

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
 Department of Electrical Engineering and Computer Science
 6.001—Structure and Interpretation of Computer Programs
 Fall Semester, 1998

Lecture Notes – November 12, 1998

Analysis and Nondeterministic Computation

Syntactic Analysis

The metacircular evaluator may perform a lot of redundant computation. Why is the following so inefficient?

```
(define (factorial n)
  (if (= n 1)
      1
      (* (factorial (- n 1)) n)))

(factorial 10)
```

We can redesign the evaluator to ANALYZE each expression only once, and allow us to EXECUTE it as many times as we want. We essentially *curry* the `mc-eval` procedure to separate these two notions. The `analyze` procedure creates an *execution object*, which is a procedure that can be used to compute an evaluation by applying it to an environment.

```
;; Evaluation consists of (1) analysis, followed by (2) execution in an environment.
(define (mc-eval exp env)
  ((analyze exp) env))

;; Analyze returns (lambda (env) ...)
(define (analyze exp)
  (cond ((self-evaluating? exp)
        (analyze-self-evaluating exp))
        ((quoted? exp) (analyze-quoted exp))
        ((variable? exp) (analyze-variable exp))
        ((assignment? exp) (analyze-assignment exp))
        ((definition? exp) (analyze-definition exp))
        ((if? exp) (analyze-if exp))
        ((lambda? exp) (analyze-lambda exp))
        ((begin? exp) (analyze-sequence (begin-actions exp)))
        ((cond? exp) (analyze (cond->if exp)))
        ((application? exp) (analyze-application exp))
        (else
         (error "Unknown expression type -- ANALYZE" exp))))
```

```

(define (analyze-self-evaluating exp)
