

ICS/CSE 141: Programming Languages

More Lambda-Calculus & Scheme

School of Information and Computer Science
University of California, Irvine

Chris Stork

`cstork@ics.uci.edu`

Notes on L^AT_EXing

To typeset λ -term equations

- either use the `verbatim` environment with the syntax as defined in the second lecture
- or use the `align*` environment (math mode)

```
\begin{align*}
& (\lambda f . \lambda x. f (f x)) (\lambda y. y+1) \\\
=& \lambda x . (\lambda y. y+1) ((\lambda y. y+1) x) \\\
=& \lambda x . (\lambda y. y+1) (x+1) \\\
=& \lambda x . (x+1)+1
\end{align*}
```

$$\begin{aligned}
 & (\lambda f. \lambda x. f(fx))(\lambda y. y + 1) \\
 = & \lambda x. (\lambda y. y + 1)((\lambda y. y + 1)x) \\
 = & \lambda x. (\lambda y. y + 1)(x + 1) \\
 = & \lambda x. (x + 1) + 1
 \end{aligned}$$

Overview

1. More Lambda-Calculus & Scheme
2. Garbage Collection

More Lambda-Calculus & Scheme

λ -Calculus Review

λ -terms: $M ::= x \mid MM \mid \lambda x.M$

The *binding operator* λ binds the *variable* x in the λ -term $\lambda x.M$.
 M is called the *scope* of x .

Free variables of M :

$$\text{FV}(x) = \{x\}$$

$$\text{FV}(MN) = \text{FV}(M) \cup \text{FV}(N)$$

$$\text{FV}(\lambda x.M) = \text{FV}(M) - x$$

Free variables can't be renamed!

Expressions in Scheme

Function call (function arg1 ... argn)

- evaluate each of the arguments
- pass list of argument values to function
- **strict** evaluation

Special forms do not eval all arguments

- Example (cond (p1 e1) ... (pn en))
 - proceed from left to right
 - find the first p_i with value true, eval this e_i
 - otherwise *undefined*
 - **lazy** evaluation
- Example (quote a)
 - does not evaluate a

McCarthy's 1960 Paper

Interesting paper with

- good language ideas, succinct presentation
- some feel for historical context
- insight into language design process

Important concepts

- Interest in symbolic computation influenced design
- Use of simple machine model
- Attention to theoretical considerations
 - Recursive function theory, Lambda calculus
- Various good ideas: Programs as data, garbage collection

Declarations

Lots of programming languages allow **local declarations** e.g. in C++ we can declare constants in a block and use them

Simple form for declarations: “**let** $x = M$ **in** N ”

- Inside N each occurrence of x stands for M
- **let** $x = 10$ **in** $x + 1 \Rightarrow 10 + 1$

We can **encode** the **let** construct in λ -calculus

- **let** $x = M$ **in** $N \Leftrightarrow (\lambda x.N)M$
- Compare: $(\lambda x.(x + 1))10 \Rightarrow 10 + 1$

Definitions in Scheme

Demonstrate `let`, `let*`!

```
> (let ((x 1) (y 2)) (+ x y))
3
> (let ((x 1) (y (- 2 x))) (+ x y))
BUG: reference to undefined identifier: x
> (let ((x 100))
    (let ((x 1) (y (- 2 x))) (+ x y)))
-97
> (let ((x 100))
    (let* ((x 1)
           (y (- 2 x)))
          (+ x y)))
2
```

Lookup the exact definitions in R5RS!

Recursive Functions

Scheme's `letrec`, `define`

```
> (letrec ((fac (lambda (n)
                 (if (<= n 1)
                     1
                     (* n (fac (- n 1)))))))
    (fac 6))
```

720

```
> (define fac (lambda (n)
                (if (<= n 1)
                    1
                    (* n (fac (- n 1))))))
```

```
> (fac 6)
```

720

```
> (define (fac n)
    (if (<= n 1)
        1
        (* n (fac (- n 1)))))
```

```
> (fac 6)
```

720

Recursion over Lists

```
> (define (len l)
    (if (null? l)
        0
        (+ 1 (len (cdr l)))))
> (len '(1 2 3))
3
```

Recursion vs. Iteration

Scheme has **no looping/iteration** construct. Use recursion instead.

