

6.001 SICP Interpretation

- Parts of an interpreter
- Arithmetic calculator
- Names
- Conditionals and if
- Store procedures in the environment
- Environment as explicit parameter
- Defining new procedures

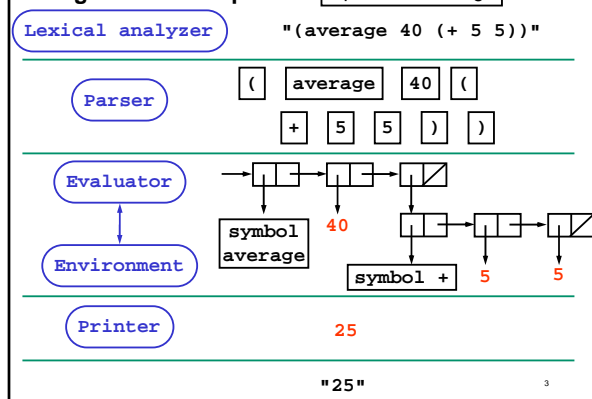
1

Why do we need an interpreter?

- Abstractions let us bury details and focus on use of modules to solve large systems
- Need to unwind abstractions at execution time to deduce meaning
- Have seen such a process – Environment Model
- Now want to describe that process as a procedure

2

Stages of an interpreter



3

Role of each part of the interpreter

- **Lexical analyzer**
 - break up input string into "words" called tokens
- **Parser**
 - convert linear sequence of tokens to a tree
 - like diagramming sentences in elementary school
 - also convert self-evaluating tokens to their internal values
 - #f is converted to the internal false value
- **Evaluator**
 - follow language rules to convert parse tree to a value
 - read and modify the **environment** as needed
- **Printer**
 - convert value to human-readable output string

4

Goal of lecture

- Implement an interpreter
- Only write evaluator and environment
 - use scheme's **reader** for lexical analysis and parsing
 - use scheme's **printer** for output
 - to do this, our language must look like scheme
- Call the language **scheme***
 - All names end with a star
- Start with interpreter for simple arithmetic expressions
- Progressively add more features

5

1. Arithmetic calculator

Want to evaluate arithmetic expressions of two arguments, like:

(plus* 24 (plus* 5 6))

6

1. Arithmetic calculator

```
(define (tag-check e sym) (and (pair? e) (eq? (car e) sym)))
(define (sum? e) (tag-check e 'plus*))

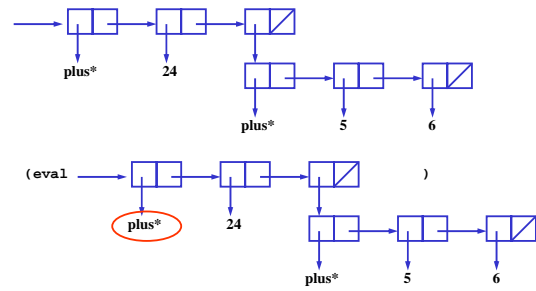
(define (eval exp)
  (cond
    ((number? exp) exp)
    ((sum? exp) (eval-sum exp))
    (else (error "unknown expression " exp))))

(define (eval-sum exp)
  (+ (eval (cadr exp)) (eval (caddr exp))))

(eval '(plus* 24 (plus* 5 6)))
```

7

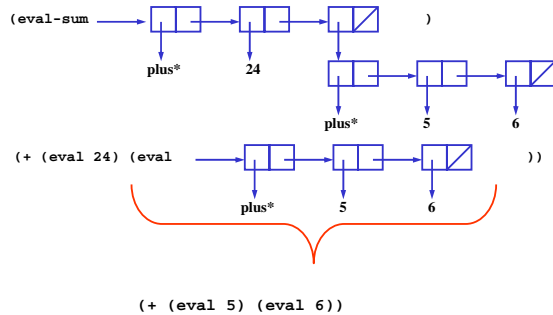
We are just walking through a tree ...



sum? checks the tag

8

We are just walking through a tree ...



9

1. Arithmetic calculator

```
(plus* 24 (plus* 5 6))
```

- What are the argument and return values of `eval` each time it is called in the evaluation of line 17?

