Hierarchical Structures

CSE 413, Autumn 2005
Programming Languages

http://www.cs.washington.edu/education/courses/413/05au/

References

• Section 2.2.2, 2.3.1, Structure and Interpretation of Computer Programs

• Sections 4.1.2, 6.1, 6.3.3, Revised5 Report on the Algorithmic Language Scheme (R5RS)

Lists are a basic abstraction

• Using list to build lists, we can build data structures of increasing complexity
• Nested lists
  » one or more of the elements of the list can also be lists themselves
  » (list 1 2 (list 3 4) 5)

List structure

(define a (list 4 5 6))
(define b (list 7 a 8))

car = "this element"
cdr = "rest of the elements"
Printed representation of a list

- Lists are so fundamental to Scheme that the interpreter assumes that any data structure that uses pairs is probably a list.
- The printed representation of a pair uses a “.” to separate the car and the cdr elements.
  - (cons 3 4) => (3 . 4)
- But when printing a list, the complexity of the pair is suppressed for clarity when possible.
  - (cons 3 '()) => (3)

Printing pairs and lists

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

List structure

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

List structure and cons

- (list 2 (list 4 6)) => (2 (4 6))
- (cons 2 (list 4 6)) => (2 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
  - details of the implementation should not leak into the statement of the problem solution
- For example, a tree structure can be built in many different ways in many different languages

Further abstraction

- The more we can map into the problem domain the better
- A layer of abstraction can hide much or all of the messy details of implementation
  - easier to understand
  - easier to replace the implementation
- Lists are an abstraction implemented with pairs
- Trees are an abstraction implemented with lists

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: \((+ 1 (* 2 (- 3 5)))\)
- Expression tree:
  - \(+\)
  - \(1\)
  - \(*\)
  - \(2\)
  - \((-\)
  - \(3\)
  - \(5\)
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

Each node is represented by a 3-element list

Constructors and accessors

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

(def (operator exp)
  (car exp))

(def (left exp)
  (cadr exp))

(def (right exp)
  (caddr exp))

Evaluator

(def (eval-expr exp)
  (if (not (pair? exp))
      exp
      ((operator exp)
       (eval-expr (left exp))
       (eval-expr (right exp))))))

; note that this code expects the operators
; to be the actual functions, not text symbols
Symbols and expressions

- We've been using symbols and lists of symbols to refer to values of all kinds in our programs
  
  (+ a 3)
  (inc b)
- Scheme evaluates the symbols and 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

Manipulating symbols, not values

- What if we want to manipulate the symbols, and not the value of the symbols
  
  » perhaps evaluate after all the manipulation is done
- We 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

Special form: **quote**

(quote ⟨datum⟩)

or '⟨datum⟩

- This expression always evaluates to **datum**
  
  » datum is the external representation of the object
- The **quote** form tells Scheme to treat the given expression as a data object directly, rather than as an expression to be evaluated

Quote examples

(define a 1)

<table>
<thead>
<tr>
<th>a</th>
<th>=&gt; 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(quote a)</td>
<td>=&gt; a</td>
</tr>
</tbody>
</table>

(define b (+ a a))

<table>
<thead>
<tr>
<th>b</th>
<th>=&gt; 2</th>
</tr>
</thead>
</table>

(define c (quote (+ a b)))

<table>
<thead>
<tr>
<th>c</th>
<th>=&gt; (+ a b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(car c)</td>
<td>=&gt; +</td>
</tr>
<tr>
<td>(cadr c)</td>
<td>=&gt; a</td>
</tr>
<tr>
<td>(caddr c)</td>
<td>=&gt; b</td>
</tr>
</tbody>
</table>

| a is a symbol whose value is the number 1 |
| b is a symbol whose value is the number 2 |
| c is a symbol whose value is the list (+ a b) |
quote can be abbreviated: '

'a  =>  a
'(a b)  =>  (+ a b)
'()  =>  ()
(null? '())  =>  #t

'(1 2 3 4)  =>  (1 2 3 4)
'(a (b (c)))  =>  (a (b (c)))
(car '(1 2 3 4))  =>  1
(cdr '(1 2 3 4))  =>  (2 3 4)

lists are easily expressed as quoted objects

Building lists with symbols

What would the interpreter print in response to evaluating each of the following expressions?

(list 'a 'b)
(cons 'a (list 'b))
(cons 'a (cons 'b '()))
(cons 'a '(b))
'(a b)

Building lists with symbols

Comparing items

Scheme provides several different means of comparing objects

Do two numbers have the same value?

- (= a b)  use (= ...) for numbers

Are two objects the same object in memory?

- (eq? a b)

Do two objects have the same value?

- (eqv? a b)  use (eqv? ...) for everything else

Do the corresponding elements have the same values?

- (equal? list-a list-b) applies eqv? recursively
(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))         => (a)
(member '(1 3) '(1 (1 3) 3)) => ((1 3) 3)
(member 'b '(a (b) c))       => #f
(member '(b) '(a (b) c))     => ((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

logical expression tree
(1*(2*(3-5)))

our data structure
'( + 1 (* 2 ( - 3 5 )))
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-expr '(+ 1 2))

(eval-op (+ 1 2))

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 - * +)