Class Information

INFORMATION and REMINDERS

• Homework 5 is due tomorrow, Thursday November 9, at 11:59pm

• Homework 6 will be posted on Friday.

• Second programming project will be posted next week.

• Final exam: Wednesday, December 20, 4:00-7:00pm.

DO YOU HAVE A CONFLICT?

http://sasundergrad.rutgers.edu/forms/final-exam-conflict

  – More than two (2) final exams on one calendar day
  – More than two (2) final exams scheduled in consecutive periods
  – Two final exams scheduled for the same exam period.
Pure Functional Languages

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
Lists in Scheme

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

Note: (a.b) is not a list!
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
READ-EVAL-PRINT Loop

The Scheme interpreters on the ilab machines are called mzscheme, racket, and drracket. “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")
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)):
   (a) Evaluate cons to obtain a function
   (b) Evaluate 'b to obtain b itself
   (c) Evaluate '(c d) to obtain (c d) itself
   (d) 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
Scheme Programming and Emacs

You can invoke the interpreter **mzscheme** Scheme interpreter on the ilab cluster from within **emacs** by executing the commands: **ESC-x run-scheme**.

Typically, you want to split your emacs window into two parts (**CTRL-x 2**), and then edit your Scheme file in one window, and execute it in the other. To read a Scheme program into the interpreter, say (**load “<name>.ss”**). You can switch between windows by saying **CTRL-x o**.

You can save the “scheme interpreter” window into a file to inspect it later, i.e., to keep a record on what you have done. This may be useful during debugging.
Defining Global Variables

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

```scheme
> (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

\[
(\text{define } \textit{fcn-name} \ (\text{lambda} \ (\textit{fcn-params}) \\
\quad \textit{expression}))
\]

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

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

Evaluating \textit{(atom? '(a))}:  
1. Obtain function value for \textit{atom?}  
2. Evaluate \textit{'(a)} obtaining \textit{(a)}  
3. Evaluate \textit{(not (pair? object))} 
   a) Obtain function value for \textit{not}  
   b) Evaluate \textit{(pair? object)}  
      i. Obtain function value for \textit{pair?}  
      ii. Evaluate \textit{object} obtaining \textit{(a)}  
          Evaluates to \textit{#t}  
          Evaluates to \textit{#f}  
  Evaluates to \textit{#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: cond

(cond (<condition1> <result1>))
   (cond (<condition2> <result2>))
   ...
   (cond (<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>
Recursive Scheme Functions: Abs-List

- \( \text{abs-list '}(1 \ -2 \ -3 \ 4 \ 0) \Rightarrow (1 \ 2 \ 3 \ 4 \ 0) \)
- \( \text{abs-list '}() \Rightarrow () \)

\[
\text{(define abs-list }
\text{ (lambda (l}
\text{ )}
\text{ )}
\]
Recursive Scheme Functions: Append

(append '1 2 '3 4 5) \Rightarrow (1 2 3 4 5)
(append '1 2 '3 (4) 5) \Rightarrow (1 2 3 (4) 5)
(append () '1 4 5) \Rightarrow (1 4 5)
(append '1 4 5 () ) \Rightarrow (1 4 5)
(append () () ) \Rightarrow ()

(define append
    (lambda (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.
Equality Checking for Lists

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

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

- \text{(equal? ’a ’a)} evaluates to \text{#t}
- \text{(equal? ’a ’b)} evaluates to \text{#f}
- \text{(equal? ’(a) ’(a))} evaluates to \text{#t}
- \text{(equal? ’((a)) ’(a))} evaluates to \text{#f}
Next Lecture

Next time:

- higher-order functions; map and reduce
- lambda calculus