CS 565 September 21, 1997
Jens Palsberg Readings: EOPL Chapter 5, 7

Lecture Note 5:

An interpreter, written in Scheme, for the core of Java

Contents

1 The Key Ideas

- Environments
- Closures
- Cells
- Recursive Environments

2 A Simple Language and its Interpreter

$$
\begin{array}{rcl}
\textit{Expression} & ::= & \textit{LitExp} \\
& | & \textit{VarExp} \\
& | & \textit{AppExp} \\
\textit{LitExp} & ::= & \textit{INTEGER\_LITERAL} \\
\textit{VarExp} & ::= & \textit{IDENTIFIER} \\
\textit{AppExp} & ::= & (\ \textit{Expression}\ \textit{Expression}_1 \ldots\ \textit{Expression}_n\ )
\end{array}
$$

```
Concrete Syntax:

(ghgh (x 137) (foo 12 2 46))


Abstract Syntax:

(AppExp
  (VarExp ghgh)
  ((AppExp
     (VarExp x)
     ((LitExp 137)))
   (AppExp
     (VarExp foo)
     ((LitExp 12)
      (LitExp 2)
      (LitExp 46)))))
```

The Interpreter

```
(define eval-exp
  (lambda (exp)
    (record-case exp
      (LitExp (const) const)
      (VarExp (id) (apply-env init-env id))
      (AppExp (rator rands)
        (let ((proc (eval-exp rator))
              (args (eval-rands rands)))
          (apply-proc proc args)))
      (else (error 'eval-exp "Bad abstract syntax")))))

(define eval-rands
  (lambda (rands)
    (map eval-exp rands)))

(define run-string
  (lambda (string)
    (eval-exp (parse-exp (scan-string string)))))
```

3 Adding Primitive Procedures

$$
\begin{array}{rcl}
\textit{Procedure} & ::= & \textit{PrimitiveProc} \\
\textit{PrimitiveProc} & ::= & \textit{Primop} \\
\textit{Primop} & ::= & \texttt{+-op} \\
 & | & \texttt{--op} \\
 & | & \texttt{*-op} \\
 & | & \texttt{+1-op} \\
 & | & \texttt{-1op}
\end{array}
$$

```
(define-record PrimitiveProc (primop))

(define apply-proc
  (lambda (proc args)
    (record-case proc
      (PrimitiveProc (primop)
        (apply-primop primop args))
      (else (error 'apply-proc "Bad Procedure")))))

(define apply-primop
  (lambda (primop args)
    (case primop
      ((+-op)  (+ (car args) (cadr args)))
      ((--op)  (- (car args) (cadr args)))
      ((*-op)  (* (car args) (cadr args)))
      ((+1-op) (+ (car args) 1))
      ((-1-op) (- (car args) 1))
      (else (error 'apply-primop "Unknown Primop")))))
```

Building environments from finite functions

```
;;; the-empty-env : env
;;; extend-env    : names * vals * env -> env
;;; apply-env     : env * name -> val

(define the-empty-env the-empty-ff)
(define extend-env extend-ff*)
(define apply-env apply-ff)
```

Building the initial environment:

```
(define init-pairs
  '((+    +-op)
    (ghgh +-op)                    ; an oddball name
    (-    --op)
    (*    *-op)
    (add1 +1-op)
    (sub1 -1-op)))

(define build-init-env
  (lambda (pairs)
    (extend-env
      (map car pairs)
      (map make-PrimitiveProc (map cadr pairs))
      the-empty-env)))

(define init-env (build-init-env init-pairs))
```

4 Adding `if`

$$\begin{array}{rcl} \mathit{Expression} & ::= & \mathit{IfExpression} \\ \mathit{IfExpression} & ::= & \texttt{if}\ \mathit{Expression}\ \texttt{then}\ \mathit{Expression}\ \texttt{else}\ \mathit{Expression} \end{array}$$

```
;; abstract syntax:

(define-record IfExpression (test-exp true-exp false-exp))

;; concrete syntax:

if x then (+ x y) else if (+ x 1) then z else foo


;; add to interpreter:

(IfExpression (test-exp true-exp false-exp)
  (if (true-val? (eval-exp test-exp))
      (eval-exp true-exp)
      (eval-exp false-exp)))

;; auxiliary:

(define true-val?
  (lambda (val)
    (not (zero? val))))
```

