function. Thus:

λx. x = λy. y

λx. ($\underline{f}$ ($\underline{f}$ $\underline{x}$)) = λx'. ($\underline{f}$ ($\underline{f}$ $\underline{x'}$))

λf. λx. (f x) = λg. λy. (g y)

Reduction Rules

Three famous reduction rules: α, β, η

α-reduction is renaming of binder variables – it doesn't really "reduce" the term.

β-reduction resembles call-by-name, and is based on the substitution rule:

(λV.T1 T2) $\rightarrow_\beta$ T1 [V ← T2]

η-reduction is not so common:
λV.(T V) $\rightarrow_\eta$ T if V $\notin$ free(T)

Computation = β-Reduction

$(\underline{(\lambda f.\ \lambda x.\ (f\ (f\ x))\ \ \lambda x.\ x)}\ \ a)$

$\Rightarrow \underline{(\lambda x.\ (\lambda x.x\ \ (\lambda x.\ x\ \ x)\ )\ \ a)}$

$\Rightarrow (\lambda x.x\ \ \underline{(\lambda x.x\ \ a)}\ )$

$\Rightarrow (\lambda x.x\ \ a)$

$\Rightarrow a$

Another β-Reduction

$(\underline{(\lambda f.\ \lambda x.\ (f\ (f\ x))\ \ \lambda x.x)}\ \ a)$

$\Rightarrow (\lambda x.\ (\lambda x.x\ \ \underline{(\lambda x.x\ \ x)\ )}\ \ a)$

$\Rightarrow (\lambda x.\ \underline{(\lambda\ x\ .\ x\ \ x)\ }\ a)$

$\Rightarrow (\lambda x.\ x\ \ a)$

$\Rightarrow a$

Yet Another β-Reduction

$(\underline{(\lambda f.\ \lambda x.\ (f\ (f\ x))\ \ \lambda x.\ x)}\ \ a)$

$\Rightarrow (\lambda x.\ \underline{(\lambda x.\ x\ \ (\lambda x.\ x\ \ x)\ )\ }\ a)$

$\Rightarrow \underline{(\lambda x.\ \ (\lambda x.\ x\ \ x)\ \ a)}$

$\Rightarrow (\lambda x.\ x\ \ a)$

$\Rightarrow a$

Confluence Property

"If a lambda term T reduces to two terms T1 and T2, then T1 and T2 can be reduced to a common term U."

$$\begin{tikzcd}
 & T \arrow[dl, "*"'] \arrow[dr, "*"] & \\
T_1 \arrow[dr, "*"'] & & T_2 \arrow[dl, "*"] \\
 & U &
\end{tikzcd}$$

Unique Normal Form

If a term T reduces to a term U, and U cannot be reduced any further (by η- or β-reductions), then U is said to be in normal form.

Normal Form: The normal form of a term is unique if it exists. (Uniqueness is up to renaming of bound variables.)

Proof by Contradiction

Suppose T has two normal forms N_1 and N2:

$T \rightarrow^* N_1$ and $T \rightarrow^* N_2$

By Confluence Property,

$N_1 \rightarrow^* U$ and $N_2 \rightarrow^* U$

But N1 and N2 are irreducible, hence must be the same except for alpha-reductions, i.e., variable renaming.

Nontermination is Posssible!

(λ x . (x x)   λ x . (x x))

⇒

(λ x . (x x)   λ x . (x x))

⇒

(λ x . (x x)   λ x . (x x))

⇒

...

Leftmost Reductions

- How should we reduce a term in order that the normal form can be derived, if it exists?
- Answer: Choose the leftmost "redex" at every step.

- Let Ω = (λx . (x x)   λx. (x x))
- Then, (λx.a Ω) ➔ a, by leftmost reduction
- A nonterminating reduction sequence is:

  (λx.a Ω) ➔ (λx.a Ω) ➔ ...

Different Reduction Orders

- Leftmost Innermost
- Parallel Innermost
- Rightmost Innermost
- Parallel Outermost
- Leftmost Outermost = Leftmost

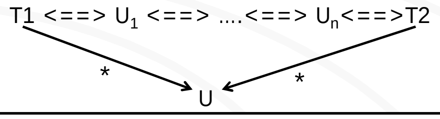
Church-Rosser Property

Defn: T1 <==> T2 if T1 ==> T2 or T2 ==> T1, where ==> uses one of the three reduction rules.

