CS450 - Structure of Higher Level Languages

Metalinguistic Abstraction

November 2, 2020

Programming Languages Are Means to Model Complex Problems

- To design a complex system (of any kind) we need several general techniques:
- Combine primitive elements to form compound objects
- Abstract compound objects to form higher-level building blocks
- Preserve modularity by adopting appropriate large-scale views of system structure.
- We have used scheme for this purpose, but as our problems become more complex, we may need to resort to new languages that help us express new ideas more effectively.

Metalinguistic Abstraction

- **Metalinguistic abstraction**, establishing new languages, plays an important role in all branches of engineering design.
- It is particularly important to computer programming – we can formulate new languages, and we can also implement these languages by constructing evaluators.
- An evaluator (or interpreter) for a programming language is a procedure that, when applied to an expression of the language, performs the actions required to evaluate that expression.
- It is **very** important to remember that the evaluator, which determines the meaning of expressions in a programming language, is just another program.

Metalinguistic Abstraction

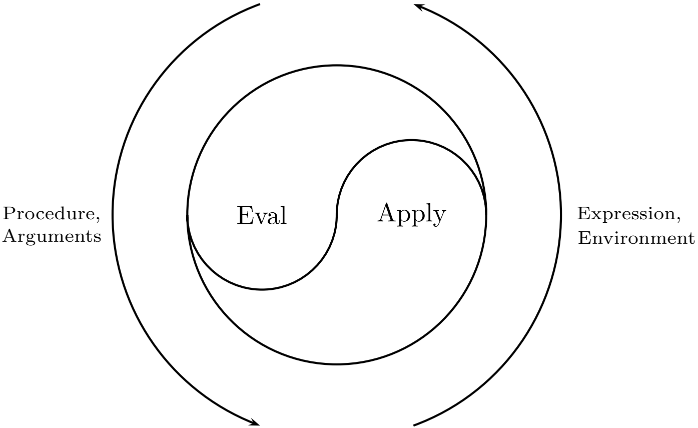
- As a matter of fact, we can think of any program as a "mini-language".
- The complex number package we mentioned earlier is the core of a language that deals with complex numbers, their representations and math operations, using primitives and building higher-level abstractions.
- We can also think of the RSA system in HW3 this way...
- There are several other examples in the book – a digital logic simulator, a polynomial manipulation system etc.

Metalinguistic Abstraction

- In what follows we will use scheme to explore the ability of languages to build other languages.
- We will implement evaluators as procedures.
- Lisp is especially suitable due to its ability to represent and manipulate symbolic expressions.
- We will build an evaluator for Lisp itself.
- The evaluator is a subset of the Scheme language used in the text.
- It is rather simple, yet capable of executing most of the programs in the text...

The Core of the Evaluator: Eval/Apply

The evaluation process can be described as the interplay between two procedures: `eval` and `apply`.



Eval

- Eval takes as arguments an expression and an environment.
- It classifies the expression and directs its evaluation.
- Eval is structured as a case analysis of the syntactic type of the expression to be evaluated.
- We express the determination of the type of an expression abstractly, making no commitment to any particular representation for the various types of expressions.
- The way we implement it allows us to change the syntax of the language by using the same evaluator, but with a different collection of syntax procedures.

Types of Expressions

- **Primitive expressions:**
  - Self evaluating objects (like numbers), are evaluated to themselves.
  - For variables, look up in the environment to find their values.
- **Special forms:**
  - Quoted expressions are evaluated to the expression that was quoted.
  - An assignment to (or a definition of) a variable recursively calls `eval` to compute the new value to be associated with the variable. The environment is modified accordingly.
  - An `if` expression requires special processing of its parts, so as to evaluate the consequent if the predicate is true, or the alternative otherwise.

Types of Expressions

- **Special forms:** (cont.)
  - A `lambda` expression must be transformed into an applicable procedure by packaging together the parameters and body with the environment of the evaluation.
  - A `begin` expression requires evaluating its sequence of expressions in the order in which they appear.
  - A `cond` is transformed into a nested `if` and evaluated.
- **Combinations:**
  - For a procedure application, `eval` must recursively evaluate the operator part and the operands of the combination.
  - The resulting procedure and arguments are passed to apply, which handles the actual procedure application.

