Hierarchical Structures

CSE 413, Autumn 2002
Programming Languages

http://www.cs.washington.edu/education/courses/413/02au/

Readings and References

• Reading
  » Section 2.2.2, Structure and Interpretation of Computer Programs, by Abelson, Sussman, and Sussman

• Other References

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 are themselves lists
    » (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"
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)
- `(cons 3 '()) => (3)

But when printing a list, the complexity of the pair is suppressed for clarity when possible:

- `(cons 3 '()) => (3)

This is a valid data structure,
but it is not a well formed list.

List structure and `cons`

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

This is a well formed list.

List structure

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

This is a valid data structure,
Recursive tree structure

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

• This list has two elements
  » the literal 2 and the list (4 6)
• The sublist also has two elements
  » the literals 4 and 6
• We can think of lists, and lists of lists, as tree structures
  » all the elements in one list are siblings

(depth x)

; x is a tree node. It is defined by a list that contains the node at this entry, plus all the sibling tree nodes to the right of this node.
; The value at this node is (car x).
; The list of siblings to the right is (cdr x).

(define (depth x)
  (cond ((null? x) 0)
        ((not (pair? x)) 0)
        (else (max (+ 1 (depth (car x)))
                   (depth (cdr x))))))

(fringe x)

; pick the leaves off a tree defined as lists of lists
(define (fringe m)
  (cond ((null? m) m)
        ((not (pair? m)) (list m))
        (else (append (fringe (car m)) (fringe (cdr m))))))

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 of a pair structure
• Trees are an abstraction of a list structure
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

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)
Constructors and accessors

(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))

(define a (make-exp + 1 2))

Evaluator

(define (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