MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall 2002

Project 3

Issued: Week 11 / Day 5

Due:
Part I, Week 12 / Day 5, by evening
Part II, Week 14 / Day 5, by evening

Reading: *SICP* Sections 4.1–4.1.6, 4.2, and attached code files `c-eval.scm` and `desugar.scm`.

An Explicit Continuation Evaluator for Scheme

Yet Another Scheme Interpreter

Introduction

You will be working with a Scheme interpreter similar in many ways to those described in *SICP*, Chapter 4. Without a good understanding of how an interpreter is structured, it is very easy to become confused between the programs that the interpreter is evaluating, the procedures that implement the interpreter itself, and underlying Scheme procedures called by the interpreter. You will need to study carefully Chapter 4 through subsection 4.2.2 and the attached code to do this project.

In particular, the new Scheme interpreter `c-eval`[1] used in this project has essentially the same procedures for handling syntax and environments as the basic "meta-circular" interpreter `m-eval` from lecture and Section 4.1.1 of *SICP*. Like the other interpreters we have considered, the definition of `c-eval` is also a conditional branch on the type of expression being evaluated. But `c-eval` differs from the other interpreters because it is written in "continuation style." This means its individual expression evaluation procedures are defined to *pass* the value of an expression to an explicit *continuation procedure* instead of simply returning the value. We'll explain this in detail in Part II of the project. You won't need to understand continuation style to complete Part I.

The reason for introducing this new explicit continuation interpreter is that it provides a simple way to explain and implement features that affect the overall "flow of control" in an interpreter. For example, Scheme's `error` procedure is the simplest flow of control operator. When `error` is called, the evaluation aborts directly to the Scheme read-eval-print loop, without returning to any of the procedures awaiting values.

[1]In order to avoid confusing the `eval` and `apply` of this interpreter with the `eval` and `apply` of the underlying Scheme system, we have renamed these interpreter procedures `c-eval` and `c-apply`.

There is a dramatic difference in the ease with which the recursive meta-circular interpreter and the explicit continuation interpreter can be extended to handle calls to `error`. It could be done in the recursive `m-eval` interpreter, but would require revising nearly every procedure call in the interpreter code to test if the call returns an error value; this would be very cumbersome. But in the explicit continuation interpreter, you will see that it requires inserting only a few lines of code in two places to install `error`.

Using the Interpreter

The Scheme code for this Project comes in two files:

- `c-eval.scm` contains the new explicit continuation interpreter `c-eval`. This interpreter handles only the five special forms `if`, `define`, `lambda`, `set!`, and `begin`.

  The `c-eval` has been extended to handle "dotted" parameters (we used these in Project 2; see the Scheme manual for more on how these work).

- `desugar.scm` contains procedures that transform a Scheme expression with various special forms such as `and`, `cond`, and `let` into an expression containing at most the five special forms `if`, `define`, `lambda`, `set!`, and `begin`. Every expression to be evaluated is "desugared" first.

  In this way, an interpreter that directly evaluates only these five special forms still winds up being able to handle nearly all the Scheme special forms.

Here's how to use the extended interpreter:

- Start up Scheme/Edwin and load the files `c-eval.scm` and `desugar.scm`.

  Since the interpreter's environments are represented as elaborate circular list structures, you should set Scheme's printer so that it will not go into an infinite loop when asked to print a circular list. This can be done by evaluating the expressions

  ```
  (set! *unparser-list-depth-limit* 7)
  (set! *unparser-list-breadth-limit* 10)
  ```

  which you can find at the end of the file `c-eval.scm`. You may want to alter the values of these limits to vary how much list structure will be printed as output.

- To start the `c-eval` read-eval-print loop, evaluate `(start-c-eval 1)`. This starts the read-eval-print loop with a freshly initialized global environment. The argument to `start-c-eval`, in this case `1`, serves as a label that appears in the input prompts and outputs from the interpreter.

  To have `c-eval` evaluate an expression, you may type the expression into the `*scheme*` buffer after the `;;C-Eval1>>` input prompt and then type `C-x C-e`. (Be careful using `M-z` to evaluate the expression in the `*scheme*` buffer—the presence of the interpreter prompts can confuse the `M-z` mechanism. But `M-z` and `M-o` work fine for sending expressions to the `*scheme*` buffer from other Scheme-mode buffers.)

