Class Information

REMINDERS

• Homework 6 has been posted. Due on Tuesday, November 17.

• Wednesday, November 25, is officially a Friday, so there will be lecture and recitation.
Review: 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  \((\text{contents of address part of register})\)
- **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.,
  
  \((+ 5 3) = 8\), \((- 5 3) = 2\)
  
  \((\ast 5 3) = 15\), \((/ 5 3) = 1.6666666\)

- = < > comparison operators for numbers

- Explicit type determination and test functions:
  
  \(\Rightarrow\) All return Boolean values: \(\#f\) 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`
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:

\[(f \ arg_1 \ arg_2 \ \ldots \ arg_n)\]

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 `<name>.ss` and then read it into the Scheme interpreter by saying at the interpreter prompt: `(load "<name>.ss")`
READ-EVAL-PRINT Loop Example

How to think about the execution.

> (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 b (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

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

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

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

Evaluating \textbf{(atom? '(a))}:
1. Obtain function value for \textbf{atom}?
2. Evaluate \textbf{'(a)} obtaining \textbf{(a)}
3. Evaluate \textbf{(not (pair? object))}
   a) Obtain function value for \textbf{not}
   b) Evaluate \textbf{(pair? object)}
      i. Obtain function value for \textbf{pair}?
      ii. Evaluate \textbf{object} obtaining \textbf{(a)}

Evaluates to \textbf{#t}
Evaluates to \textbf{#f}
Evaluates to \textbf{#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 ; 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>
Conditional Execution: cond

(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

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

\begin{verbatim}
(define abs-list
  (lambda (l)
    ))
\end{verbatim}
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)
    
    )

)
Next Lecture

Things to do:

- Homework problem set 6 has been posted.
- Project 2 (Scheme) will be posted soon; start programming in Scheme!

Next time:

- Higher order functions
- Project overview