Class Information

• Homework 6 will be posted later today.
• All test cases for project 1 have been released.
• If a small issue in project 1 results in your code not working for most test cases, you can submit a patch (using the Linux utility tool `patch` to generate the difference of two versions of a file) **by the end of this week.** We will review the change before we determine if you will get partial credit back.
Special (Primitive) Functions

- **eq?**: identity on names (atoms)
- **null?**: is list empty?
- **car**: select first element of the list
  (contents of address part of register)
- **cdr**: select rest of the list
  (contents of decrement part of register)
- **(cons element list)**: constructs lists by adding **element** to the front of **list**
- **quote or '**: produces constants

*Do not evaluate the ' the content after '. Treat them as list of literals.*
Quotes Inhibit Evaluation

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

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

;; If we unquote the first argument
> ( cons a (cons 'b '(c d)) )
a: undefined;
cannot reference undefined identifier
context …
Review: 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):

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)
      iii. Evaluates to #t
   c) Evaluates to #f
4. Evaluates to #f
Review: 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) '())))
Review: Conditional Execution \textit{cond}

\begin{itemize}
\item Evaluate conditions in order until obtaining one that returns a \#t value
\item Evaluate and return the corresponding result
\item If none of the conditions returns a true value, evaluate and return \texttt{<else-result>}
\end{itemize}

\texttt{(cond \langle\texttt{condition1}\rangle \ \langle\texttt{result1}\rangle)}
\begin{itemize}
\item \texttt{(\langle\texttt{condition2}\rangle \ \langle\texttt{result2}\rangle)}
\item \ldots
\item \texttt{(\langle\texttt{conditionN}\rangle \ \langle\texttt{resultN}\rangle)}
\item \texttt{(\texttt{else \langle\texttt{else-result}\rangle})}; \textit{optional else clause}
\end{itemize}
(define abs-val
  (lambda (x)
    (cond ((>= x 0) x)
          ((else (- x))))))

Example 2

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

Example 1
Recursive Scheme Functions: abs-List

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

- (abs-list '(1 -2 -3 4 0)) ⇒ (1 2 3 4 0)
- (abs-list '()) ⇒ ()
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 does not work for lists.
Why not?

• (cons 'a '()) produces a new list
• (cons 'a '()) produces another new list
• eq? checks whether two arguments are the same
• (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 a list, 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.

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

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

```
(define equal?  
  (lambda (x y)    
    (or (and (atom? x) (atom? y) (eq? x y)) 
        (and (not (atom? x)) (not (atom? y)) 
         (equal? (car x) (car y)) 
         (equal? (cdr x) (cdr y)))  
  )
  )
)  
```
Equality Checking for Lists

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

(define equal? (lambda (x y) (or (and (atom? x) (atom? y) (eq? x y)) (and (not (atom? x)) (not (atom? y)) (equal? (car x) (car y)) (equal? (cdr x) (cdr y)))))

)
Equality Checking for Lists

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

(define equal?
  (lambda (x y)
    (or ( and (atom? x) (atom? y) (eq? x y) )
        ( and (not (atom? x))(not (atom? y))
          (equal? (car x) (car y))
          (equal? (cdr x) (cdr y))
        )
    )
  )
)
Equality Checking for Lists

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

```
(define equal? 
  (lambda (x y) 
    (or (and (atom? x) (atom? y) (eq? x y)) 
        (and (not (atom? x)) (not (atom? y))) 
        (equal? (car x) (car y)) 
        (equal? (cdr x) (cdr y)))
  )
)
```
Equality Checking for Lists

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

(define equal?
  (lambda (x y)
    (or ( and (atom? x) (atom? y) (eq? x y) )
        ( and (not (atom? x)) (not (atom? y))
            (equal? (car x) (car y))
            (equal? (cdr x) (cdr y))
        )
    )
  )
)
Equality Checking for Lists

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

```
(define equal? 
  (lambda (x y) 
    (or (and (atom? x) (atom? y) (eq? x y))  
      (and (not (atom? x)) (not (atom? y)) 
        (equal? (car x) (car y)) 
        (equal? (cdr x) (cdr y)))))
  )
)
```
For lists, need a comparison function to check for the same structure in two lists.