  For example,

```
(start-c-eval 1)

;;C-Eval1>>
(if (not (= 1 2)) (list 3) 4)
                                  ;USER TYPES C-x C-e:
;;C-Eval1 value: (3)

;;C-Eval1>>
```

shows an interaction with the `c-eval` interpreter.

- It may be helpful to keep various files of procedure definitions you want to run the interpreter on. When your Edwin Scheme buffer is running the `c-eval` read-eval-print loop, you can then visit such a file and type `M-o` to have all the expressions in the file evaluated by `c-eval`.
- You can exit or interrupt `c-eval` by typing `ctrl-c ctrl-c`. To restart it with the recent definitions still intact, evaluate `(restart-c-eval)`.

  The `c-eval` interpreter has a limited error system, and evaluation errors will often bounce you back into ordinary Scheme. In this case, you can also restart the current interpreter—with its global environment still intact—by evaluating `(restart-c-eval)`.
- To undertand or debug a procedure definition, it can be helpful to have a *trace* of successive arguments to which the procedure is applied during an evaluation. MIT Scheme supports this activity with a builtin procedure `trace-entry`. You will be asked to trace interpreter procedures in several problems in this project. (You may also want trace your own procedures to debug them for this assignment.)

  Applying `trace-entry` to a defined procedure will cause it to report its successive arguments during a computation. (Note that you should ask the underlying Scheme evaluator to "trace" things – we have not installed tracing into the `c-eval` evaluator.) For example:

```
(define (map1 f l)
  (if (null? l) nil (cons (f (car l)) (map1 f (cdr l)))))
;Value: "map1 --> #[compound-procedure 3 map1]"

(trace-entry map1)
;Value: #[unspecified-return-value]

(map1 inc '(1 2 3))

[Entering #[compound-procedure 3 map1]
    Args: #[compiled-procedure 5 ("arith" #x3A) #xF #x10BED93]
          (1 2 3)]
[Entering #[compound-procedure 3 map1]
    Args: #[compiled-procedure 5 ("arith" #x3A) #xF #x10BED93]
          (2 3)]
```

```
[Entering #[compound-procedure 3 map1]
    Args: #[compiled-procedure 5 ("arith" #x3A) #xF #x10BED93]
          (3)]
[Entering #[compound-procedure 3 map1]
    Args: #[compiled-procedure 5 ("arith" #x3A) #xF #x10BED93]
          #f]
;Value: (2 3 4)
```

Apply `untrace` to the procedure to turn off tracing. There is also a `trace-exit` procedure for tracing successive values returned:

```
(untrace map1)
;Value: #[unspecified-return-value]

(trace-exit map1)
;Value: #[unspecified-return-value]

(map1 inc '(1 2 3))

[#f
      <== #[compound-procedure 3 map1]
    Args: #[compiled-procedure 5 ("arith" #x3A) #xF #x10BED93]
          #f]
[(4)
      <== #[compound-procedure 3 map1]
    Args: #[compiled-procedure 5 ("arith" #x3A) #xF #x10BED93]
          (3)]
[(3 4)
      <== #[compound-procedure 3 map1]
    Args: #[compiled-procedure 5 ("arith" #x3A) #xF #x10BED93]
          (2 3)]
[(2 3 4)
      <== #[compound-procedure 3 map1]
    Args: #[compiled-procedure 5 ("arith" #x3A) #xF #x10BED93]
          (1 2 3)]
;Value: (2 3 4)
```

Part I, due: **Week 12 / Day 5**

Getting Started

Problem 1: Start the `c-eval` interpreter and evaluate a few simple expressions and definitions. It's a good idea to make an intentional error and practice restarting the read-eval-print loop. Show a transcript of your tests.

Problem 2: Define some simple recursive procedure (eg, `factorial`, `fibonacci`, ...) and trace the arguments to `c-eval` and `c-apply` when your procedure is applied to some argument. Turn in this trace as part of your solutions. (Note, to do the tracing, you want to enable tracing from the underlying Scheme environment, then return to `c-eval` and evaluate your procedure; similarly, you need to return to the underlying Scheme evaluator to turn off the tracing). Notice how the arguments are passed into your procedure at each call.

The rest of this project involves defining various Scheme procedures—usually to extend the interpreter in some way. For every problem that asks you to produce Scheme code, you should be sure to include printouts illustrating its correct execution on appropriate examples. If your solution is buggy or incomplete, you should include comments illustrating what examples your code *can* and *cannot* handle, and explaining the way you intended your code to work.

