Principles of Programming Languages

Topic: Functional Programming B

Professor Lou Steinberg

Spring 2015

Review: Scheme

Scheme is a *dynamic* language

- **function definition is an <u>executable</u> operation**

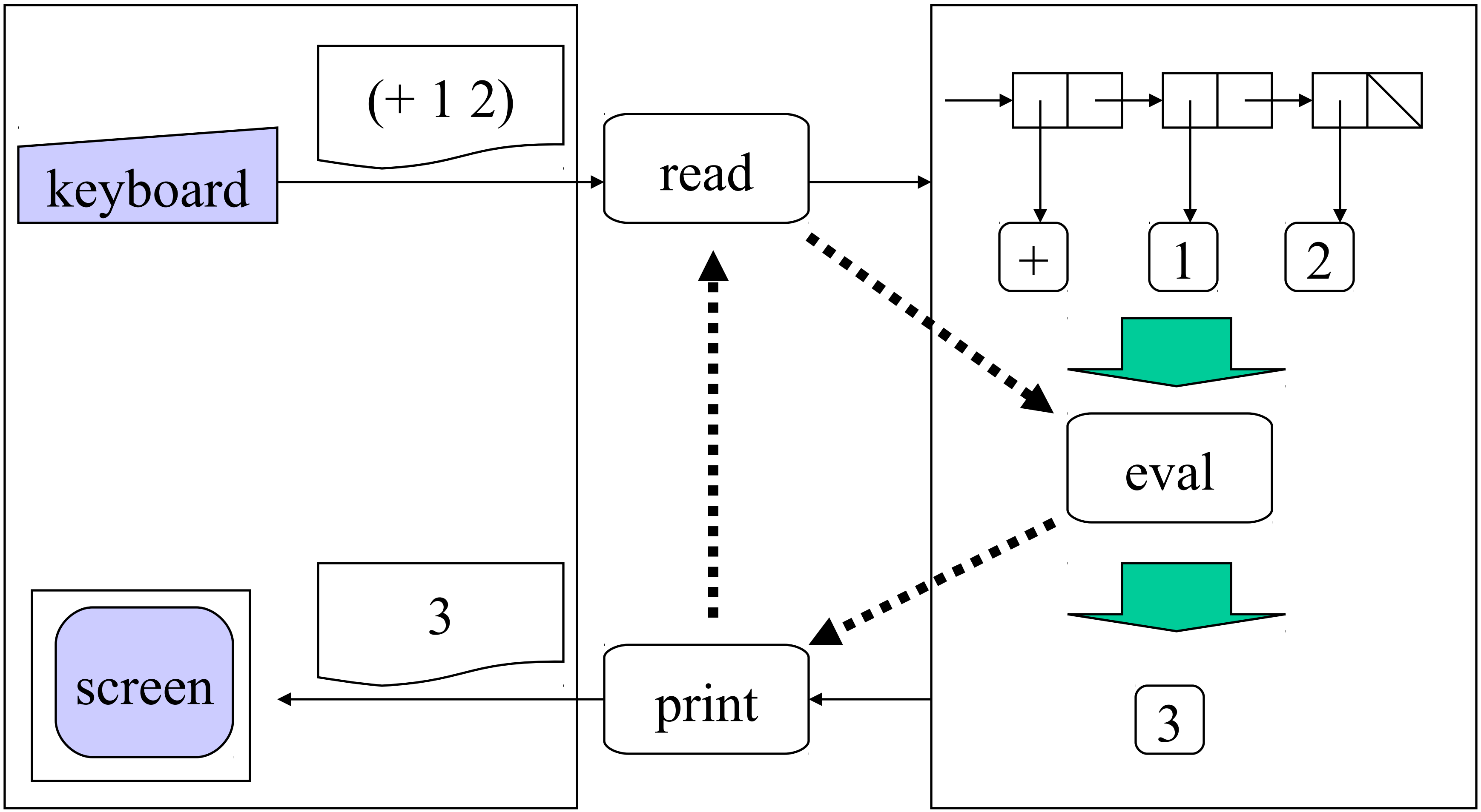
Scheme is an *interactive* language

- **"Read-eval-print loop"**

Scheme is a *functional* language

- **A program is an expression to be evaluated**
- **Functions are data like any other data**

