CS 314 Principles of Programming Languages

Lecture 15

Zheng Zhang

Department of Computer Science
Rutgers University

Wednesday 26th October, 2016
Class Information

- Homework 6 will be posted today.
- Midterm November 2, Wednesday, in class.
Homework 3 Extra-credit

▶ >75% Credit: Rich Chen, David Domingo, Jason Ramirez, Tian Si, Tianyi Yu
▶ >50%: Michael Dilks, Akhila Narayan, Xinran Shi,
▶ >25%: Elina Chhabra, Heman Gandhi, Chaozhang Huang, Christopher Orthmann, Patel Rooshhil, Andre Pereira, Farah, Sultana
▶ Those who tried: Chinmoyi Bhushan, Patel Yashvi, Ayush Joshi
Scott: Chapter 11

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

1. **Referential transparency**: The value of a function application is independent of the context in which it occurs
2. The concept of assignment is **not** part of functional programming
3. Control flow is governed by function calls and conditional expressions
4. All storage management is implicit
5. Functions are **First Class Values**
A program includes:

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

E.g. in Scheme:

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

> (length '(A LIST OF 5 THINGS))
5
```
- Developed in 1975 by G. Sussman and G. Steele
- A version of LISP
- Simple syntax, small language
- Closer to initial semantics of LISP as compared to COMMON LISP
- Provides basic list processing tools
- Allows functions to be first class objects
Expressions are written in prefix, parenthesized form

- (function arg₁ arg₂ ... argₙ)
- (+ 4 5)
- (+ (* 3 4) (- 5 3))

Operational semantics: In order to evaluate an expression:

1. evaluate function to a function value
2. evaluate each argᵢ in order to obtain its value
3. apply the function value to these values
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.
Does the above S-expression grammar look right?
Lists in Scheme

The building blocks for lists are pairs or cons-cells. Lists use the empty list ( ) as an “end-of-list” marker.
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
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
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))) =
Special (Primitive) Functions

- car and cdr can break up any list:
  - (car (cdr (cdr ’((a) b (c d))))) =
  - (caddr ’((a) b (c d)))

- cons can construct any list:
  - (cons ’a ’()) =
  - (cons ’d ’(e)) =
  - (cons ’(a b) ’(c d)) =
  - (cons ’(a b c) (((a) b)) =
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
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

(define <fcn-name> (lambda (<fcn-params>)
  <expression>))

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

(define atom?
  (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) '())))
Conditional Execution: \texttt{cond}

\begin{verbatim}
(cond (<condition1> <result1>)
  (<condition2> <result2>)
  ...
  (<conditionN> <resultN>)
  (else <else-result>)) ; optional else clause
\end{verbatim}

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 '()))))
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 arg₁ arg₂ ...argₙ)
1. evaluate f to obtain a function
2. evaluate each argᵢ 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 <name>.ss and then read it into the Scheme interpreter by saying at the interpreter prompt: (load "<name>.ss")
> (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)
More Scheme and higher-order functions