CS 565, Jens Palsberg Sep 7, 1999

Lecture Note 2: Interpreters and Operational Semantics

Contents

1 Interpreters 2

2 Stack Machine Code Grammar 3

3 A Stack Machine Interpreter 6

4 MiniScheme Grammar 8

5 A MiniScheme Interpreter 12

6 MiniJava Grammar 25

7 A MiniJava Interpreter 30

8 Lambda-calculus 40

9 Operational Semantics 41

1 Interpreters

We will write interpreters using four ideas:

- Environments
- Cells
- Closures
- Recursive Environments

We will begin with a simple example that does not require any of these ideas: an interpreter for a stack machine.

We will then study interpreters for five subsets of Scheme, and we will conclude with an interpreter for MiniJava.

All of the interpreters will be written in Scheme.

2 Stack Machine Code Grammar

Let us write an interpreter for a stack machine. The machine will have two components: an *action* and a *stack*. The stack contains the data in the machine. We will represent the stack as a list of Scheme values, with the top of the stack at the front of the list.

The action represents the instruction stream being executed by the machine. Actions are defined by the grammar:

$$
\begin{array}{rcl}
\mathit{Action} & ::= & \texttt{halt} \\
& | & \texttt{incr;}\ \mathit{Action} \\
& | & \texttt{add;}\ \mathit{Action} \\
& | & \texttt{push}\ \mathit{Integer}\ ;\ \mathit{Action} \\
& | & \texttt{pop;}\ \mathit{Action}
\end{array}
$$

We will write an interpreter `eval-action` which takes an action and a stack and returns the value produced by the machine at the completion of the action. We will adopt the convention that the machine produces a value by leaving it on the top of the stack when it halts.

Specification of Operations

We want to write a *specification* for `eval-action`. The specification should say what (`eval-action` a s) does for each possible value of a.

The specification is given by the following equations.

```
(eval-action halt s) = (car s)

(eval-action incr; a (v w ...)) =
   (eval-action a (v+1 w ...))

(eval-action add; a (v w x ...)) =
   (eval-action a ((v+w) x ...))

(eval-action push v; a (w ...)) =
   (eval-action a (v w ...))

(eval-action pop; a (v w ...)) =
   (eval-action a (w ...))
```

Representation of Operations

To write Scheme code to implement the specification of `eval-action`, we need to specify a representation of the type of actions. Let us make a simple choice:

$$\begin{aligned}
\lceil \texttt{halt} \rceil &= \texttt{(halt)} \\
\lceil \texttt{incr;}\ a \rceil &= \texttt{(incr .}\ \lceil a \rceil\texttt{)} \\
\lceil \texttt{add;}\ a \rceil &= \texttt{(add .}\ \lceil a \rceil\texttt{)} \\
\lceil \texttt{push}\ v\ \texttt{;}\ a \rceil &= \texttt{(push}\ v\ \texttt{.}\ \lceil a \rceil\texttt{)} \\
\lceil \texttt{pop;}\ a \rceil &= \texttt{(pop .}\ \lceil a \rceil\texttt{)}
\end{aligned}$$

so an action is represented as a list of instructions. This is like bytecode: a typical action is

```
(push 3 push 4 add halt)
```

3 A Stack Machine Interpreter

```
(define eval-action
  (lambda (action stack)
    (let ((op-code (car action)))
      (case op-code
        ((halt)
         (car stack))
        ((incr)
         (eval-action (cdr action)
            (cons (+ (car stack) 1) (cdr stack))))
        ((add)
         (eval-action (cdr action)
           (cons (+ (car stack) (cadr stack))
                 (cddr stack))))
        ((push)
         (let ((v (cadr action)))
           (eval-action (cddr action)
                        (cons v stack))))
        ((pop)
         (eval-action (cdr action) (cdr stack)))
        (else
          (error 'eval-action
                 "unknown op-code:" op-code))))))
```

Running the Interpreter

```
(define start
  (lambda (action)
    (eval-action action '())))
```

```
> (start '(push 3 push 4 add halt))
7
```