5 Adding `let`

```
let x = 5
    y = 6
in  (+ x y)
```

```
let z = 5 x = 3
in let x = 4
       y = (+ x z)          % here x = 3
   in (* z (+ x y))         % here x = 4
```

Grammar:

$$\begin{array}{rcl} \textit{Expression} & ::= & \textit{LetExpression} \\ \textit{LetExpression} & ::= & \texttt{let}\ \textit{Declaration}_1 \ldots\ \textit{Declaration}_n\ \texttt{in}\ \textit{Expression} \\ \textit{Declaration} & ::= & \textit{Identifier}\ \texttt{=}\ \textit{Expression} \end{array}$$

Abstract syntax:

```
(define-record LetExpression (decls body))
(define-record Declaration (var exp))
```

Interpreter for `let`

```
(define eval-exp
  (lambda (exp env)
    (record-case exp
      (LitExp (constant) constant)
      (VarExp (id) (apply-env env id))
      (IfExpression (test-exp true-exp false-exp)
        (if (true-val? (eval-exp test-exp env))
            (eval-exp true-exp env)
            (eval-exp false-exp env)))
      (LetExpression (decls body)
        (let ((ids  (map Declaration->var decls))
              (exps (map Declaration->exp decls)))
          (let ((new-env (extend-env ids
                              (eval-rands exps env) env)))
            (eval-exp body new-env))))
      (AppExp (rator rands)
        (let ((proc (eval-exp rator env))
              (args (eval-rands rands env)))
          (apply-proc proc args)))
      (else (error 'eval-exp "Bad abstract syntax")))))

(define eval-rands
  (lambda (rands env)
    (map (lambda (exp) (eval-exp exp env)) rands)))

(define run
  (lambda (string)
    (eval-exp (scan&parse string) init-env)))
```

6 Adding User-Defined Procedures

$$
\begin{array}{rcl}
\mathit{Expression} & ::= & \mathit{ProcExp} \\
\mathit{ProcExp} & ::= & \texttt{proc} \ \texttt{(} \ \mathit{Identifier}_1 \ldots \ \mathit{Identifier}_n \ \texttt{)} \ \mathit{Expression}
\end{array}
$$

```
let f = proc (y z) (* y (+ x z))
    g = proc (u) (+ u 5)
in (f (g 3) 17)

(proc (u) (+ u 5)  3)

let x = 5
in  let f = proc (y z) (* y (+ x z))
        g = proc (u) (+ u x)
        x = 38
    in  (f (g 3) 17)

let f = proc (y) (y (y 5))
    g = proc (z) (+ z 8)
in  (f g)

let f = proc (y) proc (z) (y (y z))
    g = proc (z) (+ z 8)
in ((f g) 5)

let f = proc (y z) (y (y z))
    g = proc (z) (+ z 8)
in  (f g 5)
```

Representing User-Defined Procedures

```
(define-record closure (formals body env))

(define apply-proc
  (lambda (proc args)
    (record-case proc
      (PrimitiveProc (primop) (apply-primop primop args))
      (closure (formals body env)
        (eval-exp body (extend-env formals args env)))
      (else (error 'apply-proc "Bad Procedure" proc)))))

(define eval-exp
  (lambda (exp env)
    (record-case exp
      (ProcExp (formals body) (make-closure formals body env))
      ...
      )))

(define apply-proc
  (lambda (proc args)
    (record-case proc
      (PrimitiveProc (primop) (apply-primop primop args))
      (closure (formals body env)
        (eval-exp body (extend-env formals args env)))
      (else (error 'apply-proc "Bad Procedure" proc)))))
```

7 Adding Assignment

Simulating cells:

```
(define make-cell
  (lambda (value)
    (cons '*cell value)))

(define deref-cell cdr)

(define set-cell! set-cdr!) ; danger!
```