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

- (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:

\[
\text{(define } f (\text{lambda} \ (g \ x) \ (g \ x)) \text{)}
\]

- (f number? 0)
- (f length '(1 2))
- (f (lambda (x) (* 2 3)) 3)
Functions as arguments:

(\(\text{define f (lambda (g x) (g x))}\) )

\[
\begin{align*}
\text{• } (\text{f number? 0}) & \Rightarrow (\text{number? 0}) \Rightarrow \#t \\
\text{• } (\text{f length '(1 2)}) & \Rightarrow (\text{length '(1 2)}) \Rightarrow 2 \\
\text{• } (\text{f (lambda (x) (* 2 3)) 3}) & \Rightarrow ((\text{lambda (x) (* 2 3)) 3}) \Rightarrow (* 2 3) \Rightarrow 6
\end{align*}
\]
Scheme: Functions as First Class Values (Higher-Order)

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 parameters 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)

Computation completes when S-expression cannot be further reduced.
Higher-Order Functions (Cont.)

Functions as returned value:

(define plusn
    (lambda (n)
        (lambda (x) (+ n x)))
)

- \textbf{(plusn 5)} evaluates to a function that adds 5 to its argument:

\begin{quote}
\textit{Question:} How would you write down the value of \textit{(plusn 5)}?
\end{quote}

- \textbf{((plusn 5) 6)}
In general, any n-ary function

\[(\text{lambda} \ (x_1 \ x_2 \ \ldots \ x_n) \ e)\]
can be rewritten as a nest of \(n\) unary functions:

\[(\text{lambda} \ (x_1)\]

\[\quad (\text{lambda} \ (x_2) \]

\[\quad \quad (\ldots \ (\text{lambda}(x_n) \ e) \ \ldots \ ))\]
Higher-Order Functions (Cont.)

In general, any n-ary function

$$(\lambda (x_1 x_2 \ldots x_n) e)$$

can be rewritten as a nest of $n$ unary functions:

$$(\lambda (x_1)$$

$$(\lambda (x_2)$$

$$(\lambda (x_3)$$

$$\ldots (\lambda (x_n) e) \ldots )))$$

This translation process is called currying. It means that having functions with multiple parameters do not add anything to the expressiveness of the language:

$$( ((\lambda (x_1 x_2 \ldots x_n) e) v_1 v_2 \ldots v_n)$$

$$\downarrow$$

$$((\lambda (x_1)$$

$$((\lambda (x_2)$$

$$\ldots ((\lambda (x_n) e )\ldots)) v_1) v_2) \ldots v_n)$$
Higher-order Functions: map

(define map
  (lambda (f l)
    (if (null? l)
        '()
        (cons (f (car l)) (map f (cdr l)) ) ) )
  )

- **map** takes two arguments: a function and a list
- **map** builds a new list by applying the function to every element of the (old) list
Higher-order Functions: map

(map)

• map takes two arguments: a function and a list
• map builds a new list by applying the function to every element of the (old) list

(define map
  (lambda (f l)
    (if (null? l)
        '( )
        (cons ( f (car l) ) (map f (cdr l))) )
  )
)

(cons ( f (car l) ) (map f (cdr l)))
Higher-Order Functions: map

• Example:
  (map \texttt{abs} '(-1 2 -3 4)) \Rightarrow (1 2 3 4)
  (map \texttt{(lambda} (x) (+ 1 x))) '(-1 2 3) \Rightarrow (0 3 4)

• Actually, the built-in \texttt{map} can have more than two arguments:
  (map + '1 2 3) '4 5 6)) \Rightarrow (5 7 9)
Things to do: