Today’s Lecture: the Metacircular Evaluator

- Today we’ll look at a complete Scheme interpreter written in Scheme.
- Why?
  - An interpreter makes things explicit, e.g., procedures and procedure application in the environment model.
  - Provides a precise definition for what the Scheme language means.
  - Describing a process in a computer language forces precision and completeness.
  - Sets the foundation for exploring variants of Scheme.
    - Today: lexical vs. dynamic scoping.
    - Next time: eager vs. lazy evaluation.

The Core Evaluator

1. eval/apply
2. core

Last Lecture

- Last time, we built up an interpreter for a new language, scheme*
  - Conditionals (if*)
  - Names (define*)
  - Applications
  - Primitive procedures
  - Compound procedures (lambda*)
- Everything still works if you delete the stars from the names.
  - So we actually wrote (most of) a Scheme interpreter in Scheme.
  - Seriously nerdly, eh?

Metacircular evaluator

(Scheme implemented in Scheme)

```
(define (m-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp) (make-procedure (lambda-parameters exp)
                                   (lambda-body exp)
                                   env))
        ((begin? exp) (eval-sequence (begin-actions exp) env))
        ((cond? exp) (cond (if exp env)))
        ((application? exp) (m-apply (m-eval (operator exp) env)
                        (list-of-values (operands exp) env)))))
```

Pieces of Eval&Apply

```
(define (m-eval exp env)
  (cond ((self-evaluating? exp) exp)
        (variable? exp) (lookup-variable-value exp env))
        (quoted? exp) (text-of-quotation exp))
        (assignment? exp) (eval-assignment exp env))
        (definition? exp) (eval-definition exp env))
        (if? exp) (eval-if exp env))
        (lambda? exp) (make-procedure (lambda-parameters exp)
                                   (lambda-body exp)
                                   env))
        (begin? exp) (eval-sequence (begin-actions exp) env))
        (cond? exp) (cond (if exp env)))
        (application? exp) (m-apply (m-eval (operator exp) env)
                        (list-of-values (operands exp) env)))))
```

primitives
special forms
application
Pieces of Eval&Apply

(define (list-of-values exps env)
  (cond ((no-operands? exps) `())
        (else (cons (m-eval (first-operand exps) env)
                    (list-of-values (rest-operands exps) env)))))

Pieces of Eval&Apply

(define (m-apply procedure arguments)
  (cond ((primitive-procedure? procedure)
          (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment (procedure-parameters procedure) arguments
           (procedure-environment procedure)))
         (else (error "Unknown procedure type -- APPLY" procedure))))

Side comment – procedure body

• The procedure body is a sequence of one or more expressions:

(define (foo x)
  (do-something (+ x 1))
  (* x 5))

• In m-apply, we eval-sequence the procedure body.

Pieces of Eval&Apply

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (m-eval (first-exp exps) env))
        (else (m-eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))

Pieces of Eval&Apply

(define (m-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp) (make-procedure (lambda-parameters exp)
                                      (lambda-body exp)
                                      env))
        ((begin? exp) (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (m-apply (m-eval (operator exp) env)
                  (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))

Pieces of Eval&Apply

(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                        (m-eval (assignment-value exp) env) env))

Pieces of Eval&Apply

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (m-eval (definition-value exp) env) env))
Pieces of Eval&Apply

(define (m-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
          (make-procedure (lambda-parameters exp)
                          (lambda-body exp) env))
        ((begin? exp) (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
          (m-apply (m-eval (operator exp) env)
                   (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))

Pieces of Eval&Apply

(define (eval-if exp env)
  (if (m-eval (if-predicate exp) env)
      (m-eval (if-consequent exp) env)
      (m-eval (if-alternative exp) env)))

Syntactic Abstraction

- Semantics
  - What the language **means**
  - Model of computation
- Syntax
  - Particulars of writing expressions
  - E.g. how to signal different expressions
- Separation of syntax and semantics: allows one to easily alter syntax

Basic Syntax

(define (tagged-list? exp tag)
  (and (pair? exp) (eq? (car exp) tag)))

• Routines to detect expressions
  (define (if? exp) (tagged-list? exp 'if))
  (define (lambda? exp) (tagged-list? exp 'lambda))
  (define (application? exp) (pair? exp))

• Routines to get information out of expressions
  (define (operator app) (car app))
  (define (operands app) (cdr app))

• Routines to manipulate expressions
  (define (no-operands? args) (null? args))
  (define (first-operand args) (car args))
  (define (rest-operands args) (cdr args))

Example – Changing Syntax

• Suppose you wanted a “verbose” application syntax, i.e.,
  instead of
  
        (<proc> <arg1> <arg2> ...)

  use

        (CALL <proc> ARGS <arg1> <arg2> ...)

• Changes – only in the syntax routines!

  (define (application? exp) (tagged-list? exp 'CALL))
  (define (operator app) (cadr app))
  (define (operands app) (cdddr app))

Implementing "Syntactic Sugar"

• Idea:
  - Easy way to add alternative/convenient syntax
  - Allows us to implement a simpler “core” in the evaluator, and support the alternative syntax by translating it into core syntax

  “let” as sugared procedure application:

  (let ((<name1> <val1>)
        (<name2> <val2>))
    <body>)

    (lambda (<name1> <name2>) <body>)
    <val1> <val2>
Detect and Transform the Alternative Syntax

```scheme
(define (m-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp)
         (lookup-variable-value exp env))
        ((quoted? exp)
         (text-of-quotation exp))
        ...)
  ((let? exp)
   (m-eval (let->combination exp) env))
  ((application? exp)
   (m-apply (m-eval (operator exp) env)
            (list-of-values (operands exp) env)))
  (else (error "Unknown expression" exp)))
```

