Functional Programming with Scheme

Read: Scott, Chapter 11.1-11.3

Lecture Outline

- Functional programming languages
- Scheme
  - S-expressions and lists
    - cons, car, cdr
  - Defining functions
  - Examples of recursive functions
    - Shallow vs. deep recursion
  - Equality testing

Racket/PLT Scheme/DrScheme

- Download Racket
  (was PLT Scheme (was DrScheme))
  - http://racket-lang.org/
  - Run DrRacket
  - Languages => Choose Language => Other Languages => Legacy Languages: R5RS
- One additional textbook/tutorial:
  - Teach Yourself Scheme in Fixnum Days by Dorai Sitaram:

  https://ds26gte.github.io/tyscheme/index.html

First, Imperative Languages

- The concept of assignment is central
  - `X:=5; Y:=10; Z:=X+Y; W:=f(Z);`
  - Side effects on memory
- Program semantics (i.e., how the program works): state-transition semantics
  - A program is a sequence of assignment statements with effect on memory (i.e., state)

```
C := 0;
for I := 1 step 1 until N do
  t := a[I]*b[I];
  C := C + t;
```

Imperative Languages

- Functions (also called "procedures", "subroutines", or routines) have side effects:

  Roughly:
  - A function call affects visible state; i.e., a function call may change state in a way that affects execution of other functions; in general, function call cannot be replaced by result
  - Also, result of a function call depends on visible state; i.e., function call is not independent of the context of the call

Imperative Languages

- Functions are, traditionally, not first-class values
  - A first-class value is one that can be passed as argument to functions, and returned as result from functions
    - In a language with assignments, it can be assigned into a variable or structure
  - Are functions in C first-class values?
  - As languages become more multi-paradigm, imperative languages increasingly support functions as first-class values (JS, R, Python, Java 8, C++11)

Functional Languages

- Program semantics: reduction semantics
  - A program is a set of function definitions and their application to arguments

Function composition

Def IP = (Insert +) ° (ApplyToAll *) ° Transpose

IP <<1,2,3>,<6,5,4>> is

(Insert +) ((ApplyToAll *) (Transpose <<1,2,3>,<6,5,4>>))

(Insert +) ((ApplyToAll *) <<1,6>,<2,5>,<3,4>>)

(Insert +) <6,10,12>

28

Functional Languages

- In pure functional languages, there is no notion of assignment, no notion of state
  - Variables are bound to values <u>only</u> through parameter associations
  - No side effects!
- Referential transparency
  - Roughly:
    - Result of function application is independent of context where the function application occurs; function application can be replaced by result

Functional Languages

- Functions are first-class values
  - Can be returned as value of a function application
  - Can be passed as an argument
  - In a language with assignment, can be assigned into variables and structures

  - Unnamed functions exist as values

Lecture Outline

- Functional programming languages
- Scheme
  - S-expressions and lists
    - cons, car, cdr
  - Defining functions
  - Examples of recursive functions
    - Shallow vs. deep recursion
  - Equality testing

Lisp and Scheme

- Lisp is the second oldest high-level programming language!
  - Simple syntax
  - Program code and data have same syntactic form
    - The S-expression
  - Function application written in prefix form

    (e1 e2 e3 … ek) means
    - Evaluate e1 to a function value
    - Evaluate each of e2,…,ek to values
    - Apply the function to these values