4 MiniScheme Grammar

$$
\begin{array}{rcl}
\mathit{Expression} & ::= & \mathit{IntegerLiteral} \\
 & | & \mathit{TrueLiteral} \\
 & | & \mathit{FalseLiteral} \\
 & | & \mathit{PlusExpression} \\
 & | & \mathit{IfExpression} \\
 & | & \mathit{LetExpression} \\
 & | & \mathit{Identifier} \\
 & | & \mathit{Assignment} \\
 & | & \mathit{ProcedureExp} \\
 & | & \mathit{Application} \\
 & | & \mathit{RecExpression} \\
\mathit{IntegerLiteral} & ::= & \mathit{INTEGER\_LITERAL} \\
\mathit{TrueLiteral} & ::= & \texttt{\#t} \\
\mathit{FalseLiteral} & ::= & \texttt{\#f} \\
\mathit{PlusExpression} & ::= & \texttt{(}\ \texttt{+}\ \mathit{Expression}\ \mathit{Expression}\ \texttt{)} \\
\mathit{IfExpression} & ::= & \texttt{(}\ \texttt{if}\ \mathit{Expression}\ \mathit{Expression}\ \mathit{Expression}\ \texttt{)} \\
\mathit{LetExpression} & ::= & \texttt{(}\ \texttt{let}\ \texttt{(}\ \mathit{Declaration}^{*}\ \texttt{)}\ \mathit{Expression}\ \texttt{)} \\
\mathit{Assignment} & ::= & \texttt{(}\ \texttt{set!}\ \mathit{Identifier}\ \mathit{Expression}\ \texttt{)} \\
\mathit{ProcedureExp} & ::= & \texttt{(}\ \texttt{lambda}\ \texttt{(}\ \mathit{Identifier}^{*}\ \texttt{)}\ \mathit{Expression}\ \texttt{)} \\
\mathit{Application} & ::= & \texttt{(}\ \mathit{Expression}\ \mathit{Expression}^{*}\ \texttt{)} \\
\mathit{RecExpression} & ::= & \texttt{(}\ \texttt{letrec}\ \texttt{(}\ \mathit{RecDeclaration}^{*}\ \texttt{)}\ \mathit{Expression}\ \texttt{)} \\
\mathit{Declaration} & ::= & \texttt{(}\ \mathit{Identifier}\ \mathit{Expression}\ \texttt{)} \\
\mathit{RecDeclaration} & ::= & \texttt{(}\ \mathit{Identifier}\ \mathit{ProcedureExp}\ \texttt{)}
\end{array}
$$

Records

If we load the file `recscm.scm` from code directory of the CS 565 webpage, and write, say,

```
> (define-record person (name age country))
```

then we can create and inspect records:

```
> (define me (make-person "Jens" 33 "Denmark"))
> me
(person "Jens" 33 "Denmark")
> (record-case me
    (person (a b c)
      (display b)))
33
> (person->age me)
33
```

By executing

```
jtb -scheme minischeme.jj
```

we get the record definitions on the next page, in the file records.scm. We also get a SchemeTreeBuilder visitor.

Record Definitions

```
(define-record IntegerLiteral (Token))
(define-record TrueLiteral (Token))
(define-record FalseLiteral (Token))
(define-record PlusExpression (Token1 Token2
   Expression1 Expression2 Token3))
(define-record IfExpression (Token1 Token2
   Expression1 Expression2 Expression3 Token3))
(define-record LetExpression (Token1 Token2
   Token3 List Token4 Expression Token5))
(define-record Identifier (Token))
(define-record Assignment (Token1 Token2
   Identifier Expression Token3))
(define-record ProcedureExp (Token1 Token2
   Token3 List Token4 Expression Token5))
(define-record Application (Token1 Expression
   List Token2))
(define-record RecExpression (Token1 Token2
   Token3 List Token4 Expression Token5))
(define-record Declaration (Token1 Identifier
   Expression Token2))
(define-record RecDeclaration (Token1 Identifier
   ProcedureExp Token2))
```

Syntax Trees

If we run the automatically generated SchemeTreeBuilder visitor on the MiniScheme program

```
(if #t ((lambda (x y) x) 5 6) #f)
```

then we get the syntax tree:

```
(define root '(Goal
  (IfExpression "(" "if" (TrueLiteral "#t" )
    (Application "("
      (ProcedureExp "(" "lambda" "("
                      ( (Identifier "x" )
                        (Identifier "y" ) ) ")"
                      (Identifier "x" ) ")" )
      ( (IntegerLiteral "5" )
      (IntegerLiteral "6" ) ) ")" )
      (FalseLiteral "#f" ) ")" ) "" ))
```

Our MiniScheme interpreter will work on programs in this representation.

5 A MiniScheme Interpreter

```
(load "/homes/palsberg/.www/cs565/F98/code/recscm.scm")
(load "records")
(load "tree")

(define run
  (lambda ()
    (record-case root
      (Goal (Expression Token)
        (eval-Expression Expression))
      (else (error 'run "Goal not found")))))
```

Simple Expressions

```
(define eval-Expression
  (lambda (Expression)
    (record-case Expression
      (IntegerLiteral (Token)
        (string->number Token))
      (TrueLiteral (Token)
        #t)
      (FalseLiteral (Token)
        #f)
      (PlusExpression (Token1 Token2 Expression1
                       Expression2 Token3)
        (+ (eval-Expression Expression1)
           (eval-Expression Expression2)))
      (IfExpression (Token1 Token2 Expression1
                     Expression2 Expression3 Token3)
        (if (eval-Expression Expression1)
            (eval-Expression Expression2)
            (eval-Expression Expression3)))
      (else (error 'eval-Expression
                   "Expression not found")))))
```

Environments

To handle names, let, etc., we will use environments.