Definition of Eval (in text)

```
(define (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)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Definition of Eval

- In most Lisp implementations, `eval` is implemented by dispatching on type.
- This allows more flexibility in adding new types of expressions.
- The way it is implemented here requires us to edit the definition of `eval` whenever we add a new type.
- For our purposes we will use a slightly different version (see handout).

Definition of Apply

- Apply takes two arguments, a procedure and a list of arguments to which the procedure should be applied.
- Apply classifies procedures into two kinds: It calls apply-primitive-procedure to apply primitives;
- Compound procedures are applied by sequentially evaluating the expressions that make up the body of the procedure.
- The environment for the evaluation of the body of a compound procedure is constructed by extending the base environment carried by the procedure to include a frame that binds the parameters of the procedure to the arguments to which the procedure is to be applied.

Definition of Apply (in text)

```
(define (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))))
```

The CS450 Implementation

- In HW6: Special forms to be stored in a 1-D lookup table, like the one we saw earlier on.
- The table is wrapped up in a "dispatch on type" procedure (that's not called dispatch!), which supports insert, lookup and display.
- The `eval` implementation, named `xeval`, uses the table to look up special forms.
- We use tagged data (remember that?) to represent different kinds of expressions (same as the text).

The CS450 Implementation

The values of the operands of an expression are being evaluated in sequence.

```
(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
      (cons (xeval (first-operand exps) env)
            (list-of-values (rest-operands exps) env))))

define (eval-if exp env)
  (if (true? (xeval (if-predicate exp) env))
      (xeval (if-consequent exp) env)
      (xeval (if-alternative exp) env)))
```

Evaluating Different Expressions

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

(define (eval-assignment exp env)
  (let ((name (assignment-variable exp)))
    (set-variable-value! name
 (xeval (assignment-value exp) env)
 env)
  name))    ;; A & S return 'ok

  (define (eval-definition exp env)
  (let ((name (definition-variable exp)))
    (define-variable! name
      (xeval (definition-value exp) env)
      env)
  name))     ;; A & S return 'ok
```

Representing Expressions

```
(define (self-evaluating? exp)
  (or (number? exp)
      (string? exp)
      (boolean? exp) ))

(define (variable? exp) (symbol? exp))

(define (quoted? exp)
  (tagged-list? exp 'quote))

(define (text-of-quotation exp) (cadr exp))

(define (tagged-list? exp tag)
  (if (pair? exp)
      (eq? (car exp) tag)
      #f))
```

Representing Assignments

```
(define (assignment? exp)
  (tagged-list? exp 'set!))

(define (assignment-variable exp) (cadr exp))

(define (assignment-value exp) (caddr exp))
```

Representing Definitions

```
(define (definition? exp)
  (tagged-list? exp 'define))

(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)
      (caadr exp)))

(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp)
                   (cddr exp))))
```

Representing Definitions

- A definition can either be

  `(define <var> <value>)`

  or

  `(define (<var> <par_1> ... <par_n>) <body>)`
- In the second case, the variable is the `caadr` of the expression (the name of the function)
- The value in this case is turned into a lambda expression.
- `(cdadr exp)` is the list of parameters.
- `(cddr exp)` is the body.

Representing Lambda Expressions

```
(define (lambda? exp) (tagged-list? exp 'lambda))

(define (lambda-parameters exp) (cadr exp))
(define (lambda-body exp) (cddr exp))

(define (make-lambda parameters body)
  (cons 'lambda (cons parameters body)))
```

Notice that the list must have at least one other element (except the tag lambda).

Representing If Statements

```
(define (if? exp) (tagged-list? exp 'if))

(define (if-predicate exp) (cadr exp))

(define (if-consequent exp) (caddr exp))

(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      #f))

(define (make-if predicate consequent alternative)
   (list 'if predicate consequent alternative))
```

Representing Sequences

```
(define (begin? exp) (tagged-list? exp 'begin))

(define (begin-actions exp) (cdr exp))

(define (last-exp? seq) (null? (cdr seq)))
(define (first-exp seq) (car seq))
(define (rest-exps seq) (cdr seq))

(define (sequence->exp seq)
  (cond ((null? seq) seq)
        ((last-exp? seq) (first-exp seq))
        (else (make-begin seq))))

(define (make-begin seq) (cons 'begin seq))
```

User Defined Procedures

Procedure applications – any compound expression that is not one of the above expression types.

```
(define (application? exp) (pair? exp))
(define (operator exp) (car exp))
(define (operands exp) (cdr exp))

(define (no-operands? ops) (null? ops))
(define (first-operand ops) (car ops))
(define (rest-operands ops) (cdr ops))
```