    (+ 1 3) evaluates to 4

History

Lisp
1950's
John McCarthy

Scheme
1975
Guy Steele
Gerald Sussman

Common Lisp

dynamic scoping

lexical scoping
functions as first-class values

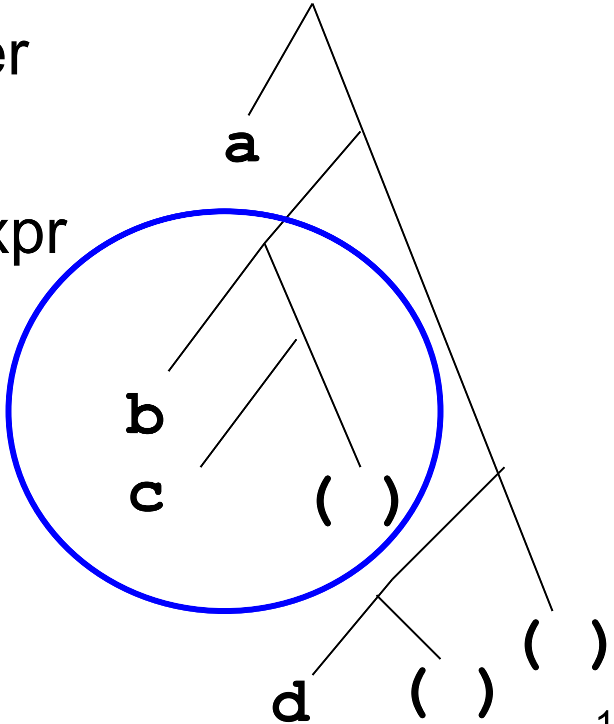
Why Scheme?

- Simple syntax! Great to introduce core functional programming concepts
  - Reduction semantics
  - Lists and recursion
  - <u>Higher order functions</u>
  - Evaluation order
  - Parametric polymorphism
- Later we'll see Haskell and new concepts
  - Algebraic data types and pattern matching
  - Lazy evaluation
  - Type inference

S-expressions

S-expr ::= Name | Number | ({ S-expr })

- Name is a symbolic constant (a string of chars which starts off with anything that can't start a Number)
- Number is an integer or real number
- List of zero or more S-expr's
- E.g., `(a (b c) (d))` is a list S-expr



List Functions

- `car` and `cdr`
  - Given a list, they decompose it into first element, rest-of-list portions
  - E.g., `car` of `(a (b c) (d))` is `a`
  - E.g., `cdr` of `(a (b c) (d))` is `((b c)(d))`
- `cons`
  - Given an element and a list, `cons` builds a new list with the element as its `car` and the list as its `cdr`
  - `cons` of `a` and `(b)` is `(a b)`
- `()` is the empty list

Quoting

- **`** or **quote** prevents the Scheme interpreter from evaluating the argument

(quote (+ 3 4)) yields **(+ 3 4)**

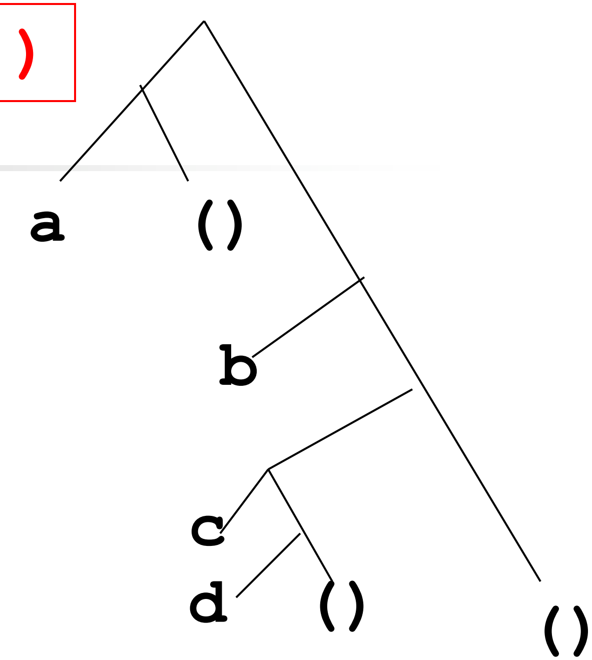
`(+ 3 4) yields **(+ 3 4)**

Whereas **(+ 3 4)** yields **7**

- Why do we need quote?

Questions

`((a) b (c d))`



`(car '(a b c))` **yields** ?

`(car '((a) b (c d)))` **yields** ?

`(cdr '(a b c))` **yields** ?

`(cdr '((a) b (c d)))` **yields** ?

