Review: sum the items in a list

\[
\text{add-items} \text{ (list 2 5 4)}
\]

\[
2
\]

\[
5
\]

\[
4
\]

\[
\text{define} \text{ (add-items} \text{ m)}
\]

\[
(\text{if} \text{ (null? m)}
\]

\[
0
\]

\[
+ \text{ (car m)} \text{ (add-items} \text{ (cdr m))})
\]

Exercise #1: Write a function to find the maximum element of a list. Assume list is non-empty.

\[
\text{define} \text{ (find-max m)}
\]

Review: multiply each list element by 2

\[
\text{double-all} \text{ (list 4 0 -3)}
\]

\[
\text{define} \text{ (double-all} \text{ m)}
\]

\[
(\text{if} \text{ (null? m)}
\]

\[
'()
\]

\[
(\text{cons} \text{ (*} \text{ 2} \text{ (car m)} \text{)} \text{ (double-all} \text{ (cdr m))})
\]

\[
(\text{cons} 8 \text{ (cons} 0 \text{ (cons} -6 \text{ '()})))
\]

Exercise #2: Write a function to concatenate two lists.

\[
\text{concat} \text{ (list 1 2 3) (list 7 8 9)}
\]

\[
(1 \text{ 2} \text{ 3} \text{ 7} \text{ 8} \text{ 9})
\]

\[
\text{define} \text{ (concat} \text{ x} \text{ y)}
\]

Exercise #3: Write a function that removes all the negative numbers from a list

\[
\text{remove-neg} \text{ (list 1 -7 8 -9)}
\]

\[
(1 \text{ 8})
\]

\[
\text{define} \text{ (remove-neg m)}
\]
Exercise #4: Write a tail-recursive solution to exercise #1 (or non-tail-recursive if your solution already was)

(define (concat x y)
  (cons x (cons y '())))

References

- Section 2.2.2, 2.3.1, Structure and Interpretation of Computer Programs
- Sections 4.1.2, 6.1, 6.3.3, Revised Report on the Algorithmic Language Scheme (R5RS)

Printing pairs and lists

- (cons 3 4) => (3 . 4)
- (cons 3 (cons 4 '())) => (3 4)

List structure

- (list 4 6) => (4 6)
- (list 2 (list 4 6)) => (2 (4 6))

List structure and cons

- (list 2 (list 4 6)) =>
- (cons 2 (list 4 6)) =>

Using lists to build abstract data types

- We know how lists are constructed and we know how to represent them
- We want to build abstract data structures
  - the use of lists is actually an implementation detail
- For example, a tree structure can be built in many different ways in many different languages
Expression trees

- In Scheme, we often use constructors and accessors to abstract away the underlying representation of data (which is usually a list)
- For example, consider arithmetic expression trees
- A binary expression is
  » an operator: +, -, *, / and two operands
- An operand is
  » a number or another expression

Expression tree example

- Infix notation: \((1 + (2 * (3 - 5)))\)
- Scheme prefix notation: \(+ 1 (* 2 (- 3 5))\)

Represent expression with a list

- For this example, we are restricting the type of expression somewhat
  » Operators in the tree are all binary
  » All of the leaves (operands) are numbers
- Each node is represented by a 3-element list
  » (operator left-operand right-operand)
- Recall that the operands can be
  » numbers (explicit values)
  » other expressions (lists)

Expressions as trees, trees as lists

Constructors and accessors

```
(define (make-exp op left right)
  (list op left right))

(define (operator exp)
  (define (left exp)
    (define (right exp)
      (define a (make-exp + 1 2))
      (+ 1 2)
      1 2)
  )

(define (eval-expr exp)
  (if (not (list? exp))
      exp
      (operator exp)
      (eval-expr (left exp))
      (eval-expr (right exp))))
```

Evaluator

```scheme
(eval-expr (make-exp + 1 2))
```

```scheme
(define (eval-exp expr)
  (if (not (list? expr))
      expr
      ((operator expr)
       (eval-exp (left expr))
       (eval-exp (right expr))))
```

```scheme
(eval-exp (make-exp + 1 2))
```
### Why quote?

