

Short Road Map

1. First project: Do you need an extension? Right now, deadline is Tuesday, Oct. 18.
2. Midterm exam: 80 minutes; closed book; in class. Thursday, November 3?
3. Heads-up: No class on Tuesday, November 22 (officially a Thursday). Will need to schedule a make-up.

Review - Lambda calculus

- formalism for studying ways in which functions can be formed, combined, and used for computation
- **computation** is defined as rewriting rules (operational semantics)
- the syntactic notion of computation was developed first; a mathematical semantics followed much later

Examples:

$f(x) = x+2$	$\lambda x.x+2$	different notation
$(\lambda x.x+2) 1$	$1+2 = 3$	function application and substitution
$(\lambda x.x) (\lambda y.y)$		arguments and returned “values” can be functions
$\lambda x.xx$		untyped lambda calculus $f(x) = x(x)$

Lambda calculus and functional programming

Lambda calculus is the theoretical foundation of pure functional programming (no side effects, *referencial transparency*)

Functional programming: functions are *first class citizens*

- can be a return value
- can be passed as arguments
- can be put into a data structure
- value of an expression can be a function

$((\lambda x.x) (\lambda x.1)) (\lambda y.y)$

Currying functions

$f : D^n \rightarrow D$ can be transformed to

$$f : \underbrace{D \rightarrow (D \rightarrow \dots (D \rightarrow D)) \dots}_{n\text{-times}}$$

Examples:

$$f(x, y) = x + y$$

$$f : \mathcal{N} \times \mathcal{N} \rightarrow \mathcal{N}$$

$$f'(x) = g_x(y)$$

$$f' : \mathcal{N} \rightarrow (\mathcal{N} \rightarrow \mathcal{N})$$

$$g_x : \mathcal{N} \rightarrow \mathcal{N}$$

$$g_2 : \mathcal{N} \rightarrow \mathcal{N}$$

$$g_x(y) = x + y$$

$$g_2(y) = 2 + y$$

$g_x(y)$ “freezes” the first argument value x .

This is sometimes referred to as **partial evaluation**.

Here is an example:

$$(\lambda x. \lambda y. x + y) 2 3 =$$

$$(\lambda y. 2 + y) 3 =$$

$$2 + 3 = 5$$

To simplify discussion, all n -ary functions are curried, i.e., are represented by a sequence of one-place functions.

Examples

What are the λ -terms represented by

$\lambda xyz.x(yz)t$

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

Examples

What are the λ -terms represented by

$\lambda_{xyz}.x(yz)t$ is

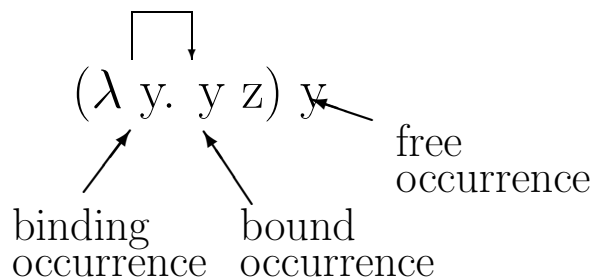
$(\lambda x.(\lambda y.(\lambda z.((x(yz))t))))$

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

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

Free and bound variables

Abstraction $(\lambda x. M)$ “binds” variable x in “body” M .
You can think of this as a declaration of variable x with scope M .



Let M, N be λ -terms and x is a variable. The set of *free variables of M* , $\text{free}(M)$, is defined inductively as follows:

- $\text{free}(x) = \{x\}$
- $\text{free}(M N) = \text{free}(M) \cup \text{free}(N)$
- $\text{free}(\lambda x.M) = \text{free}(M) - \{x\}$

Free and bound variables

Note:

- a variable can occur free and bound in a λ - term.
See example above

$$\lambda x. \overbrace{\lambda y. (\lambda z. xyz)}^{y \text{ is bound}} y$$

$y \text{ is free}$

“free” is relative to a λ -subterm

- formal definition of *bound*: exercise

Function application as substitution

The result of applying an abstraction $(\lambda x.M)$ to an argument N is formalized by a special form of textual substitution.

$$(\lambda x.M)N \cong [N/x]M$$

Informally: N replaces all free occurrences of x in M .