An environment is a function whose domain is a finite set of identifiers: $\{\ (s_1, v_1), \ldots, (s_n, v_n)\ \}$.

```
;;; interface Environment {
;;;   type Env
;;;   empty-Env  : Env
;;;   apply-Env  : Env * Id -> Value
;;;   extend-Env : Id * Value * Env -> Env
;;; }
```

$$
\begin{array}{ll}
\texttt{(empty-env)} & = \lceil \emptyset \rceil \\
\texttt{(apply-env } \lceil f \rceil\ s\texttt{)} & = f(s) \\
\texttt{(extend-env } s\ v\ \lceil f \rceil\texttt{)} & = \lceil g \rceil, \\
 & \text{where } g(s') = \begin{cases} v & s' = s \\ f(s') & \text{otherwise} \end{cases}
\end{array}
$$

Environments

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

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

(define extend-env
  (lambda (id value env)
    (cons (list (Identifier->Token id) value)
          env)))

(define extend-env-list
  (lambda (ids vals env)
    (if (null? ids)
        env
        (extend-env-list
          (cdr ids)
          (cdr vals)
          (extend-env (car ids) (car vals) env)))))
```

Environments

```
(define run
  (lambda ()
    (record-case root
      (Goal (Expression Token)
        (eval-Expression Expression (empty-env)))
      (else (error 'run "Goal not found")))))
```

Let Expressions

```
(define eval-Expression
  (lambda (Expression env)
    (record-case Expression
      ...
      (IfExpression (Token1 Token2 Expression1
                     Expression2 Expression3 Token3)
        (if (eval-Expression Expression1 env)
            (eval-Expression Expression2 env)
            (eval-Expression Expression3 env)))
      (LetExpression (Token1 Token2 Token3
                     List Token4 Expression Token5)
        (let*
           ((ids  (map Declaration->Identifier List))
            (exps (map Declaration->Expression List))
            (vals (map (lambda (Expression)
                         (eval-Expression
                           Expression env))
                       exps))
            (new-env (extend-env-list ids vals env)))
          (eval-Expression Expression new-env)))
      (Identifier (Token)
        (apply-env env Token))
      (else (error ...)))))
```

Cells

In Scheme we can assign to all variables. To handle this, we will use cells.

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

(define deref-cell cdr)

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

When we extend an environment, we will create a cell, store the initial value in the cell, and bind the identifier to the cell.

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

Assignments

```
(define eval-Expression
  (lambda (Expression env)
    (record-case Expression
      ...
      (Identifier (Token)
        (deref-cell (apply-env env Token)))
      (Assignment (Token1 Token2 Identifier
                   Expression Token3)
        (set-cell!
          (apply-env env
                     (Identifier->Token Identifier))
          (eval-Expression Expression env))
        'ok)
      (else (error ...)))))
```

Closures

To represent user-defined procedures, we will use closures.

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

(define eval-Expression
  (lambda (Expression env)
    (record-case Expression
      ...
      (ProcedureExp (Token1 Token2 Token3
                     List Token4 Expression Token5)
        (make-closure List Expression env))
      (Application (Token1 Expression List Token2)
        (let*
           ((clos (eval-Expression Expression env))
            (ids  (closure->formals clos))
            (vals (map (lambda (Exp)
                          (eval-Expression Exp env))
                        List))
            (static-env (closure->env clos))
            (new-env
              (extend-env-list ids vals static-env)))
            (body (closure->body clos))
          (eval-Expression body new-env)))
      (else (error ...)))))
```

Recursive Environments

To handle recursive definitions, we need two kinds of environment records. Normal environments contain cells. A recursive environment contains a `RecDeclarationList`. If one looks up a recursively-defined procedure, then it gets closed in the environment frame that contains it:

```
(define-record empty-env ())
(define-record normal-env (ids vals env))
(define-record rec-env (recdecl-list env))

(define empty-env make-empty-env)

(define extend-env-list
  (lambda (ids vals env)
    (make-normal-env
      (map cadr ids)
      (map (lambda (value) (make-cell value))
           vals)
      env)))
```

Letrec

```
(define eval-Expression
  (lambda (Expression env)
    (record-case Expression
      ...
      (RecExpression (Token1 Token2 Token3
                      List Token4 Expression Token5)
        (eval-Expression
          Expression
          (make-rec-env List env)))
      (else (error ...)))))
```

Recursive Environments

```
(define apply-env
  (lambda (env id)
    (record-case env
      (empty-env ()
        (error 'apply-env "Unbound identifier"))
      (normal-env (id-list vals old-env)
       (if (member id id-list)
           (ribassoc id id-list vals)
           (apply-env old-env id)))
      (rec-env (recdecl-list old-env)
        (let ((id-list
                (map (lambda (RecDeclaration)
                       (cadr (RecDeclaration->Identifier
                               RecDeclaration)))
                     recdecl-list)))
          (if (member id id-list)
            (let* ((RecDeclaration
                     (ribassoc id id-list recdecl-list))
                   (ProcedureExp (RecDeclaration->ProcedureExp
                                   RecDeclaration)))
              (make-cell  ; apply-env must return a cell
                (make-closure
                  (ProcedureExp->List ProcedureExp)
                  (ProcedureExp->Expression ProcedureExp)
                  env)))
            (apply-env old-env id))))
      (else
        (error 'apply-env "unknown environment frame")))))
```

Recursive Environments

```
(define ribassoc
  (lambda (id id-list vals)
    (cond
      ((null? id-list)
         (error 'ribassoc "id not found"))
      ((equal? id (car id-list))
         (car vals))
      (else
         (ribassoc id (cdr id-list) (cdr vals))))))
```

