# SICP section 4.3.1

December 28, 2007 at 21:01 Tags SICP

The material in section 4.3 is far from simple, and requires careful reading of the book in order to understand it fully[1]. Also, if you *really* want to grasp what's going under the hood of nondeterministic evaluation, there is no alternative to reimplementing the `amb` evaluator with your own hands following the instructions in section 4.3.3

In order to test my solutions to exercises in sections 4.3.1 and 4.3.2 I've implemented the `amb` evaluator in Common Lisp. Below are some selected parts with explanations.

## Understanding the `amb` evaluator

From "Structure of the evaluator"

> A success continuation is a procedure of two arguments: the value just obtained and another failure continuation to be used if that value leads to a subsequent failure. A failure continuation is a procedure of no arguments.

This is a crucial point, and it's valuable to understand why this is so. From an earlier paragraph, in "Execution procedures and continuations"

> It is the job of the success continuation to receive a value and proceed with the computation. Along with that value, the success continuation is passed another failure continuation, which is to be called subsequently if the use of that value leads to a dead end. It is the job of the failure continuation to try another branch of the nondeterministic process.

The simple expressions presented also demand some explanations. Compare the original `analyze-self-evaluating`:

```
(defun analyze-self-evaluating (exp)
  (lambda (env) exp))
```

With the new version:

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

What's going on here? Why is `succeed` called? Because:

> [...] 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.

So yes, even the simplest evaluations must obey these rules. A self evaluating expression always succeeds, hence `succeed` is called, passing its value on. Also, recall that `succeed` is a procedure of two arguments: the value and a `fail` continuation.

Here's `analyze-if`:

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

It is not immediately obvious (at least for me!) why it works this way. Consider, however, what is the result of each call to `analyze.`. It is a procedure that accepts `(env succeed fail)` as arguments and is expected to call `succeed` if it results in a value and `fail` otherwise. But without first calling `pproc` we don't know here if there is a value, so we call `pproc` with a `succeed` and `fail` continuations of its own. The `succeed` continuation we provide it performs the *actual* job of the `if` form[2].

The new `analyze-assignment` is:

```
(defun analyze-assignment (exp)
  (let ((var (assignment-variable exp))
        (vproc (analyze. (assignment-value exp))))
    (lambda (env succeed fail)
      (funcall 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)
                (funcall fail2)))))
        fail))))
```

When I first read the description of the `amb` evaluator I thought that it's probably a lot of trouble "rolling back" assignments from failed paths. However, in `analyze-assignment`, continuation-passing style solves the problem with elegance.

If the call to `vproc` succeeds, we continue by actually assigning the value and passing on a `fail` continuation that resets the old value in case this path ever becomes failed. That's it – so simple.

At last, let's inspect `analyze-amb`:

```
(defun analyze-amb (exp)
  (let ((cprocs (mapcar #'analyze. (amb-choices exp))))
    (lambda (env succeed fail)
      (labels (
          (try-next (choices)
            (if (null choices)
              (funcall fail)
              (funcall (car choices)
                env
                succeed
                (lambda ()
                  (try-next (cdr choices)))))))
        (try-next cprocs)))))
```

This one simply tries all the choices it was given in order, until one of them succeeds. It iterates through the choices by passing on a `fail` continuation that tries the next choice.

You can download the full `amb` evaluator here: evaluator_amb.lisp. Now we're ready to solve the exercises of section 4.3.1

### Exercise 4.35

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

Now we can use `a-pythagorean-triple-between` like this[3]:

```
;;; Amb-Eval input:
(a-pythagorean-triple-between 1 6)

;;; Starting a new problem
;;; Amb-Eval value:
(3 4 5)
```

### Exercise 4.36

To understand why that wouldn't work, we'll modify `a-pythagorean-triple-between` as suggested and add a printout[4] to show which values it tries:

```
(interpret
  '(define (aptb low)
    (let ((i (an-integer-starting-from low)))
      (let ((j (an-integer-starting-from low)))
        (let ((k (an-integer-starting-from low)))
          (format true "pyt: ~a ~a ~a~%" i j k)
          (require (= (+ (* i i) (* j j)) (* k k)))
          (list i j k))))))
```

```
(aptb 1)

;;; Starting a new problem pyt: 1 1 1
pyt: 1 1 1
pyt: 1 1 2
pyt: 1 1 3
pyt: 1 1 4
pyt: 1 1 5
pyt: 1 1 6
pyt: 1 1 7
pyt: 1 1 8
pyt: 1 1 9

...
...
```

Whoops! Now it should be quite clear why providing an infinite range wouldn't work. As the authors noted in the book, the `amb` evaluator employs *depth-first search* to backtrack, so it tries to exhaust each path before trying another one. In this case, it tries to exhaust `k` – an impossible task[5].