Grammar:

Expression ::= `set!` *Identifier Expression*

Abstract Syntax

```
(define-record assign-exp (ident rhs-exp))
```

Environments Containing Cells

An environment will be a finite function containing cells.

```
(define eval-exp
  (lambda (exp env)
    (record-case exp
      ;; change:
      (VarExp (id) (deref-cell (apply-env env id)))
      ...
      (assign-exp (ident rhs-exp)
(set-cell!
  (apply-env env ident)
  (eval-exp rhs-exp env)))
      ...)))
```

And need to change `extend-env` to put the values in cells:

```
(define extend-env
  (lambda (names values env)
    (extend-ff* names (map make-cell values) env)))
```

8 Adding letrec

$$\begin{array}{rcl} \textit{Expression} & ::= & \textit{LetRecExpression} \\ \textit{LetRecExpression} & ::= & \texttt{letrec}\ \textit{RecDecl}_1 \ldots \textit{RecDecl}_n\ \texttt{in}\ \textit{Expression} \\ \textit{RecDecl} & ::= & \texttt{(}\ \textit{Identifier}\ \textit{Identifier}_1 \ldots \textit{Identifier}_n\ \texttt{)}\ \texttt{=}\ \textit{Expression} \end{array}$$

```
letrec
  (fact x) = if (zero? x) then 1 else (* x (fact (sub1 x)))
in (fact 6)
```

```
letrec
  (even x) = if (zero? x) then 1 else (odd (sub1 x))
  (odd x)  = if (zero? x) then 0 else (even (sub1 x))
in (odd 13)
```

Abstract syntax:

```
(define-record LetRecExpression (RecDecls body))
(define-record RecDecl (var formals body))
```

Interpreter line:

```
(LetRecExpression (RecDecls body)
  (eval-exp body (extend-rec RecDecls env)))
```

Two kinds of environment records. Ordinary environments contain cells. Recursive environments contain `RecDecls`. If you look up a `RecDecl`, it gets closed in the environment frame that contains it:

```
(define-record the-empty-env ())
(define-record extended-env (names vals old-env))
(define-record extended-rec-env (RecDecls old-env))

;; we'll use more familiar names for these:

(define extend-env make-extended-env)
(define extend-rec make-extended-rec-env)
```

```
(define apply-env
  (lambda (env var-sym)
    (record-case env
      (the-empty-env ()
        (error 'apply-env "Unbound identifier"))
      (extended-env (names values old-env)
       (if (memq var-sym names)
           (ribassoc var-sym names vals)
           (apply-env old-env var-sym)))
      (extended-rec-env (decls old-env)
        (let ((names (map RecDecl->var decls)))
          (if (memq var-sym names)
            (let ((value (ribassoc var-sym names decls)))
              (make-cell  ; apply-env must return a cell
                (make-closure
                  (RecDecl->formals value)
                  (RecDecl->body value)
                  env)))
            (apply-env old-env var-sym))))
      (else
        (error 'apply-env "unknown environment frame")))))
```

Exercise: avoid doing the `(map RecDecl->var decls)` on every lookup doing the search explicitly in the existing structure.

Variable lookup is the most frequent interpreter operation in Scheme-like languages. What else could be done to speed up variable lookup?