6 MiniJava Grammar

Goal	::=	*MainClass*
		*TypeDeclaration**
MainClass	::=	`class` *Identifier* `{`
		`public static void main`
		`( String [ ]` *Identifier* `) {`
		MethodBody
		`}`
		`}`
TypeDeclaration	::=	*InterfaceDeclaration*
	|	*InterfaceExtends*
	|	*ClassDeclaration*
InterfaceDeclaration	::=	`interface` *Identifier* *InterfaceBody*
InterfaceExtends	::=	`interface` *Identifier*
		`extends` *Identifier* *InterfaceBody*
InterfaceBody	::=	`{` *InterfaceMember** `}`
InterfaceMember	::=	*ResultType* *Identifier* `(` *Type* *Identifier* `) ;`

ClassDeclaration	::=	`class` *Identifier* `implements` *Identifier* `{`
		*ClassBodyDeclaration** `}`
ClassBodyDeclaration	::=	*FieldDeclaration*
	|	*MethodDeclaration*
FieldDeclaration	::=	*Type* *Identifier* `;`
MethodDeclaration	::=	`public` *ResultType* *Identifier*
		`(` *Type* *Identifier* `) {` *MethodBody* `}`
MethodBody	::=	*StatementListReturn*
	|	*StatementList*

MiniJava Grammar

$$
\begin{array}{rcl}
\mathit{Type} & ::= & \mathit{BooleanType} \\
 & | & \mathit{IntegerType} \\
 & | & \mathit{Identifier} \\
\mathit{BooleanType} & ::= & \texttt{boolean} \\
\mathit{IntegerType} & ::= & \texttt{int} \\
\mathit{ResultType} & ::= & \mathit{VoidType} \\
 & | & \mathit{Type} \\
\mathit{VoidType} & ::= & \texttt{void}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Block} & ::= & \texttt{\{}\ \mathit{StatementList}\ \texttt{\}} \\
\mathit{StatementListReturn} & ::= & \mathit{StatementList}\ \texttt{return}\ \mathit{Expression}\ \texttt{;} \\
\mathit{StatementList} & ::= & \mathit{Statement}^{*} \\
\mathit{Statement} & ::= & \mathit{Block} \\
 & | & \mathit{AssignmentStatement} \\
 & | & \mathit{IfStatement} \\
 & | & \mathit{MessageSendStatement} \\
 & | & \mathit{PrintStatement} \\
\mathit{AssignmentStatement} & ::= & \mathit{Identifier}\ \texttt{=}\ \mathit{Expression}\ \texttt{;} \\
\mathit{IfStatement} & ::= & \texttt{if}\ \texttt{(}\ \mathit{Expression}\ \texttt{)}\ \mathit{Statement} \\
 & & \quad \texttt{else}\ \mathit{Statement} \\
\mathit{MessageSendStatement} & ::= & \mathit{MessageSend}\ \texttt{;} \\
\mathit{PrintStatement} & ::= & \texttt{System.out.println(}\ \mathit{Expression}\ \texttt{)}\ \texttt{;}
\end{array}
$$

MiniJava Grammar

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

Record Definitions

```
(define-record Goal (MainClass List Token))
(define-record MainClass (Token1 Identifier1 Token2
   Token3 Token4 Token5 Token6
   Token7 Token8 Token9 Token10
   Identifier2 Token11 Token12 MethodBody
   Token13 Token14))
(define-record InterfaceDeclaration (Token Identifier
   InterfaceBody))
(define-record InterfaceExtends (Token1 Identifier1 Token2
   Identifier2 InterfaceBody))
(define-record InterfaceBody (Token1 List Token2))
(define-record InterfaceMember (ResultType Identifier1 Token1
   Type Identifier2 Token2 Token3))
(define-record ClassDeclaration (Token1 Identifier1 Token2
   Identifier2 Token3 List Token4))
(define-record FieldDeclaration (Type Identifier Token))
(define-record MethodDeclaration (Token1 ResultType
   Identifier1 Token2 Type Identifier2 Token3
   Token4 MethodBody Token5))
(define-record BooleanType (Token))
(define-record IntegerType (Token))
(define-record VoidType (Token))
(define-record Block (Token1 StatementList Token2))
(define-record StatementListReturn (StatementList Token1
   Expression Token2))
(define-record StatementList (List))
```

```
(define-record AssignmentStatement (Identifier Token1
   Expression Token2))
(define-record IfStatement (Token1 Token2 Expression
   Token3 Statement1 Token4 Statement2))
(define-record MessageSendStatement (MessageSend Token))
(define-record PrintStatement (Token1 Token2 Expression
   Token3 Token4))
(define-record CompareExpression (PrimaryExpression1 Token
   PrimaryExpression2))
(define-record PlusExpression (PrimaryExpression1 Token
   PrimaryExpression2))
(define-record MinusExpression (PrimaryExpression1 Token
   PrimaryExpression2))
(define-record MessageSend (PrimaryExpression Token1
   Identifier Token2 Expression Token3))
(define-record IntegerLiteral (Token))
(define-record TrueLiteral (Token))
(define-record FalseLiteral (Token))
(define-record Identifier (Token))
(define-record ThisExpression (Token))
(define-record AllocationExpression (Token1 Identifier Token2
   Token3))
(define-record BracketExpression (Token1 Expression Token2))
```

