CS61A – Homework 7.2	**Kurt Meinz**
University of California, Berkeley	Summer 2003

Topic: Nondeterministic evaluator

Lectures: Wednesday August 6, Thursday August 7

Reading: Abelson & Sussman, Section 4.3 (Pages 412–437)

In this homework you will gain experience modifying the nondeterministic evaluator. Most of this assignment is very much "below the line." Two versions of the amb evaluator are available:

- `~cs61a/lib/ambeval.scm` — This is the nondeterministic interpreter from the book, based on the analyzing evaluator.
- `~cs61a/lib/vambeval.scm` — This is a version of the nondeterministic interpreter based on the metacircular evaluator. This is also the version described in lecture. Most students find this one easier to understand. (The "v" is for vanilla.)

Copy whichever version you wish to use to do the homework into a file `hw7-2.scm` and make all modifications in this file. Clearly indicate what you changed. When you are done, you will have a nondeterministic interpreter that supports `quit`, `permanent-set!`, `or` and `if-fail`. You should include test cases either in this file (commented out), or a separate file called `tests`. Please put your answer to Question 1 into a file `question1.scm`. Submit all files electronically. The assignment is due at **8 PM on Sunday, August 10**.

All problems that ask you to add something to the nondeterministic evaluator have very short solutions. You should not be writing a lot of code at all! Wrapping your brain around continuations is the tricky part.

Question 1. Read and complete Exercise 4.42 in SICP. This is the only "above the line" problem on the homework.

Question 2. We'd like to be able to quit the amb evaluator at *any point* in the execution of a program. Add a `quit` feature to the nondeterministic evaluator that immediately returns control to STk. **It must be a clean exit—don't cause an error!** The return value of `quit` is up to you; ours returns the string "Have a nice day." The following are some examples of how `quit` should behave; `quit` must exit the amb evaluator not just from the toplevel, but from any depth in the evaluation (the bars separate different sessions with the evaluator):

```
;;; Amb-Eval input:
(quit)                                  ;; exit from toplevel
;;; Starting a new problem
"Have a nice day"
STk>
```

```
;;; Amb-Eval input:
(list 1 2 (quit) 3)                      ;; exit from subexpression evaluation
;;; Starting a new problem
"Have a nice day"
STk>
```

The question continues on the next page.

```
;;; Amb-Eval input:
(define (factorial n)
  (if (= n 0)
      (begin (newline) (quit))          ;; exit from arbitrarily deep recursion
      (begin (display n)
             (display " ")
             (* n (factorial (- n 1))))))

;;; Amb-Eval input:
(factorial 14)

;;; Starting a new problem 14 13 12 11 10 9 8 7 6 5 4 3 2 1
"Have a nice day"

STk>
```

Hint: Remember that control flow is done via continuations in the nondeterministic evaluator. To continue the computation you must invoke the success continuation; to backtrack you invoke the fail continuation. What if you call neither?

Question 3. One of the really neat things about the nondeterministic evaluator is that variable assignments are "undone" when backtracking occurs. Backtracking occurs automatically when `(amb)` is encountered; it also can be forced when the user types `try-again`. Therefore, assignments can be undone by saying `try-again`. Watch:

```
;;; Amb-Eval input:
(define neo 2)                                  ;; return value omitted

;;; Amb-Eval input:
(define trinity 4)

;;; Amb-Eval input:
(define cypher 6)

;;; Amb-Eval input:
(begin (set! neo (* neo neo))
       (set! trinity (* trinity trinity))
       (set! cypher 'bloody-rat)
       (list neo trinity cypher))

;;; Starting a new problem
;;; Amb-Eval value:
(4 16 bloody-rat)                               ;; clearly the assignment takes effect

;;; Amb-Eval input:
try-again                                       ;; but it is not permanent

;;; There are no more values of ...

;;; Amb-Eval input:
(list neo trinity cypher)

;;; Starting a new problem
;;; Amb-Eval value:
(2 4 6)                                         ;; back to their old values
```

Sometimes, however, we want assignments to be permanent. Add a special form `permanent-set!` that is just like `set!` but does not get rolled back when backtracking occurs.

The question continues on the next page.

