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

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
  – `#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. We are just walking through a tree ...

(sum? checks the tag)

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

1. More Complex Expressions

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

2. Evaluation of define*

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

2. 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

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. Evaluation of if*

\[
\text{eval}\ '\left(\text{if}\ (\text{greater}\ y\ 6)\ (\text{plus}\ y\ 2)\ 15\right)\\
= \text{eval}\ '\left(\text{greater}\ y\ 6\right)\\
= \text{eval}\ 'y*\\
= 9\\
= \text{eval}\ 6\\
= \#t\\
= \text{eval}\ '\left(\text{plus}\ y\ 2\right)\\
= \text{eval}\ 'y*\\
= 9\\
= \text{eval}\ 2\\
= 2\\
= 11\\
= 11
\]

### 4. Store operators in the environment

- Want to add lots of operators but keep \text{eval} short
- Operations like \text{plus*} and \text{greater*} are similar
- \text{eval} the first subexpression of an application
- Value of that name is a procedure
- Approach:
  - \text{eval} the first subexpression of an application
  - put a name in the environment for each operation
  - value of that name is a procedure
  - apply procedure to operands

#### 4. Store operators in the environment

\[
\begin{align*}
\text{define} & \quad \text{application?} e (\text{pair?} e) \\
\text{define} & \quad \text{eval}\ e \\
& \quad \text{(cond} \ldots \\
& \quad \left\{ \begin{array}{l}
(\text{number?} e) \quad \text{exp} \\
(\text{symbol?} e) \quad \text{lookup}\ e \\
(\text{define?} e) \quad \text{eval-define}\ e \\
(\text{if?} e) \quad \text{eval-if}\ e \\
(\text{application?} e) \quad \text{apply}\ (\text{eval}\ (\text{car}\ e)) \quad \text{map}\ \text{eval}\ (\text{cdr}\ e)) \\
\ldots \end{array} \right) \\
& \quad \text{else} \quad \text{error}\ "\text{unknown expression } e"
\end{align*}
\]
4. Environment after previous page

<table>
<thead>
<tr>
<th>names</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>true*</td>
<td>#t</td>
</tr>
<tr>
<td>greater*</td>
<td></td>
</tr>
<tr>
<td>plus*</td>
<td></td>
</tr>
</tbody>
</table>

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

4. Evaluation of plus*

\[
\text{eval '(plus* 9 6)}\]
\[
\text{(apply (eval 'plus*) (map eval '(9 6)))}
\]
\[
\text{(apply '(primitive #[add]) (list (eval 9) (eval 6)))}
\]
\[
\text{apply '(primitive #[add]) '(9 6)}
\]
\[
\text{(scheme-apply '(primitive #[add]) '(9 6))}
\]
\[
\text{(scheme-apply #[add] '(9 6))}
\]
\[
\text{15}
\]

4. Evaluation of plus*

\[
\text{eval '(if* true* 10 15))}
\]
\[
\text{(eval-if '(if* true* 10 15))}
\]
\[
\text{(let ((test (eval 'true*))) (cond ...))}
\]
\[
\text{(let ((test (lookup 'true*))) (cond ...))}
\]
\[
\text{(let ((test #t)) (cond ...)}
\]
\[
\text{(eval 10)}
\]
\[
\text{10}
\]
\[
\text{Apply is never called!}
\]

4. Things to observe

- applications must be last case in eval
  - no tag check
- apply is never called
  - applications evaluate all subexpressions
  - expressions that need special handling, like if*, gets their own case in eval

5. Environment as explicit parameter

- change from
  \[
  \text{(eval '(plus* 6 4))}
  \]
  to
  \[
  \text{(eval '(plus* 6 4) environment)}
  \]
- 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

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

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. Implementation of lambda*

(eval '(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!

6. Defining a named procedure

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

6. Implementation of apply (1)

(apply (eval 'twice* GE) (extend-env-with-new-frame '(x*) '4 GE))

6. Implementation of apply (2)

```scheme
(apply (eval 'plus* E1)
  (map (lambda (e) (eval e E1)) '(x* x*))

(apply '(primitive #[add]) (list (eval 'x* E1)
  (eval 'x* E1)))

(apply '(primitive #[add]) '(4 4))

(scheme-apply #\[add] '(4 4))
```

8

6. Implementation of environment model

- 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 (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 (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