Representing cond

Cond is syntactically transformed into a nest of if expressions.

```
(define (cond? exp) (tagged-list? exp 'cond))

(define (cond-clauses exp) (cdr exp))

(define (cond-else-clause? clause)
  (eq? (cond-predicate clause) 'else))

(define (cond-predicate clause) (car clause))

(define (cond-actions clause) (cdr clause))

(define (cond->if exp)
  (expand-clauses (cond-clauses exp)))
```

Representing cond (Cont.)

```
(define (expand-clauses clauses)
  (if (null? clauses)
      #f  ; no else clause -- return #f
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
          (if (null? rest)
              (sequence->exp (cond-actions first))
              (error "ELSE clause isn't last -- COND->IF "
                     clauses))
          (make-if (cond-predicate first)
                   (sequence->exp (cond-actions first))
                   (expand-clauses rest))))))
```

Truth Values and Procedure Objects

```
(define (true? x)
  (not (eq? x #f)))

(define (false? x)
  (eq? x #f))

;;; Procedures
(define (make-procedure parameters body env)
  (list 'procedure parameters body env))

(define (user-defined-procedure? p)
  (tagged-list? p 'procedure))

(define (procedure-parameters p) (cadr p))
(define (procedure-body p) (caddr p))
(define (procedure-environment p) (cadddr p))
```

Representing Environments

- An environment is a list of frames.
- The enclosing environment is the cdr of the current environment.
- Each frame is represented as a pair of lists:
  1. a list of the variables bound in that frame, and
  2. a list of the associated values.
- For HW6 it is **crucial** to understand how environments are represented.

Representing Environments

```
(define (enclosing-environment env) (cdr env))

(define (first-frame env) (car env))

(define the-empty-environment '())

(define (make-frame variables values)
  (cons variables values))

(define (frame-variables frame) (car frame))
(define (frame-values frame) (cdr frame))

(define (add-binding-to-frame! var val frame)
  (set-car! frame (cons var (car frame)))
  (set-cdr! frame (cons val (cdr frame))))
```

Extending an Environment

- Creating a new frame with a set of variables and values.
- Making the base environment the enclosing environment of the new frame.

```
(define (xtend-environment vars vals base-env)
  (if (= (length vars) (length vals))
    (cons (make-frame vars vals) base-env)
    (if (< (length vars) (length vals))
       (error "Too many arguments supplied " vars vals)
       (error "Too few arguments supplied " vars vals))))
```

Looking up a Variable's Value

Scan current frame, if not found – go to the enclosing environment.

```
(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 " var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

Set a Variable's Value

Change value first time it's found.

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

Defining a Variable's Value

Add a binding to current frame, or change value if exists already.

```
(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars)
              (add-binding-to-frame! var val frame))
             ((eq? var (car vars))
              (set-car! vals val))
             (else (scan (cdr vars) (cdr vals)))))
    (scan (frame-variables frame)
          (frame-values frame))))
```

Initial Environment Setup

- The global environment starts up as containing primitive procedures only.
- In HW6 you will need to modify that, and separate the primitive procedure installation from the initial setup.
- In the current setup there are only four primitive procedures installed.
- Think what it means for other primitive procedures... (hint for HW6).

Initial Environment Setup

```
(define (setup-environment)
  (let ((initial-env
         (xtend-environment (primitive-procedure-names)
                            (primitive-procedure-objects)
                            the-empty-environment)))
    initial-env))

(define (primitive-procedure? proc)
  (tagged-list? proc 'primitive))

(define (primitive-implementation proc) (cadr proc))

(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
;;      more primitives
        ))
```

Initial Environment Setup

```
(define (primitive-procedure-names)
  (map car
       primitive-procedures))

(define (primitive-procedure-objects)
  (map (lambda (proc) (list 'primitive (cadr proc)))
       primitive-procedures))

;;; Here is where we rely on the underlying Scheme
;;; implementation to know how to apply
;;; a primitive procedure.

(define (apply-primitive-procedure proc args)
  (apply (primitive-implementation proc) args))
```

The Main Driver Loop

- `read` returns an internal representation of the next expression.
- It does not evaluate anything.
- `xeval` does the actual evaluation.

```
(define input-prompt "s450==> ")

(define (s450)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (xeval input the-global-environment)))
      (user-print output)))
  (s450))

(define (prompt-for-input string)
  (newline) (newline) (display string))
```

The Main Driver Loop

```
(define (user-print object)
  (if (user-defined-procedure? object)
      (display (list 'user-defined-procedure
                     (procedure-parameters object)
                     (procedure-body object)
                     '<procedure-env>))
      (display object)))

(define the-global-environment (setup-environment))

(display "... loaded the metacircular evaluator.
(s450) runs it.")
(newline)
```