To represent objects we will use:

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

7 A MiniJava Interpreter

```
(load "/homes/palsberg/.www/cs565/F98/code/recscm.scm")
(load "records")
(load "tree")

(define run
  (lambda ()
    (record-case root
      (Goal (MainClass List Token)
        (run-MainClass MainClass List))
      (else (error 'run "Goal not found")))))

(define run-MainClass
  (lambda (MainClass class-env)
    (record-case MainClass
      (MainClass (Token1 Identifier1 Token2
                  Token3 Token4 Token5 Token6
                  Token7 Token8 Token9 Token10
                  Identifier2 Token11 Token12 MethodBody
                  Token13 Token14)
        (eval-MethodBody MethodBody class-env (empty-env)
                         'no-this))
      (else (error 'run-MainClass "MainClass not found")))))
```

eval-MethodBody

```
(define eval-MethodBody
  (lambda (MethodBody class-env env this)
    (record-case MethodBody
      (StatementListReturn (StatementList Token1 Expression
                            Token2)
        (eval-StatementList StatementList class-env env this)
        (eval-Expression Expression class-env env this))
      (StatementList (List)
        (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 (List)
        (for-each
          (lambda (Statement)
            (eval-Statement Statement class-env env this))
          List)))))
```

eval-Statement

```
(define eval-Statement
  (lambda (Statement class-env env this)
    (record-case Statement
      (Block (Token1 StatementList Token2)
        (eval-StatementList StatementList class-env env this))
      (AssignmentStatement (Identifier Token1 Expression
                            Token2)
        (set-cell!
          (apply-env env (Identifier->Token Identifier))
          (eval-Expression Expression class-env env this)))
      (IfStatement (Token1 Token2 Expression Token3 Statement1
                    Token4 Statement2)
        (if (eval-Expression Expression class-env env this)
            (eval-Statement Statement1 class-env env this)
            (eval-Statement Statement2 class-env env this)))
      (MessageSendStatement (MessageSend Token)
        (eval-Expression MessageSend class-env env this))
      (PrintStatement (Token1 Token2 Expression Token3 Token4)
        (display (eval-Expression Expression
                                  class-env env this))
        (newline))
      (else (error 'eval-Statement "Statement not found")))))
```

eval-Expression

```
(define eval-Expression
  (lambda (Expression class-env env this)
    (record-case Expression
      (CompareExpression   (PrimaryExpression1 Token
                            PrimaryExpression2)
        (< (eval-Expression PrimaryExpression1
                            class-env env this)
           (eval-Expression PrimaryExpression2
                            class-env env this)))
      (PlusExpression      (PrimaryExpression1 Token
                            PrimaryExpression2)
        (+ (eval-Expression PrimaryExpression1
                            class-env env this)
           (eval-Expression PrimaryExpression2
                            class-env env this)))
      (MinusExpression     (PrimaryExpression1 Token
                            PrimaryExpression2)
        (- (eval-Expression PrimaryExpression1
                            class-env env this)
           (eval-Expression PrimaryExpression2
                            class-env env this)))
```

eval-Expression

```
(MessageSend (PrimaryExpression Token1 Identifier Token2
              Expression Token3)
  (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
                   (find-method CBDL Identifier)))
    (record-case MethodDeclaration
      (MethodDeclaration (Token1 ResultType Identifier1
         Token2 Type Identifier2 Token3
         Token4 MethodBody Token5)
         (eval-MethodBody
              MethodBody class-env
              (extend-env Identifier2
                          argument (object->env receiver))
              receiver)))))
```

eval-Expression

```
(IntegerLiteral (Token)
  (string->number Token))
(TrueLiteral ()
  #t)
(FalseLiteral ()
  #f)
(Identifier (Token)
  (deref-cell (apply-env env Token)))
(ThisExpression (Token)
  this)
(AllocationExpression (Token1 Identifier Token2 Token3)
  (make-object Identifier
               (allocate (apply-class-env class-env
                                          Identifier))))
(BracketExpression (Token1 Expression Token2)
  (eval-Expression Expression class-env env this))
(else (error 'eval-Expression "Expression not found")))))
```

Finding Classes

```
;;; search procedures

(define apply-class-env
                        ;; returns a ClassBodyDeclarationList
  (lambda (class-env class-id)
                        ;; class-env is a TypeDeclarationList
    (if (null? class-env)
        (error 'apply-class-env "Class not found")
        (record-case (car class-env)
          (ClassDeclaration (Token1 Identifier1 Token2
                             Identifier2 Token3 List Token4)
            (if (equal? Identifier1 class-id)
                List
                (apply-class-env (cdr class-env) class-id)))
          (else (apply-class-env (cdr class-env) class-id))))))
```

