



COMPUTER SCIENCE 320

Introduction to Programming Languages

Variations on a Scheme: Notes on

Dynamic Binding and Normal Order Evaluation

Dynamic Binding

In the usual Scheme implementation, the environment in which a procedure is *evaluated* is statically determined from the environment in which the procedure is *defined*.

Why not the following instead? Let the environment in which evaluation occurs be determined from the *dynamic* environment in which the application occurs:

```
(define (accumulate op nullvalue lst)
  (acc lst))
```

```
(define (acc lst)
  (if (null? lst)
      nullvalue
      (op (car lst)
          (acc (cdr lst)))))
```

This instead of the usual

```
(define (accumulate op nullvalue lst)
  (if (null? lst)
      nullvalue
      (op (car lst)
          (accumulate op nullvalue (cdr lst)))))
```

How does the Scheme implementation need to change to accommodate this *dynamic binding* discipline? *How and where are frames attached to the environment?*

Dynamic Binding: Eval and Apply

```
(define (eval exp env)
  (cond ...
    ((lambda? exp)
     (make-procedure (lambda-parameters exp)
                      (lambda-body exp)))
    ...
    ((application? exp)
     (apply (eval (operator exp) env)
             (list-of-values (operands exp) env)
             env))
    (else
     (error "Unknown expression type -- EVAL" exp))))

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

Normal Order Evaluation

In normal order evaluation, we delay the computation of everything we can. *Delaying* means the construction of a *thunk*: a data object having (unevaluated) *code*, and an *environment* in which it is to be (later) evaluated.

When a user-defined (compound) procedure is *applied*, we bind each argument to a thunk.

Procrastination stops at the following points:

Read-eval-print loop: You shouldn't print a thunk—instead, you should evaluate it.

Applying compound procedures: You can't apply a thunk—you need to force its evaluation so you get a procedure to apply.

Applying primitive procedures: You can't add two thunks, or take the **car** of one. What about **cons**? (A design decision.)

Conditionals: When evaluating an **if** expression, a thunk isn't good enough—you need to evaluate it to see if it is true or false.

Driver loop

```
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output
           (actual-value input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
    (driver-loop)))
```

If an expression evaluates to a thunk, procedure **actual-value** forces evaluation of the thunk until a value is produced.

```

(define (eval exp env)
  (cond ...
    ((application? exp)
     (apply (actual-value (operator exp) env)
             (operands exp)
             env))
    (else
     (error "Unknown expression type -- EVAL" exp))))

```

Procedure *apply* takes unevaluated arguments and their environment as parameters.

```

(define (apply procedure arguments env)
  (cond ((primitive-procedure? procedure)
        (apply-primitive-procedure
         procedure
         (list-of-arg-values arguments env))) ; changed
        ((compound-procedure? procedure)
        (eval-sequence
         (procedure-body procedure)
         (extend-environment
          (procedure-parameters procedure)
          (list-of-delayed-args arguments env) ; changed
          (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))

```

In compound procedure application, the parameters are thunked. In primitive procedure application, all thunked parameters are forced.

List of values:

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

List of thunks:

```
(define (list-of-delayed-args exps env)
  (if (no-operands? exps)
      '()
      (cons (delay-it (first-operand exps) env)
            (list-of-delayed-args (rest-operands exps)
                                  env))))
```

In evaluating a conditional, force evaluation of the predicate:

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

Thunks

```
(define (actual-value exp env)
  (force-it (eval exp env)))
```

```
(define (force-it obj)
  (if (thunk? obj)
      (actual-value (thunk-exp obj) (thunk-env obj))
      obj))
```

```
(define (delay-it exp env)
  (list 'thunk exp env))
```

```
(define (thunk? obj)
  (tagged-list? obj 'thunk))
```

```
(define (thunk-exp thunk) (cadr thunk))
(define (thunk-env thunk) (caddr thunk))
```

Why thunks need recursive forcing:

```
(define (A x) (B x))
(define (B y) (C y))
(define (C z) (D z))
(define (D w) (1+ w))
```

```
(A 10)
```