The Scheme Language

- Chapter 15, mainly section 15.5
The program is a collection of functions
- A function computes and returns a value
- No side-effects (i.e., no state changes)
- No program variables whose values change
  - Basically, no assignments

Languages: LISP, Scheme (dialect of LISP from MIT, mid-70s), ML, Haskell, ...

- A function can be an actual parameter of another function
- A function can be the return value of another function
- A function could be an element of a list
- A function can be created at run time
Using Scheme

- **Read**: you enter an expression
- **Eval**: the interpreter evaluates the expression
- **Print**: the interpreter prints the resulting value
- **stdsun**: at the prompt, type `scheme48`
  - `type your expression here`
  - the interpreter prints the value here
  - `,help`
  - `,exit`
Data Objects in Scheme

- **Atoms**
  - Numeric constants: 5, 20, -100, 2.788
  - Booleans: #t (true) and #f (false)
  - String constants: “hi there”
  - Character constants: #\a
  - **Symbols**: f, x, +, *, foo, bar, null?, set!

- **S-expressions**
  - Lists are a special case of S-expressions
Evaluation of Atoms

- Numeric constants, string constants, and character constants evaluate to themselves
  
  > 4.5 > #t
  4.5 #t
  > "This is a string" > #f
  "This is a string" #f

- Symbols do not have values to start with
  - They may get “bound” to values

  > x
  Error: undefined variable x
Lists

- List elements are atoms, other lists, and general S-expressions
  - e.g. ( (3 4) 5 (6) )

- Lists are **heterogeneous**: the elements do not have to be of the same type

- Empty list () - has zero elements
  - Operations **car** and **cdr** are not defined for an empty list - run-time error

- Evaluation of a list: **function application** (i.e., function call) - more later ...
Primitive Operations for Lists

- **car** for a list produces the first element of the list (the list head)
  - e.g. for ((A B) (C D) E) will produce (A B)

- **cdr** produces the tail of the list: a list containing all elements except the first
  - e.g. for ((A B) (C D) E) will produce ((C D) E)

- **cons** adds to the beginning of the list
  - cons of A and (B C) is (A B C)
  - e.g., cons of car of x and cdr of x is x
S-expressions

- Every atom is an S-expression
- () is an S-expression (empty S-expression)
- If s1 and s2 are S-expressions, so is the pair (s1 . s2)
  - Essentially, a binary tree: left child is the tree for s1, and right child is the tree for s2
  - Atoms and () are leaves of the tree
  - E.g.: (3 . 5), ((3 . 4) . (5 . 6)), (3 . (5 . ()))
Primitive Functions for S-expressions

- **car**: unary; produces the S-expression corresponding to the left child of the argument
  - Not defined for atoms and for ()

- **cdr**: unary; produces the S-expression corresponding to the right child of the argument
  - Not defined for atoms and for ()

- **cons**: binary; produces a new S-expr with left child = 1st arg and right child = 2nd arg
Lists as S-Expressions

- A list is a special S-expression in which the rightmost leaf is ()
  - e.g. ( (3 . 4) . (5 . ()))

- The S-expressions “hanging” from the rightmost “spine” of the tree are the elements of the list
  - e.g. (3 . 4) and 5
  - The list is written as ( (3 . 4) 5)

- S-expression ( (3 . ()) . ( (4 . ()) . (5 . ())))) is the list ( (3) (4) 5)
More Examples of Lists

\[(A \ B \ C) = (A \cdot (B \cdot (C \cdot ()))))\]
\[((A \ B) \ C) = ((A \cdot (B \cdot ()))) \cdot (C \cdot ())\]
\[(A \ B \ (C \ D)) = (A \cdot (B \cdot ((C \cdot (D \cdot ()))) \cdot ())))\]
\[((A)) = ((A \cdot ())) \cdot ()\]
\[(A \ (B \cdot C)) = (A \cdot ((B \cdot C) \cdot ())))\]
Data vs. Code

- Interpreter for an imperative language: the input is code+data, the output is data (values)
- Everything in Scheme is an S-expression
  - The “program” we are executing is an S-exp
  - The output of the program is also an S-exp
    - Data and code are really the same thing
- Example: an expression that represents function application is a list (f p1 p2 ...)
  - f is an S-expression representing the function we are calling
  - p1 is an S-expression representing the first actual parameter, etc.
Function Application

- (+ 5 6)
  - This S-expression is a “program”; here + is the built-in function “add”
  - The evaluation by the interpreter produces the S-expression 11

- Function application: (f p1 p2 …)
  - The interpreter evaluates S-exp f, p1, p2, etc.
  - The interpreter invokes the resulting function on the resulting values
Examples

\[ (+ 5 6) \]
11

\[ (+ (+ 3 5) (* 4 4)) \]
24

\[ (+ 5 #t) \]
Error, because “add” is defined only for numeric atoms

\[ (\text{car} \ 5) \]
Error, car is not defined for atoms

\[ (\text{cdr} \ 5) \]
Same here

\[ (\text{cons} \ 4 \ 5) \]
'(4 . 5)
Quoting an Expression

- When the interpreter sees a non-atom, it tries to evaluate it as if it were a function call
  - But for (5 6), what does it mean? Error ...

