Chapter 6

Section 4.3 in the SICP book

Metalinguistic Abstraction (more variations on Scheme)

Variations on the Interpreter (ctd.)

Slightly less meta
"circular"

MC-eval

Delay arguments into
"thunks"

Yield multiple results
from a computation

Normal Order Evaluation

```
(define (try a b)
  (if (= a 0) 1 b))

(try 0 (/ 1 0)
```

Non-deterministic Computing

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

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

The Amb-evaluator: Abstractions

```
(amb)
```

Always fails

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

All of them

Example: All integers

```
(define (an-integer-starting-from n)
  (amb n (an-integer-starting-from (+ n 1))))
```

Additional abstractions

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

(define (an-element-of items)
  (require (not (null? items)))
  (amb (car items) (an-element-of (cdr items))))
```

Simple Example

```
  ;;; Amb-Eval input:
(prime-sum-pair '(1 3 5 8) '(20 35 110))
  ;;; Starting a new problem
  ;;; Amb-Eval value:
(3 20)
  ;;; Amb-Eval input:
try-again
  ;;; Amb-Eval value:
(3 110)
  ;;; Amb-Eval input:
try-again
  ;;; Amb-Eval value:
(8 35)
  ;;; Amb-Eval input:
try-again
  ;;; There are no more values of
(prime-sum-pair (quote (1 3 5 8)) (quote (20 35 110)))
  ;;; Amb-Eval input:
(prime-sum-pair '(19 27 30) '(11 36 58))
  ;;; Starting a new problem
  ;;; Amb-Eval value:
(30 11)
```

```
(define (prime-sum-pair list1 list2)
  (let ((a (an-element-of list1))
        (b (an-element-of list2)))
    (require (prime? (+ a b)))
    (list a b)))
```

Extended Example: Difficult People

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?

Extended Example: Difficult People

```
(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))))
```

∀

Constraint Satisfaction Problem

Declarative Style

Running the Example

```
  ;;; Amb-Eval input:
(begin
  (define (require p)
    ...)
  (define (distinct? items)
    ...)
  (define (multiple-dwelling)
    ...)
  ;;; Starting a new problem
  ;;; Amb-Eval value:
  ok
  ;;; Amb-Eval input:
(multiple-dwelling)
  ;;; Starting a new problem
  ;;; Amb-Eval value:
  {{baker 3} {cooper 2} {fletcher 4} {miller 5} {smith 1}}
try-again
  ;;; There are no more values of
  {multiple-dwelling}
```

Enter all bits at once

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

Solve the problem

Extended Example: NLP

```
(define nouns '(noun student professor cat class))
(define verbs '(verb studies lectures eats sleeps))
(define articles '(article the a))
(define (parse-sentence)
  (list 'sentence
        (parse-noun-phrase)
        (parse-word verbs)))
(define (parse-noun-phrase)
  (list 'noun-phrase
        (parse-word articles)
        (parse-word nouns)))
(define (parse-word word-list)
  (require (not (null? *unparsed*)))
  (require (memq (car *unparsed*) (cdr word-list)))
  (let ((found-word (car *unparsed*)))
    (set! *unparsed* (cdr *unparsed*))
    (list (car word-list) found-word)))
(define *unparsed* '())
(define (parse input)
  (set! *unparsed* input)
  (let ((sent (parse-sentence)))
    (require (null? *unparsed*))
    sent))
```

word classification

grammar

e.g., "the cat"

modifies unparsed input list!

Extended Example: NLP

```
;;; Amb-Eval input:
(begin
  (define (require p)
    ...)
  (define nouns '(noun student professor cat class))
  (define verbs '(verb studies lectures eats sleeps))
  (define articles '(article the a))
  (define (parse-sentence)
    ...)
  (define (parse-noun-phrase)
    ...)
  (define (parse-word word-list)
    ...)
  (define *unparsed* '())
  (define (parse input)
    ...))
  ;;; Starting a new problem
  ;;; Amb-Eval value:
  ok
  ;;; Amb-Eval input:
(parse '(the cat eats))
  ;;; Starting a new problem
  ;;; Amb-Eval value:
  {sentence {noun-phrase {article the} {noun cat}} {verb eats}}
```

Enter all bits at once

Extended Example(ctd.)

Examples of ambiguity