9 MiniJava and its Interpreter

$$
\begin{array}{rcl}
\mathit{Goal} & ::= & \mathit{TypeDeclaration}_1 \ \ldots \ \mathit{TypeDeclaration}_n \\
\mathit{TypeDeclaration} & ::= & \mathit{ClassDeclarationExtends} \\
 & | & \mathit{ClassDeclaration} \\
\mathit{ClassDeclarationExtends} & ::= & \texttt{class}\ \mathit{Identifier} \\
 & & \quad \texttt{extends}\ \mathit{Identifier}\ \mathit{ClassBody} \\
\mathit{ClassDeclaration} & ::= & \texttt{class}\ \mathit{Identifier}\ \mathit{ClassBody} \\
\mathit{ClassBody} & ::= & \texttt{\{}\ \mathit{ClassBodyDeclaration}_1 \ \ldots \\
 & & \quad \mathit{ClassBodyDeclaration}_n\ \texttt{\}} \\
\mathit{ClassBodyDeclaration} & ::= & \mathit{ConstructorDeclaration} \\
 & | & \mathit{FieldDeclaration} \\
 & | & \mathit{MethodDeclaration} \\
\mathit{ConstructorDeclaration} & ::= & \mathit{Identifier}\ \texttt{( )}\ \mathit{Block} \\
\mathit{FieldDeclaration} & ::= & \texttt{private}\ \mathit{Identifier}\ \texttt{;} \\
\mathit{MethodDeclaration} & ::= & \texttt{public}\ \mathit{Identifier}\ \texttt{(}\ \mathit{Identifier}\ \texttt{)} \\
 & & \quad \texttt{\{}\ \mathit{MethodBody}\ \texttt{\}} \\
\mathit{Block} & ::= & \texttt{\{}\ \mathit{StatementList}\ \texttt{\}} \\
\mathit{MethodBody} & ::= & \mathit{StatementListReturn} \\
 & | & \mathit{StatementList} \\
\mathit{StatementListReturn} & ::= & \mathit{StatementList}\ \texttt{return}\ \mathit{Expression}\ \texttt{;} \\
\mathit{StatementList} & ::= & \mathit{Statement}_1 \ \ldots \ \mathit{Statement}_n \\
\mathit{Statement} & ::= & \mathit{Block} \\
 & | & \mathit{AssignmentStatement} \\
 & | & \mathit{IfStatement} \\
 & | & \mathit{PrintStatement} \\
\mathit{AssignmentStatement} & ::= & \mathit{Identifier}\ \texttt{=}\ \mathit{Expression}\ \texttt{;} \\
\mathit{IfStatement} & ::= & \texttt{if}\ \texttt{(}\ \mathit{Expression}\ \texttt{)}\ \mathit{Statement} \\
 & & \quad \texttt{else}\ \mathit{Statement} \\
\mathit{PrintStatement} & ::= & \texttt{System.out.println(}\ \mathit{Expression}\ \texttt{)}\ \texttt{;}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Expression} & ::= & \mathit{EqualityExpression} \\
 & | & \mathit{CompareExpression} \\
 & | & \mathit{PlusExpression} \\
 & | & \mathit{MessageSend} \\
 & | & \mathit{PrimaryExpression} \\
\mathit{EqualityExpression} & ::= & \mathit{PrimaryExpression}\ \texttt{==}\ \mathit{PrimaryExpression} \\
\mathit{CompareExpression} & ::= & \mathit{PrimaryExpression}\ \texttt{<}\ \mathit{PrimaryExpression} \\
\mathit{PlusExpression} & ::= & \mathit{PrimaryExpression}\ \texttt{+}\ \mathit{PrimaryExpression} \\
\mathit{MessageSend} & ::= & \mathit{PrimaryExpression}\ \texttt{.}\ \mathit{Identifier}\ \texttt{(}\ \mathit{Expression}\ \texttt{)} \\
\mathit{PrimaryExpression} & ::= & \mathit{IntegerLiteral} \\
 & | & \mathit{TrueLiteral} \\
 & | & \mathit{FalseLiteral} \\
 & | & \mathit{Identifier} \\
 & | & \mathit{ThisExpression} \\
 & | & \mathit{SuperExpression} \\
 & | & \mathit{AllocationExpression} \\
\mathit{IntegerLiteral} & ::= & \mathit{INTEGER\_LITERAL} \\
\mathit{TrueLiteral} & ::= & \texttt{true} \\
\mathit{FalseLiteral} & ::= & \texttt{false} \\
\mathit{Identifier} & ::= & \mathit{IDENTIFIER} \\
\mathit{ThisExpression} & ::= & \texttt{this} \\
\mathit{SuperExpression} & ::= & \texttt{super}\ \texttt{.}\ \mathit{Identifier}\ \texttt{(}\ \mathit{Expression}\ \texttt{)} \\
\mathit{AllocationExpression} & ::= & \texttt{new}\ \mathit{Identifier}\ \texttt{(}\ \texttt{)}
\end{array}
$$

