6.001 SICP
Interpretation

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

Why do we need an interpreter?
• Abstractions let us bury details and focus on use of modules to solve large systems
• We need a process to unwind abstractions at execution time to deduce meaning
• We have already seen such a process – the Environment Model
• Now want to describe that process as a procedure

Stages of an interpreter

Lexical analyzer
Parser
Evaluator
Environment
Printer

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
  • e.g., #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

Goal of today’s 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 resemble Scheme
• Call the language scheme*
  • All names end with a star to distinguish from Scheme names
• Start with interpreter for simple arithmetic expressions
  • Progressively add more features

1. Arithmetic calculator

Want to evaluate arithmetic expressions of two arguments, like:

\[(\text{plus}\ast\text{ 24} \ (\text{plus}\ast\text{ 5} \ 6))\]
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)))

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

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

We are just walking through a tree ...

We are just walking through a tree ...

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 this expression?

1. Things to observe

2. Names
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))

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* 4 5))

How many times is eval called in these two evaluations?

Evaluation of page 2 lines 36 and 37

- Show argument and return values of eval for each call
- Show the environment each time it changes

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

(eval '(plus* x* 2))
  (eval 'x*) == 9
  (eval 2) == 2

===> undefined

names values
x* 9

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, and checks if its value is true or false

- What are the argument and return values of eval each time it is called in the expression above?
We are just walking through a tree ...

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** either on the consequent or on the alternative
  - but not both
  - this is the mechanism that makes **if** a special form

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

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

```
4. Store operators in the environment

(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)))
```

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>z*</td>
<td>9</td>
</tr>
<tr>
<td>true*</td>
<td>#t</td>
</tr>
<tr>
<td>greater*</td>
<td></td>
</tr>
<tr>
<td>plus*</td>
<td></td>
</tr>
</tbody>
</table>

Environment after eval 4 line 36

```
(eval '(define* z* 9))
(eval '(plus* 9 6))
(eval '(if* true* 10 15))
```

```
(define environment (make-table))
(table-put! environment ‘greater* (make-primitive ‘greater*))
(table-put! environment ‘plus* (make-primitive ‘plus*))
```
Evaluation of eval 4 line 37

```
(eval '(plus* 9 6))
(apply (eval 'plus*) (map eval '(9 6)))
(apply '(primitive #*[add]) (list (eval 9) (eval 6)))
(scheme-apply (get-scheme-procedure '(primitive #*[add])) (9 6))
(scheme-apply #*[add] '(9 6))
```

15

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 the 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*, get 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 now have extra argument
  - lookup and define use environment from argument
  - No other change from evaluator 4
  - Only nontrivial code: case for apply? in eval

6. Defining new procedures

- Want to add new procedures
- For example, a scheme* procedure:
  ```
  (define* twice* (lambda* (x*) (plus* x* x*))
  ```
  `(twice* 4)

- Strategy:
  - Add a case for lambda* to eval
  - the value of lambda* is a compound procedure
  - Extend apply to handle compound procedures
  - Implement environment model
6. Defining new procedures

(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))
   ((lambda? exp)      (eval-lambda exp env))
   ((application? exp) (apply (eval (car exp) env) (map (lambda (e) (eval e env)) (cdr exp))))
   (else (error "unknown expression " exp))))

(define (eval-lambda exp env)
  (let ((args (cadr exp))
         (body (caddr exp)))
    (make-compound args body env)))

(define (apply operator operands)
  (cond ((primitive? operator)
          (scheme-apply (get-scheme-procedure operator) operands))
       ((compound? operator)
        (eval (body operator)
              (extend-env-with-new-frame
               (parameters operator) operands
               (env operator))))
       (else (error "operator not a procedure: " operator))))

;; ADT that implements the "double bubble"
(define compound-tag 'compound)
(define (make-compound parameters body env)
  (list compound-tag parameters body env))
(define (compound? exp) (tag-check exp compound-tag))
(define (parameters compound) (cadr compound))
(define (body compound)       (caddr compound))
(define (env compound)        (cadddr compound))

Implementation of lambda*

(eval '(lambda* (x*) (plus* x* x*)) GE)
(make-compound '(x*) '(plus* x* x*) GE)
(list 'compound '(x*) '(plus* x* x*) GE)

Implementation of apply

(eval '(twice* 4) GE)
(apply (eval 'twice* GE)
       (map (lambda (e) (eval e GE)) '(4)))
(apply (list 'compound '(x*) '(plus* x* x*) GE)
       '(4))
(eval '(plus* x* x*)
     (extend-env-with-new-frame '(x*) '(4) GE)
     E1)

Implementation of environment model

- Environment = list<

A

name value
x*  4
GE

E1
Environment model code (part of eval 6)

```scheme
(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))))

; 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)
```

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

Cute Punchline

- Everything in these lectures would still work if you deleted the stars from the names.
- We just wrote (most of) a Scheme interpreter in Scheme.
- Seriously nerdy, eh?
  - The language makes things explicit
    - e.g., procedures and procedure app in environment
  - More generally
    - Writing a precise definition for what the Scheme language means
    - Describing computation in a computer language forces precision and completeness
    - Sets the foundation for exploring variants of Scheme