Defn: T1 <==>* T2 uses <==> 0 or more times.

Church-Rosser Property: If T1 <==>* T2 then there is a term U s.t. T1 ==>* U and T2 ==>* U.

Diagram:



T1 <==> U_1 <==> ....<==> U_n<==>T2

*

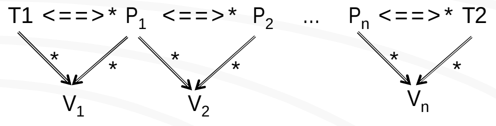
*

U

Relation between <==>* and ==>*

Note: T1 <==>* T2 does NOT imply T1 ==>* T2 or T2 ==>* T1.

In reality, given T1 <==>* T2, the situation is:



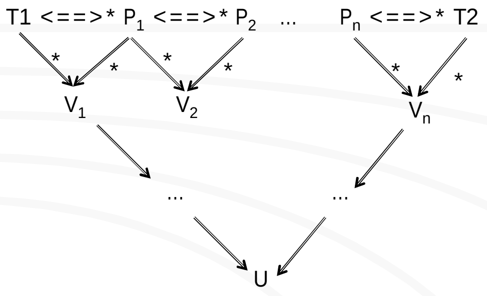
T1 <==>* P_1 <==>* P_2 ... P_n <==>* T2

* * * * * *

V_1 V_2 V_n

Confluence implies Church-Rosser

Proof (informal):



T1 <==>* P_1 <==>* P_2 ... P_n <==>* T2

* * * * * *

V_1 V_2 V_n

...   ...

U

Church-Rosser implies Confluence

Proof (easy): Given:

$$\begin{tikzcd} & T \arrow[dl, "*"'] \arrow[dr, "*"] & \\ T_1 & & T_2 \end{tikzcd}$$

Therefore: T_1 <==>* T_2

Therefore:

$$\begin{tikzcd} T_1 \arrow[dr, "*"'] & & T_2 \arrow[dl, "*"] \\ & U & \end{tikzcd}$$

by Church-Rosser

Data Representation

The boolean type can be represented as follows:

λx. λy. x ➔ can represent “true”

λx. λy. y ➔ can represent “false”

Representation of “not” operator:

λb.((b false) true)

Justification of “not” operator

We must show:

a. (not true) ➔* false

b. (not false) ➔* true

For example, (not true), i.e.,

(λb.((b false) true) true)

if = λb. λt. λe.((b t) e)

We can justify if-then-else by showing:

a. (((if true) T1) T2) ➔* T1

b. (((if false) T1) T2) ➔* T2

Example:

(((λb. λt. λe.((b t) e) true) T1) T2)

Note on Syntax

- Lisp syntax:

(and T1 T2)

(if B T1 T2)

...

Lambda calculus:

((and T1) T2)

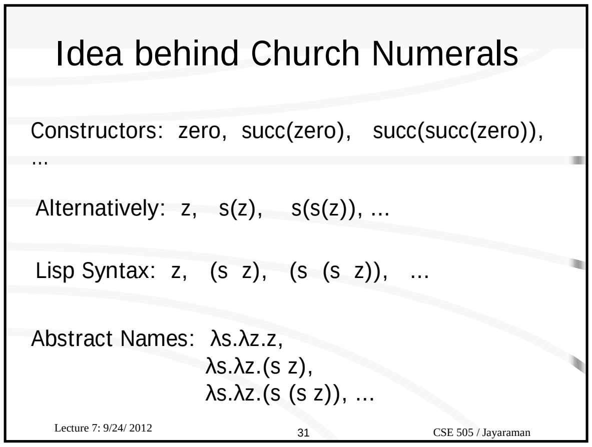
(((if B) T1) T2)

...

What datatype can this represent?

- λ f . λ x . x
- λ f . λ x . (f x)
- λ f . λ x . (f (f x))
- λ f . λ x . (f (f (f x)))
- ....

These are called Church Numerals.