- Scheme evaluates the symbols/lists that we give it
  » numbers evaluate to themselves
  » symbols evaluate to their current value
  » lists are evaluated as expressions defining procedure calls on a sets of actual arguments
- We sometimes need a way to say "use this symbol or list as it is, don’t evaluate it"
- Special form quote
  >\[(define a 1)\]
  >\[a => 1\]
  >\[(quote a) => a\]

### Quote examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Evaluate to</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(define a 1)</code></td>
<td>a</td>
</tr>
<tr>
<td><code>(quote a)</code></td>
<td>a</td>
</tr>
<tr>
<td><code>(define b (+ a a))</code></td>
<td>b</td>
</tr>
<tr>
<td><code>(define c (quote (+ a b)))</code></td>
<td>c</td>
</tr>
<tr>
<td><code>(car c)</code></td>
<td></td>
</tr>
<tr>
<td><code>(cadr c)</code></td>
<td></td>
</tr>
<tr>
<td><code>(caddr c)</code></td>
<td></td>
</tr>
</tbody>
</table>

### quote can be abbreviated: ‘

- `(a)` => a
- `(+ a b)` => (+ a b)
- `()` => ()
- `(null? '())` => #t
- `'((1 2 3) 4)` => (1 2 3 4)
- `(car '((1 2 3) 4))` => 1
- `(cadr '((1 2 3) 4))` => ((2 3) 4)

### Building lists with symbols

- What would the interpreter print in response to evaluating each of the following expressions?
  1. `(list 'a 'b)`
  2. `(cons 'a (list 'b))`
  3. `(cons 'a (cons 'b '()))`
  4. `(cons 'a (b()))`
  5. `'(a b)`

### Comparing items

- Scheme provides several different means of comparing objects
  » Do two numbers have the same value?
    * `(= a b)`
  » Are two objects the same object in memory?
    * `(eq? a b)`
  » Do two objects have the same value?
    * `(eqv? a b)`
  » Do the corresponding elements have the same values?
    * `(equal? list-a list-b)`
(member item s)

; find an item of any kind in a list s
; return the sublist that starts with the item
; or return #f

(define (member item s)
  (cond
   ((null? s) #f)
   ((equal? item (car s)) s)
   (else (member item (cdr s))))

(member 'a '(c d a))         =>
(member '(1 3) '(1 (1 3) 3)) =>
(member 'b '(a (b) c))       =>
(member '(b) '(a (b) c))     =>

Recall: Expression tree example

infix notation (1 * (2 * (3 - 5)))
Scheme prefix notation (+ 1 (* 2 (- 3 5)))

expression tree

Represent expression with a list

• Each node is represented by a 3-element list
  » (operator left-operand right-operand)
• Operands can be
  » numbers (explicit values)
  » other expressions (lists)
• In previous implementation, operators were the actual procedures
  » This time, we will use symbols throughout

Expressions as trees, trees as lists

our data structure
'(+(1(*2(-35))))

Constructor and accessor functions

(define (make-exp op left right)
  (list op left right))

(define (operator exp)
  (car exp))

(define (left exp)
  (cadr exp))

(define (right exp)
  (caddr exp))

(eval-op op)

(define (eval-op op)
  (cond
   ((eqv? op '^) expt)
   (else (eval op))))

(define (eval-expr exp)
  (if (not (list? exp))
      exp
      ((eval-op (operator exp))
       (eval-expr (left exp))
       (eval-expr (right exp))))

(eval-expr '(+ 1 (* 2 (- 3 5))))
Traversing a binary tree

- Recall the definitions of traversal
  - pre-order
    - this node, left branch, right branch
  - in-order
    - left branch, this node, right branch
  - post-order
    - left branch, right branch, this node

```
(1+(2*(3-5)))
```

Output expression in post-fix order

```
define (post-order exp)
  (if (not (pair? exp))
    (list exp)
    (append
      (post-order (left exp))
      (post-order (right exp))
      (list (operator exp))))

(define f '+ 1 (* 2 (- 3 5)))
(post-order f)
(1 2 3 5 - * +)
```