MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1999

Recitation – Wednesday, April 28

1. Explicit Control Evaluator: The Important Contracts

Function	Assumption	Promise
eval-dispatch	Expression in EXP, Environment in ENV, continuation in CONT	End up at CONT with result in VAL
apply-dispatch	Procedure in PROC, list of arguments in ARGL, continuation at the top of the stack.	End up at CONT with result in VAL and the stack popped
eval-sequence	Sequence of expressions in UNEV, Environment in ENV, continuation at the top of the stack.	Evaluate the expressions in sequence, and end up at CONT with result of the final expression in VAL and the stack popped.

2. Adding AND to the Explicit Control Evaluator

We can also add `and` to our explicit-control evaluator. We first add a clause to the evaluator dispatch

```
eval-dispatch
  ...
  (test (op and?) (reg exp))
  (branch (label ev-and))
  ...
```

`EV-AND` presumes that the `EXP` register holds the expression to be evaluated, the `ENV` register holds the current environment pointer, and the `CONT` register holds the place to jump to.

The key is to write register machine code to implement `AND` starting at the label `EV-AND`. Fill in the missing spots. Assume we've got the primitives `first-conjunct` and `second-conjunct`.

```
ev-and
  (assign unev )
  (assign exp )
  (save continue)
  (save env)
  (save unev)
  (assign continue eval-after-first)
  (goto )
eval-after-first
  (restore )
  (restore )
  (test (op true?) (reg val))
  (branch (label eval-second-arg))
  (assign val #f)
  (restore )
  (goto (reg continue))
eval-second-arg
  (assign exp )
  (assign continue after-second)
  (goto )
after-second
  
  (goto (reg continue))
```

3. Tail Recursion

Does this `ev-and` routine handle tail recursion? For example, consider the scheme code below. What result (if any) do we get when we evaluate `(list?  x)` in our regular scheme? How about a scheme built on top of the above explicit-control evaluator?

```
(define (list? x)
  (or (null? x)
      (and (pair? x) (list? (cdr x)))))
(define z (list 1))
(set-cdr! z z)
(list? z)
```

To see how this is working, let's evaluate the expression `(and #t (and #f #t))`.

How could we change the code above so that it handles tail-recursion? Hint: remove lines 19 through 23 and add two lines in their place.

```
19.
20.
```

Can we get rid of the new line 19 by moving another line somewhere?

Could we remove line 14 without changing the value returned by the code? Why or why not?

How can we get rid of line 18 by changing another line?