Finding Methods

```
(define find-method          ;; returns a MethodDeclaration
  (lambda (ClassBodyDeclarationList method-id)
    (if (null? ClassBodyDeclarationList)
        (error 'find-method "Method not found")
        (record-case (car ClassBodyDeclarationList)
          (FieldDeclaration (Type Identifier Token)
            (find-method (cdr ClassBodyDeclarationList)
                         method-id))
          (MethodDeclaration (Token1 ResultType Identifier1
                              Token2 Type Identifier2 Token3
                              Token4 MethodBody Token5)
            (if (equal? Identifier1 method-id)
                (car ClassBodyDeclarationList)
                (find-method (cdr ClassBodyDeclarationList)
                             method-id))))))))
```

Allocating Objects

```
(define allocate          ;; returns an env
  (lambda (ClassBodyDeclarationList)
    (allocate-CBDL ClassBodyDeclarationList (empty-env))))

(define allocate-CBDL     ;; returns an env
  (lambda (ClassBodyDeclarationList env)
    (if (null? ClassBodyDeclarationList)
        env
        (record-case (car ClassBodyDeclarationList)
          (FieldDeclaration (Type Identifier Token)
            (allocate-CBDL (cdr ClassBodyDeclarationList)
                           (extend-env Identifier
                                       '*undefined env)))
          (MethodDeclaration (Token1 ResultType Identifier1
                              Token2 Type Identifier2 Token3
                              Token4 MethodBody Token5)
            (allocate-CBDL (cdr ClassBodyDeclarationList)
                           env))))))
```

Environments and Cells

```
;;; env

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

(define apply-env
  (lambda (env id)
    (display env) (newline) (display id) (newline)
    (cadr (assoc id env))))

(define extend-env
  (lambda (id value env)
    (cons (list (Identifier->Token id) (make-cell value))
          env)))

;;; Cells

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

(define deref-cell cdr)

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

8 Lambda-calculus

Here is a tiny subset of Scheme, called the lambda-calculus:

$$\begin{array}{rcl} \mathit{Expression} & ::= & \mathit{Identifier} \mid \mathit{ProcExp} \mid \mathit{AppExp} \\ \mathit{ProcExp} & ::= & \texttt{(lambda (} \ \mathit{Identifier} \ \texttt{)} \ \mathit{Expression} \ \texttt{)} \\ \mathit{AppExp} & ::= & \texttt{(} \ \mathit{Expression} \ \mathit{Expression} \ \texttt{)} \end{array}$$

The only values that can be computed are *procedures*, and the only computation that can take place is *procedure application*.

The traditional syntax for procedures in the lambda-calculus uses the Greek letter λ (lambda), and the grammar for the lambda-calculus can be written:

$$\begin{array}{rcl} e & \in & Expression \\ e & ::= & x \mid \lambda x.e \mid e_1 e_2 \\ x & \in & Identifier \quad \text{(infinite set of variables)} \end{array}$$

In the lambda-calculus, brackets are only used for grouping of expressions. There is a standard convention for saving brackets that says that the body of a λ-abstraction extends "as far as possible." For example, $\lambda x.xy$ is short for $\lambda x.(xy)$ and not $(\lambda x.x)y$. Moreover, $e_1e_2e_3$ is short for $(e_1e_2)e_3$ and not $e_1(e_2e_3)$.

9 Operational Semantics

An interpreter can be viewed as an executable description of the semantics of a programming language.

We will now present more formal ways of describing the operational semantics of programming languages. We will cover two of the main approaches:

- **Big-step semantics** with environments.
- **Small-step semantics** with syntactic substitution.

Big-step semantics with environments is close in spirit to the interpreters we have seen earlier.

Small-step semantics with syntactic substitution formalizes the inlining of a procedure call as an approach to computation.

We will give the semantics for the following extension of the lambda-calculus:

$$
\begin{array}{lcl}
e & \in & Expression \\
e & ::= & x \mid \lambda x.e \mid e_1 e_2 \mid c \mid \mathsf{succ}\ e \\
x & \in & Identifier \quad \text{(infinite set of variables)} \\
c & \in & IntegerConstant
\end{array}
$$

Big-Step Semantics

Here is a big-step semantics with environments for the lambda-calculus.

We will use the notions of environment and value.

$$
\begin{array}{rcl}
w, v & \in & Value \\
v & ::= & c \mid \langle \lambda x.e, \rho \rangle \\
\rho & \in & Environment \\
\rho & ::= & x_1 \mapsto v_1, \ldots, x_n \mapsto v_n
\end{array}
$$

The semantics is specified by five rules:

$$\rho \vdash x \triangleright v \quad (\rho(x) = v) \tag{1}$$

$$\rho \vdash \lambda x.e \triangleright \langle \lambda x.e, \rho \rangle \tag{2}$$

$$\frac{\rho \vdash e_1 \triangleright \langle \lambda x.e, \rho' \rangle \quad \rho \vdash e_2 \triangleright v \quad \rho', x \mapsto v \vdash e \triangleright w}{\rho \vdash e_1 e_2 \triangleright w} \tag{3}$$

$$\rho \vdash c \triangleright c \tag{4}$$

$$\frac{\rho \vdash e \triangleright c_1}{\rho \vdash \mathsf{succ}\ e \triangleright c_2} \quad (\lceil c_2 \rceil = \lceil c_1 \rceil + 1) \tag{5}$$

Small-Step Semantics

The idea of small-step semantics:

- One step of computation =
  either one **primitive** operation,
  or **inline** one procedure call.

We can do steps of computation in different orders:

```
> (define foo
    (lambda (x y) (+ (* x 3) y)))
> (foo (+ 4 1) 7)
22
```

Let us calculate:

```
      (foo
        (+ 4 1) 7)

=>    ((lambda (x y) (+ (* x 3) y))
        (+ 4 1) 7)

=>    (+ (* (+ 4 1) 3) 7)

=>    22
```

Small-Step Semantics

We can also calculate like this:

```
     (foo
       (+ 4 1) 7)

=>   (foo
       5 7)

=>   ((lambda (x y) (+ (* x 3) y))
       5 7)

=>   (+ (* 5 3) 7)

=>   22
```

Free Variables

A variable x *occurs free* in an expression E if and only if

- E is a variable reference and E is the same as x; or
- E is of the form `(`$E_1 E_2$`)` and x occurs free in E_1 or E_2; or
- E is of the form `(lambda (`y`)` E'`)`, where y is different from x and x occurs free in E'.

Example: no variables occur free in the expression

```
(lambda (y) ((lambda (x) x) y))
```

Another example: the variable `y` occurs free in the expression

```
((lambda (x) x) y)
```

Standard terminology: an expression is *closed* if it does not contain free variables.

Procedure Application

Now consider an application:

`(` *rator* *rand* `)`

To actually perform a procedure application, we have to evaluate the operator (*rator*) to a procedure. This gives:

`( (lambda (`*var*`)` *body* `)` *rand* `)`

The basic Scheme rule for this situation is:

The operand (*rand*) is evaluated to a value *before* the procedure call takes place.

Example:

```
    ((lambda (x) x)
     ((lambda (y) (+ y 9)) 5))

=>  ((lambda (x) x) (+ 5 9))

=>  ((lambda (x) x) 14)

=>  14
```

This is called call-by-value reduction.

The slogan is: "always evaluate the arguments first".

This is the evaluation principle for Scheme and ML.

An Alternative Rule

Consider again the situation:

`( (lambda (`*var*`)` *body* `)` *rand* `)`

Let us now change the Semantics completely!

Here is a new evaluation principle:

The procedure call takes place without evaluating the operand *rand* at all.

Example:

```
    ((lambda (x) x)
     ((lambda (y) (+ y 9) 5))

=>  ((lambda (y) (+ y 9)) 5)

=>  (+ 5 9)

=>  14
```

This is called call-by-name reduction.

The slogan is: "don't work if you can avoid it; be lazy!".

This is the evaluation principle for Miranda and Haskell.

Does it make a Difference?

Now we have one language with two different evaluation principles. That is: two semantics!

What if we take a program and first run it with call-by-value reduction (just run it in Scheme) and then run it again but this time with call-by-name reduction (have to build a new system for this case . . .).

Is there at least one program where the two runs will give different results?

Answer: the following is true of all programs:

- If the run with call-by-value reduction terminates, then the run with call-by-name reduction terminates. (But the converse is in general false).
- If both runs terminate, then they give the same result.

Call-by-value can be too Eager

Sometimes call-by-value reduction fails to terminate, even though call-by-name reduction terminates.

Notice that:

```
(define delta (lambda (x) (x x)))
```

```
    (delta delta)

=>  (delta delta)

=>  (delta delta)

=>  ...
```

Consider the program:

```
((lambda (y) 3) (delta delta))
```

Call-by-value reduction fails to terminate because it continues forever with trying to evaluate the operand.

Call-by-name reduction terminates, simply because it throws away the argument:

```
    ((lambda (y) 3) (delta delta))

=>  3
```

Call-by-value is more Efficient

Consider this call-by-name reduction:

```
    ((lambda (x) (x (x '((2 3)))))
     ((lambda (w) w) car))

=>  (((lambda (w) w) car)
     (((lambda (w) w) car) '((2 3))))

=>  (car (((lambda (w) w) car) '((2 3))))

=>  (car (car '((2 3))))
```

And now, with call-by-value reduction:

```
    ((lambda (x) (x (x '((2 3)))))
     ((lambda (w) w) car))

=>  ((lambda (x) (x (x '((2 3)))))
     car)

=>  (car (car '((2 3))))
```

The second computation is one step shorter!

Why? Because in the first computation, there were taken two copies of the argument before the procedure application `((lambda (w) w) car)` took place. That gave an extra step.

Beta-Redex

A procedure call which is ready to be inlined is of the form:

`( (lambda (`*var*`)` *body* `)` *rand* `)`

Such a pattern is called a *beta-redex*.

We can generalize call-by-value reduction and call-by-name reduction by allowing the next redex chosen to be *any* beta-redex, also one in the body of a **`lambda`**.

The process of inlining a beta-redex for some redex is called *beta-reduction*. The idea is:

`( (lambda (`*var*`)` *body* `)` *rand* `)`

$\Rightarrow$ *body*`[`*var*`:=`*rand*`]`

Here, *body*`[`*var*`:=`*rand*`]` intuitively means "the result is like *body* except that all occurrences of *var* is replaced by *rand*".

It is when we allow arbitrary beta-reduction to take place, that the tiny subset of Scheme is truly the *lambda-calculus*.

Name Clashes

Care must be taken to avoid name clashes.

Example:

```
((lambda (x)
   (lambda (y) (x y)))
 (y 5))
```

should *not* be transformed into

```
 (lambda (y) ((y 5) y))
```

The reference to `y` in `(y 5)` should remain free!

The solution is to change the name of the inner variable name `y` to some name, say `z`, that does not occur free in the argument `y 5`.