Read-Eval-Print Loop



Review: Expressions

A program is an expression to be evaluated

An expression is:

- **A literal constant: 3, 3.1416, “hello”**
- **A variable that has been bound to some value: x, ?a, +**
- **A function application: (+ x 1)**
- **A special form: (lambda (x) (+ x 1))**

A function application is written as a list: (+ 3 5)

- **Evaluate first element of this list → function to apply: addition**
- **Evaluate rest of elements of the list → arguments to apply the function to: 3 and 5**

Function Application

- **E.g., to evaluate (+ 5 (- 7 3)):**

  + => [machine code for addition] **+ is predefined**

  5 => 5 **literals eval to themselves**

  (- 7 3) => 4 **evaluated recursively**

  apply [machine code for addition] to 5 and 4

  => 9

Special Forms

- **Exceptions to function application rules**
- **Special symbols as car of list**
  - **quote, if, lambda, a few others**
  - **Evaluation rules depend on car of list**
- **quote: don't evaluate, just return arg**

  (quote a) => a

  (quote (+ 3 4)) => (+ 3 4) (a list, not a number)
  - **quote is extra special:**

    'a is read in as if it were (quote a)

    '(+ 3 4) is read in as if it were (quote (+ 3 4))

Lambda

- **A lambda expression creates and returns a new function**

(lambda (x) (+ x x))

list of parameters **expression**

You can use this expression-whose-value-is-a function just like you can use a variable-whose-value-is-a-function:

((lambda (x) (+ x x)) 4)

function argument

Lambda

- E.g. evaluate ((lambda (x y)(- (+ x y) 1)) 3 5)
  - Evaluate (lambda (x y)(- (+ x y) 1) => [function: ...]
  - Evaluate 3 => 3 and 5 => 5
  - Apply [function: …] to 3 and 5
    - Create binding context with x bound to 3 and y to 5
    - In this context evaluate (- (+ x y) 1) => 7
  - Return 7

Top-Level Definitions

- **(define b (+ 3 5)) assigns 8 to variable b**
- **(define double (lambda (x) (+ x x)) is similar**
- **(define (double x)(+ x x)) is shorthand for the line above**

Booleans and if

- #f represents false, #t represents true
  - in fact, everything besides #f represents true
- (if (even? n) (/ n 2) (/ (- n 1) 2))
  - Evaluate (even? n)
  - if result is true, evaluate (/ n 2) and return the result
  - otherwise evaluate (/ (- n 1) 2) and return the result
- (if (zero? x) 1 (/ 1 x))