One approach to solve it is to generate the triplets in a more balanced fashion, using *breadth-first search* on the tree of possibilities:

```
(interpret
  '(define (nextc trp)
    (let ((a (car trp))
          (b (car (cdr trp)))
          (c (car (cdr (cdr trp)))))
      (cond ((= a b c)
             (list 1 1 (+ c 1)))
            ((= a b)
             (list 1 (+ b 1) c))
            (else
             (list (+ a 1) b c))))))
```

This procedure will iterate over triplets, given the current one. Here's a sample order it generates:

```
(1 1 1)
(1 1 2)
(1 2 2)
(2 2 2)
(1 1 3)
(1 2 3)
(2 2 3)
(1 3 3)
(2 3 3)
(3 3 3)
(1 1 4)
(1 2 4)
(2 2 4)
(1 3 4)
(2 3 4)
```

Now, we're ready for the implementation:

```
(interpret
  '(define (a-triplet)
    (a-triplet-rec '(1 1 1))))

(interpret
  '(define (a-triplet-rec trp)
    (amb
      trp
      (a-triplet-rec (nextc trp)))))

(interpret
  '(define (aptb)
    (let ((trp (a-triplet)))
      (let ((i (car trp))
            (j (car (cdr trp)))
            (k (car (cdr (cdr trp)))))
        (require (= (+ (* i i) (* j j)) (* k k)))
        (list i j k)))))
```

Here's a sample interaction:

```
;;; Amb-Eval input:
(aptb)

;;; Starting a new problem
;;; Amb-Eval value:
(3 4 5)

;;; Amb-Eval input:
try-again

;;; Amb-Eval value:
(6 8 10)

;;; Amb-Eval input:
try-again

;;; Amb-Eval value:
(5 12 13)

;;; Amb-Eval input:
try-again

;;; Amb-Eval value:
(9 12 15)

;;; Amb-Eval input:
try-again

;;; Amb-Eval value:
(8 15 17)

;;; Amb-Eval input:
try-again

;;; Amb-Eval value:
(12 16 20)

;;; Amb-Eval input:
try-again

;;; Amb-Eval value:
(15 20 25)

;;; Amb-Eval input:
try-again

;;; Amb-Eval value:
(7 24 25)

;;; Amb-Eval input:
try-again

;;; Amb-Eval value:
(10 24 26)
```

Although it works, I have a nagging feeling that this solution is not optimal and there may be more elegant solutions to this problem out there. If you find one, let me know!

### Exercise 4.37

Yes, Ben's implementation is much more efficient than the one in exercise 4.35, because it throws away many irrelevant results, using two techniques:

1. It rejects triplets in which the sum of squares of `i` and `j` is higher than the square of `high`. This is called *tree pruning*.

2. Instead of running on all `k`, it tries to check whether the square root of the squares of `i` and `j` is an integer. If it is, we have a solution.

[1] I think it took me at least 4 times of reading through the instructions in section 4.3.3 before I really understood what's going on.

[2] Perhaps a clearer way to look at it would be: we have to decide if we have a result and so which of the continuations to call. But without first executing `pproc` we can't know. So we pass the decision to `pproc` itself by means of a continuation, asking it to perform the work of `if` in case of a success.

[3] The `amb` evaluator is very call-stack-heavy, so `clisp` can't handle it and reports a stack overflow. I used the experimental SBCL 1.0.9 build for Windows to generate this result.

[4] To make `format` work in the evaluated Scheme, I added it as a primitive procedure to the evaluator.

[5] This problem is somewhat similar with the one we had with infinite streams in section 3.5, where we solved it by writing the `interleave` procedure.

---





Back to top