Record Definitions

```
(define-record Goal (TypeDeclarationList))
(define-record ClassDeclarationExtends (ClassId BaseId ClassBody))
(define-record ClassDeclaration (Identifier ClassBody))
(define-record ClassBody (ClassBodyDeclarationList))
(define-record ConstructorDeclaration (Identifier Block))
(define-record FieldDeclaration (Identifier))
(define-record MethodDeclaration (MethodId ArgumentId MethodBody))
(define-record Block (StatementList))
(define-record StatementListReturn (StatementList Expression))
(define-record StatementList (StatementList))
(define-record AssigmentStatement (Identifier Expression))
(define-record IfStatement (Expression ThenStatement ElseStatement))
(define-record PrintStatement (Expression))
(define-record EqualityExpression (PrimaryExpression1 PrimaryExpression2))
(define-record CompareExpression (PrimaryExpression1 PrimaryExpression2))
(define-record PlusExpression (PrimaryExpression1 PrimaryExpression2))
(define-record MessageSend (PrimaryExpression MethodId Expression))
(define-record IntegerLiteral (literal))
(define-record TrueLiteral ())
(define-record FalseLiteral ())
(define-record Identifier (id))
(define-record ThisExpression ())
(define-record SuperExpression (Identifier Expression))
(define-record AllocationExpression (Identifier))

(define-record object (class-id env))
```

Example

A MiniJava program:

```
class C {
  public main(y) {
    System.out.println( new D().m(4) );
  }
}

class D {
  private y;
  private z;
  public m(x) {
    if (true) {z=5;} else z=7;
    y=x; return y+z;
  }
}
```

The syntax tree for the above MiniJava program:

```
(define goal '(Goal
  ((ClassDeclaration (Identifier C)
     (ClassBody
       ((MethodDeclaration (Identifier Main) (Identifier y)
          (StatementList
            ((PrintStatement
               (MessageSend
                 (AllocationExpression (Identifier D))
                 (Identifier m)
                 (IntegerLiteral 4)))))))))
   (ClassDeclaration (Identifier D)
     (ClassBody
       ((FieldDeclaration (Identifier y))
        (FieldDeclaration (Identifier z))
        (MethodDeclaration (Identifier m) (Identifier x)
          (StatementListReturn
            (StatementList
              ((IfStatement (TrueLiteral )
                 (Block (StatementList
                   ((AssignmentStatement
                      (Identifier z) (IntegerLiteral 5)))))
                 (AssignmentStatement
                   (Identifier z) (IntegerLiteral 7)))
               (AssignmentStatement
                 (Identifier y) (Identifier x))))
            (PlusExpression
              (Identifier y) (Identifier z)))))))))
```

A visitor which produces syntaxtrees:

```
package visitor;
import syntaxtree.*;
import java.util.*;
import java.io.*;

public class ToSchemeVisitor extends Visitor {
   PrintWriter out = null;

   //
   // f0 -> ( TypeDeclaration() )*
   // f1 -> <EOF>
   //
   public void visitGoal(Goal n) {
      try {
        out = new PrintWriter(
                     new FileOutputStream("tree.scm"),true);
      }
      catch(IOException e) {
        System.out.println(
             "IOException while trying to open file output");
      }
      out.write( "(define goal '(Goal (" );
      n.f0.accept(this);
      n.f1.accept(this);
      out.println( ")))" );
      out.flush();
      out.close();
   }
```

```
//
// f0 -> ( Statement() )*
//
public void visitStatementList(StatementList n) {
   out.write( "(StatementList (" );
   n.f0.accept(this);
   out.write( "))" );
}

//
// f0 -> Block()
//       | AssignmentStatement()
//       | IfStatement()
//       | PrintStatement()
//
public void visitStatement(Statement n) {
   n.f0.accept(this);
}
```

```
//
// f0 -> Identifier()
// f1 -> "="
// f2 -> Expression()
// f3 -> ";"
//
public void visitAssignmentStatement(AssignmentStatement n) {
   out.write( "(AssignmentStatement " );
   n.f0.accept(this);
   n.f1.accept(this);
   n.f2.accept(this);
   n.f3.accept(this);
   out.write( ")" );
}
```

```
//
// f0 -> "if"
// f1 -> "("
// f2 -> Expression()
// f3 -> ")"
// f4 -> Statement()
// f5 -> "else"
// f6 -> Statement()
//
public void visitIfStatement(IfStatement n) {
   out.write( "(IfStatement " );
   n.f0.accept(this);
   n.f1.accept(this);
   n.f2.accept(this);
   n.f3.accept(this);
   n.f4.accept(this);
   n.f5.accept(this);
   n.f6.accept(this);
   out.write( ")" );
}
```

```
//
// f0 -> "System.out.println("
// f1 -> Expression()
// f2 -> ")"
// f3 -> ";"
//
public void visitPrintStatement(PrintStatement n) {
   out.write( "(PrintStatement " );
   n.f0.accept(this);
   n.f1.accept(this);
   n.f2.accept(this);
   n.f3.accept(this);
   out.write( ")" );
}
```