The following problems in Part I require only small additions or modifications to the code in the problem set files (if you find yourself making major changes or defining many procedures, you're on the wrong track). They do not require an understanding of the continuation passing style, which we take up in Part II. However, to do Part I, you will have to study the files and develop a clear understanding of the basic features of the interpreter.

Problem 3: The interpreter does not handle the special form `cond`. Modify (i.e., change existing procedures and add any new procedure) the procedures in `desugar.scm` to desugar `cond` into an expression that `c-eval` can evaluate. Make sure that your `desugar-cond` procedure correctly desugars all the conditional expressions in `c-eval.scm` (if it doesn't you won't be able to the final problems in Part I). Be sure to create any appropriate data abstractions.

Problem 4: Look up the official specification of the special form `or`. Modify the procedures in `desugar.scm` to desugar `or` into an expression that `c-eval` can evaluate. Remember that the individual expressions in the body of `or` are supposed to be *evaluated at most once*:

```
(or
 (begin (display "once ") #f)
 (begin (display "and ") #f)
 (begin (display "only") 'done))
once and only
;Value: done
```

(One way to desugar `or` involves using `let`—which can itself be desugared later. In this case there can be "false capture" errors if the `let` variable already appears in the `or` expression being desugared. For this problem, it is OK to assume that any variables you use in desugaring do not already appear. Note that this is not the only way to desugar an `or` expression.)

Procedure Values

There are three kinds of procedure values in the explicit continuation interpreter:

1. `c-eval` *primitive* procedures. These are bound by the interpreter's `make-initial-environment` procedure (cf., `c-eval.scm`) to underlying Scheme procedures. The underlying Scheme procedures may be Scheme primitives like `+`, or Scheme compound procedures like `c-pair?`.

2. `c-eval` *compound* procedures. These are the values of `lambda` expressions evaluated in `c-eval`. It can also be convenient to have `make-initial-environment` install some compound procedures in the global environment on startup. There is one example of this in the project code: the procedure `append` is predefined when the interpreter starts.

3. `c-eval` *special* procedures `apply`, `error` and `exit`.

Problem 5: Notice that the `make-initial-environment` procedure in `c-eval.scm` binds many interpreter variables such as `+`, `symbol?`, ... to corresponding procedures built into Scheme. But the interpreter's variable `pair?` is bound to a special `c-pair?` procedure. Give an example of what would go wrong if the interpreter's variable `pair?` was bound instead to Scheme's built-in `pair?` procedure.

Now temporarily modify `make-initial-environment` so the interpreter's variable `pair?` is bound to to Scheme's built-in `pair?` procedure, and run the interpreter on your example. Then restore the original definition of `initial-environment` and verify that your example is handled correctly.

Problem 6: Extend `c-eval` so that the language interpreted by `c-eval` includes builtin predicates `procedure?` and `list?` (similar to `pair?`). Make sure your predicates do not confuse `c-eval` procedures, which are represented as lists, with `c-eval` representations of genuine list values.

For example, evaluating `(procedure? (list (list 'procedure) +))` in the interpreter should return `#f`—even though its argument evaluates to a list that looks like the interpreter's representation of a compound procedure. Conversely, evaluating `(list? +)` in the interpreter should also return `#f`, even though the interpreter's representation for the primitive procedure `+` is a list.

Problem 7: Add a new "special" procedure `eval` to the interpreter. Evaluating the expression

```
(eval '<expr>)
```

should lead to the same computation as evaluating `<expr>` in **the global environment**. For example,

```
;;C-Eval1>>
(let ((x 3))
  (eval '(define x 9))
  x)

;;C-Eval1 value: 3

;;C-Eval1>> x

;;C-Eval1 value: 9
```

Hint: Model your answer after the `apply` special procedure. You do not really need to understand how continuation passing works to do this. Note how `apply` is inserted into the initial environment, and how it is actually handled as part of the "special" procedures.

Problem 8: Now that `cond` and `or` can be desugared, `c-eval` is very nearly general enough to *evaluate itself.*

Evaluate the procedures in your modified `desugar.scm` file in Scheme (in Edwin, just visit the file and do `M-o`). Now start the `c-eval` read-eval-print loop, and at the `;;C-Eval1>>` input prompt, evaluate the procedures in `c-eval.scm`. This means you have now evaluated all of these procedures within the continuation evaluator. Then at the final `;;C-Eval1>>` input prompt, type `(start-c-eval 2)`. What happens?