What can go wrong?

Example: Assume we have constants and arithmetic operation “+” in our lambda calculus

$$\begin{aligned} (\lambda a.\lambda b.a+b)2\ x &\cong \\ (\lambda b.2+b)x &\cong \\ 2+x \end{aligned}$$

What about:

$$\begin{aligned} (\lambda a.\lambda b.a+b)b\ 3 &\cong \\ (\lambda b.b+b)3 &\cong \\ 3+3 &\cong \\ 6 \end{aligned}$$

Function application as substitution

We need **capture free** substitution

(1) If the free variables of N have no bound occurrence in M , then $[N/x]M$ is formed by replacing all free occurrences of x in M by N .

Example: $(\lambda b.2+b) x \cong [x/b]2+b \cong 2+x$

(2) Otherwise, if variable y is free in N and bound in M , replace the binding and bound occurrences of y in M by new (fresh) variable z . Repeat until case (1) applies.

$$\begin{aligned} (\lambda a. \lambda b. a+b) b &\cong \\ \lambda a. \lambda z. a+z) b &\cong \\ (\lambda z. b+z) & \end{aligned}$$

Function application as substitution

Examples:

$$\begin{array}{lll} [u/x] x & \cong & u \quad \text{u not bound in } x \\ [u/x] \lambda x.xu & \cong & \lambda x.xu \quad \text{x not free in } \lambda x.xu \\ [u/x] \lambda u.x & \cong & [u/x] \lambda z.x \quad \text{u is bound in } \lambda u.x \\ & \cong & \lambda z.u \\ [u/x] \lambda u.u & \cong & [u/x] \lambda z.z \\ & \cong & \lambda z.z \end{array}$$

Capture-free substitution

Formal Definition

Let x, y, z denote variables and M, N, P arbitrary λ -terms.

$$\begin{aligned} [N/y] x &= x && \text{if } y \neq x \\ &= N && \text{if } y = x \end{aligned}$$

$$\begin{aligned} [N/x](\lambda x.M) &= (\lambda x.M) \\ [N/x](\lambda y.M) &= \lambda y.[N/x]M && \text{where } x \neq y, \\ &&& y \notin \text{free}(N), \\ [N/x](\lambda y.M) &= \lambda z.[N/x] [z/y]M && \text{where } x \neq y, \\ &&& y \in \text{free}(N), \\ &&& z \notin \text{free}(M) \\ &&& z \notin \text{free}(N) \end{aligned}$$

$$[N/x](M P) = ([N/x]M [N/x]P)$$

Substitution — Examples

$$[y/x] ((\lambda z.zx)(\lambda x.x)) =$$

$$[\lambda x.xy/x] ((\lambda y.xy)z) =$$

Substitution — Examples

$$\begin{aligned} [y/x] ((\lambda z.zx)(\lambda x.x)) &= \\ (([y/x] \lambda z.zx) ([y/x] \lambda x.x)) &= \\ (\lambda z.zy) (\lambda x.x) & \end{aligned}$$

$$\begin{aligned} [\lambda x.xy/x] ((\lambda y.xy)z) &= \\ (([\lambda x.xy/x] \lambda y.xy) [\lambda x.xy/x] z) &= \\ ((\lambda z.[\lambda x.xy/x][z/y] xy) z) &= \\ ((\lambda z.[\lambda x.xy/x] xz) z) &= \\ ((\lambda z.(\lambda x.xy) z) z) & \end{aligned}$$

Function application

Computation in the lambda calculus is based on the concept or **reduction** (rewriting rules). The goal is to “simplify” an expression until it can no longer be further simplified.

$$(\lambda x.M)N \quad \Rightarrow_{\beta} \quad [N/x]M \quad (\beta\text{-reduction})$$

$$(\lambda x.M) \quad \Rightarrow_{\alpha} \quad \lambda y.[y/x]M \quad (\alpha\text{-reduction})$$

if $y \notin \text{free}(M)$

$$(\lambda x.M)_x \quad \Rightarrow_{\eta} \quad M \quad (\eta\text{-reduction})$$

if $x \notin \text{free}(M)$

Note:

- An equivalence relation can be defined based on \cong -convertible λ -terms. “Reduction” rules really work both ways, but we are interested in reducing the complexity of λ -term (\rightarrow direction).
- α -reduction does not reduce the complexity.
- β -reduction: corresponds to application, models computation.
- η -reduction: one can remove (or introduce) levels of indirections in function applications.