cond

- ```
  (cond ( <bool 1>      <expr 1> )
          …
        ( <bool n-1>  <expr n-1> )
        ( else          <expr n>))
  ```
- **Evaluate bools until one returns true, eval & return corresponding expr, otherwise return value of <expr n>**

```
(cond ((null? lst)         'zilch)
      ((null? (cdr lst)) 'one)
      (else               (car '(many lots)))))
```
```

not

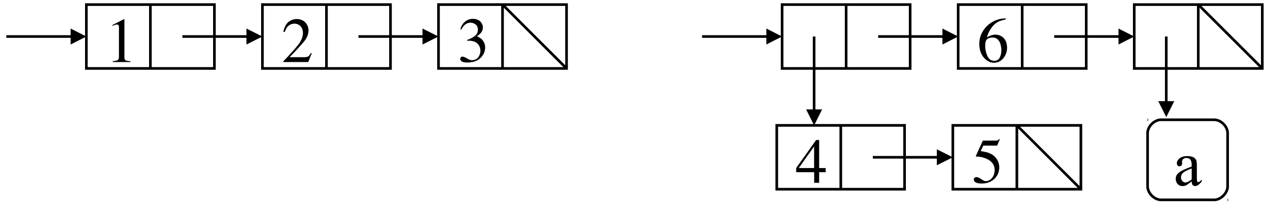
- `(not #f) => #t, (not #t) => #f`
  - `(not ...)` **is not a special form**

or, and

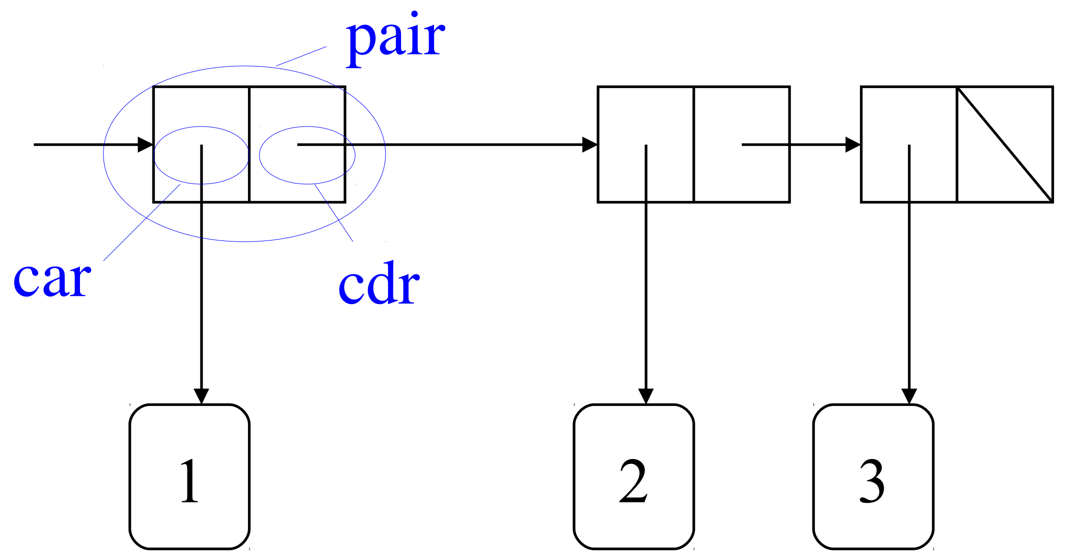
- `(and <expr1> … <exprn>)`
  - Evaluate exprs 1 thru n until one returns #f
  - If an expr returns #f, so does `and`
  - otherwise, `and` returns value of <exprn>
  - (and …) is a special form
- `(or <expr1> … <exprn>)`
  - Evaluate exprs 1 thru n until one returns true
  - If an expr returns a true value, then `or` returns that value
  - Otherwise `or` returns false
  - (or …) is a special form

Lists

- **Scheme data types:**
  - **Lists**
  - **Symbols**
  - **Numbers**
  - **Etc.**
- **External representation: (1 2 3) or ((4 5) 6 a)**
  - **elements, separated by whitespace, surrounded by ( )**
- **Internal representation: singly-linked list**



Scheme: Lists



- (car '(1 2 3)) => 1
- (cdr '(1 2 3)) => (2 3)
- (cons '1 '(2 3)) => (1 2 3)

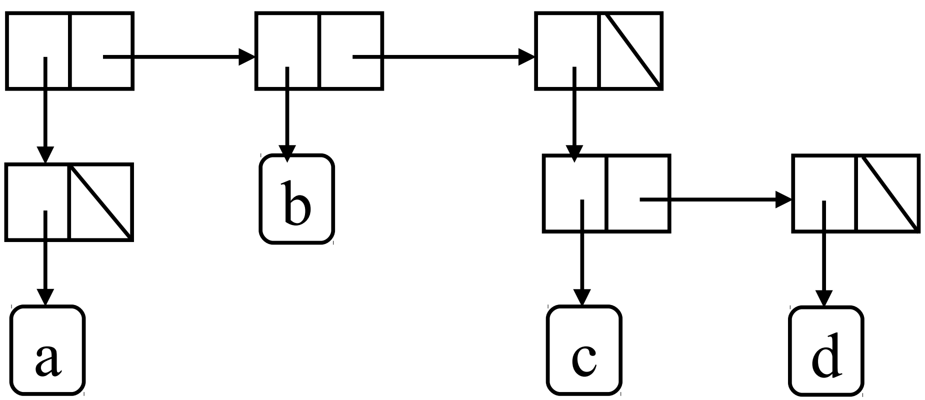
Lists

Examples:

(car ‘((a) b (c d))) => (a)

(car (car ‘((a) b (c d)))) => a

(car (car (car ‘((a) b (c d))))) => *error*



((a) b (c d))

C (a|d)+ r

- (cadr x) means (car (cdr x))

  (cadr '(a b c)) => b
- (cdadr x) means (cdr (car (cdr x)))

  (cdadr '(a (b c d) e)) => (c d)
- (cadadr x) means (car (cdr (car (cdr x))))
  - But if you use it you are probably doing something wrong
- (cdadadr x) is not defined

Lists

Examples:

(cdr ‘((a) b (c d))) => (b (c d))

(cadr ‘((a) b (c d))) => b

(cddr ‘((a) b (c d))) => ((c d))

(cdddr ‘((a) b (c d))) => ( )

(cddddr ‘((a) b (c d))) => *error*

Lists

Examples:

(cons ‘(a b c) ‘((a) b (c d)))   => ((a b c) (a) b (c d))

(cons ‘d ‘(e))   => (d e)

(cons ‘(a b) ‘(c d))   => ((a b) c d)

(cons ‘a (cons ‘b (cons 'c '( )))) => (a b c)

Dynamic typing

What is the type of the car field of a pair?

- **variables and fields don't have types, only values have types**
- **A value is represented by a data structure with**
  - **a type code and**
  - **a value**
- **Depending on the type, the value is either**
  - **immediate data (e.g. integer)**
  - **a pointer to the actual data in the heap (e.g. a pair)**

Symbols

- Scheme data types:
  - Lists
  - Symbols
  - Numbers
  - Etc.
- x, a, and cdr are symbols
- so are 3ab, +, and a-b
- A symbol can be part of a list ((a) b ((c) d)))
- To evaluate a symbol, look up its binding as a variable

Symbols

- The symbol horse is the same symbol wherever it appears in a program
  - (eq? 'horse 'horse) => #t
  - The read function uses a hash table to keep track of the symbols it has already created

Pure Functional Programming

No side effects

- No assignments
  - Variables get values via parameter binding
  - Assignment vs binding
- No iteration
  - Flow of control via if and recursion
- No explicit free
  - Reclaim storage via garbage collection
- Functions are a first class data type
  - Store in data structures, use as function arguments and values

Pure Functional Programming

- **“Look, Ma, No Hands!”**
  - **No assignment statements!**
  - **No iteration!**
- **How is it possible to write programs in a language like this?**
  - **Parameter binding**
  - **Recursion**

Binding Local Variables

How to achieve:

```
x := a
y := b
```

<expression involving values of `x` and `y`>

A solution using lambda expressions:

```
((lambda (x y)
      <expression involving x and y >)
  a
  b)
```

let

```
(let ((a 3)
      (b (+ 4 1)))
  (* a b))
```

eqivalent to:

```
 ((lambda (a b)(* a b))
   3
   (+ 4 1))
```

=> 15

let

- ```
  (let ((<var1> <val1>)
          …
        (<varn> <valn>))
     <expr>)
  ```
- ```
  (let ((a 3)
        (b 4))
     (* a b)) => 12
  ```

let

```
(define (quad a b c)
   (let ((discr (- (* b b)(* 4 a c)))
         (twoa (* 2 a)))
      (list (/ (- (- b)(sqrt discr))
               twoa)
            (/ (+ (- b)(sqrt discr))
               twoa))))
```

See quad.scm in Resources > Scheme

let and let*

```
(let ((f (lambda (x) (+ x x)))
      (y 3))
   (f y)   )
=> 6
```

Scope of f

```
(let ((f (lambda (x) (+ x x)))
      (y (f 4)))
   (f y))
=> **error**
```

reference to undefined identifier: f

let and let*

```
(let* ((f (lambda (x) (+ x x))
       (g (lambda (x) (* 2 (f x)))))
  (g 3))
⇒12
```

Scope of f

letrec

