CSE 341
Lecture 16

More Scheme:
lists; helpers; let/let*;
higher-order functions; lambdas

slides created by Marty Stepp
http://www.cs.washington.edu/341/
Lists

(list expr2 ... exprN)
'(value1 value2 ... valueN) ; all literals

• Scheme lists are created with the list procedure
  ▪ the empty list is written as null
  ▪ if you forget list, Scheme will "run" your list as a procedure call, with first element as the procedure name

> (define L (1 2 3 4))
procedure application: expected procedure, given: 1; arguments were: 2 3 4
> (define L (list 1 2 3 4))
> L
(1 2 3 4)
List procedures to access elements

(car list) ; 1st element (ML hd) "car"
(cdr list) ; all but 1st (ML tl) "could-er"
cadr, cddr, ; 2nd element; all but first 2
caddr, cdddr, ...
  ▪ these bizarre names come from IBM 704 computer op-codes
  ▪ others: first, second, third, ..., rest, last

• Examples:
  > (define L (list 1 2 3 4 5))
  > (car L)
  1
  > (cdr L)
  (2 3 4 5)
More list procedures

(cons \textit{expr} \textit{list}) ; \textit{expr} :: list
(append \textit{list1} \textit{list2} \ldots \textit{listN}) ; \textit{list1} @ \textit{list2}
(length \textit{list})
(null? \textit{list}) ; \textit{list} = [] ?

- others: drop, filter, flatten, foldl, foldr, make-list, map, member, partition, remove-duplicates, split-at, take, take-right, values

- Examples:
  > (cons 4 (1 2 3))
  (4 1 2 3)
  > (append (1 2) (30 40 50) (6 7))
  (1 2 30 40 50 6 7)
Scheme exercise

• Define a procedure `sum` that accepts a list as a parameter and computes the sum of the elements of the list.
  
  - `(sum (list 1 2 3 4 5))` should evaluate to 15

• solution:

```scheme
(define (sum lst)
  (if (null? lst)
      0
      (+ (car lst) (sum (cdr lst)))))
```
Scheme exercise

• Define a procedure `range` that accepts `min/max` integers and returns the list `(min, min+1, ..., max-1, max)`.
  
  (range 2 7) should return (2 3 4 5 6 7)

• solution:

  (define (range x y)
    (if (> x y) ()
      (cons x (range (+ x 1) y)))))
Define a procedure named \texttt{x+y^2} that accepts two numbers \texttt{x} and \texttt{y} as parameters and returns \((x + y)^2\).

\begin{itemize}
  \item Example: \((x+y^2 \ 3 \ 4)\) returns 49
\end{itemize}

\texttt{; not an ideal solution...}

\texttt{(define (x+y^2 x y) (* (+ x y) (+ x y)))}
let expressions

; define vars --> use them
(let ((name expr) ... (name expr)) expr)

• Defines a local symbol that can be used in the last expr
  ▪ notice that it is a list of (name expr) pairs to be defined

Example:
; better
(define (x+y^2 x y)
  (let ((sum (+ x y))) (* sum sum)))
• Define a procedure named \texttt{math!!} that accepts two numbers \( x \) and \( y \) as parameters and returns:
  \begin{itemize}
    \item \((x + y)^2 \times (x + y + 2)^2\)
    \item Example: \((\texttt{math!! 3 2})\) returns 1225
  \end{itemize}

\begin{verbatim}
; not an ideal solution...
(define (math!! x y)
  (* (* (+ x y) (+ x y))
      (* (+ x y 2) (+ x y 2))))
\end{verbatim}
Limitation of let expressions

(define (math!! x y)
  (let ((
    (sum (+ x y))
    (sum2 (+ sum 2)))
    (* sum sum sum2 sum2)))

> (math!! 2 3)
reference to undefined identifier: sum

• Problem: A symbol declared within a let list cannot refer to the other symbols defined in that same list!
let* expressions

; define vars --> use them
(let* ((name expr) ... (name expr)) expr)

- same as let, but the symbols are evaluated in sequence, so later ones can refer to earlier ones
  - slower / restricts evaluation order, so not default

Example:
(define (math!! x y)
  (let* ((sum (+ x y))
          (sum2 (+ sum 2))
          (* sum sum sum2 sum2)))
(define
 (outer-name param1 param2 ... paramN)
  (define (inner-name param1 ... paramN) expr1)
  expr2)

• You can define a local helper with a nested define
  ▪ the outer function's expr2 can call the inner procedure
  ▪ the inner procedure is visible only to the outer procedure
  ▪ analogous to let with functions in ML
Helper procedure example

; least common multiple of integers a and b
; \( \text{lcm}(a, b) = \frac{a \times b}{\text{gcd}(a, b)} \)
(define (lcmult a b)
  (define (gcd x y)
    (if (= y 0) x
     (gcd y (remainder x y))))
  (/ (* a b) (gcd a b)))

- (had to name it lcmult because lcm already exists)
Higher-order procedures

; apply procedure f to each element of lst
(map f lst)

; retain only elements where p returns #t
(filter p lst)

; reduce list; f takes 2 elements -> 1
(foldl f initialValue lst)
(foldr f initialValue lst)

• equivalent to ML's map/List.filter/fold*
• each takes a procedure (or "predicate") to apply to a list
Higher-order exercise

• Implement our own versions of map and filter, named mapx and filterx.
  ▪ e.g. (map f '(1 2 3 4 5))
  ▪ e.g. (filter p '(1 2 3 4 5))
; Applies procedure f to every element of lst.
(define (mapx f lst)
  (if (null? lst)
      ()
      (cons (f (car lst)) (mapx f (cdr lst))))

; Uses predicate p to keep/exclude elements of lst.
(define (filterx p lst)
  (cond ((null? lst) ()))
  ((p (car lst)) (cons (car lst)
                         (filterx p (cdr lst))))
  (else (filterx p (cdr lst))))
Anonymous procedures ("lambdas")

(lambda (param1 ... paramN) expr)

- defines an anonymous local procedure
  - you can pass a lambda to a higher-order function, etc.
  - analogous to ML's: fn(params) => expr

- Example (retain only the even elements of a list):
  (filter (lambda (n) (= 0 (modulo n 2)))
    (range 0 100))
• Using higher-order procedures and lambdas, find the sum of the factors of 24.
  ▪ Hint: First get all the factors in a list, then add them.

• Solution:
  \[(\text{foldl} + \theta
   \quad (\text{filter} \ (\lambda (n)
     \quad (= \theta (\text{modulo} 24 \ n)))
   \quad (\text{range} 1 24)))\]