CS 314 Principles of Programming Languages

Lecture 16

Zheng Zhang

Department of Computer Science
Rutgers University

Friday 28th October, 2016
Reminder: midterm exam this coming Wednesday class (11/2).

Homework 6 posted, due Saturday 11/5 11:55pm.
Special (Primitive) Functions

- eq?: identity on names (atoms)
- null?: is list empty?
- car: selects first element of list \((\text{contents of address part of register})\)
- cdr: selects rest of list \((\text{contents of decrement part of register})\)
- (cons element list): constructs lists by adding element to front of list
- quote or ’: produces constants
Other Functions

- +, - *, / numeric operators, e.g.,
  \[(+ 5 3) = 8, (- 5 3) = 2\]
  \[(* 5 3) = 15, (/ 5 3) = 1.6666666\]
- =, <, > comparison operators for numbers
- Explicit type determination and test functions:
  \[⇒\] All return Boolean values: #f and #t
  - (number? 5) evaluates to #t
  - (zero? 0) evaluates to #t
  - (symbol? 'sam) evaluates to #t
  - (list? '(a b)) evaluates to #t
  - (null? '()) evaluates to #t

Note: SCHEME is a strongly typed language.
Other Functions

- `(number? 'sam)` evaluates to `#f`
- `(null? '(a))` evaluates to `#f`
- `(zero? (- 3 3))` evaluates to `#t`
- `(zero? '(- 3 3))` ⇒ type error
- `(list? (+ 3 4))` evaluates to `#f`
- `(list? '(+ 3 4))` evaluates to `#t`
READ-EVAL-PRINT Loop

The Scheme interpreters on the ilab machines are called mzscheme, racket, and dracket. “drracket” is an interactive environment, the others are command-line based. For example: Type mzscheme, and you are in the READ-EVAL-PRINT loop. Use Control D to exit the interpreter.

**READ:** Read input from user:
- a function application

**EVAL:** Evaluate input:
- \((f \ arg_1 \ arg_2 \ldots \ arg_n)\)
  1. evaluate \(f\) to obtain a function
  2. evaluate each \(arg_i\) to obtain a value
  3. apply function to argument values

**PRINT:** Print resulting value:
- the result of the function application

You can write your Scheme program in file \(<\text{name}>.ss\) and then read it into the Scheme interpreter by saying at the interpreter prompt: \(\text{(load "<name>.ss")}\)
READ-EVAL-PRINT Loop Example

> (cons 'a (cons 'b '(c d)))
(a b c d)

1. Read the function application
   (cons 'a (cons 'b '(c d)))
2. Evaluate cons to obtain a function
3. Evaluate ’a to obtain a itself
4. Evaluate (cons ’b ’(c d)):
   4.1 Evaluate cons to obtain a function
   4.2 Evaluate ’b to obtain b itself
   4.3 Evaluate ’(c d) to obtain (c d) itself
   4.4 Apply the cons function to b and (c d) to obtain (b c d)
5. Apply the cons function to a and (b c d) to obtain (a b c d)
6. Print the result of the application:
   (a b c d)
Quotes Inhibit Evaluation

;; Same as before:
> (cons 'a (cons 'b '(c d)))
(a b c d)

;; Now quote the second argument:
> (cons 'a '(cons 'b '(c d)))
(a cons (quote b) (quote (c d)))

;; Instead, un-quote the first argument:
> (cons a (cons 'b '(c d)))
ERROR: unbound variable: a
Defining Global Variables

The define constructs extends the current interpreter environment by the new defined (name, value) association.

> (define foo '(a b c))
#<unspecified>

> (define bar '(d e f))
#<unspecified>

> (append foo bar)
(a b c d e f)

> (cons foo bar)
((a b c) d e f)

> (cons 'foo bar)
(foo d e f)
Defining Scheme Functions

\[
\text{(define <fcn-name> (lambda (<fcn-params>)
    \text{<expression>}))}
\]

Example: Given function pair? (true for non-empty lists, false o/w) and function not (boolean negation):

\[
\text{(define atom?}
    \text{(lambda (object) (not (pair? object))))}
\]

Evaluating (atom? '(a)):
1. Obtain function value for atom?
2. Evaluate '(a) obtaining (a)
3. Evaluate (not (pair? object))
   a) Obtain function value for not
   b) Evaluate (pair? object)
      i. Obtain function value for pair?
      ii. Evaluate object obtaining (a)
      Evaluates to #t
   Evaluates to #f
Evaluates to #f
Conditional Execution: if

(if <condition> <result1> <result2>)