- We can tell the interpreter to evaluate an expression to itself
  - (quote (5 6)) or simply '(5 6)
  - Evaluates to the S-expression (5 6)
  - The resulting expression is printed by Scheme as '(5 6)
Examples

\> (+ (+ 3 5) (car (7 . 8)))

Errors

1> Ctrl-D

\> (+ (+3 5) (car '(7 . 8)))

15

\> (car (7 10))

Errors

1> (car '(7 10))

7

1> (+ (car '(7 10)) (cdr '(7 10)))

Errors

2> (+ (car '(7 10)) (cdr '(7 . 10)))

17
More Examples

> (cons (car '(7 . 10)) (cdr '(7 . 10)))
'(7 . 10)
> (cons (car '(7 10)) (cdr '(7 . 10)))
'(7 . 10)
> (cons (car '(7 . 10)) (cdr '(7 10)))
'(7 10)
> (cons (car '(7 10)) (cdr '(7 10)))
'(7 10)

> a
Error
> 'a
'a
> (car '(A B))
'a
> (cdr '(A B))
'(b)
> (cons 'a '(b))
'(a b)
> (cons 'a 'b)
'(a . b)
More Examples

> (equal? #t #f)  > (equal? '() #f)
#f  #f
> (equal? #t #t)  > (equal? (+ 7 5) (+ 5 7))
#t  #t
> (equal? (cons 'a '(b)) '(a b))  // #f for eqv? and eq?
#t
> (pair? '(7 . 10))  > (pair? 7)  > (pair? '())
#t  #f  #f
> (null? '())  > (null? #f)  > (null? '(b))
#t  #f  #f
<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(even? 7)</code></td>
<td>#f</td>
</tr>
<tr>
<td><code>(even? (+ 7 7))</code></td>
<td>#t</td>
</tr>
<tr>
<td><code>(= 5 6)</code></td>
<td>#f</td>
</tr>
<tr>
<td><code>(= 4.5 4.5 4.5)</code></td>
<td>#t</td>
</tr>
<tr>
<td><code>(= 'a 'b)</code></td>
<td>Error</td>
</tr>
<tr>
<td><code>(even? 8)</code></td>
<td>#t</td>
</tr>
<tr>
<td><code>(even 7)</code></td>
<td>Error</td>
</tr>
<tr>
<td><code>(even? 'a)</code></td>
<td>Error</td>
</tr>
<tr>
<td><code>(= 5 6)</code></td>
<td>#t</td>
</tr>
<tr>
<td><code>(&gt; 5 6)</code></td>
<td>#f</td>
</tr>
<tr>
<td><code>(= 4.5 4.5 4.7)</code></td>
<td>#f</td>
</tr>
</tbody>
</table>
Conditional Expressions

- \((\text{cond} \ (b_1 \ e_1) \ (b_2 \ e_2) \ldots \ (b_n \ e_n))\)
- First evaluate \(b_1\). If not \(#f\), evaluate \(e_1\) and this is the value to the conditional.
- If \(b_1\) evaluates to \(#f\), evaluate \(b_2\), etc.
- If all \(b\) evaluate to \(#f\): error
  - That’s why we usually have \(#t\) as the last \(b\).
- \((\text{if} \ b \ e_1 \ e_2): \text{if} \ b \ \text{does not evaluate to} \ #f, \ \text{evaluate} \ e_1; \ \text{otherwise, evaluate} \ e_2\)
  - i.e. anything other than \(#f\) is “true”
Function Definition

> (define (double x) (+ x x)) ; no values returned

> (double 7)   > (double 4.4)   >(double '(7))
14     8.8       Error

> (define (mydiff x y) (cond ((= x y) #f) (#t #t))) ; no values returned

> (mydiff 4 5)   > (mydiff 4 4)   > (mydiff '(4) '(4))
#t     #f       ???
Member of a List?

In text file mbr.ss create the following:

; this is a comment
; (mbr x list): is x a member of the list?
(define (mbr x list)
  (cond
    ( (null? list) #f )
    ( #t (cond
      ( (equal? x (car list)) #t )
      ( #t (mbr x (cdr list)) ) )
  )
)

Or we could use just one "cond"...
Member of a List?