Can compose these operators in a short-hand manner. Can reach arbitrary list element by composition of **`car`**'s and **`cdr`**'s.

`(car (cdr (cdr '((a) b (c d)) )))`

can also be written

`(caddr '((a) b (c d)) )`

`(car (cdr (cdr '((a) b (c d)) ))) =`

`(car (cdr '( b (c d))) = (car '((c d))) = (c d)`

Questions

- Recall **`cons`**
  - E.g., **`(cons 'a '(b c))`** **yields** **`(a b c)`**

`(cons 'd '(e))` **yields** ?

`(cons '(a b) '(c d))` **yields** ?

`(cons '(a b c) '((a) b (c d)))` **yields** ?

Type Predicates

- Note the quote: it prevents evaluation of the argument

`(symbol? `sam)` yields `#t`  `(symbol? 1)` yields `#f`

`(number? `sam)` yields `#f`  `(number? 1)` yields `#t`

`(list? `(a b))` yields `#t`  `(list? `a)` yields `#f`

`(null? `())` yields `#t`  `(null? `(a b))` yields `#f`

`(zero? 0)` yields `#t` `(zero? 1)` yields `#f`

Can compose these.

`(zero? (- 3 3))` yields `#t` Note that since this language is fully parenthesized, there are no precedence problems in expressions!

Question

- What is the typing discipline in Scheme?
  - Static or dynamic?
  - Answer: Dynamic typing. Variables are bound to values of different types at runtime. All type checking done at runtime.

Lecture Outline

- Functional Programming Languages
- Scheme
  - S-expressions and lists
    - cons, car, cdr
  - Defining functions
  - Examples of recursive functions
    - Shallow vs. deep recursion
  - Equality testing

Scheme: Defining Funcitons

Fcn-def ::= (define (Fcn-name {Param}) S-expr)

Fcn-name should be a new name for a function.

Param should be variable(s) that appear in the S-expr which is the function body.

Fcn-def ::= (define Fcn-name Fcn-value)

Fcn-value ::= (lambda ({Param}) S-expr)

where Param variables are expected to appear in the S-expr; called a lambda expression.

Examples

```
(define (zerocheck? x)
  (if (= x 0) #t #f) )
```

If-expr ::= (if S-expr0 S-expr1 S-expr2)
where S-expr0 must evaluate to a boolean value; if that value is **#t**, then the If-expr yields the result of S-expr1, otherwise it yields the result of S-expr2.

(zerocheck? 1) yields #f,
(zerocheck? (* 1 0)) yields #t

Examples

```
(define (atom? object)
  (not  (pair? object)) )
```

Here **`pair?`** is a built-in type predicate. It yields **`#t`** if the argument is a non-trivial S-expr (i.e., something one can take the **`cdr`** of). It yields **`#f`** otherwise.

`not` is the built-in logical operator.

What does **`atom?`** do?

Examples

```
(define square (lambda (n) (*  n  n)))
```

- Associates the Fcn-name square with the function value `(lambda (n)  (*  n  n))`
- Lambda calculus is a formal theory of functions
  - Set of functions definable using lambda calculus (Church 1941) is same as set of functions computable as Turing Machines (Turing 1930's)