  (lambda (env) exp))

(define (analyze-quoted exp)
  (let ((qval (text-of-quotation exp)))
    (lambda (env) qval)))

(define (analyze-variable exp)
  (lambda (env) (lookup-variable-value exp env)))

(define (analyze-assignment exp)
  (let ((var (assignment-variable exp))
        (value-exe (analyze (assignment-value exp))))
    (lambda (env)
      (set-variable-value! var (value-exe env) env)
      'ok)))

(define (analyze-if exp)
  (let ((pred-exe (analyze (if-predicate exp)))
        (con-exe (analyze (if-consequent exp)))
        (alt-exe (analyze (if-alternative exp))))
    (lambda (env)
      (if (true? (pred-exe env))
          (con-exe env)
          (alt-exe env)))))

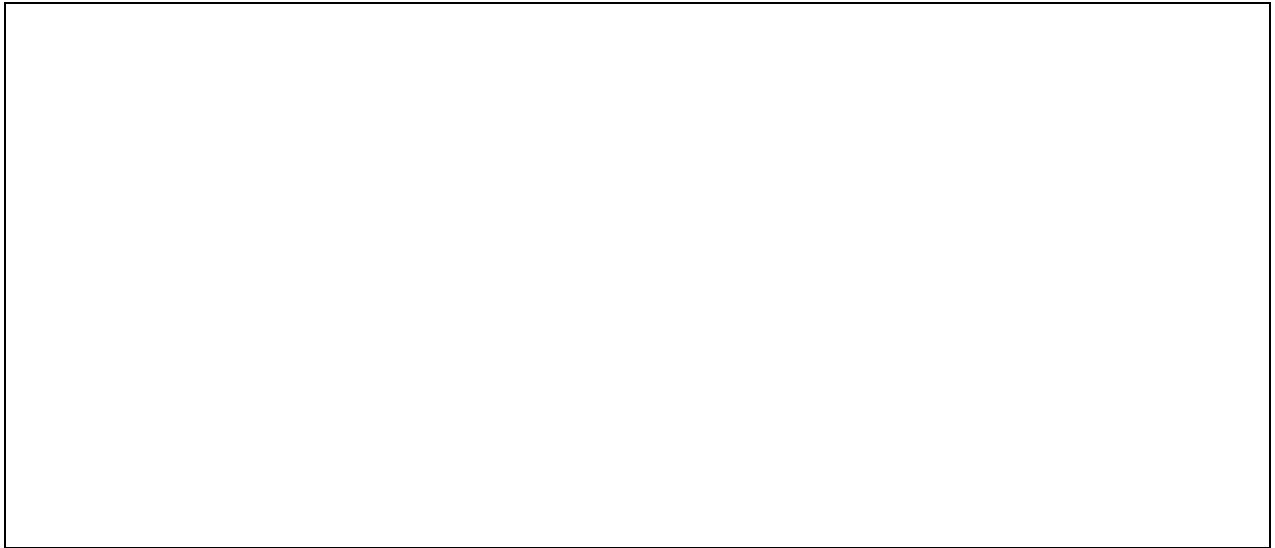
(define (analyze-lambda exp)
  (let ((vars (lambda-parameters exp))
        (body-exe (analyze-sequence (lambda-body exp))))
    (lambda (env) (make-procedure vars body-exe env))))

(define (analyze-application exp)
  (let ((op-exe (analyze (operator exp)))
        (arg-exes (map analyze (operands exp))))
    (lambda (env)
      (execute-application (op-exe env)
                           (map (lambda (arg-exe) (arg-exe env))
                                arg-exes)))))

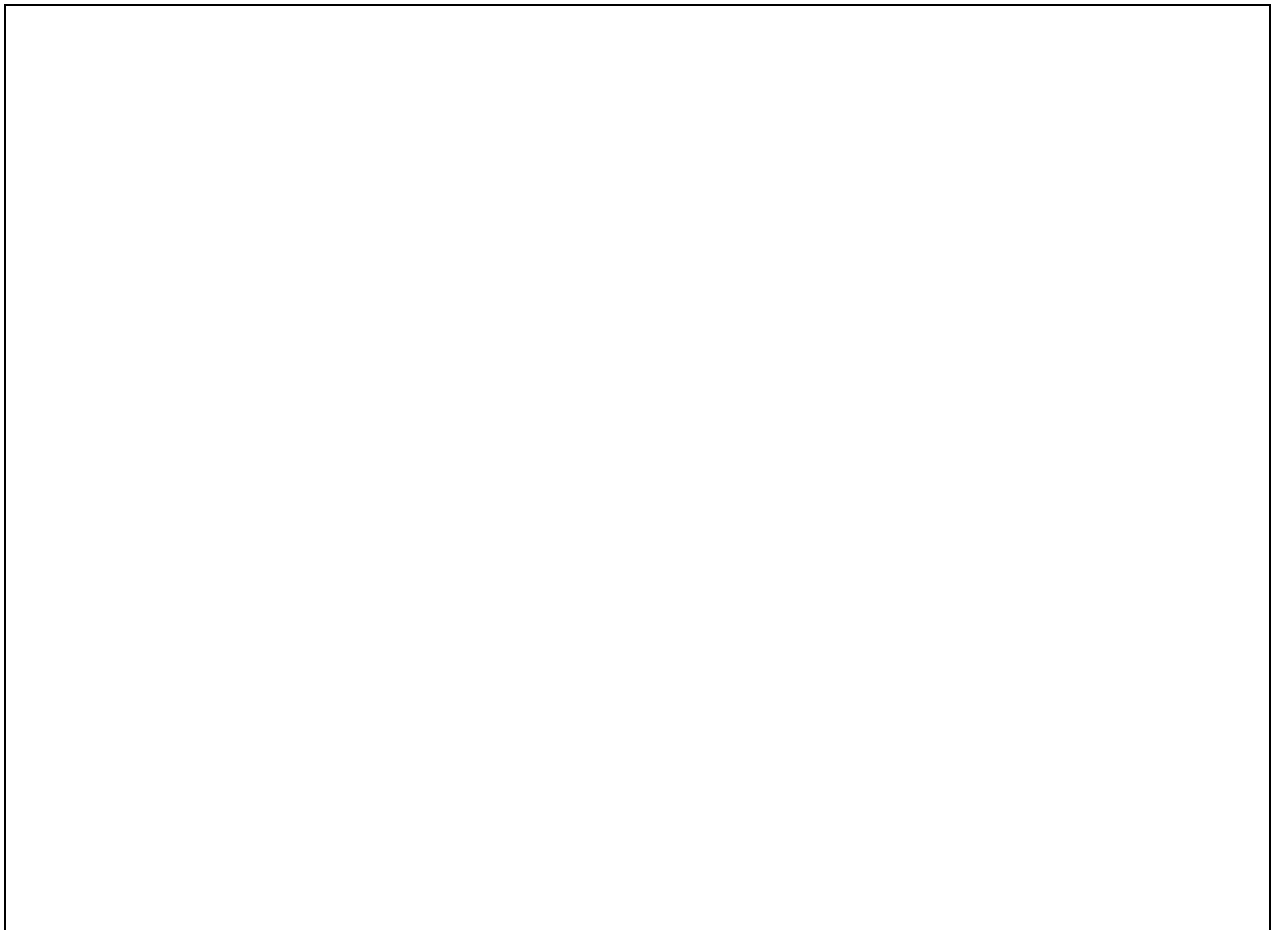
(define (execute-application proc args)
  (cond ((primitive-procedure? proc)
        (apply-primitive-procedure proc args))
        ((compound-procedure? proc)
         ((procedure-body proc)
          (extend-environment (procedure-parameters proc)
                            args
                            (procedure-environment proc))))
        (else
         (error "Unknown procedure type -- EXECUTE-APPLICATION" proc))))