1. Evaluate <condition>
2. If the result is a “true value” (i.e., anything but #f), then evaluate and return <result1>
3. Otherwise, evaluate and return <result2>

(define abs-val
  (lambda (x)
    (if (>= x 0) x (- x))))

(define rest-if-first
  (lambda (e l)
    (if (eq? e (car l)) (cdr l) '())))

Zheng Zhang 11 CS@Rutgers University
Conditional Execution: cond

(cond (<condition1> <result1>))
   ((<condition2> <result2>))
   ...
   ((<conditionN> <resultN>))
   (else <else-result>)) ; optional else clause

1. Evaluate conditions in order until obtaining one that returns a true value
2. Evaluate and return the corresponding result
3. If none of the conditions returns a true value, evaluate and return <else-result>
(define abs-val
  (lambda (x)
    (cond ((>= x 0) x)
      (else (- x)))))

(define rest-if-first
  (lambda (e l)
    (cond ((null? l) '())
      ((eq? e (car l)) (cdr l))
      (else '()))))
Recursive Scheme Functions: Abs-List

- `(abs-list '(1 -2 -3 4 0)) ⇒ (1 2 3 4 0)`
- `(abs-list '()) ⇒ ()`

```
(define abs-list
  (lambda (l)
    (if (null? l)
        '()
        (cons (abs-val (car l)) (abs-list (cdr l))))))
```

Zheng Zhang 14 CS@Rutgers University
Recursive Scheme Functions: Append

(append '(1 2) '(3 4 5) ⇒ (1 2 3 4 5)
(append '(1 2) '(3 (4) 5) ⇒ (1 2 3 (4) 5)
(append '() '(1 4 5)) ⇒ (1 4 5)
(append '(1 4 5) '()) ⇒ (1 4 5)
(append '() '()) ⇒ ()

(define append
  (lambda (x y)
    (cond ((null? x) y)
          ((null? y) x)
          (else (cons (car x)
                      (append (cdr x) y))))))
Equality Checking

The eq? predicate doesn’t work for lists.
Why not?

1. (cons ’a ’()) produces a new list
2. (cons ’a ’()) produces another new list
3. eq? checks if its two arguments are the same
4. (eq? (cons ’a ’()) (cons ’a ’())) evaluates to #f

Lists are stored as pointers to the first element (car) and the rest of the list (cdr). This elementary “data structure”, the building block of lists, is called a pair.

Symbols are stored uniquely, so eq? works on them.

Zheng Zhang
Equality Checking for Lists

For lists, need a comparison function to check for the same structure in two lists

\[
(\text{define equal?} \\
(\lambda (x \ y) \\
 (\text{or} \ (\text{and} \ (\text{atom?} \ x) \ (\text{atom?} \ y) \ (\text{eq?} \ x \ y)) \\
 (\text{and} \ (\text{not} \ (\text{atom?} \ x)) \ (\text{not} \ (\text{atom?} \ y)) \\
 (\text{equal?} \ (\text{car} \ x) \ (\text{car} \ y)) \\
 (\text{equal?} \ (\text{cdr} \ x) \ (\text{cdr} \ y))))))
\]

- (equal? 'a 'a) evaluates to #t
- (equal? 'a 'b) evaluates to #f
- (equal? '(a) '(a)) evaluates to #t
- (equal? '((a)) '(a)) evaluates to #f
Functions as arguments:
(define f (lambda (g x) (g x)))

▶ (f number? 0)
⇒ (number? 0) ⇒ #t
▶ (f length '(1 2))
⇒ (length '(1 2)) ⇒ 2
▶ (f (lambda (x) (* 2 x)) 3)
⇒ ((lambda (x) (* 2 x)) 3)
⇒ (* 2 3) ⇒ 6

REMINDER: Computation, i.e., function application is performed by reducing the initial S-expression (program) to an S-expression that represents a value. Reduction is performed by substitution, i.e., replacing formal by actual arguments in the function body. Examples for S-expressions that directly represent values, i.e., cannot be further reduced:

▶ function values (e.g.: (lambda(x) e))
▶ constants (e.g.: 3, #t)
Higher-order Functions (Cont.)

Functions as returned values:
(define plusn
  (lambda (n) (lambda (x) (+ n x))))

▶ (plusn 5) evaluates to a function that adds 5 to its argument

Question: How would you write down the value of (plusn 5)?

▶ ((plusn 5) 6) ⇒ 11
Next Lecture

Reading: Scott Chapter 11.1 to 11.3

More Scheme and higher-order functions