Reduction — example

$(\lambda xyz.(xz)(yz)) (\lambda xy.x) (\lambda xy.x)$

$(\lambda yz.((\lambda xy.x)z)(yz)) (\lambda xy.x)$

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

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

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

$(\lambda yz.z)(\lambda xy.x)$

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

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

$\lambda z.z$

Reduction

- A subterm of the form $(\lambda x.M)N$ is called a *redex* (reduction expression).
- A reduction is any sequence of β -reductions and α -reductions.
- A term that cannot be β -reduced is said to be in β -normal form (**normal form**).
- A subterm that is an abstraction or a variable is said to be in **head normal form**.

Does a normal form always exist?

Examples:

$(\lambda x.xx)(\lambda x.xx)$

$(\lambda x.xxx)(\lambda x.xxx)$

Church–Rosser Property

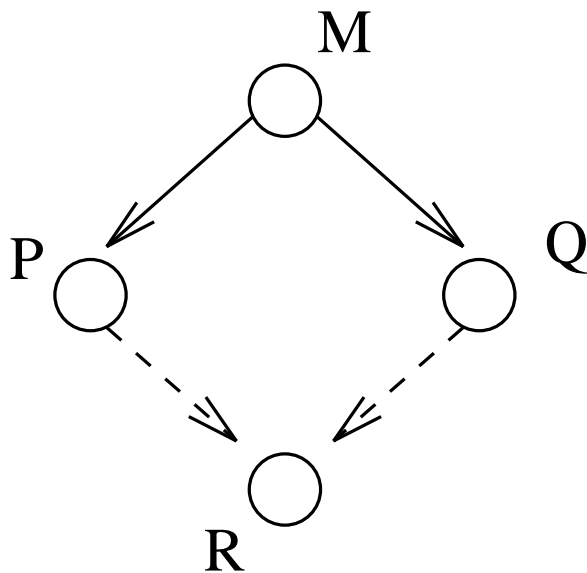
Informal: If an expression reduces to two normal forms, they must be identical modulo α -conversion.

Theorem (CR-I)

Let M , P , Q , and R be λ -terms.

If $M \Rightarrow^* P$ and $M \Rightarrow^* Q$, then

$\exists R$ such that $P \Rightarrow^* R$ and $Q \Rightarrow^* R$



diamond property

Corollary: If a normal form exists for M , it must be unique. Proof?

Reduction strategies

If P is not in normal form, what redex to choose?

Does it make a difference (remember CR-I)?

Yes, since there are infinite computations possible

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

Note:

- It is undecidable whether a λ -term has a normal form, i.e., whether it will reduce to a normal form.
- You can think of the normal form as the “value” to which a λ -term evaluates
- Two λ -terms are equal if they have the same normal form.

Reduction strategies & CR-II

- If A and B are two redexes in a λ -term M and the first occurrence of λ in A is to the left of the first occurrence of λ in B , then A is to the left of B .
- If A is a redex in M , and it is to the left of all other redexes, then A is the leftmost redex of M .

Theorem (CR-II)

Let X and Y be λ -terms.

If $X \Rightarrow^* Y$ and Y is in normal form, then $X \Rightarrow^* Y$ using only leftmost redexes.

Normal Form, c-b-n, c-b-v

- Choosing always the left-most redex is called **normal order**.
- **normal order** specifies the redex that is chosen next for a given λ -term. **c-b-n** and **c-b-v** specify what to do with a redex of the form $(\lambda x.M)N$.
 - **c-b-n**: use β -reduction (don't "touch" N).
 - **c-b-v**: need a notion of value. Only after " N " is reduced to a value, the β -reduction is performed. What's a value? Let's pick head-normal form.

Question: Reduce the following λ -term using c-b-n and c-b-v

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

Scott refers to **c-b-v** as **applicative-order evaluation**, **c-b-n** is called **normal-order evaluation** of a redex. **lazy evaluation** is a specific combination of both approaches, where arguments are evaluated at most once (memoization).

call-by-name vs. call-by-value

left-most redex, call-by-name:

- guaranteed to reach a normal form if it exists.
- can result in some parameter being evaluated several times or never.

left-most redex, call-by-value:

- efficient — assuming parameter is used at least once.
- can lead to nonterminating computation.

Functional languages typically use call-by-value because of its efficiency.

Programming in lambda calculus

The lambda calculus has very few constructs and it is therefore easy to reason *about it*.

Question: Is the lambda calculus too simple, i.e., can we express all computable functions in the lambda calculus?

Remember: Computation in the lambda calculus is a sequence of applications of reduction rules (mostly β -reductions).

Logical constants and operations:

true $\equiv \lambda a.\lambda b.a$ *select-first*

false $\equiv \lambda a.\lambda b.b$ *select-second*

cond $\equiv \lambda m.\lambda n.\lambda p.((p\ m)n)$

not $\equiv \lambda x.((x\ \text{false})\ \text{true})$

and $\equiv \lambda x.\lambda y.((x\ y)\ \text{false})$

or $\equiv \lambda x.\lambda y. ((x\ \text{true})\ y)$

Programming in lambda calculus

What about data structures?

data structures:

pairs can be represented as

$$[M . N] \equiv \lambda z.((z M) N)$$

$$\mathbf{first} \equiv \lambda x.(x \text{ true}) \quad (\mathit{car})$$

$$\mathbf{second} \equiv \lambda x.(x \text{ false}) \quad (\mathit{cdr})$$

$$\mathbf{build} \equiv \lambda x.\lambda y.\lambda z.((z x) y) \quad (\mathit{cons})$$

What do we need to represent lists?

- *empty list*
- *isEmpty?* function

Programming in lambda calculus

What about arithmetic constants and operations?

There are many options here. Let's look at the system proposed by Church:

$$0 \equiv \lambda fx.x$$

$$1 \equiv \lambda fx.(f x)$$

$$2 \equiv \lambda fx.(f (f x))$$

...

$$n \equiv \lambda fx.\underbrace{(f(f(\dots(f x)\dots))}_{n \text{ times}}) \equiv \lambda fx.(f^n x)$$

The natural number **n** is represented as a function that applies a function *f* *n*-times to its argument *x*.

$$\mathbf{succ} \equiv \lambda m.(\lambda fx.(f (m f x)))$$

$$\mathbf{add} \equiv \lambda mn.(\lambda fx.((m f) (n f x)))$$

$$\mathbf{mult} \equiv \lambda mn.(\lambda fx.((m (n f)) x))$$

$$\mathbf{isZero?} \equiv \lambda m.((m (\text{true false})) \text{true})$$

Note: **pred** function can also be encoded, but the encoding is rather complicated

Programming in lambda calculus

Examples:

$$\begin{aligned}(\text{mult } 2 \ 3) &= \\((\lambda mn.(\lambda fx.((m \ (n \ f)) \ x))) \ 2 \ 3) &= \\ \lambda f_0 x_0.((2 \ (\boxed{3 \ f_0})) \ x_0) &= \\ \lambda f_0 x_0.((2 \ ((\lambda fx.(f \ (f \ (f \ x)))) \ f_0)) \ x_0) &= \\ \lambda f_0 x_0.((2 \ (\lambda x.(f_0 \ (f_0 \ (f_0 \ x)))) \ x_0) &= \\ \lambda f_0 x_0.(\boxed{(2 \ (\lambda x_1.(f_0^3 \ x_1)))} \ x_0) &= \\ \lambda f_0 x_0.((\lambda x.((\lambda x_1.(f_0^3 \ x_1)) \ (\boxed{((\lambda x_1.(f_0^3 \ x_1)) \ x)}))) \ x_0) &= \\ \lambda f_0 x_0.((\lambda x.(\boxed{((\lambda x_1.(f_0^3 \ x_1)) \ (f_0^3 \ x))}) \ x_0) &= \\ \lambda f_0 x_0.(\boxed{((\lambda x.(f_0^3 \ (f_0^3 \ x))) \ x_0)} &= \\ \lambda f_0 x_0.(f_0^3 \ (f_0^3 \ x_0)) &= \\ \lambda fx.(f^6 \ x) &= 6\end{aligned}$$