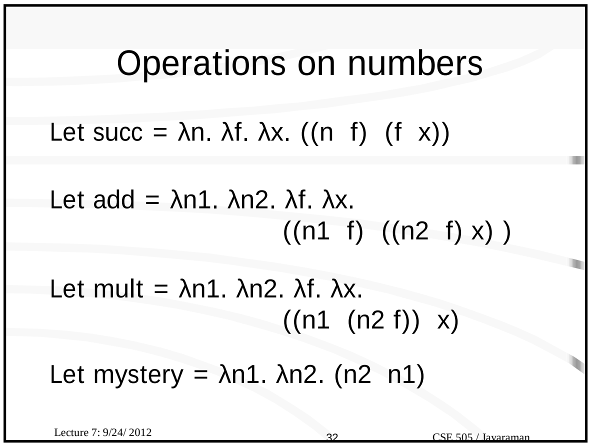
Idea behind Church Numerals

Constructors: zero, succ(zero), succ(succ(zero)), ...

Alternatively: z, s(z), s(s(z)), ...

Lisp Syntax: z, (s z), (s (s z)), ...

Abstract Names: λs.λz.z,
λs.λz.(s z),
λs.λz.(s (s z)), ...



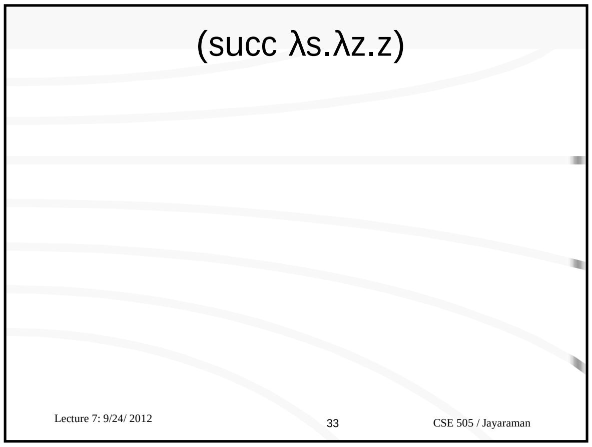
Operations on numbers

Let succ = λn. λf. λx. ((n f) (f x))

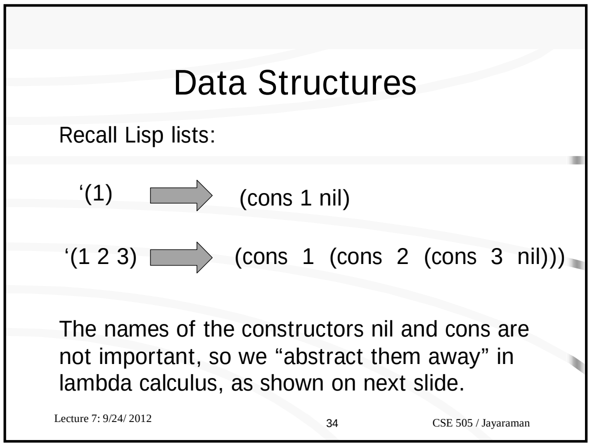
Let add = λn1. λn2. λf. λx.
((n1 f) ((n2 f) x))

Let mult = λn1. λn2. λf. λx.
((n1 (n2 f)) x)

Let mystery = λn1. λn2. (n2 n1)



(succ λs.λz.z)



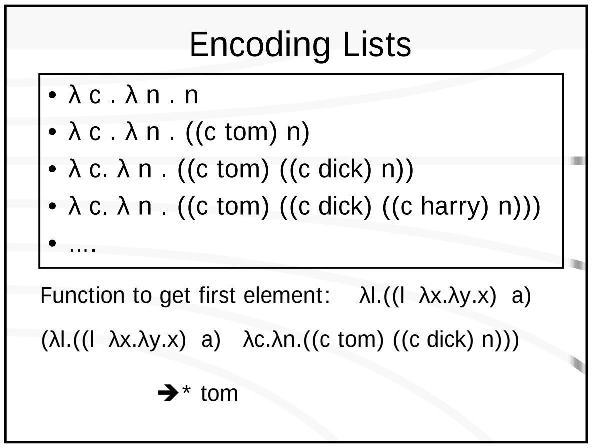
Data Structures

Recall Lisp lists:

‘(1) ⇨ (cons 1 nil)

‘(1 2 3) ⇨ (cons 1 (cons 2 (cons 3 nil)))

The names of the constructors nil and cons are not important, so we “abstract them away” in lambda calculus, as shown on next slide.