Let Syntax Transformation

FROM

```scheme
(let ((x 23)
       (y 15))
  (dosomething x y))
```

TO

```scheme
(lambda (x y) (dosomething x y))
```

Let Syntax Transformation

```scheme
(let? exp) (tagged-list? exp 'let))
(define (let-bound-variables let-exp)
  (map car (cadr let-exp)))
(define (let-values let-exp)
  (map cadr (cadr let-exp)))
(define (let-body let-exp)
  (cddr let-exp))

(define (let->combination let-exp)
  (let ((names (let-bound-variables let-exp))
        (values (let-values let-exp))
        (body (let-body let-exp)))
   (cons (make-lambda names body) values)))
```

Details of let syntax transformation

```scheme
(let ((x 23)
       (y 15))
  (dosomething x y))
```

Defining Procedures

```scheme
(define foo (lambda (x) <body>))
```

- Semantic implementation – just another define:
  ```scheme
  (define (eval-definition exp env)
    (define-variable! (definition-variable exp)
      (m-eval (definition-value exp) env))
  )
  ```

- Syntactic transformation:
  ```scheme
  (define (definition-value exp)
   (if (symbol? (cadr exp))
       (caddr exp)
       (make-lambda (caddr exp) ;formal params (cddr exp))))) ;body
  ```
How the Environment Works

- Abstractly – in our environment diagrams:
  - E2: x: 10, plus: (procedure ...)
  - E1: frame

- Concretely – our implementation (as in textbook)

```
list of variables

x

plus

10

procedure

list of values
```

Extending the Environment

- (extend-environment '(x y) '(4 5) E2)

```
E2: x: 10, plus: (procedure ...)
E1: frame
```

```
E2: x: 4, y: 5
E1: frame
```

"Scanning" the environment

- Look for a variable in the environment...
  - Look for a variable in a frame...
    - loop through the list of vars and list of vals in parallel
    - detect if the variable is found in the frame
  - If not found in frame (i.e. we reached end of list of vars), look in enclosing environment

Scanning the environment (details)

```
(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars) (env-loop (enclosing-environment env)))
            ((eq? var (car vars)) (car vals))
            (else (scan (cdr vars) (cdr vals))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable -- LOOKUP" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame) (frame-values frame)))))))
(env-loop env)
```

The Initial (Global) Environment

- setup-environment
  - (define (setup-environment)
    (let ((initial-env (extend-environment (primitive-procedure-names) (primitive-procedure-objects) the-empty-environment)))
      (define-variable! 'true #T initial-env)
      (define-variable! 'false #F initial-env)
      initial-env))
  - define initial variables we always want
  - bind explicit set of "primitive procedures"
    - here: use underlying Scheme procedures
    - in other interpreters: assembly code, hardware, ....

Read-Eval-Print Loop

- (define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (m-eval input the-global-env)))
      (announce-output output output-prompt)
      (display output)))
  (driver-loop))
  (read-eval-print loop)
Variations on a Scheme

- More (not-so) stupid syntactic tricks
  - Let with sequencing
    \[
    \text{let* (} \begin{align*}
    (x & 4) \\
    (y & (+ x 1))
    \end{align*}
    \text{) }
    \]
  - Infix notation
    \[
    (4 * 3) + 7 \]
  - Semantic variations
    - Lexical vs dynamic scoping
      - Lexical: defined by the program text
      - Dynamic: defined by the runtime behavior

Diving in Deeper: Lexical Scope

- Scoping is about how free variables are looked up (as opposed to bound parameters)
  \[
  \text{\lambda (x) \{ x x \}} \]
  - is free
  - is bound
- How does our evaluator achieve lexical scoping?
  - environment chaining
  - procedures capture their enclosing lexical environment

Lexical Scope & Environment Diagram

\[
\begin{align*}
\text{(define (foo x y) \.....)} \\
\text{(define (bar l) \.....)}
\end{align*}
\]

Dynamic Scope & Environment Diagram

\[
\begin{align*}
\text{(define (pooh x) \.....)} \\
\text{(define (bear y) \.....)}
\end{align*}
\]

Alternative Model: Dynamic Scoping

- Dynamic scope:
  - Look up free variables in the caller's environment rather than the surrounding lexical environment
- Example:
  \[
  \begin{align*}
  \text{(define (pooh x)} \\
  \text{(bear 20))} \\
  \text{(define (bear y)} \\
  \text{(+ x y))} \\
  \text{(pooh 9)}
  \end{align*}
  \]
A "Dynamic" Scheme

(define (m-eval exp env)
  (cond
    ((self-evaluating? exp) exp)
    ((variable? exp) (lookup-variable-value exp env))
    ...)
    ((lambda? exp)
      (make-procedure (lambda-parameters exp)
                       (lambda-body exp)
                       'no-environment*)
      ) ;CHANGE: no env
    ...
    ((application? exp)
      (d-apply (m-eval (operator exp) env)
               (list-of-values (operands exp) env)
               env)
      ) ;CHANGE: add env
    (else (error "Unknown expression -- M-EVAL" exp))))

A "Dynamic" Scheme – d-apply

(define (d-apply procedure arguments calling-env)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
    ((compound-procedure? procedure)
      (eval-sequence
       (procedure-body procedure)
       (extend-environment
        (procedure-parameters procedure)
        arguments
        calling-env))
      ) ;CHANGE: use calling env
    (else (error "Unknown procedure" procedure)))))

Summary

- Scheme Evaluator – Know it Inside & Out
- Techniques for language design:
  - Interpretation: eval/apply
  - Semantics vs. syntax
  - Syntactic transformations
- Able to design new language variants!
  - Lexical scoping vs. Dynamic scoping