$$\begin{aligned}(\text{isZero? } 0) &= \\(\boxed{((\lambda fx.x) \ (\lambda y.\text{false}))} \ \text{true}) &= \\((\lambda x.x) \ \text{true}) &= \text{true}\end{aligned}$$

(isZero? n) where $n > 0$?

Recursion in lambda calculus

Does this make sense?

$$\mathbf{f} \equiv \dots \mathbf{f} \dots$$

In lambda calculus, such an equation does not define a term. How to find a λ - term that does “satisfy” the recursive definition?

Example:

$$\mathbf{add} \equiv \lambda mn. \\ (\text{cond } m \ (\mathbf{add} \ (\text{succ } m) \ (\text{pred } n)) \ (\text{isZero? } n))$$

Just to make things easier to read, we will write instead:

$$\mathbf{add} \equiv \lambda mn. \\ \text{if } (\text{isZero? } n) \text{ then } m \text{ else } (\mathbf{add} \ (\text{succ } m) \ (\text{pred } n))$$

This is not a valid definition of a λ - term. What about this one?

$$\text{add} \equiv \lambda \mathbf{f}. (\lambda mn. \\ \text{if } (\text{isZero? } n) \text{ then } m \text{ else } (\mathbf{f} \ (\text{succ } m) \ (\text{pred } n)))$$

Claim: The fixed point of the above function is what we are looking for.

Function fixed points

The fixed points of a function g is the set of values $fix_g = \{x | x = g(x)\}$.

Examples:

function g	fix_g
$\lambda x.6$	$\{6\}$
$\lambda x.(6 - x)$	$\{3\}$
$\lambda x.((x*x) + (x-4))$	$\{-2, 2\}$
$\lambda x.x$	entire domain of f
$\lambda x.(x+1)$	$\{ \}$

Is there a λ -term Y that “computes” a fixed point of a function $F = \lambda f.(\dots f \dots)$, i.e., $YF = F(YF)$?

YES. Y is called the **fixed point combinator**.

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

$$\begin{aligned} YF &= ((\lambda f.((\lambda x.f(x x)) (\lambda x.f(x x)))) F) \\ &= (\lambda x.F(x x)) (\lambda x.F(x x)) \\ &= F((\lambda x.F(x x)) (\lambda x.F(x x))) \\ &= F(YF) \end{aligned}$$

The Y-combinator

Example:

$F \equiv \lambda f.(\lambda mn.$
if (isZero? n) then m else (\mathbf{f} (succ m) (pred n)))

$((YF) 3 2) =$

$((\lambda f.((\lambda x.f(x x)) (\lambda x.f(x x)))) F) 3 2) =$

$(\boxed{((F((\lambda x.F(x x)) (\lambda x.F(x x))))} 3 2) =$

$((\lambda mn.$ if (isZero? n) then m else

$((\lambda x.F(x x)) (\lambda x.F(x x)))$ (succ m) (pred n))) 3 2) =

if (isZero? 2) then 3 else

$((\lambda x.F(x x)) (\lambda x.F(x x)))$ (succ 3) (pred 2)) =

$(\boxed{((\lambda x.F(x x)) (\lambda x.F(x x)))} 4 1) =$

$((F((\lambda x.F(x x)) (\lambda x.F(x x)))) 4 1) =$

if (isZero? 1) then 4 else

$((\lambda x.F(x x)) (\lambda x.F(x x)))$ (succ 4) (pred 1)) =

$(\boxed{((\lambda x.F(x x)) (\lambda x.F(x x)))} 5 0) =$

$((F((\lambda x.F(x x)) (\lambda x.F(x x)))) 5 0) =$

if (isZero? 0) then 5 else

$((\lambda x.F(x x)) (\lambda x.F(x x)))$ (succ 5) (pred 0)) = **5**

The Y-combinator example (cont.)