```
(letrec ((f (lambda (x)
              (if (null? x) 0
                     (+ 1 (f (cdr x))))) ))
   (f '(a b c d)))
=> 4
```

Scope of f

Recursion in place of iteration

```
(define (count-down n)
  (if (<= n 0)
      (display "boom")
      (begin
        (display n)
        (display #\newline)
        (count-down (- n 1)))))
```

```
public static void
        count-down(int n){
  while (n > 0){
    System.out.println(n);
    n  = n-1;
  }
  System.out.println(“boom”);
```

Recursion in place of iteration

- **Note how repeated assignment to one iterative variable become a single binding to each of many (recursive) variables**

Lists

- A list is
  - the empty list or
  - a car which is anything and
    a cdr which is a list
- Note recursion in definition

Recursive functions

- **Often the structure of a recursive function on a list parallels recursive structure of list definition**

```
(define (foo lst)
   (if (null? lst)  base-case
     (fn (car lst) (foo (cdr lst)) ) ))


(define (sum lst)
   (if (null? lst) 0
      (+ (car lst)(sum (cdr lst)) ) ))
```

Recursive Functions

```
(define (sum lst)
  (if (null? lst) 0
     (+ (car lst)(sum (cdr lst)) ) ))
```

```
lst = (4 2 5)
lst =   (2 5)
lst =     (5)
lst =     ( )
```

Recursive Functions

```
(define (sum lst)
   (if (null? lst) 0
      (+ (car lst)(sum (cdr lst)) ) ))
```

lst = (4 2 5)   => (+ 4 7) => 11

lst =   (2 5)   => (+ 2 5) => 7

lst =   (5)   => (+ 5 0) => 5

lst =   ( ) => 0

Recursive Functions

Another example of the same pattern

```
(define (count2s lst)
  (cond ( (null? lst) 0)
            ( (eq? (car lst) 2)
              (+ 1 (count2s (cdr lst))))
            (else  (count2s (cdr lst)))))
```

Recursive Functions

- Really?

```
(define (count2s lst)
  (cond ( (null? lst) 0)



        ( (eq? (car lst) 2)
          (+ 1 (count2s
                 (cdr lst))) )
        (  else (count2s
                 (cdr lst)))  ))
```

- Yes

```
(define (count2sC lst)
  (if (null? lst) 0
     ((lambda
        (first recursive-result)
         (if (eq? first 2)
            (+ 1 recursive-result )
           recursive-result  ))
      (car lst)
      (count2sC (cdr lst)))))
```

Recursive Functions

```
(define (count2s lst)
  (cond ( (null? lst) 0)
           ( (eq? (car lst) 2)
              (+ 1 (count2s (cdr lst))))
          (  else (count2s (cdr lst)))))
```

```
lst = (1 2 3)       =>  1
lst =    (2 3)      => (+ 1 0) => 1
lst =       (3)    =>  0
lst =        ( )  =>  0
```

Recursive Functions

```
(define (length lst)
  (if (null? lst) 0
    (+ 1 (length (cdr list)))))
```

Recursive Functions

```
(define (incr-all lst)
   (if (null? lst) '( )
       (cons (+ (car lst) 1)
                 (incr-all (cdr lst)))))

(define (remove0 lst)
  (if (null? lst) '( )
    (if (= (car lst) 0)  (remove0 (cdr lst))
       (cons (car lst) (remove0 (cdr lst))))))
```

Built in functions

- **append**
- **eq?, equal?**
- **reverse**
- **member**
- **assoc**

Assoc

- **Assoc-list is a data structure**
  - **Stores an association symbol -> data**
  - **((s1 d1) (s2 d2) …)**
  - **(assoc ‘a ‘((b 3) (a horse) (pi 3.14))) =>(a horse)**

```
(define (assoc key a-list)
   (cond ((null? a-list) #f)
            ((eq? key (caar a-list)) (car a-list))
            ( else (assoc key (cdr a-list)))))
```

Finite State Machine Simulator

- **See scheme > dfa-simulator.scm**