## Topic #5: Hierarchical Structures

CSE 413, Autumn 2004

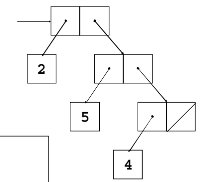
Programming Languages

http://www.cs.washington.edu/education/courses/413/04au/

1

## Review: sum the items in a list

```
(add-items (list 2 5 4))
```

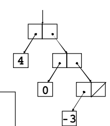



```
(define (add-items m)
  (if (null? m)
      0
      (+ (car m) (add-items (cdr m)))))
```

2

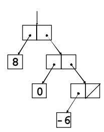
## Review: multiply each list element by 2

```
(double-all (list 4 0 -3))
```




```
(define (double-all m)
  (if (null? m)
      '()
      (cons (* 2 (car m)) (double-all (cdr m)))))
```

```
(cons 8 (cons 0 (cons -6 '())))
```




3

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

(define (find-max m)

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

Example: (concat (list 1 2 3) (list 7 8 9) ) → (1 2 3 7 8 9)

(define (concat x y)

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

(remove-neg (list 1 -7 8 -9)) → (1 8)

(define (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)

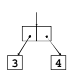
## References

- Section 2.2.2, 2.3.1, *Structure and Interpretation of Computer Programs*
- Sections 4.1.2, 6.1, 6.3.3, *Revised[5] Report on the Algorithmic Language Scheme (R5RS)*

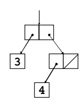


## Printing pairs and lists

`(cons 3 4) => (3 . 4)`



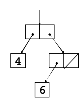

`(cons 3 (cons 4 '())) => (3 4)`



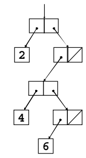



## List structure

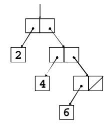
`(list 4 6) => (4 6)`




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




`(list 2 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

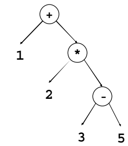


## Expression tree example

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

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

expression tree



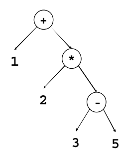



## Represent expression with a list

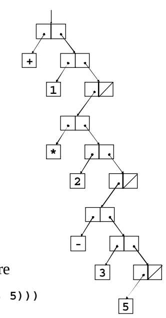
- For this example, we are restricting the type of expression somewhat
  - » Operators in the tree are all binary
  - » All of the leaves (operands) are numbers
- Each node is represented by a 3-element list
  - » (operator left-operand right-operand)
- Recall that the operands can be
  - » numbers (explicit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



## Expressions as trees, trees as lists




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



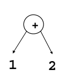

our data structure
`(list + 1 (list * 2 (list - 3 5)))`

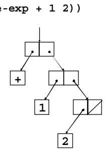


## Constructors and accessors

```
(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))`

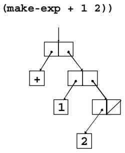





## Evaluator

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




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

## 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
```

19

## Quote examples

```
(define a 1)
a                    =>
(quote a)            =>

(define b (+ a a))
b                    =>

(define c (quote (+ a b)))
c                    =>
(car c)              =>
(cadr c)             =>
(caddr c)            =>
```

20

## 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)
```

21

## 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)
```

22

## Building lists with symbols

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

```
(cons '(a) '(b))


(list '(a) '(b))
```

23

## 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)`

24

## `(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))      =>
```

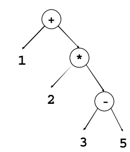
25

## Recall: Expression tree example

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

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

expression tree



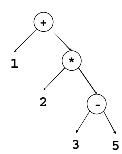

26

## Represent expression with a list

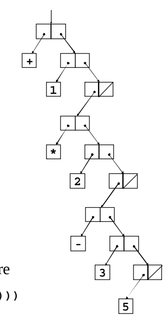
- Each node is represented by a 3-element list
  - » (operator left-operand right-operand)
- Operands can be
  - » numbers (explicit values)
  - » other expressions (lists)
- In previous implementation, operators were the actual procedures
  - » This time, we will use symbols throughout

27

## Expressions as trees, trees as lists




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



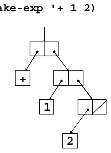

our data structure
`'(+ 1 (* 2 (- 3 5)))`

28

## 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))
```

`(make-exp '+ 1 2)`

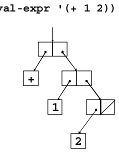



29

## `eval-op` and `eval-expr`

```
(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)))))
```

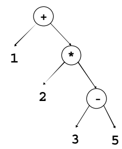
`(eval-expr '(+ 1 2))`




30

# 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)))

31

# 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 - * +)
```

32