You can use `permanent-set!` to count the number of times the nondeterministic evaluator backtracks:

```
;;; Amb-Eval input:
(define count 0)                  ;; return value omitted
;;; Amb-Eval input:
(let ((x (an-element-of '(a b c)))
      (y (an-element-of '(a b a))))
  (permanent-set! count (+ 1 count))
  (require (not (eq? x y)))
  (list x y count))
;;; Starting a new problem
;;; Amb-Eval value:
(a b 2)
;;; Amb-Eval input
try-again
;;; Amb-Eval value:
(b a 4)
```

Hint: This question does not ask you to add new functionality, but to subtract from what's already there. Find the line(s) in `eval-assignment` that implement this undo effect and get rid of them. The failure continuation is a good place to look.

Question 4. Add the `or` special form to the nondeterministic evaluator by writing an evaluation procedure `eval-or` that handles it. **Do not add `or` as a derived expression.** As in regular Scheme, `or` should take any number of arguments and return the value of the first one that is true, or #f if none are.

You should model `eval-or` very heavily on `get-args` (code from `vambeval.scm`):

```
(define (get-args exps env succeed fail)
  (if (null? exps)
      (succeed '() fail)
      (ambeval (car exps)
               env
               (lambda (arg fail2)                    ;; first success continuation
                 (get-args (cdr exps)
                           env
                           (lambda (args fail3)       ;; second success continuation
                             (succeed (cons arg args) fail3))
                           fail2))
               fail)))
```

Like `list-of-values` in the MCE, the job of `get-args` is to evaluate a sequence of Scheme expressions, `exps`, and return a list of their values:

```
STk> (get-args '((+ 2 3) (first 'neo) (bf 'trinity))
               the-global-environment
               (lambda (result fail-cont) result)
               (lambda () 'failed))
(5 n rinity)
```

There are two success continuations. The first one is invoked if evaluating the very first expression in the sequence *does not* cause a failure; in this case, `arg` refers to the value of that first expression. The second one is invoked if the remaining expressions in the sequence were evaluated without failure; in this case, `args` is a list of their values. Notice how the list of values is built up in this second success continuation by consing `arg` into `args`.

The question continues on the next page.

A good place to start is by adding this clause to `ambeval`

```
((or? exp) (eval-or (cdr exp) env succeed fail))   ;; cdr to strip off "or" tag
```

and defining `eval-or` to do exactly what `get-args` does. Of course this means that `or` will evaluate all of its arguments and return a list of their results, which is not quite what we want, but it's a start! Try it out. Then tinker with this `eval-or` to make it behave as specified above. Here are some sample calls:

```
STk> (eval-or '((= 2 3) (list 1 2) this-should-not-be-evaluated)
              the-global-environment
              (lambda (result fail-cont) result)
              (lambda () 'failed))
(1 2)
STk> (eval-or '((= 2 3) (amb) this-should-not-be-evaluated)
              the-global-environment
              (lambda (result fail-cont) result)
              (lambda () 'failed))
failed
STk> (eval-or '()
              the-global-environment
              (lambda (result fail-cont) result)
              (lambda () 'failed))
#f
```

And here is how `or` can be used in the interpreter:

```
;;; Amb-Eval input:
(or (amb 1 2 #f) 'hello)
;;; Starting a new problem
;;; Amb-Eval value:
1
;;; Amb-Eval input:
try-again
;;; Amb-Eval value:
2
;;; Amb-Eval input:
try-again
;;; Amb-Eval value:
hello
;;; Amb-Eval input:
try-again
;;; There are no more values of
(or (amb 1 2 #f) 'hello)
```

The assignment continues on the next page.

Question 5. Read and complete Exercise 4.52 in the book. This question is more difficult than the others since you'll need to come up with the `if-fail` special form from scratch. Assuming your function for handling `if-fail` is called `eval-if-fail` and takes the entire expression as argument, here is how you might test it in isolation:

```
STk> (eval-if-fail '(if-fail (amb) 'hello)
                   the-global-environment
                   (lambda (result new-fail) result)
                   (lambda () 'failed))
hello
STk> (eval-if-fail '(if-fail (amb) (amb))
                   the-global-environment
                   (lambda (result new-fail) result)
                   (lambda () 'failed))
failed
```

Hint: To make something happen on failure, you must put it into the fail continuation.