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

Recitation 23 Solutions
Explicit Control Eval

```
eval-dispatch
  (test (op self-evaluating?) (reg exp))
  (branch (label ev-self-eval))
  (test (op variable?) (reg exp))
  (branch (label ev-variable))
  (test (op quoted?) (reg exp))
  (branch (label ev-quoted))
  (test (op assignment?) (reg exp))
  (branch (label ev-assignment))
  (test (op definition?) (reg exp))
  (branch (label ev-definition))
  (test (op if?) (reg exp))
  (branch (label ev-if))
  (test (op lambda?) (reg exp))
  (branch (label ev-lambda))
  (test (op begin?) (reg exp))
  (branch (label ev-begin))
  (test (op application?) (reg exp))
  (branch (label ev-application))
  (goto (label unknown-expression-type))

ev-self-eval
  (assign val (reg exp))
  (goto (reg continue))
ev-variable
  (assign val (op lookup-variable-value) (reg exp) (reg env))
  (goto (reg continue))
ev-quoted
  (assign val (op text-of-quotation) (reg exp))
  (goto (reg continue))
ev-lambda
  (assign unev (op lambda-parameters) (reg exp))
  (assign exp (op lambda-body) (reg exp))
  (assign val (op make-procedure)
              (reg unev) (reg exp) (reg env))
  (goto (reg continue))

ev-application
  (save continue)
  (save env)
  (assign unev (op operands) (reg exp))
  (save unev)
  (assign exp (op operator) (reg exp))
  (assign continue (label ev-appl-did-operator))
  (goto (label eval-dispatch))
ev-appl-did-operator
  (restore unev)
  (restore env)
  (assign argl (op empty-arglist))
  (assign proc (reg val))
  (test (op no-operands?) (reg unev))
  (branch (label apply-dispatch))
  (save proc)
ev-appl-operand-loop
  (save argl)
  (assign exp (op first-operand) (reg unev))
  (test (op last-operand?) (reg unev))
  (branch (label ev-appl-last-arg))
  (save env)
  (save unev)
  (assign continue (label ev-appl-accumulate-arg))
  (goto (label eval-dispatch))
```

```
ev-appl-accumulate-arg
  (restore unev)
  (restore env)
  (restore argl)
  (assign argl (op adjoin-arg) (reg val) (reg argl))
  (assign unev (op rest-operands) (reg unev))
  (goto (label ev-appl-operand-loop))

ev-appl-last-arg
  (assign continue (label ev-appl-accum-last-arg))
  (goto (label eval-dispatch))
ev-appl-accum-last-arg
  (restore argl)
  (assign argl (op adjoin-arg) (reg val) (reg argl))
  (restore proc)
  (goto (label apply-dispatch))
apply-dispatch
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-apply))
  (test (op compound-procedure?) (reg proc))
  (branch (label compound-apply))
  (goto (label unknown-procedure-type))

primitive-apply
  (assign val (op apply-primitive-procedure)
              (reg proc)
              (reg argl))
  (restore continue)
  (goto (reg continue))

compound-apply
  (assign unev (op procedure-parameters) (reg proc))
  (assign env (op procedure-environment) (reg proc))
  (assign env (op extend-environment)
              (reg unev) (reg argl) (reg env))
  (assign unev (op procedure-body) (reg proc))
  (goto (label ev-sequence))

ev-begin
  (assign unev (op begin-actions) (reg exp))
  (save continue)
  (goto (label ev-sequence))

ev-sequence
  (assign exp (op first-exp) (reg unev))
  (test (op last-exp?) (reg unev))
  (branch (label ev-sequence-last-exp))
  (save unev)
  (save env)
  (assign continue (label ev-sequence-continue))
  (goto (label eval-dispatch))
ev-sequence-continue
  (restore env)
  (restore unev)
  (assign unev (op rest-exps) (reg unev))
  (goto (label ev-sequence))
ev-sequence-last-exp
  (restore continue)
  (goto (label eval-dispatch))

ev-if
  (save exp)
  (save env)
  (save continue)
  (assign continue (label ev-if-decide))
  (assign exp (op if-predicate) (reg exp))
  (goto (label eval-dispatch))
ev-if-decide
  (restore continue)
  (restore env)
```

```
  (restore exp)
  (test (op true?) (reg val))
  (branch (label ev-if-consequent))
ev-if-alternative
  (assign exp (op if-alternative) (reg exp))
  (goto (label eval-dispatch))
ev-if-consequent
  (assign exp (op if-consequent) (reg exp))
  (goto (label eval-dispatch))

ev-assignment
  (assign unev (op assignment-variable) (reg exp))
  (save unev)
  (assign exp (op assignment-value) (reg exp))
  (save env)
  (save continue)
  (assign continue (label ev-assignment-1))
  (goto (label eval-dispatch))
ev-assignment-1
  (restore continue)
  (restore env)
  (restore unev)
  (perform
   (op set-variable-value!) (reg unev) (reg val) (reg env))
  (assign val (const ok))
  (goto (reg continue))

ev-definition
  (assign unev (op definition-variable) (reg exp))
  (save unev)
  (assign exp (op definition-value) (reg exp))
  (save env)
  (save continue)
  (assign continue (label ev-definition-1))
  (goto (label eval-dispatch))

ev-definition-1
  (restore continue)
  (restore env)
  (restore unev)
  (perform
   (op define-variable!) (reg unev) (reg val) (reg env))
  (assign val (const ok))
  (goto (reg continue))
```

Problems

Trace through evaluating the following examples:

1. ```
   (define a 5)
   (if (> a 4) 'yes 'no)
   ```

2. ```
   (define f (lambda (x) (f (+ x 1))))
   (f 0)
   ```
   Will this expression run forever or will it exhaust the finite stack space?

Adding And

Let's add `and` to the ec-eval. First add a clause to `eval-dispach`:

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

`ev-and` assumes 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 return to.

3. Fill in the missing spots in starred lines of `ev-and`. Assume that primitives `and-first` and `and-second` to extract parts from the expressions.

```
1. ev-and
2.   (assign unev  (op and-second) (reg exp) )
3.   (assign exp  (op and-first) (reg exp) )
4.   (save continue)
5.   (save env)
6.   (save unev)
7.   (assign continue eval-after-first)
8.   (goto  (label eval-dispatch) )
9. eval-after-first
10.  (restore  unev )
11.  (restore  env )
12.  (test (op true?) (reg val))
13.  (branch (label eval-second-arg))
14.  (assign val #f)
15.  (restore  continue )
16.  (goto (reg continue))
17. eval-second-arg
18.  (assign exp  (reg unev) )
19.  (assign continue after-second)
20.  (goto  (label eval-dispatch) )
21. after-second
22.   (restore continue)
23.  (goto (reg continue))
```

4. 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 trace through evaluating:

```
(and #t (and #f #t))
```

5. 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. (restore continue) 20. (goto (label eval-dispatch))

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

   Move line 15 after line 11

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

   Yes, if true? is defined as anything that is not #f

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

   Change line 10 to (restore exp)