Symbols

CSE 413, Autumn 2002
Programming Languages

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

Readings and References

- Reading
  - » Section 2.3.1, *Structure and Interpretation of Computer Programs*, by Abelson, Sussman, and Sussman
- Other References
  - » Sections 4.1.2, 6.1, 6.3.3, *Revised^5 Report on the Algorithmic Language Scheme (R5RS)*

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

a is a symbol whose value is the number 1

```
(define b (+ a a))
b                      => 2
```

b is a symbol whose value is the number 2

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

c is a symbol whose value is the list (+ a b)

quote can be abbreviated: '

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

a single quote has the exact same effect as the `quote` form

```
'(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)
```

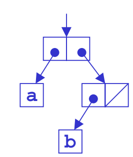
lists are easily expressed as quoted objects

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)
```

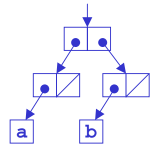
(a b)



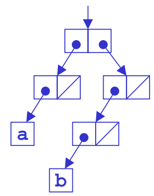
Building lists with symbols

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

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




```
(list '(a) '(b))          ((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?
    `(eq? a b), (eqv? a b)`
  - » Are the corresponding elements the same objects? Comparison is done recursively if elements are lists.
    `(equal? list-a list-b)`

(member item s)

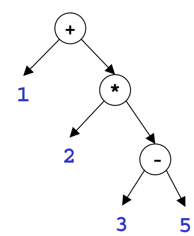
```
; find an item of any kind in a list s
; return the sublist that starts with the item
; or return #f

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


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


Recall: Expression tree example

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

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

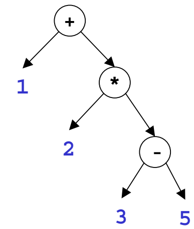
expression tree



Represent expression with a list

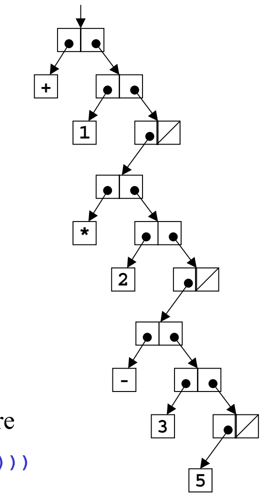
- Each node is represented by a 3-element list
  - » (operator left-operand right-operand)
- Operands can be
  - » numbers (explicit values)
  - » other expressions (lists)
- In previous implementation, operators were the actual procedures
  - » This time, we will use symbols throughout

Expressions as trees, trees as lists



logical expression tree

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



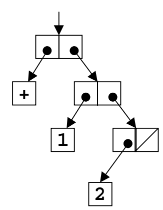
our data structure

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

eval-expr

```
(define (eval-op op)
  (cond
    ((eq? op '+) +)
    ((eq? op '-) -)
    ((eq? op '/) /)
    ((eq? op '*) *)))

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

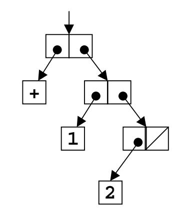
`(eval-expr '(+ 1 2))`



evaluator

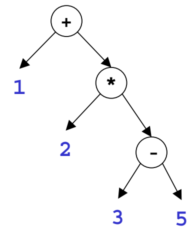
```
(define (evaluator exp)
  (if (not (pair? exp))
      exp
      ((eval (operator exp))
       (eval-expr (left exp))
       (eval-expr (right exp)))))
```

`(evaluator '(+ 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)))`

Traverse the expression tree

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

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

```
(in-order f)
(1 + 2 * 3 - 5)
```

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

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