A MiniJava Interpreter

```
;;; search procedures

(define search-Goal
  (lambda (goal class-id)
    (record-case goal
      (Goal (TypeDeclarationList)
        (search-TypeDeclarationList
          TypeDeclarationList class-id))
      (else (error 'eval-Goal "Goal not found")))))

(define search-TypeDeclarationList
  (lambda (TypeDeclarationList class-id)
    (if (null? TypeDeclarationList)
          (error 'search-TypeDeclarationList "Class not found")
        (record-case (car TypeDeclarationList)
          (ClassDeclarationExtends (ClassId BaseId ClassBody)
            ...)
          (ClassDeclaration (ClassId ClassBody)
            (if (eq? (Identifier->id ClassId) class-id)
                (ClassBody->ClassBodyDeclarationList ClassBody)
                (search-TypeDeclarationList
                  (cdr TypeDeclarationList) class-id)))))))
```

```
;;; CBDL abbreviates ClassBodyDeclarationList

(define search-CBDL
  (lambda (ClassBodyDeclarationList method-id)
    (if (null? ClassBodyDeclarationList)
        (error 'search-CBDL "Method not found")
        (record-case (car ClassBodyDeclarationList)
          (ConstructorDeclaration (Identifier Block)
            (search-CBDL (cdr ClassBodyDeclarationList)
                         method-id))
          (FieldDeclaration (Identifier)
            (search-CBDL (cdr ClassBodyDeclarationList)
                         method-id))
          (MethodDeclaration (MethodId ArgumentId MethodBody)
            (if (equal? (Identifier->id MethodId) method-id)
                (car ClassBodyDeclarationList)
                (search-CBDL (cdr ClassBodyDeclarationList)
                             method-id))))))))
```

```
;;; allocation procedure

(define allocate-CBDL
  (lambda (ClassBodyDeclarationList env)
    (if (null? ClassBodyDeclarationList)
        env
        (record-case (car ClassBodyDeclarationList)
          (ConstructorDeclaration (Identifier Block)
            (allocate-CBDL (cdr ClassBodyDeclarationList)
                           env))
          (FieldDeclaration (Identifier)
            (allocate-CBDL (cdr ClassBodyDeclarationList)
                           (extend-env
                             (Identifier->id Identifier)
                             (make-cell '*undefined) env)))
          (MethodDeclaration (MethodId ArgumentId MethodBody)
            (allocate-CBDL (cdr ClassBodyDeclarationList)
                           env))))))
```

```
;;; evaluation procedures

(define eval-Block
  (lambda (block class-env env this)
    (record-case block
      (Block (StatementList)
        (eval-StatementList StatementList class-env env this))
      (else (error 'eval-Block "Block not found")))))

(define eval-MethodBody
  (lambda (methodbody class-env env this)
    (record-case methodbody
      (StatementListReturn (StatementList Expression)
        (begin
          (eval-StatementList StatementList class-env env this)
          (eval-Expression Expression class-env env this)))
      (StatementList (StatementList)
        (eval-StatementList methodbody class-env env this))
      (else (error 'eval-MethodBody "MethodBody not found")))))

(define eval-StatementList
  (lambda (statementlist class-env env this)
    (record-case statementlist
      (StatementList (stmtlist)
        (for-each
          (lambda (statement)
              (eval-Statement statement class-env env this))
          stmtlist)))))
```

```
(define eval-Statement
  (lambda (statement class-env env this)
    (record-case statement
      (Block (StatementList)
        (eval-StatementList StatementList class-env env this))
      (AssignmentStatement (Identifier Expression)
