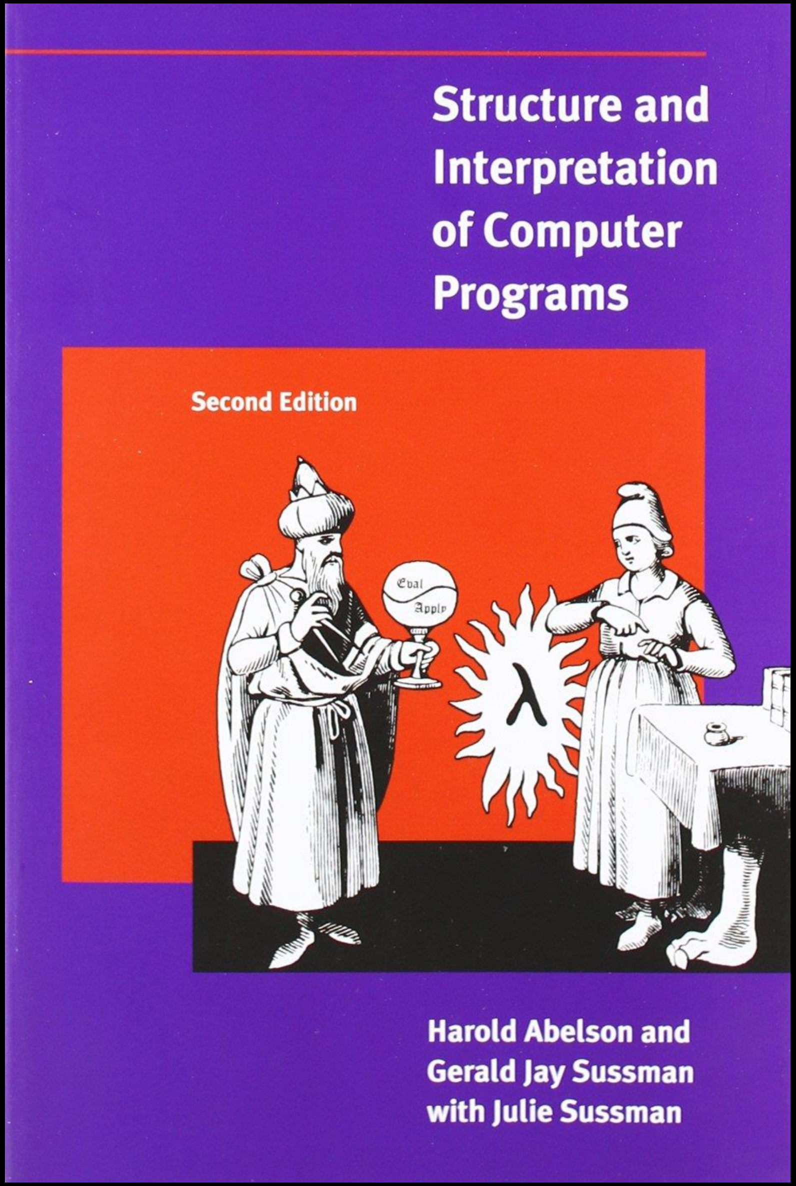


# Structure and Interpretation of Computer Programs

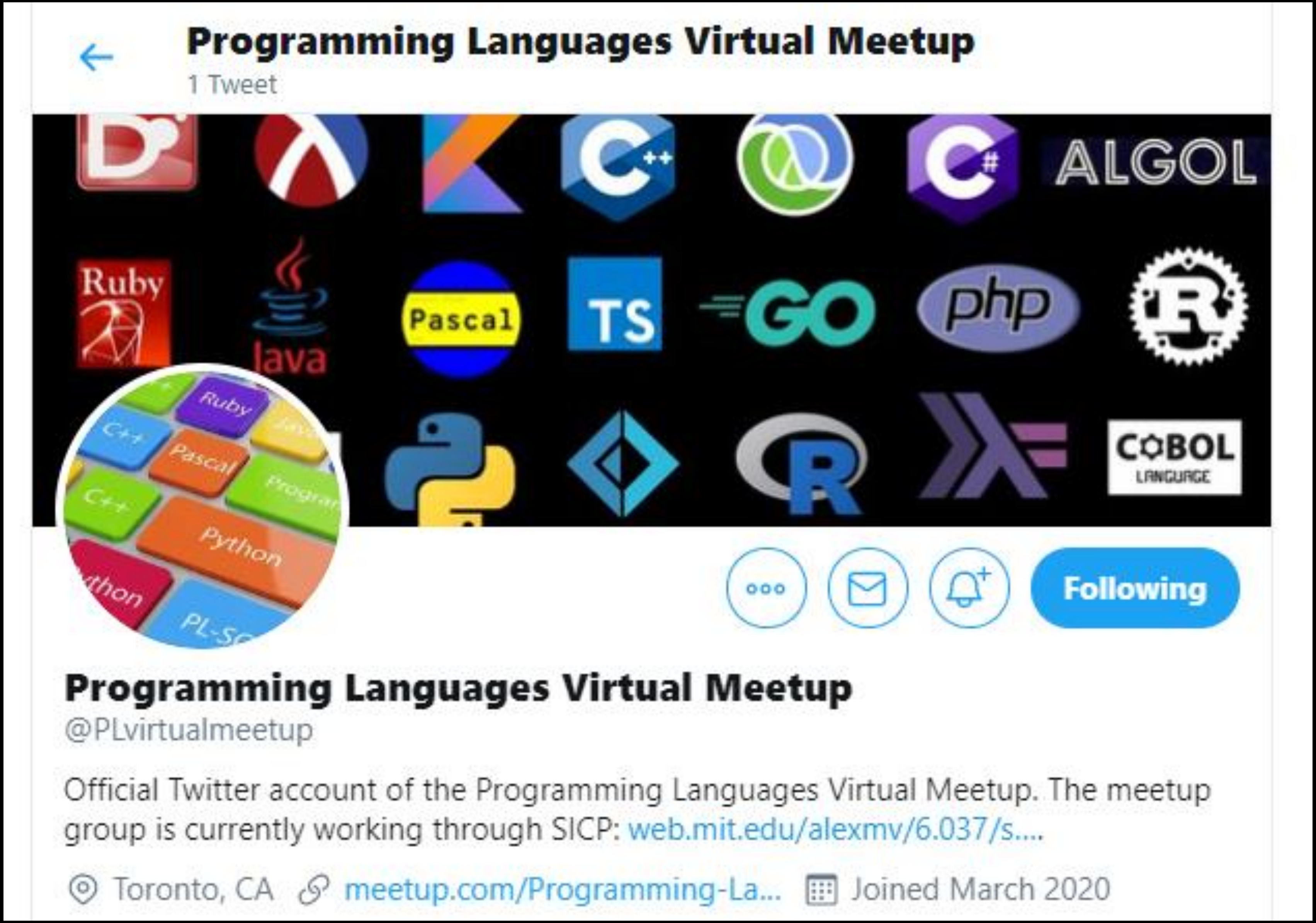
# Chaper 4.3

Before we start …

Friendly Environment Policy

Berlin Code of Conduct

DISCORD



Programming Languages Virtual Meetup

1 Tweet

Programming Languages Virtual Meetup

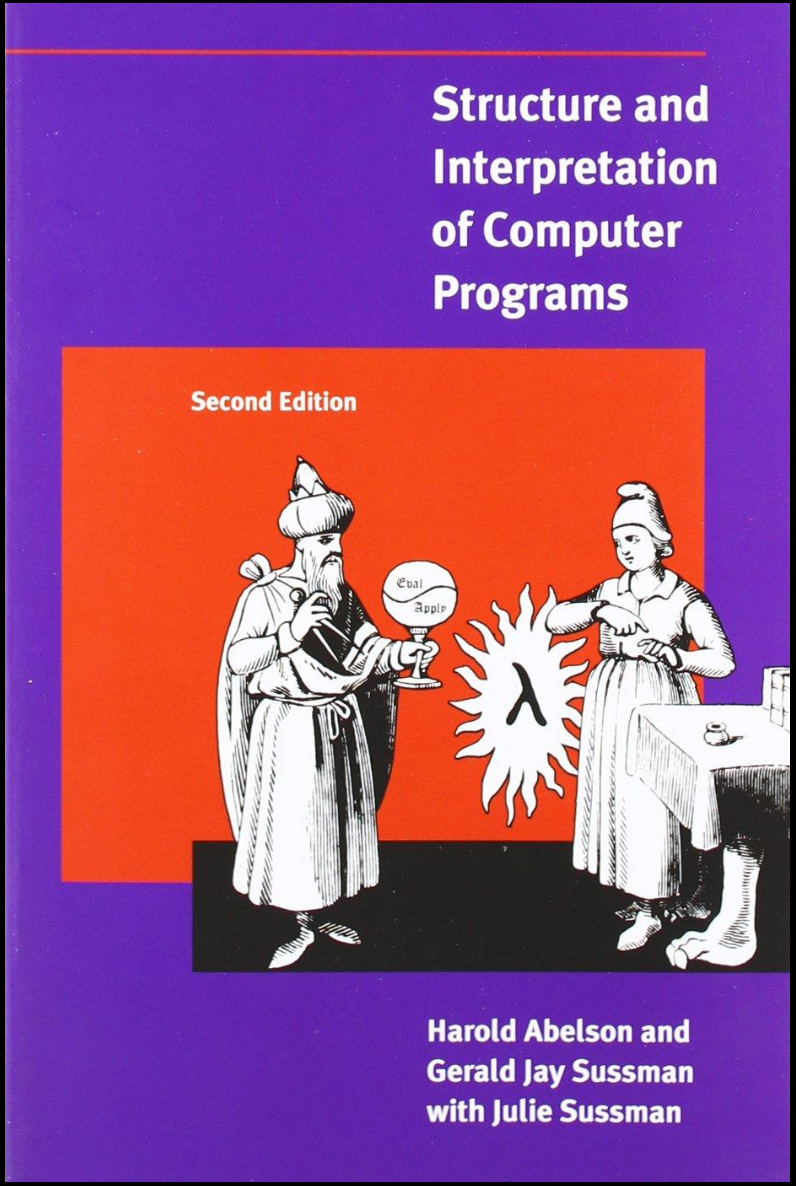
@PLvirtualmeetup

Official Twitter account of the Programming Languages Virtual Meetup. The meetup group is currently working through SICP: web.mit.edu/alexmv/6.037/s....

Toronto, CA meetup.com/Programming-La... Joined March 2020

Java

Meetup


Structure and Interpretation of Computer Programs
Second Edition
Harold Abelson and Gerald Jay Sussman with Julie Sussman

# Structure and Interpretation of Computer Programs

# Chaper 4.3

In this section, we extend the Scheme evaluator to support a programming paradigm called *nondeterministic computing* by building into the evaluator a facility to support automatic search. This is a much more profound change to the language than the introduction of lazy evaluation in Section 4.2.