Efficient due to **tail calls** which don't use up stack space!

A **tail call** is a recursive call whose value is returned as the procedure's value.

(See R5RS for exact definition!)

Currying

A single λ -abstraction defines a function with one “argument.”

What do we do if we need a function with two or more “arguments”?

Idea: We can view a function of two arguments as

- A **basic** construct:

$$f : A \times B \rightarrow C \qquad f(a, b) = c$$

- A **combination** of two **simpler** functions:

$$f : A \rightarrow (B \rightarrow C) \qquad f(a) = g, g(b) = c$$

The “spicier” approach is called **currying**. . . (Haskell Curry)

Currying II

This allows **two** interpretations of $\lambda x.\lambda y.(x + y)$:

- function with **two** arguments a and b returning $a + b$
- function with **one** argument a returning
 - a function with **one** argument b returning $a + b$
- $(\lambda x.(\lambda y.(x + y)))ab = (\lambda y.(a + y))b = (a + b)$

Summary & Examples

- The λ -**calulus** uses currying to build functions of multiple arguments (which explains left-associativity of application)
- In **Scheme** currying is possible but cumbersome
- In **ML** and **Haskell** currying is supported better, and we'll use it more later...

Booleans

What are booleans?

- constants: **true**, **false** (carrier set B)
- operations: **not**: $B \rightarrow B$, **and**: $B \times B \rightarrow B$, ...
- if-then-else: **if** c **then** t **else** e ($c \in B$; t, e any expression)

Modeling **if** as a function:

- depending on value of c , return either t or e^a
- function of three arguments, **if**(c, t, e):
 $\lambda c. \lambda t. \lambda e. cte$

^aRead c as “condition,” t as “then value,” and e as “else value.”

Booleans II

What is our **goal**? How should **if** work?

- **if true then t else e** $\Rightarrow t$
- **if false then t else e** $\Rightarrow e$

How can we do this in our **encoding** for $\lambda c.\lambda t.\lambda e.cte$?

Eventually, c is **applied** to t and e

- If c represents **true**, it should return t (the **first** argument)
- If c represents **false**, it should return e (the **second** argument)

Booleans III

How can we define functions like that?

- The function $\lambda x.\lambda y.x$ returns its **first** argument
 - $(\lambda x.\lambda y.x)ab \Rightarrow (\lambda y.a)b \Rightarrow a$
- The function $\lambda x.\lambda y.y$ returns its **second** argument
 - $(\lambda x.\lambda y.y)ab \Rightarrow (\lambda y.y)b \Rightarrow b$

To make **if** work, we define

- **true** := $\lambda x.\lambda y.x$
- **false** := $\lambda x.\lambda y.y$

Surprise: In λ -calculus, even **true** and **false** are **functions!**

Abstract Machines and Garbage Collection

Semantics

- Natural Language
 - “If the condition of a `while` is `true`, its body is executed once, etc.”
- Operational Semantics
 - Explain constructs in terms of **simpler** ones, e.g. abstract machines, λ -calculus reduction. . .
- Axiomatic Semantics
 - Explain constructs through **predicates**, e.g. pre- and post-conditions
- Denotational Semantics
 - Explain constructs through **functions**, that’s what we look at later. . .

Operational Semantics

Execution Model: Abstract Machine (or Interpreter)

- Language semantics must be defined
 - Too concrete
 - * Programs not portable, tied to specific architecture
 - * Prohibit optimization (e.g., C eval order undefined in expn)
 - Too abstract
 - * Cannot easily **estimate running time, space**
- Lisp: IBM 704, but only certain ideas
 - Address, decrement registers → cells with two parts
 - Garbage collection provides abstract view of memory