```
((lambda (x)
   (lambda (z) (x z)))
 (y 5))
```

Now we get:

```
 (lambda (z) ((y 5) z))
```

Substitution

The notation $e[x := M]$ denotes e with M substituted for every free occurrence of x in such that a way that name clashes are avoided. We will define $e[x := M]$ inductively on e.

$$
\begin{array}{rcl}
x[x := M] & \equiv & M \\
y[x := M] & \equiv & y \quad (x \not\equiv y) \\
(\lambda x.e_1)[x := M] & \equiv & \lambda x.e_1 \\
(\lambda y.e_1)[x := M] & \equiv & \lambda z.((e_1[y := z])[x := M]) \\
 & & \text{(where } x \not\equiv y \text{ and} \\
 & & z \text{ does not occur free in } e_1 \text{ or } M) \\
(e_1\ e_2)[x := M] & \equiv & (e_1[x := M])\ (e_2[x := M]) \\
c[x := M] & \equiv & c \\
(\mathsf{succ}\ e_1)[x := M] & \equiv & \mathsf{succ}\ (e_1[x := M])
\end{array}
$$

The subcase for application always creates a new variable. We can save work by observing that if x does not occur free in e, then there is nothing to do. Moreover, if y does not occur free in M, then we need not rename y to z.

The renaming of a bound variable by

$$\lambda x.e \quad \equiv \quad \lambda z.(e[z := x])$$

(z does not occur free in e), is known as *alpha-conversion.*

Small-Step Semantics

Here is a small-step semantics with syntactic substitution for the lambda-calculus.

We will use the notion of a *value*. A value is either a λ-abstraction $\lambda x.e$ or an integer constant c. We use v to range over values.

A small-step call-by-value operational semantics for the language is given by the reflexive, transitive closure of the relation $\rightarrow_V$:

$$\rightarrow_V \subseteq \; Expression \times Expression$$

$$(\lambda x.e)v \rightarrow_V e[x := v] \tag{6}$$

$$\frac{e_1 \rightarrow_V e_1'}{e_1 e_2 \rightarrow_V e_1' e_2} \tag{7}$$

$$\frac{e_2 \rightarrow_V e_2'}{v\ e_2 \rightarrow_V v\ e_2'} \tag{8}$$

$$\mathsf{succ}\ c_1 \rightarrow_V c_2 \quad (\lceil c_2 \rceil = \lceil c_1 \rceil + 1) \tag{9}$$

$$\frac{e_1 \rightarrow_V e_2}{\mathsf{succ}\ e_1 \rightarrow_V \mathsf{succ}\ e_2} \tag{10}$$

Example

```
(define i
 ((lambda (k1)
    ((lambda (k4) (k4 (lambda (x) (lambda (k5) (k5 x)))))
     (lambda (v1)
       ((lambda (k2) (k2 (lambda (y) (lambda (k3) (k3 y)))))
        (lambda (v2) ((v1 v2) k1)))
     )))
  (lambda (a) a)))
```

Let us calculate `(i 2)`.

```
(i 2)
=>
(((lambda (k4) (k4 (lambda (x) (lambda (k5) (k5 x)))))
   (lambda (v1)
     ((lambda (k2) (k2 (lambda (y) (lambda (k3) (k3 y)))))
      (lambda (v2) ((v1 v2) (lambda (a) a))))
   ))
  2)
=>
(((lambda (v1)
     ((lambda (k2) (k2 (lambda (y) (lambda (k3) (k3 y)))))
      (lambda (v2) ((v1 v2) (lambda (a) a))))
   )
     (lambda (x) (lambda (k5) (k5 x))))
  2)
=>
```

Example

```
(((lambda (k2) (k2 (lambda (y) (lambda (k3) (k3 y)))))
      (lambda (v2) ((
                (lambda (x) (lambda (k5) (k5 x)))
                        v2) (lambda (a) a))))
  2)
=>
(((lambda (v2) ((
                (lambda (x) (lambda (k5) (k5 x)))
                        v2) (lambda (a) a)))
                    (lambda (y) (lambda (k3) (k3 y))))
  2)
=>
((((lambda (x) (lambda (k5) (k5 x)))
                        (lambda (y) (lambda (k3) (k3 y)))
                          ) (lambda (a) a))
  2)
=>
(((lambda (k5) (k5 (lambda (y) (lambda (k3) (k3 y)))))
                            (lambda (a) a))
  2)
=>
(((lambda (a) a) (lambda (y) (lambda (k3) (k3 y))))
  2)
=>
((lambda (y) (lambda (k3) (k3 y)))
  2)
=>
(lambda (k3) (k3 2))
```