The nondeterministic program evaluator implemented below is called the `amb` evaluator because it is based on a new special form called `amb`. We can type the above definition of `prime-sum-pair` at the `amb` evaluator driver loop (along with definitions of `prime?`, `an-element-of`, and `require`) and run the procedure as follows:

```
;;; Amb-Eval input:
(prime-sum-pair '(1 3 5 8) '(20 35 110))
;;; Starting a new problem
;;; Amb-Eval value:
(3 20)
```

The value returned was obtained after the evaluator repeatedly chose elements from each of the lists, until a successful choice was made.

### 4.3.1 Amb and Search

To extend Scheme to support nondeterminism, we introduce a new special form called `amb`.[43] The expression

```
(amb ⟨e₁⟩ ⟨e₂⟩ ... ⟨eₙ⟩)
```

returns the value of one of the $n$ expressions $\langle e_i \rangle$ "ambiguously." For example, the expression

```
(list (amb 1 2 3) (amb 'a 'b))
```

can have six possible values:

```
(1 a) (1 b) (2 a) (2 b) (3 a) (3 b)
```

`amb` with a single choice produces an ordinary (single) value.

```
(define (require p) (if (not p) (amb)))
```

**Exercise 4.35:** Write a procedure `an-integer-between` that returns an integer between two given bounds. This can be used to implement a procedure that finds Pythagorean triples, i.e., triples of integers $(i, j, k)$ between the given bounds such that $i \leq j$ and $i^2 + j^2 = k^2$, as follows:

```
(define (a-pythagorean-triple-between low high)
  (let ((i (an-integer-between low high)))
    (let ((j (an-integer-between i high)))
      (let ((k (an-integer-between j high)))
        (require (= (+ (* i i) (* j j)) (* k k)))
        (list i j k)))))
```

```
(define (an-interger-between low high)
  (require (<= low high))
  (amb low (an-interger-between (+ low 1) high)))
```

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?