Trace of Evaluation

```
(define (atom? object)
  (not  (pair? object)) )
```

`(atom? `(a))`

-obtain function value corresponding to **`atom?`**

-evaluate **`` `(a) ``** obtaining **`(a)`**

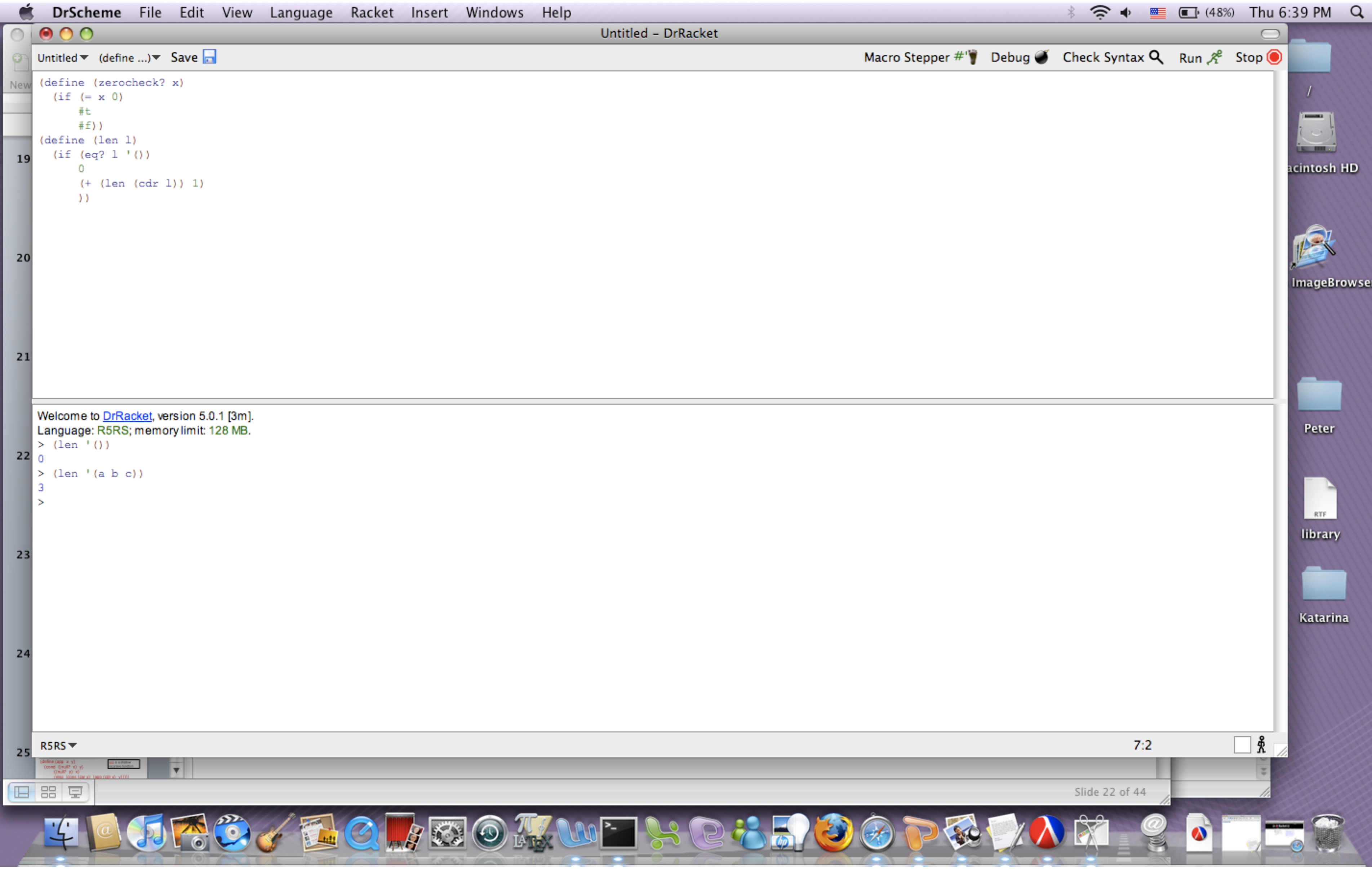
-evaluate **`(not (pair? '(a)))`**

- -obtain function value corresponding to **`not`**
- -evaluate **`(pair? '(a))`**
    - -obtain function value corresponding to **`pair?`**
    - -evaluate **`'(a)`** obtaining **`(a)`**
    - -return value **`#t`**
- -return **`#f`**

-return **`#f`**

Read-Eval-Print Loop (REPL)