```

Draw the execution object used in `((analyze 'y) t-g-e)`:



Draw the execution object used in `((analyze '(if #t 1 2)) t-g-e)`



Nondeterministic Computing - Amb Eval

Nondeterminism can be introduced into Scheme with the new special form `amb`. The idea is that when the interpreter reaches an expression of the form `(amb e1 e2)` it “nondeterministically” selects one of expressions e_1 or e_2 and continues the evaluation with that expression. Since the language with `amb` is nondeterministic, a given expression can have many different possible executions and many different possible values. In general, the special form `amb` can take any number of arguments, one of which is nondeterministically selected at run time.

As an example consider

```
(define (a-number-between low high)
  (cond ((= low high) low)
        (else (amb low (a-number-between (+ low 1) high)))))
```

The expression `(a-number-between 1 10)` has ten different possible values, each of which is an integer between 1 and 10.

Consider the following procedure.

```
(define (a-pythagorean-triple-between low high)
  (let ((n (a-number-between low high))
        (m (a-number-between low high))
        (p (a-number-between low high)))
    (cond ((= (+ (* n n) (* m m)) (* p p))
           (list n m p))
          (else (amb)))))
```

The expression `(a-pythagorean-triple-between 1 10)` has 1000 different possible executions. All but four of these executions fail. The four non-failing executions have values (3 4 5), (4 3 5), (6 8 10), and (8 6 10). In general, if `(n m p)` is a non-failing value of the expression `(a-pythagorean-triple-between low high)` then `n`, `m`, and `p` are numbers between `low` and `high` such that $n^2 + m^2 = p^2$. In other words, there exists a right triangle whose sides have length `n`, `m`, and `p`.

One way to solve such problems is to exhaustively search out all possibilities, filtering out those that violate some constraint. We can more conveniently express the requirement that a particular predicate expression `p` is true by **requiring** its truth:

```
(define (require p)
  (cond ((not p) (amb))
        (else 'ok)))

(define (a-pythagorean-triple-between low high)
  (let ((n (a-number-between low high))
        (m (a-number-between low high))
        (p (a-number-between low high)))
    (require (= (+ (* n n) (* m m)) (* p p)))
    (list n m p)))
```

We are often interested in determining whether or not a given nondeterministic expression has a nonfailing value. One might try running the probabilistic interpreter over and over again hoping to find a nonfailing value. A much better approach is to *systematically search* the space of all possible executions. We will play with a modified version of the evaluator which we call the *search* or *amb evaluator*. The amb evaluator systematically searches the space of all possible executions of a given expression. When the amb evaluator encounters an application of `amb` it initially selects the first argument. If this selection does not result in a nonfailing execution, then the evaluator “backs up” and tries the second argument. The amb evaluator is used to implement a read-eval-print loop with some unusual properties. The read-eval-print loop reads an expression and prints the value of the first nonfailing execution. For example, consider the following interaction.

```
;;; Amb-Eval input:
(a-number-between 1 10)
;;; Starting a new problem
;;; Amb-Eval value:
1

;;; Amb-Eval input:
try-again
;;; Amb-Eval value:
2

;;; Amb-Eval input:
(a-pythagorean-triple-between 1 10)
;;; Starting a new problem
;;; Amb-Eval value:
(3 4 5)
```

The read-eval-print loop allows the user to ask for alternative executions of an expression. If the first execution is not desired, or if one simply wants to see the value of the next successful execution, one can ask the interpreter to back up and attempt to generate a second nonfailing execution.

An example where searching over possible evaluations is useful is given by the following elementary logic puzzle:

Multiple Dwelling ¹

Baker, Cooper, Fletcher, Miller, and Smith live on different floors of an apartment house that contains only five floors.

Baker does not live on the top floor.

Cooper does not live on the bottom floor.

Fletcher does not live on either the top or the bottom floor.

Miller lives on a higher floor than does Cooper.