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?

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| B | | | ✓ | | ✗ |
| C | ✗ | ✓ | | | ✗ |
| F | ✗ | | | ✓ | ✗ |
| M | ✗ | ✗ | | | ✓ |
| S | ✓ | | | | |

**Exercise 4.38:** Modify the multiple-dwelling procedure to omit the requirement that Smith and Fletcher do not live on adjacent floors. How many solutions are there to this modified puzzle?

```
(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)))
    (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))))
```

```
;; '((1 2 4 3 5) (1 2 4 5 3) (1 4 2 5 3) (3 2 4 5 1) (3 4 2 5 1))
;; (actual generated this list from 4.42)
```

**Exercise 4.41:** Write an ordinary Scheme program to solve the multiple dwelling puzzle.

```racket
;; Exercise 4.41 (page 570)

(require threading)

(define (is-multiple-dwelling? lst)
  (let ((baker    (first  lst))
        (cooper   (second lst))
        (fletcher (third  lst))
        (miller   (fourth lst))
        (smith    (fifth  lst)))
    (cond ((= baker 5)       #f)
          ((= cooper 1)      #f)
          ((= fletcher 5)    #f)
          ((= fletcher 1)    #f)
          ((< miller cooper) #f)
          ((= (abs (- smith fletcher)) 1)  #f)
          ((= (abs (- fletcher cooper)) 1) #f)
          (else #t))))

(define (multiple-dwelling)
  (~>> (permutations (range 1 6))
       (filter is-multiple-dwelling?)))
```

**Exercise 4.42:** Solve the following "Liars" puzzle (from Phillips 1934):

Five schoolgirls sat for an examination. Their parents—so they thought—showed an undue degree of interest in the result. They therefore agreed that, in writing home about the examination, each girl should make one true statement and one untrue one. The following are the relevant passages from their letters:

- Betty: "Kitty was second in the examination. I was only third."
- Ethel: "You'll be glad to hear that I was on top. Joan was 2nd."
- Joan: "I was third, and poor old Ethel was bottom."
- Kitty: "I came out second. Mary was only fourth."
- Mary: "I was fourth. Top place was taken by Betty."

What in fact was the order in which the five girls were placed?

```racket
;; Exercise 4.42 (page 570)

(define (is-possible? lst)
  (let ((betty (first  lst))
        (ethel (second lst))
        (joan  (third  lst))
        (kitty (fourth lst))
        (mary  (fifth  lst)))
    (cond ((eq? (= kitty 2) (= betty 3)) #f)
          ((eq? (= ethel 1) (= joan  2)) #f)
          ((eq? (= joan  3) (= ethel 5)) #f)
          ((eq? (= kitty 2) (= mary  4)) #f)
          ((eq? (= mary  4) (= betty 1)) #f)
          (else #t))))

(define (liars-puzzle)
  (~>> (permutations (range 1 6))
       (filter is-possible?)))

(liars-puzzle) ; '((3 5 2 1 4))
```

We will construct the `amb` evaluator for nondeterministic Scheme by modifying the analyzing evaluator of Section 4.1.7.[55] As in the analyzing evaluator, evaluation of an expression is accomplished by calling an execution procedure produced by analysis of that expression. The difference between the interpretation of ordinary Scheme and the interpretation of nondeterministic Scheme will be entirely in the execution procedures.