        (set-cell! (apply-env env (Identifier->id Identifier))
                   (eval-Expression
                     Expression class-env env this)))
      (IfStatement (Expression ThenStatement ElseStatement)
        (if (eval-Expression Expression class-env env this)
            (eval-Statement ThenStatement class-env env this)
            (eval-Statement ElseStatement class-env env this)))
      (PrintStatement (Expression)
        (display (eval-Expression Expression class-env env this))
        (newline))
      (else (error 'eval-Statement "Statement not found")))))
```

```
(define eval-Expression
  (lambda (expression class-env env this)
    (record-case expression
      (EqualityExpression (PrimaryExpression1 PrimaryExpression2)
        (eq? (eval-Expression PrimaryExpression1
                              class-env env this)
             (eval-Expression PrimaryExpression2
                              class-env env this)))
      (CompareExpression (PrimaryExpression1 PrimaryExpression2)
        (<   (eval-Expression PrimaryExpression1
                              class-env env this)
             (eval-Expression PrimaryExpression2
                              class-env env this)))
      (PlusExpression (PrimaryExpression1 PrimaryExpression2)
        (+   (eval-Expression PrimaryExpression1
                              class-env env this)
             (eval-Expression PrimaryExpression2
                              class-env env this)))
      (IntegerLiteral (literal) literal)
      (TrueLiteral () #t)
      (FalseLiteral () #f)
      (Identifier (id)
        (deref-cell (apply-env env id)))
      (ThisExpression () this)
      (SuperExpression (Identifier Expression)
        ...)
      (AllocationExpression (Identifier)
        (make-object (Identifier->id Identifier)
                     (allocate (apply-class-env class-env
                                 (Identifier->id Identifier)))))
```

```
(MessageSend (PrimaryExpression MethodId Expression)
  (let* ((receiver (eval-Expression
                     PrimaryExpression class-env env this))
         (argument (eval-Expression Expression
                        class-env env this))
         (class-id (object->class-id receiver))
         (CBDL     (apply-class-env class-env class-id))
         (MethodDeclaration (search-CBDL CBDL
                              (Identifier->id MethodId))))
    (record-case MethodDeclaration
      (MethodDeclaration (MethodId2 ArgumentId MethodBody)
         (eval-MethodBody
              MethodBody class-env
              (extend-env (Identifier->id ArgumentId)
                          argument (object->env receiver))
              receiver)))))
(else (error 'eval-Expression "Expression not found")))))
```

```
;;; auxiliary procedures

;;; class-env:  A class-env is simply a goal.

(define apply-class-env           ;; returns a CBDL
  (lambda (class-env class-id)
    (search-Goal class-env class-id)))

(define allocate
  (lambda (ClassBodyDeclarationList)
    (allocate-CBDL ClassBodyDeclarationList (empty-env))))

;;; env

(define empty-env
  (lambda ()
    '()))

(define apply-env
  (lambda (env id)
    (cadr (assq id env))))

(define extend-env
  (lambda (id value env)
    (cons (list id (make-cell value)) env)))
```

```
;;; Cells

(define make-cell
  (lambda (value)
    (cons '*cell value)))

(define deref-cell cdr)

(define set-cell! set-cdr!)

;;; the top level procedure

(define run
  (lambda (class-id)
    (eval-MethodBody
      (MethodDeclaration->MethodBody
        (search-CBDL (apply-class-env goal class-id) 'main))
      goal (extend-env 'arg 0 (empty-env)) 'no-this)))
```