```
...
(define prepositions '(prep for to in by with))
(define (parse-sentence)
  (list 'sentence
        (parse-noun-phrase)
        (parse-verb-phrase)))
(define (parse-noun-phrase)
  (define (maybe-extend noun-phrase)
    (amb noun-phrase
         (maybe-extend (list 'noun-phrase
                             noun-phrase
                             (parse-prepositional-phrase)))))
  (maybe-extend (parse-simple-noun-phrase)))
(define (parse-prepositional-phrase)
  (list 'prep-phrase
        (parse-word prepositions)
        (parse-noun-phrase)))
(define (parse-verb-phrase
  (define (maybe-extend verb-phrase)
    (amb verb-phrase
         (maybe-extend (list 'verb-phrase
                             verb-phrase
                             (parse-prepositional-phrase)))))
  (maybe-extend (parse-word verbs)))
(define (parse-simple-noun-phrase)
  (list 'simple-noun-phrase
        (parse-word articles)
        (parse-word nouns)))
```

e.g., "a cat in the class"

simple noun phrase or
noun phrase + prepositional phrase

e.g., "for the cat"

verb or
verb phrase + prepositional phrase

e.g., "eats in the class"

Extended Example(ctd.)

```
  ;;; Amb-Eval input:
(parse '(the student with the cat sleeps in the class))
  ;;; Starting a new problem
  ;;; Amb-Eval value:
  {sentence {noun-phrase {simple-noun-phrase {article the} {noun student}}
  {prep-phrase {prep with} {simple-noun-phrase {article the} {noun cat}}}}
  {verb-phrase {verb sleeps} {prep-phrase {prep in} {simple-noun-phrase
  {article the} {noun class}}}}}
  ;;; Amb-Eval input:
(parse '(the professor lectures to the student with the cat))
  ;;; Starting a new problem
  ;;; Amb-Eval value:
  {sentence {simple-noun-phrase {article the} {noun professor}} {verb-phrase
  {verb-phrase {verb lectures} {prep-phrase {prep to} {simple-noun-phrase
  {article the} {noun student}}}} {prep-phrase {prep with} {simple-noun
  phrase {article the} {noun cat}}}}}
  ;;; Amb-Eval input:
try-again
  ;;; Amb-Eval value:
  {sentence {simple-noun-phrase {article the} {noun professor}} {verb-phrase
  {verb lectures} {prep-phrase {prep to} {noun-phrase {simple-noun-phrase
  {article the} {noun student}} {prep-phrase {prep with} {simple-noun-phrase
  {article the} {noun cat}}}}}}}
```

Unambiguous

Ambiguous

Evaluator Architecture

Success continuation

Failure continuation

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

Simplistic Usage

```
(ambeval <exp>
         the-global-environment
         (lambda (value fail) value)
         (lambda () 'failed))
```

New Syntactic Category

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

((amb? exp) (analyze-amb exp))
```

Add it to the dispatcher

Simple Expressions

```
(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))))
```

succeed with value of expression, passing along failure continuation

always succeeds (else program bug)

Conditionals

```
(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
             (lambda (pred-value fail2)
               (if (true? pred-value)
                   (cproc env succeed fail2)
                   (aproc env succeed fail2)))
             fail))))
```

success

failure

what should happen after the if

Sequences

```
(define (analyze-sequence exps)
  (define (sequentially a b)
    (lambda (env succeed fail)
      (a env
        (lambda (a-value fail2)
          (b env succeed fail2))
        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))))
```

lambda-glue

call *a* with success continuation that calls b

failure

glue a list of evaluator parts together

Definition

```
(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))))
```

if evaluating initial value succeeded then definition succeeds

else definition fails

Assignment

first example of backtracking

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

success continuation that saves old value before assigning new value

failure continuation that restores old value before continuing the failure

if evaluation of assignment value fails, then assignment fails

Procedure Application

```
(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))))
```

Processing Actual Arguments

```
(define (get-args aprocs env succeed fail)
  (if (null? aprocs)
      (succeed '() fail)
      ((car aprocs) env
                    (lambda (arg fail2)
                      (get-args (cdr aprocs)
                                env
                                (lambda (args fail3)
                                  (succeed (cons arg args)
                                           fail3))
                                fail2))
                    fail)))
```

success continuation that recursively calls `get-args`

success continuation that accumulates arguments

Actual Procedure Application

```
(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))))
```

The amb special form

```
(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))))
```

The REPL

```
(define input-prompt ";;; Amb-Eval input:")
(define output-prompt ";;; Amb-Eval value:")
(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
                     (lambda (val next-alternative)
                       (announce-output output-prompt)
                       (user-print val)
                       (internal-loop next-alternative))
                     (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))))
```

success? call the internal loop again

failure? call the driver loop again

Chapter 6