## Execution procedures and continuations

Recall that the execution procedures for the ordinary evaluator take one argument: the environment of execution. In contrast, the execution procedures in the `amb` evaluator take three arguments: the environment, and two procedures called *continuation procedures*. The evaluation of an expression will finish by calling one of these two continuations: If the evaluation results in a value, the *success continuation* is called with that value; if the evaluation results in the discovery of a dead end, the *failure continuation* is called. Constructing and calling appropriate continuations is the mechanism by which the nondeterministic evaluator implements backtracking.

```scheme
(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)))
        ((let? exp) (analyze (let->combination exp)))
        ((application? exp) (analyze-application exp))
        (else (error "Unknown expression type: ANALYZE" exp))))
```

```scheme
(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)))
        ((let? exp) (analyze (let->combination exp)))
        ((amb? exp) (analyze-amb exp))
        ((application? exp) (analyze-application exp))
        (else (error "Unknown expression type: ANALYZE" exp))))
```

```scheme
;; add

(define (amb? exp) (tagged-list? exp 'amb))
(define (amb-choices exp) (cdr exp))

(define (ambeval exp env succeed fail)
  ((analyze exp) env succeed fail))
```

```
;; replace

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

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

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

(define (analyze-lambda exp)
  (let ((vars (lambda-parameters exp))
        (bproc (analyze-sequence (lambda-body exp))))
    (lambda (env succeed fail)
      (succeed (make-procedure vars bproc env) fail))))
```

```scheme
(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env succeed fail)
      (pproc env
             ;; success continuation for evaluating the predicate
             ;; to obtain pred-value
             (lambda (pred-value fail2)
               (if (true? pred-value)
                   (cproc env succeed fail2)
                   (aproc env succeed fail2)))
             ;; failure continuation for evaluating the predicate
             fail))))
```

```scheme
(define (analyze-sequence exps)
  (define (sequentially a b)
    (lambda (env succeed fail)
      (a env
         ;; success continuation for calling a
         (lambda (a-value fail2)
           (b env succeed fail2))
         ;; failure continuation for calling a
         fail)))
  (define (loop first-proc rest-procs)
    (if (null? rest-procs)
        first-proc
        (loop (sequentially first-proc
                            (car rest-procs))
              (cdr rest-procs))))
  (let ((procs (map analyze exps)))
    (if (null? procs)
        (error "Empty sequence: ANALYZE"))
    (loop (car procs) (cdr procs))))
```

```scheme
(define (analyze-definition exp)
  (let ((var (definition-variable exp))
        (vproc (analyze (definition-value exp))))
    (lambda (env succeed fail)
      (vproc env
             (lambda (val fail2)
               (define-variable! var val env)
               (succeed 'ok fail2))
             fail))))
```

```
(define (analyze-assignment exp)
  (let ((var (assignment-variable exp))
        (vproc (analyze (assignment-value exp))))
    (lambda (env succeed fail)
      (vproc 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))))
```

```
(define (analyze-application exp)
  (let ((fproc (analyze (operator exp)))
        (aprocs (map analyze (operands exp))))
    (lambda (env succeed fail)
      (fproc env
             (lambda (proc fail2)
               (get-args aprocs
                         env
                         (lambda (args fail3)
                           (execute-application
                            proc args succeed fail3))
                         fail2))
             fail))))
```

```racket
(define (get-args aprocs env succeed fail)
  (if (null? aprocs)
      (succeed '() fail)
      ((car aprocs)
       env
       ;; success continuation for this aproc
       (lambda (arg fail2)
         (get-args
          (cdr aprocs)
          env
          ;; success continuation for
          ;; recursive call to get-args
          (lambda (args fail3)
            (succeed (cons arg args) fail3))
          fail2))
       fail)))
```

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

```scheme
(define (analyze-amb exp)
  (let ((cprocs (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 cprocs))))
```

```
(define (driver-loop)
  (define (internal-loop try-again)
    (prompt-for-input input-prompt)
    (let ((input (read)))
      (if (eq? input 'try-again)
          (try-again)
          (begin
            (newline) (display ";;; Starting a new problem ")
            (ambeval
             input
             the-global-environment
             ;; ambeval success
             (lambda (val next-alternative)
               (announce-output output-prompt)
               (user-print val)
               (internal-loop next-alternative))
             ;; ambeval failure
             (lambda ()
               (announce-output
                ";;; There are no more values of")
               (user-print input)
               (driver-loop)))))))
  (internal-loop
   (lambda ()
     (newline) (display ";;; There is no current problem")
     (driver-loop))))
```

```
(define (require p) (if (not p) (amb)))
```

**Exercise 4.50:** Implement a new special form `ramb` that is like `amb` except that it searches alternatives in a random order, rather than from left to right. Show how this can help with Alyssa's problem in Exercise 4.49.

```
;; Exercise 4.50

(define (ramb? exp) (tagged-list? exp 'ramb))
(define (ramb-choices exp) (cdr exp))

(define (analyze-ramb exp)
  (let ((cprocs (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 (shuffle cprocs)))))
```

Java

Meetup