Smith does not live on a floor adjacent to Fletcher's.

Fletcher does not live on a floor adjacent to Cooper's.

Where does everyone live?

¹This puzzle is typical of a large class of puzzles. This particular one is quoted from *Superior Mathematical Puzzles*, by Howard P. Dinesman, 1968, Simon and Schuster, New York.

In this case, we can consider possible assignments of the characters to floors in a building. We will also need to be able to tell when a list is made up of distinct entries (no two are the same):

```
(define (distinct list)
  (cond ((null? list) true)
        ((null? (cdr list)) true)
        ((member (car list) (cdr list)) false)
        (else (distinct (cdr list)))))

(define (multiple-dwelling)
  (let ((baker (amb 1 2 3 4 5))
        (cooper (amb 1 2 3 4 5))
        (fletcher (amb 1 2 3 4 5))
        (miller (amb 1 2 3 4 5))
        (smith (amb 1 2 3 4 5)))
    ;; Beginning of filtration
    (require (distinct (list baker cooper fletcher miller smith)))
    (require (not (= baker 5)))
    (require (not (= cooper 1)))
    (require (not (= fletcher 5)))
    (require (not (= fletcher 1)))
    (require (> miller cooper))
    (require (not (= (abs (- smith fletcher)) 1)))
    (require (not (= (abs (- fletcher cooper)) 1)))
    (list (list 'baker baker)
          (list 'cooper cooper)
          (list 'fletcher fletcher)
          (list 'miller miller)
          (list 'smith smith))))
```

Indeed, we can try it:

```
;;; Amb-Eval input:
(multiple-dwelling)
;;; Starting a new problem
;;; Amb-Eval value:
((baker 3) (cooper 2) (fletcher 4) (miller 5) (smith 1))

;;; Amb-Eval input:
try-again
;;; There are no more values of (multiple-dwelling)
```

Amb Eval Implementation

Here we discuss the basic ideas in the implementation of the Amb evaluator; the full code is in the book. Our strategy is to extend the notion of the *execution object* from the analyze evaluator. Now it becomes a procedure that can be used to complete an evaluation, and can *also* be used to try for more values from an evaluation. This execution object takes an environment, a succeed procedure, and a fail procedure. For example, the `analyze-self-evaluating` code now becomes:

```
(define (analyze-self-evaluating exp)
  (lambda (env succeed fail)
    (succeed exp fail)))
```

Draw a picture corresponding to the execution object used in `((analyze '1) env mainloop-succeed mainloop-fail)`:



We will consider here a simplified version of the read-eval-print loop for the Amb evaluator (the full one is in the book) which simply consists of the `mainloop-succeed` and `mainloop-fail` procedures. These describe what should happen when a value is found (tell the user, and then go back for another value if the user wants one), and when no value is found (tell the user “no more values”).

```
(define (mainloop-succeed value fail-continuation)
  (display "got a value" value)
  (let ((exp (read)))
    (if (eq? exp 'try-again)
        (fail-continuation)
        (amb-eval exp t-g-e mainloop-succeed mainloop-fail))))
```

```
(define (mainloop-fail)
  (display "no more values"))
```

Assuming the user does in fact request a second value, we can use the execution object above to consider the steps that occur in the evaluation of `((analyze '1) env mainloop-succeed mainloop-fail)`.

The `amb` expression is at the heart of the backtracking abilities of the `amb` evaluator:

```
(define (analyze-amb exp)
  (let ((choice-exes (map analyze (amb-choices exp))))
    (lambda (env succeed fail)
      (define (try-next choices)
        (if (null? choices)
            (fail)
            ((car choices) env
              succeed
              (lambda () (try-next (cdr choices))))))
        (try-next choice-exes))))
```

Draw a picture representing the execution object resulting from
`((analyze '(amb 1 2)) env mainloop-succeed mainloop-fail):`



We may also need to *undo* any side-effects (i.e. assignments) when we backtrack to try a different computation path. This is taken care of in the `exe` object created by the *analyze-assignment* procedure:

```
(define (analyze-assignment exp)
  (let ((var (assignment-variable exp))
        (value-exe (analyze (assignment-value exp))))
    (lambda (env succeed fail)
      (value-exe env
        (lambda (val fail2) ; *1*
          (let ((old-value (lookup-variable-value var env)))
            (set-variable-value! var val env)
            (succeed 'ok
              (lambda () ; *2*
                (set-variable-value! var old-value env)
                (fail2))))))
        fail))))
```