Note:

- Informally, the Y-combinator allows us to get as many copies of the recursive procedure body as we need. The computation “unrolls” recursive procedure calls one at a time.
- This notion of recursion is purely syntactic.
- Question: Would our Y-combinator work with call-by-value application order?

Call-by-value lambda calculus interpreter

Procedure $eval(P)$:

case P:

“x”: x

“(λ x.M)” : λ x.M

“(M N)” : $e_1 = eval(M)$;
error if not $e_1 \equiv \lambda x.M_1$;
 $e_2 = eval(N)$;
 $e_3 = [e_2/x]M_1$; //capture-free subst.
 $eval(e_3)$

endcase

Note:

- Values are variables or λ -abstractions
- You could think of this interpreter as an operational semantics for our call-by-value application order lambda calculus (defining interpreter)

Lambda calculus — final remarks

- We can express all computable functions in our λ -calculus. However, nobody “programs” in lambda calculus. For that we have more “convenient” functional languages.
- All computable functions can be express by the following two combinators, referred to as **S** and **K**:
 - $K \equiv \lambda xy.x$
 - $S \equiv \lambda xyz.xz(yz)$

Combinatory logic is as powerful as Turing Machines.

Functional Programming

Pure Functional Languages

Fundamental concept: **application** of (mathematical) **functions** to **values**

Referential transparency:

The value of a function application is independent of the context in which it occurs

- value of $f(a, b, c)$ depends only on the values of f , a , b and c
- It does not depend on the global state of computation

⇒ all vars in function must be local, or parameters

Pure Functional Languages

1. The concept of assignment is **not** part of functional programming
 - no explicit assignment statements
 - variables bound to values only through the association of actual parameters to formal parameters in function calls
 - function calls have no side effects
 - thus no need to consider global state
2. Control flow is governed by function calls and conditional expressions
 - ⇒ no iteration
 - ⇒ recursion is widely used

Pure Functional Languages

1. All storage management is implicit
 - needs garbage collection
2. Functions are *First Class Values*
 - Can be returned as the value of an expression
 - Can be passed as an argument
 - Can be put in a data structure as a value
 - (Unnamed) functions exist as values

Pure Functional Languages

A program includes:

1. A set of function definitions
2. An expression to be evaluated

E.g. in Scheme:

```
> (define (length x)
    (if (null? x)
        0
        (+ 1 (length (rest x)))))

> (length '(A LIST OF 5 THINGS))
5
```

LISP

- Functional language developed by John McCarthy in the mid 50's
- Semantics based on *Lambda Calculus*
- All functions operate on lists or symbols: (called "S-expressions")
- Only five basic functions: list functions `cons`, `car`, `cdr`, `equal`, `atom` and one conditional construct: `cond`
- Useful for list-processing applications
- Programs and data have the same syntactic form: S-expressions
- Used in Artificial Intelligence
- SCHEME: Developed in 1975 by G. Sussman and G. Steele as a version of LISP

⇒ we are using SCHEME here

You can call the SCHEME interpreter on the ilab cluster by saying `mzscheme`.

S-expressions

S-expression ::= Atom | '(' { S-expression } ')'

Atom ::= Name | Number | #t | #f

#t

()

(a b c)

(a (b c) d)

((a b c) (d e (f)))

(1 (b) 2)

Lists have nested structure.

Special (Primitive) Functions

- `eq?`: identity on names (atoms)
- `null?`: is list empty?
- `car`: selects first element of list (*contents of address part of register*)
- `cdr`: selects rest of list (*contents of decrement part of register*)
- `(cons element list)`: constructs lists by adding `element` to front of `list`
- `quote` or `'`: produces constants

Special (Primitive) Functions

- '() is the empty list
- (car '(a b c)) =
- (car '((a) b (c d))) =
- (cdr '(a b c)) =
- (cdr '((a) b (c d))) =

Next topic: A small defining interpreter in Scheme

For next time, please practice writing small Scheme programs

Read Scott: Chapter 10

Get familiar with `mzscheme`

Check out resources on the web