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

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

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

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

1. Arithmetic calculator

Want to evaluate arithmetic expressions of two arguments, like:

(plus* 24 (plus* 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

We are just walking through a tree ...

(eval-sum (plus* 24 (plus* 5 6)))

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?

More Complex Expressions

(plus* 24 (plus* 5 6))

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

We are just walking through a tree ...

+ (eval 24) (eval )
+ (eval 5) (eval 6)

sum? checks the tag
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.

```scheme
(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?

```
(names values
x* 9
)
```

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
3. Conditionals and If

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` 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

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

```
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 (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))))
```
Evaluation of eval 4 line 36

\[(\text{eval } '(\text{define } z* 9))\]
\[(\text{eval } '(\text{plus* } z* 6))\]
\[(\text{eval } '(\text{if* } \text{true* } 10 15))\]

```
(names values
  true*       #t
  greater*   
  plus*)
```

Evaluation of eval 4 line 37

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

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
```

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*, get their own case in eval

5. Environment as explicit parameter

- Change from
  
  \[(\text{eval } '(\text{plus* } 4 5))\]

  to

  \[(\text{eval } '(\text{plus* } 4 5) \text{ environment})\]

- 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

This change is boring! Exactly the same functionality as 4.
6. Defining new procedures

- Want to add new procedures
- For example, a scheme procedure:

   ```scheme
   (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

---

**Implementation of lambda**

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

This data structure is a procedure!

---

**Defining a named procedure**

```scheme
(eval '(define* twice* (lambda* (x*) (plus* x* x*)) GE)
```

---

**Implementation of apply (1)**

```scheme
(eval '(twice* 4) GE some-other-environment)
```

---

**Implementation of apply (2)**

```scheme
(eval '(plus* x* x*) EI)
(scheme-apply #\[add\] '4 4)
```

---

```scheme
GE
```
Implementation of environment model

- Environment = list<table>

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x*</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>plus*</td>
<td>(primitive #[add])</td>
</tr>
<tr>
<td>greater*</td>
<td>(primitive #[grt])</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

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-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)

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