- Scheme interpreter runs read-eval-print loop
  - Read input from user
    - A function application
  - Evaluate input
    - (e1 e2 e3 … ek)
      - Evaluate e1 to obtain a function
      - Evaluate e2, … , ek to values
      - Execute function body using values from previous step as parameter values
      - Return value
  - Print return value


DrScheme File Edit View Language Racket Insert Windows Help
Untitled - DrRacket
Untitled (define ...) Save
Macro Stepper Debug Check Syntax Run Stop
(define (zerocheck? x)
  (if (= x 0)
      #t
      #f))
(define (len l)
  (if (eq? l '())
      0
      (+ (len (cdr l)) 1)
      ))
Welcome to DrRacket, version 5.0.1 [3m].
Language: R5RS; memory limit: 128 MB.
> (len '())
0
> (len '(a b c))
3
>
R5RS
7:2
Slide 22 of 44
Peter
library
Katarina
ImageBrowser

Conditional Execution

```
(if  e1  e2  e3)
(cond  (e1  h1) (e2  h2) … (en-1  hn-1)
  (else hn))
```

- Cond is like **if – then – else if** construct

```
(define (zerocheck? x)
     (cond ((=  x  0)  #t)  (else  #f)))
```

OR

```
(define (zchk?  x)
  (cond ((number? x) (zero? x))
          (else #f)))
```

Recursive Functions

```
(define (len x)
  (cond ((null?  x)  0)  (else  (+ 1 (len (cdr  x))))))
```

`len` is a shallow recursive function

`(len `(1 2))` should yield 2.

Trace: `(len `(1 2))` **-- top level call**

`x = (1  2)`

`(len `(2))` **-- recursive call 1**

`x = (2)`

`(len `())` -- **recursive call 2**

`x = ()`

returns 0 **-- return for call 2**

returns `(+  1  0) = 1` **--return for call 1**

returns `(+ 1  1) = 2` **-- return for top level call**

`(len '((a) b (c  d)))` **yields** what?

Recursive Functions

```
(define (app  x  y)
  (cond ((null?  x)  y)
        ((null?  y)  x)
        (else
          (cons (car x)
                (app (cdr x)  y)))))
```

`app` is a shallow recursive function

- What does **app** do?

(app ‘() ‘()) yields ?
(app ‘() ‘(1 4 5)) yields ?
(app ‘(5 9) ‘(a (4) 6)) yields ?

Exercise

```
(define (len x)
  (cond ((null?  x)  0)  (else  (+ 1 (len (cdr  x))))))
```

Write a version of **`len`** that uses **`if`** instead of **`cond`**

Write a function **`countlists`** that counts the number of list elements in a list. E.g.,

`(countlists '(a))` yields **`0`**

`(countlists '(a (b c (d)) (e)))` yields **`2`**

Recall **`(list? l)`** returns true if **`l`** is a list, false otherwise

Recursive Functions

```
(define  (fun  x)
  (cond ((null? x)  0)
        ((atom? x)  1)
        (else  (+ (fun  (car x))
                  (fun  (cdr x)))) ))
```

`fun` is a deep recursive function

What does `fun` do?

`fun` counts atoms in a list

```
(define (atomcount x)
   (cond ((null? x) 0)
         ((atom? x) 1)
         (else (+ (atomcount (car x)) (atomcount (cdr x)))) ))
```

atomcount is a deep recursive function

(atomcount ‘(a)) **yields 1**

(atomcount ‘(1 (2 (3)) (5))) **yields 4**

Trace: (atomcount ‘(1 (2 (3)))

1> (+ (atomcount 1) (atomcount ‘((2 (3)))))

2> (+ (atomcount ‘(2 (3))) (atomcount ‘( )))

3> (+ (atomcount 2) (atomcount ‘((3)))

4> (+ (atomcount ‘(3)) (atomcount ‘( )))

5> (+ (atomcount 3) (atomcount ‘( )))

1 **0**

Exercise

- Write a function **`flatten`** that flattens a list

`(flatten '(1 (2 (3))))` **yields** **`(1 2 3)`**

Lecture Outline

- Functional Programming Languages
- Scheme
  - S-expressions and lists
    - cons, car, cdr
  - Defining functions
  - Examples of recursive functions
    - Shallow vs. deep recursion
  - Equality testing

Equality Testing

eq?

