MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall 2007

Recitation 20 — 11/16/2007 Solutions
Amb Evaluator

AMB Eval

The AMB evaluator adds one new special form called `amb` which takes a set of possible values and returns one of them. For example,

`(amb 1 2 3)` could take on the value 1, 2, or 3.

`amb` called with no values represents a failure: there are no possible values, so the evaluator needs to backtrack and try another value.

In addition to the new special form, a new procedure named `require` is often useful:

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

One way to read that definition is to check the condition `p`, and if it is true, then proceed, otherwise backtrack to try some other set of values before proceeding.

Example:

```
(let ((a (amb 1 2 3 4)))
    (require (even? a))
    (require (> a 3))
    a)
```

4

How about this version?

```
(let ((a (an-element-of (list 1 2 3 4))))
    (require (even? a))
    (require (odd? a))
    a)
```

Implementing AMB as an evaluator

In the normal version of the analyze evaluator, `analyze` takes in an expression, and returns a procedure that takes in an environment as argument and carries out that operation.

In amb-eval the procedures returned are different and all look like this:

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

Rather than return a value, they all pass the value to success: nothing ever returns!

The difference is analyze-amb:

```
(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))))
```

This one does call the failure continuation, to backtrack to the last branch point.

The rest of the amb definition are in SICP [1]

[1] If you want to try out amb in DrScheme, you can either run all the code for the amb evaluator (in SICP), or you can add the following definitions which will let you use amb in normal DrScheme. Don't ask me how this works (I didn't write it!), though if you look at it you'll see that it has success and failure continuations just like the amb-eval version. Note that there are two different quote characters here: normal quote (') and backquote (`).

```
(define amb-fail '*)
(define initialize-amb-fail
  (lambda ()
    (set! amb-fail
      (lambda ()
        (error "amb tree exhausted")))))
(initialize-amb-fail)
(define-macro amb
  (lambda alts
    `(let ((+prev-amb-fail amb-fail))
       (call/cc
        (lambda (success)
          ,@(map (lambda (alt)
                   `(call/cc
                     (lambda (fail)
                       (set! amb-fail
                         (lambda ()
                           (set! amb-fail +prev-amb-fail)
                           (fail 'fail)))
                       (success ,alt))))
                 alts)
          (+prev-amb-fail))))))
```

8 Queens

Suppose you have a $n \times n$ chessboard. How can you place n queens on the board such that all the positions are safe (no other queen can capture one of the others in a single move).

For $n = 2$, there are no solutions, though for $n = 8$ there are several, one of which is:

*							
						*	
			*				
					*		
							*
	*						
				*			
		*					

Write a procedure `queens` which uses `amb` to produce solutions to this puzzle.

To begin with, notice that any solution will have exactly one queen in each row and column of the solution. As such let's represent a solution as a list of n elements: each element represents the row that that column's queen is in. For example `(queens 4)` $\Longrightarrow$ `(2 4 1 3)` is one possible solution.

The following helper procedures may be useful:

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

(define (distinct? items)
  (cond ((null? items) #t)
        ((null? (cdr items)) #t)
        ((member (car items) (cdr items)) #f)
        (else (distinct? (cdr items)))))

(define (enumerate-interval l u)
    (if (> l u) '()
        (cons l (enumerate-interval (+ l 1) u))))

(define (except n lst)
  (filter (lambda (x) (not (eq? x n))) lst))
```

This is the naive version of the solution: to do the real one we need to change generate board to filter the list of possible values based on the previous columns first.

```
(define (queens n)
  (define (safe? board)
    (define (check-col i)
      (if (= i n) #t
          (let ((row (list-ref board i)))
            (if (safe-vs row (- i 1) 1)
                (check-col (+ i 1))
                #f))))
    (define (safe-vs row col offset)
      (cond ((< col 0) #t)
            ((= (+ row offset) (list-ref board col)) #f)
            ((= (- row offset) (list-ref board col)) #f)
            (else (safe-vs row (- col 1) (+ offset 1)))))
    (check-col 1))
  (define (generate-board)
    (map
     (lambda (x)
       (an-element-of (enumerate-interval 1 n)))
     (enumerate-interval 1 n)))
  (let ((board (generate-board)))
    (require
     (distinct? board))
    (require (safe? board))
    board))
```