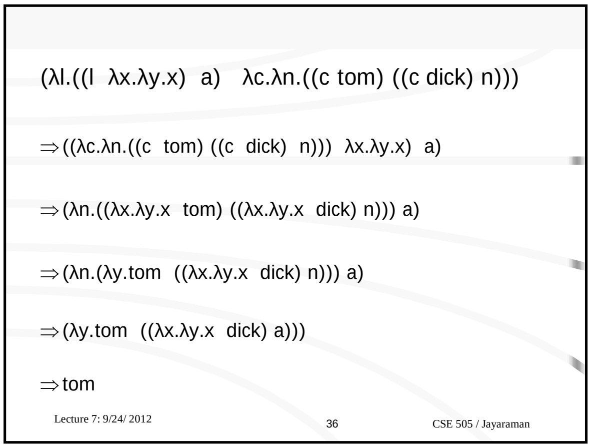
Encoding Lists

- λ c . λ n . n
- λ c . λ n . ((c tom) n)
- λ c. λ n . ((c tom) ((c dick) n))
- λ c. λ n . ((c tom) ((c dick) ((c harry) n)))
- ....

Function to get first element: λl.((l λx.λy.x) a)

(λl.((l λx.λy.x) a) λc.λn.((c tom) ((c dick) n)))

➔* tom



(λl.((l λx.λy.x) a) λc.λn.((c tom) ((c dick) n)))

⇒((λc.λn.((c tom) ((c dick) n))) λx.λy.x) a)

⇒(λn.((λx.λy.x tom) ((λx.λy.x dick) n))) a)

⇒(λn.(λy.tom ((λx.λy.x dick) n))) a)

⇒(λy.tom ((λx.λy.x dick) a)))

⇒tom

LAMBDA CALCULUS TOOL DEMO

Computability

- The language of lambda expressions is powerful enough to encode all computable functions!
- Notice that there is no recursive function definition – but this can be simulated, as will be next shown.

Recursive Definition

Consider recursive definition:

f(n) = if is0(n) then 1 else n * f(n-1)

Lisp syntax:

(defun f (n) (if (is0 n) 1 (* n (f (- n 1)))))

Lambda calculus (not quite):

letrec f = λn.(((if (is0 n)) 1)
((mult n) (f (pred n))))

Representing Recursion

letrec fact = λf. λn.(((if (is0 n)) 1)
((mult n) (fact (pred n))))

Fixed-Point Operator, Y:

let Y = λf. (λx.(f (x x)) (λx.(f (x x))

Note: Fixed point of f is an x such that f(x) = x

Non-recursive equivalent of original function: (Y t)

Y = λf. (λx.(f (x x)) λx.(f (x x)))

Y is fixed-point operator, because (for any t):

(Y t) <==>* (t (Y t))

Derivation:

(Y t) = (λf. (λx.(f (x x)) λx.(f (x x))) t)

==> (λx.(t (x x)) λx.(t (x x)))

==> (t (λx.(t (x x)) λx.(t (x x)))))

<==> (t (Y t))

Recursion and Fixed-points

fact = λn.(((if (is0 n)) 1) ((mult n) (fact (pred n)))

t = λf. λn.(((if (is0 n)) 1) ((mult n) (f (pred n))))

Why does the fixed-point of t capture f?

Fixed point g has the property: g = (t g)

g = λn.(((if (is0 n)) 1) ((mult n) (g (pred n)))

Least Fixed Point

Consider: letrec f(n) = if n=0 then 0 else f(n);

Fixed-point f1(n) = $\left\{\begin{array}{l} 0, \text{ if } n=0 \\ 1, \text{ if } n \geq 0 \end{array}\right\}$

Fixed-point f2(n) = $\left\{\begin{array}{l} 0, \text{ if } n=0 \\ 2, \text{ if } n \geq 0 \end{array}\right\}$

Least fixed-point g(n) = $\left\{\begin{array}{l} 0, \text{ if } n=0 \\ ?, \text{ if } n \geq 0 \end{array}\right\}$

Typed Lambda Calculi

Thus far, we have studied the untyped lambda calculus, i.e., no types associated with vars.

There are two well-known calculi:

- the simply-type lambda calculus
- the second-order (polymorphic) lambda calculus

Interesting, adding types causes all lambda expressions to terminate! Cannot have (x x).