Abstract Machine

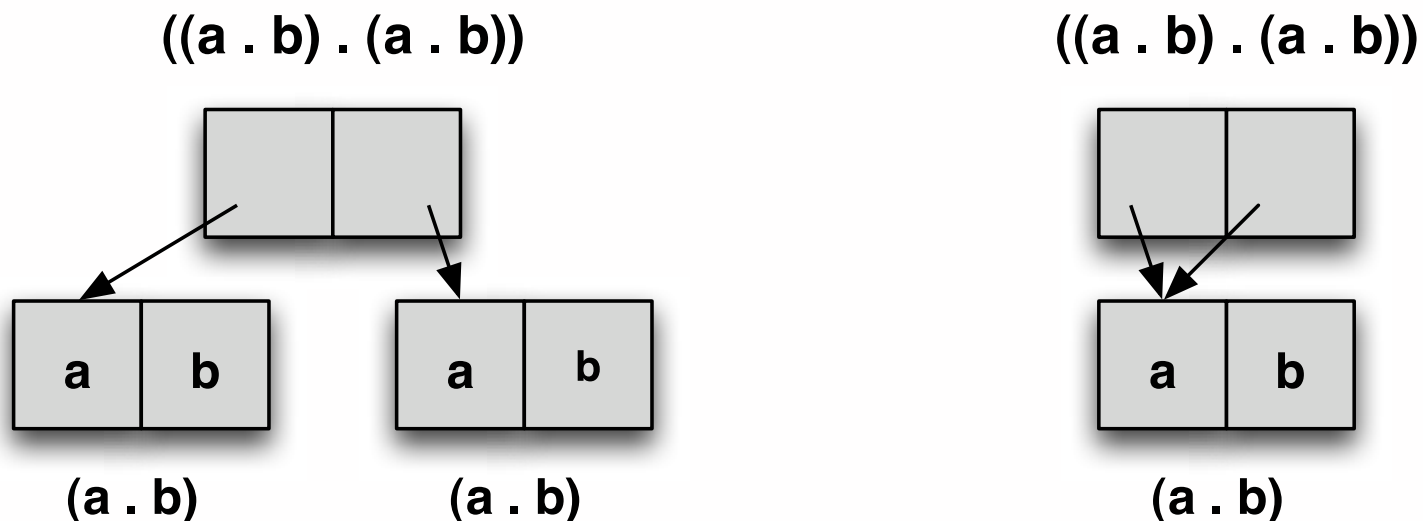
Concept

- Idealized computer, executes programs directly
- Capture programmer's mental image of execution
- Not too concrete, not too abstract

Examples

- **Fortran**
 - Flat register machine; memory arranged as linear array
 - No stacks, no recursion.
- **Algol** family
 - Stack machine, contour model of scope, heap storage
- **Smalltalk**
 - Objects, communicating by messages.

Sharing



- Both structures are printed as `((a . b) a . b)`
- Which of the two above is result of evaluating `(cons (cons 'a 'b) (cons 'a 'b))` or `((lambda (x) (cons x x)) (cons 'a 'b))` ?

Append

```
> (define (app x y)
      (if (null? x)
          y
          (cons (car x) (app (cdr x) y))))
> (app '(1 2) '(3 4))
(1 2 3 4)
```

Exploit: $(xX)Y = x(XY)$

As many allocated cons cells as elements in x .

Make sure you get this!

Interlude: Garbage Collection

Garbage

At a given point in the execution of a program P , a memory location m is garbage if no continued execution of P from this point can access location m .

Garbage Collection

- Detect garbage during program execution
- GC invoked when more memory is needed
- Decision made by run-time system, not program

This is can be very convenient. Example: in building text-formatting program, $\approx 40\%$ of programmer time on memory management ($\approx 5\%$ run-time increase)

Mark-and-Sweep Algorithm

Assume **tag bits** associated with data

Need list of heap locations named by program (**root set**)

Algorithm:

- Set all tag bits to 0
- Start from each location in root set. Follow all links, changing tag bit to 1
- Free all cells with tag = 0

Other GC Algorithms

- Reference Counting
 - Count pointers to a specific location (cons cell, object, . . .)
 - Free if reference count down to 0
 - Problem: Cyclic data structures
- Stop-and-Copy
 - Copy instead of mark (Drawing!)
- Many optimizations, e.g., generational garbage collection

Why Garbage Collection in Lisp?

McCarthy's paper says that this is "more convenient for the programmer than a system in which he has to keep track of and erase unwanted lists."

Does this reasoning apply equally well to C?

Is garbage collection "more appropriate" for Lisp than C? Why?

Thank You!