(eval 5)	5	(eval 6)	6
(eval-sum '(plus* 5 6))			11
(eval 24)	24	(eval '(plus* 5 6))	11
(eval-sum '(plus* 24 (plus* 5 6)))			35
(eval '(plus* 24 (plus* 5 6)))			35

10

1. Things to observe

- `cond` determines the expression type
- no work to do on numbers
 - scheme's reader has already done the work
 - it converts a sequence of characters like "24" to an internal binary representation of the number 24
- `eval-sum` recursively calls `eval` on both argument expressions

11

More Complex Expressions

```
(plus* 24 (plus* 5 6))
```

```
(plus* (plus* 43 (plus* 24 (plus* 5 6)))
      (plus* 43 (plus* 24 (plus* 5 6))))
```

```
(plus* (plus* 43 (plus* 24 (plus* 5 6)))
      (plus* 43 (plus* 24 (plus* 5 6))))
```

```
(define x* (plus* 5 6))
(define y* (plus 24 x*))
(define z* (plus 43 y*))
(plus z* z*)
```

12

2. Names

- Extend the calculator to store intermediate results as named values

```
(define* x* (plus* 4 5))  store result as x*
(plus* x* 2)              use that result
```
- Store bindings between names and values in a table
- What are the argument and return values of `eval` each time it is called in lines 36 and 37?
 - Show the environment each time it changes during evaluation of these two lines.

13

2. Names

```
(define (define? exp) (tag-check exp 'define*))

(define (eval exp)
  (cond
    ((number? exp) exp)
    ((sum? exp) (eval-sum exp))
    ((symbol? exp) (lookup exp))
    ((define? exp) (eval-define exp))
    (else
     (error "unknown expression " exp))))

; table ADT from prior lecture:
; make-table      void -> table
; table-get       table, symbol -> (binding | null)
; table-put!      table, symbol, anytype -> undef
; binding-value   binding -> anytype

(define environment (make-table))
```

14

2. Names ...

```
(define (lookup name)
  (let ((binding (table-get environment name)))
    (if (null? binding)
        (error "unbound variable: " name)
        (binding-value binding))))

(define (eval-define exp)
  (let ((name (cadr exp))
        (defined-to-be (caddr exp)))
    (table-put! environment name (eval defined-to-be))
    'undefined))

(eval '(define* x* (plus* 4 5)))
(eval '(plus* x* 2))
```

How many times is `eval` called in these two evaluations?

15

Evaluation of page 2 lines 36 and 37

```
(eval '(define* x* (plus* 4 5)))
  (eval '(plus* 4 5))
    (eval 4) ==> 4
    (eval 5) ==> 5
  ==> 9
==> undefined

(eval '(plus* x* 2))
  (eval 'x*) ==> 9
  (eval 2) ==> 2
==> 11
```

names values
x* 9

16

2. Things to observe

- Use scheme function `symbol?` to check for a name
 - the reader converts sequences of characters like `"x"` to symbols in the parse tree
- Can use any implementation of the `table` ADT
- `eval-define` recursively calls `eval` on the second subtree but not on the first one
- `eval-define` returns a special undefined value

17

3. Conditionals and if

- Extend the calculator to handle predicates and `if`:

```
(if* (greater* y* 6) (plus* y* 2) 15)
```
- `greater*` an operation that returns a boolean
- `if*` an operation that evaluates the first subexp, checks if value is true or false
- What are the argument and return values of `eval` each time it is called in line 32 (above)?

18

```

(define (greater? exp) (tag-check exp 'greater*))
(define (if? exp) (tag-check exp 'if*))

(define (eval exp)
  (cond ...
    ((greater? exp) (eval-greater exp))
    ((if? exp) (eval-if exp))
    (else (error "unknown expression " exp))))

(define (eval-greater exp)
  (> (eval (cadr exp)) (eval (caddr exp))))

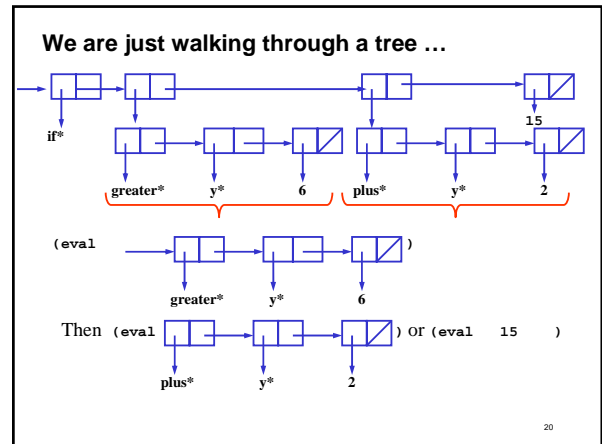
(define (eval-if exp)
  (let ((predicate (cadr exp))
        (consequent (caddr exp))
        (alternative (caddrdr exp)))
    (let ((test (eval predicate)))
      (cond
        ((eq? test #t) (eval consequent))
        ((eq? test #f) (eval alternative))
        (else (error "predicate not boolean: "
                     predicate))))))

(eval '(define* y* 9))
(eval '(if* (greater* y* 6) (plus* y* 2) 15))