- Built-in predicate that can check atoms for equal values
- Does not work on lists in the way you might expect!

eql?

- Our predicate that works on lists

```
(define (eql?  x  y)
  (or (and (atom?  x) (atom? y) (eq?  x  y))
      (and  (not (atom? x)) (not (atom? y))
            (eql?  (car x)  (car y))
            (eql?  (cdr x)  (cdr y)) )))
```

equal?

- Built-in predicate that works on lists

Examples

`(eql? '(a) '(a))` **yields** what?

`(eql? 'a 'b)` **yields** what?

`(eql? 'b 'b)` **yields** what?

`(eql? '((a)) '(a))` **yields** what?

`(eq? 'a 'a)` **yields** what?

`(eq? '(a) '(a))` **yields** what?

Models for Variables

- Value model for variables
  - A variable is a location that holds a value
    - I.e., a named container for a value
  - a := b

l-value (the location)

r-value (the value held in that location)

- Reference model for variables
  - A variable is a reference to a value
  - Every variable is an l-value
    - Requires dereference when r-value needed (usually, but not always implicit)

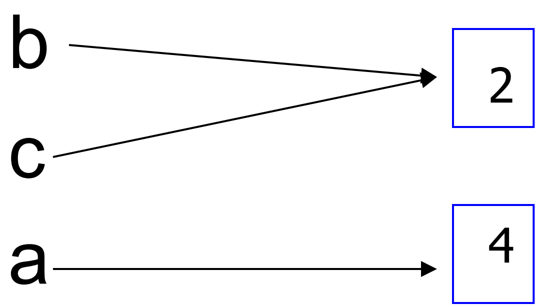
Models for Variables: Example

b := 2;

c := b;

a := b + c;

- Value model for variables
  - b := 2   b: $\boxed{2}$
  - c := b   c: $\boxed{2}$
  - a := b+c   a: $\boxed{4}$
- Reference model for variables
  - b := 2
  - c := b
  - a := b+c



Equality Testing: How does eq? work?

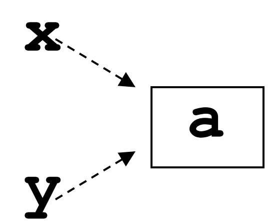
- Scheme uses the reference model for variables!

```
(define (f  x  y)  (list x  y))
```

Call (f ‘a ‘a) yields (a a)

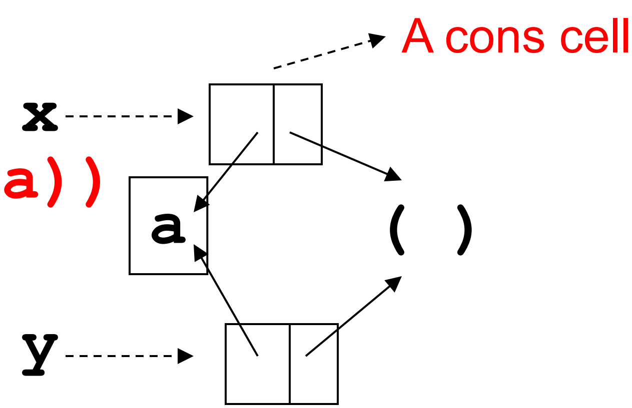
x refers to atom a and y refers to atom a.

eq? checks that x and y both point to the same place.



Call (f ‘(a) ‘(a)) yields (‘(a) ‘(a))

x and y do not refer to the same list.



Models for Variables

- C/C++, Pascal, Fortran
  - Value model
- Java
  - Mixed model: value model for simple types, reference model for class types
- JS, Python, R, etc.
  - Reference model
- Scheme
  - Reference model! `eq?` is "reference equality" (akin of Java's ==), `equal?` is value equality

The End