In the interpreter:

> (load "mbr.ss")
mbr.ss
; no values returned
> (mbr 4 '(5 6 4 7))
#t
> (mbr 8 '(5 6 4 7))
#f
Union of Two Lists

(define (uni s1 s2)
  (cond
    ((null? s1) s2)
    ((null? s2) s1)
    (#t (cond
          ((mbr (car s1) s2) (uni (cdr s1) s2))
          (#t (cons (car s1) (uni (cdr s1) s2))))))))

How about using "if" in mbr and uni?

> (uni '(4) '(2 3))
'(4 2 3)

> (uni '(3 10 12) '(20 10 12 45))
'(3 20 10 12 45)
Removing Duplicates

; x: a sorted list of numbers; remove duplicates ...

(define (unique x)
  (cond
    ( (null? x) x )
    ( (null? (cdr x)) x )
    ( (equal? (car x) (cdr x)) (unique (cdr x)) )
    ( #t (cons (car x) (unique (cdr x))) )
  )
)

> (unique '(2 2 3 4 4 5))
  (2 2 3 4 4 5) ;???
Largest Number in a List

; max number in a non-empty list of numbers
(define (maxlist L)
  (cond
   ( (null? (cdr L)) (car L) )
   ( (> (car L) (maxlist (cdr L))) (car L) )
   ( #t (maxlist (cdr L)) )
  )
)

What is the running time as a function of list size? How can we improve it?
A Different Approach

; max number in a non-empty list of numbers
(define (maxlist L) (mymax (car L) (cdr L)))
(define (mymax x L)
  (cond
    ( (null? L) x )
    ( (> x (car L)) (mymax x (cdr L)) )
    ( #t (mymax (car L) (cdr L)) )
  )
)

What is the running time as a function of list size?
Consider \((F \ p1 \ p2 \ ...)\):

- Evaluate \(p1, p2, \ldots\) using the current bindings.
- "Bind" the resulting values \(v1, v2, \ldots\) to the formal parameters \(f1, f2, \ldots\) of \(F\).
  - Add pairs \((f1,v1), (f2,v2), \ldots\) to the current set of bindings.
- Evaluate the body of \(F\) using the bindings.
  - If we see \(p1\) in the body, we will evaluate it to value \(v1\).
- After coming back from the call, the bindings for \(p1, p2, \ldots\) are destroyed.
Higher-Order Functions

(define (double x) (+ x x))
(define (twice f x) (f (f x)))
(twice double 2) ; returns 8

(define (mymap f list)
  (if (null? list) list
      (cons (f (car list))
        (mymap f (cdr list)))))

(mymap double '(1 2 3 4 5)) ; returns '(2 4 6 8 10)
Higher-Order Functions

(define (double x) (+ x x))
(define (id x) x)
((id double) 11) ; returns 22

(define (makelist f n)
  (if (= n 0) '()
      (cons f (makelist f (- n 1)))))

(makelist double 4)
; returns '(procedure double, procedure double, procedure double, procedure double)
Higher-Order Functions

(define (newmap x list)
  (if (null? list) list
      (cons ((car list) x) (newmap x (cdr list))))))
; what does this function do?

(newmap 11 (makelist double 7))
; what is the result of this function application?

(define (foo n) (newmap n (makelist double 5)))
(twice foo 9)
; how about here?
Recursion for Iterating

; Factorial function
(define (fact n)
  (if (= n 0) 1
      (* n (fact (- n 1)))))

Equivalent computation in imperative languages
f := 1;
for (i = 1; i <= n; i++) f := f * i;
QuickSort

Sort list of numbers (no duplicates)

Algorithm:

- If list is empty, we are done
- Choose pivot $n$ (e.g., first element)
- Partition list into lists $A$ and $B$ with elements $< n$ in $A$ and elements $> n$ in $B$
- Recursively sort $A$ and $B$
- Append sorted lists and $n$
Constructing the Two Sublists

(define (ltlist n list)
  (if (null? list) list
      (if (< (car list) n)
          (cons (car list) (ltlist n (cdr list)))
          (ltlist n (cdr list))))

Similarly we can define function gtlist
Sorting

(define (qsort list)
  (if (null? list) list
      (append
       (qsort (ltlist (car list) (cdr list)))
       (cons (car list) '())
       (qsort (gtlist (car list) (cdr list)))))))

(qsort '(4 3 5 1 6 2 8 7)) ; returns '(1 2 3 4 5 6 7 8)