```

3. Conditionals and If

Note: if* is stricter than Scheme's if



Evaluation of page 3 line 32

```

(eval '(if* (greater* y* 6) (plus* y* 2) 15))
  (eval '(greater* y* 6))
    (eval 'y*) ==> 9
    (eval 6) ==> 6
  ==> #t
  (eval '(plus* y* 2))
    (eval 'y*) ==> 9
    (eval 2) ==> 2
  ==> 11
==> 11

```

- ### 3. Things to observe
- `eval-greater` is just like `eval-sum` from page 1
 - recursively call `eval` on both argument expressions
 - call scheme `>` to compute value
 - `eval-if` does not call `eval` on all argument expressions:
 - call `eval` on the predicate
 - call `eval` on the consequent or on the alternative but not both
 - this is the mechanism that makes if* a _____.

- ### 4. Store operators in the environment
- Want to add lots of operators but keep `eval` short
 - Operations like `plus*` and `greater*` are similar
 - evaluate all the argument subexpressions
 - perform the operation on the resulting values
 - Call this standard pattern an **application**
 - Implement a single case in `eval` for all applications
 - Approach:
 - `eval` the first subexpression of an application
 - put a name in the environment for each operation
 - value of that name is a **procedure**
 - **apply** the procedure to the **operands**

4. Store operators in the environment

```

(define (application? e) (pair? e))

(define (eval exp)
  (cond
    ((number? exp) exp)
    ((symbol? exp) (lookup exp))
    ((define? exp) (eval-define exp))
    ((if? exp) (eval-if exp))
    ((application? exp) (apply (eval (car exp))
                               (map eval (cdr exp))))
    (else (error "unknown expression " exp))))

(define scheme-apply apply) ;; rename scheme's apply so we can reuse the name

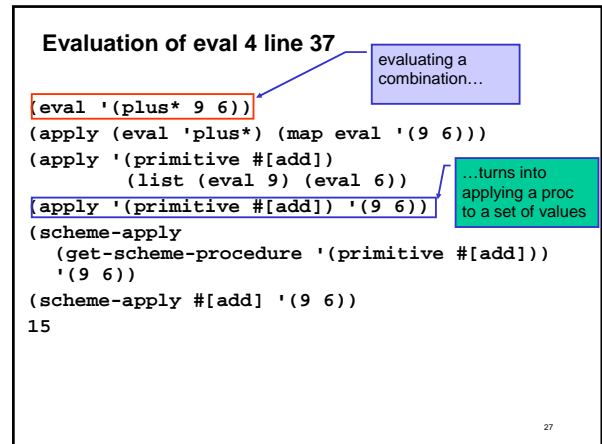
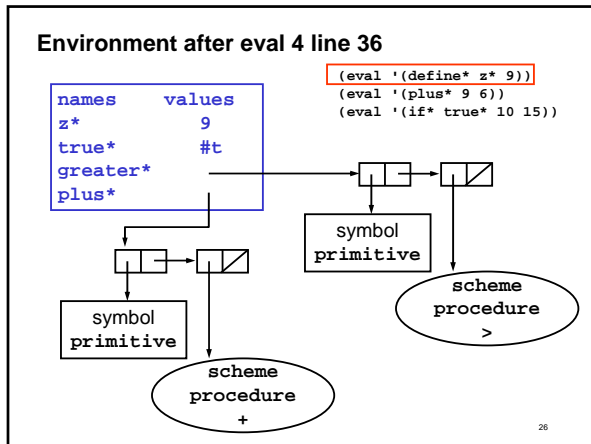
(define (apply operator operands)
  (if (primitive? operator)
      (scheme-apply (get-scheme-procedure operator) operands)
      (error "operator not a procedure: " operator)))

;; primitive: an ADT that stores scheme procedures

(define prim-tag 'primitive)
(define (make-primitive scheme-proc)(list prim-tag scheme-proc))
(define (primitive? e) (tag-check e prim-tag))
(define (get-scheme-procedure prim) (cadr prim))

(define environment (make-table))
(table-put! environment 'plus* (make-primitive +))
(table-put! environment 'greater* (make-primitive >))
(table-put! environment 'true* #t)

```



Evaluation of eval 4 line 38

```

(eval '(if* true* 10 15))
(eval-if '(if* true* 10 15))
(let ((test (eval 'true*))) (cond ...))
(let ((test (lookup 'true*))) (cond ...))
(let ((test #t)) (cond ...))
(eval 10)
10
  
```

Apply is never called!

