Class Information

INFORMATION and REMINDERS

- Homework 5 deadline extension?
Lists in Scheme

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

Note: \texttt{(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
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:

  \[-(\text{car} \ (\text{cdr} \ (\text{cdr} \ '((a) \ b \ (c \ d)))))) = \]

  \[-(\text{caddr} \ '((a) \ b \ (c \ d))) \]

• **cons** can construct any list:

  \[-(\text{cons} \ 'a \ '()) = \]

  \[-(\text{cons} \ 'd \ 'e()) = \]

  \[-(\text{cons} \ '(a \ b) \ '(c \ d)) = \]

  \[-(\text{cons} \ '(a \ b \ c) \ '((a) \ b)) = \]
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:

  \[⇒ \text{All return Boolean values: } \#f \text{ and } \#t\]

  - \((\text{number? } 5)\) evaluates to \#t
  - \((\text{zero? } 0)\) evaluates to \#t
  - \((\text{symbol? } 'sam)\) evaluates to \#t
  - \((\text{list? } '(a b))\) evaluates to \#t
  - \((\text{null? } '())\) 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`
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"**)
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` construct 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

(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: cond

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

Next time:

• More programming in Scheme
• Lambda calculus