- ### 4. Things to observe
- applications must be last case in eval
 - no tag check
 - apply is never called in line 38
 - applications evaluate all subexpressions
 - expressions that need special handling, like if*, gets their own case in eval

- ### 5. Environment as explicit parameter
- change from


```
(eval '(plus* 6 4))
```

 to


```
(eval '(plus* 6 4) environment)
```
 - all procedures that call eval have extra argument
 - lookup and define use environment from argument
 - No other change from evaluator 4
 - Only nontrivial code: case for application? in eval

5. Environment as explicit parameter

```

(define (eval exp env)
  (cond
    ((number? exp) exp)
    ((symbol? exp) (lookup exp env))
    ((define? exp) (eval-define exp env))
    ((if? exp) (eval-if exp env))
    ((application? exp) (apply (eval (car exp) env)
                               (map (lambda (e) (eval e env))
                                   (cdr exp)))))
  (else (error "unknown expression " exp))))

(define (lookup name env)
  (let ((binding (table-get env name)))
    (if (null? binding)
        (error "unbound variable: " name)
        (binding-value binding))))

(define (eval-define exp env)
  (let ((name (cadr exp))
        (defined-to-be (caddr exp)))
    (table-put! env name (eval defined-to-be env))
    'undefined))

(define (eval-if exp env)
  (let ((predicate (cadr exp))
        (consequent (caddr exp))
        (alternative (caddr exp)))
    (let ((test (eval predicate env)))
      (cond
        ((eq? test #t) (eval consequent env))
        ((eq? test #f) (eval alternative env))
        (else (error "predicate not boolean: "
                    predicate))))))
  
```

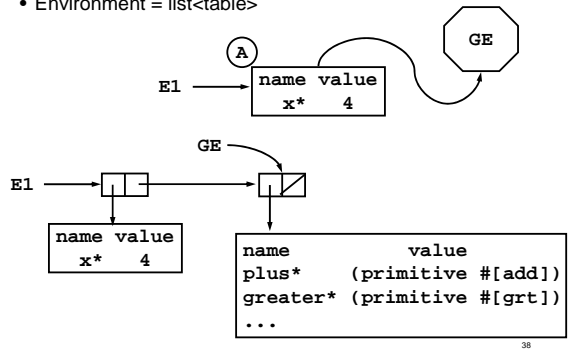
This change is boring! Exactly the same functionality as #4.

```

(eval '(define* z* (plus* 4 5)
      environment)
      (eval '(if* (greater* z* 6) 10 15)
            environment))
  
```


Implementation of environment model

- Environment = list<table>



38

; Environment model code (part of eval 6)

```

; Environment = list<table>
(define (extend-env-with-new-frame names values env)
  (let ((new-frame (make-table)))
    (make-bindings! names values new-frame)
    (cons new-frame env)))

(define (make-bindings! names values table)
  (for-each
   (lambda (name value) (table-put! table name value))
   names values))

; the initial global environment
(define GE
  (extend-env-with-new-frame
   (list 'plus* 'greater*)
   (list (make-primitive +) (make-primitive >))
   nil))

; lookup searches the list of frames for the first match
(define (lookup name env)
  (if (null? env)
      (error "unbound variable: " name)
      (let ((binding (table-get (car env) name)))
        (if (null? binding)
            (lookup name (cdr env))
            (binding-value binding)))))

; define changes the first frame in the environment
(define (eval-define exp env)
  (let ((name (cadr exp))
        (defined-to-be (caddr exp)))
    (table-put! (car env) name (eval defined-to-be env))
    'undefined))

(eval '(define* twice* (lambda* (x*) (plus* x* x*))) GE)
(eval '(twice* 4) GE)

```

39

Summary

- Cycle between eval and apply is the core of the evaluator
 - eval calls apply with operator and argument values
 - apply calls eval with expression and environment
 - no pending operations on either call
 - an iterative algorithm if the expression is iterative
- What is still missing from **scheme*** ?
 - ability to evaluate a sequence of expressions
 - data types other than numbers and booleans

Everything in these lectures would still work if you deleted the stars from the names!

40