Fix the problem by modifying `make-initial-environment` to include the missing procedure in the global environment (like the procedure `append`). Go back and repeat the process of loading the the files into the continuation evaluator and try evaluating `(start-c-eval 2)` again. You should get a `;;C-Eval2>>` input prompt, indicating that you are in the read-eval-print loop of a level 2 `c-eval` interpreter running in a level 1 `c-eval` interpreter running in Scheme!

Evaluate a couple of simple expressions at the `;;C-Eval2>>`; don't expect a speedy answer.

To return to the level 1 interpreter, evaluate `(exit)` at the level 2 prompt. Then to return to Scheme, evaluate `(exit)` at the level 1 prompt.

Turn in printouts showing that you accomplished this.

Problem 9: Finally, once we can run an evaluator within an evaluator, we can monitor the amount of work we do in evaluating expressions. One way to do this is to create a new special form, called `meter`, that we can use to explicitly meter the use of resources for a given expression, e.g., `(meter (fact 5))`. However, we are going to make things a bit easier, by altering our evaluator to meter the use resources for every expression that we type in.

You will do this in the following manner:

- Create some global variables, that you will use to count the number of times an operation is performed. For example, `*eval-count*` should be used to count the number of times that `eval` is called, and `*apply-count*` should be used to count the number of times a **primitive** procedure is applied.
- Modify your code for the evaluator so that these two variables are incremented each time their associated operation is performed.
- You need to adjust the part of the evaluator that interacts with the user, the so-called `READ-EVAL-PRINT` loop. Inside of `start-c-eval` you will see code that accomplishes this, called `driver-loop`. Look this code over carefully. You should alter this code so that your variables are reset to 0 each time `driver-loop` is called.
- Finally, you need to print out the information about usage of `eval` and `apply`. This part is a bit tricky. If you look at the call to `c-eval` within driver loop, you will see it has three arguments: the expression to evaluated (which is read in and then desugared), the environment with respect to which the expression should be evaluated, and a third argument, which is the continuation. This third argument we will explore in much more detail in the second part of the project. For this part, you simply need to think of this as a procedure that is applied to the value returned by the evaluation – in this case you can see that it prints out

the result and then calls `driver-loop` recursively. You will need to alter this continuation procedure so that it also prints out (or displays) information about the number of calls to `c-eval` and the number of primitive `apply`s.

Once you have made your changes, try reloading your evaluator code `c-eval` and `desugar`, and then try some simple expressions. Note how much effort is expended in doing this. Try defining a simple recursive, like `factorial` or `fibonacci`, and record the number of evals and applys used in applying these procedures to some argument.

Next, repeat the process of Problem 8: start up your `c-eval`uator, then evaluate the contents of your code buffer by using `M-o`; and then start up a second level evaluator. Repeat the same definition of a recursive procedure that you just did (you need to do this within this new evaluator), and applying it to the same argument. Notice that the number of evals and applys is the same. Now, exit out of this second level evaluator. Notice what gets printed out in terms of evals and applys. This is the amount of work that the evaluator itself had to do to similuate the process of evaluation!

Do the same thing with an iterative version of the procedure.

Part II, due **Week 14 / Day 5**

In this part of the project, we'll extend Scheme with some further flow of control operations that are more interesting than simply aborting computation by calling `error`. There is no direct way to incorporate such operations into recursive meta-circular interpreters. It is easy to do when continuations are explicit.

Continuation Passing

Start by considering our normal Scheme evaluator. The situation at any point during evaluation of an expression can usefully be understood as consisting of two parts: rules for finding a value of a subexpression, and rules for what to do when the value is found. It is possible to rewrite a procedure so the rules for what to do with the returned value are packaged into an explicit *continuation procedure*. Instead of returning the subexpression value, the computation "passes on" the value by applying the continuation procedure to the value.

Because values of subexpressions are always passed on, they *never get returned*, except perhaps at a final step. And since no value gets returned after a procedure call, continuation style procedures always turn out to be *tail-recursive*. We explained early in the term that tail-recursive procedures provide a stack-space saving benefit. That's important, but not our main concern.

As we indicated, the real point of having explicit continuations is to manage flow of control during computation. This can be a valuable technique in Scheme programming *regardless of whether or not we are defining an interpreter*. The added benefit of writing an interpreter in continuation passing style is that it becomes easy to add all sorts of flow of control primitives to the language being implemented by the interpreter.

As a first example of how explicit continuations work, we'll consider the recursive procedure `flatten`. Applying `flatten` to a tree returns a list of its leaves. For example, evaluation of

```
(flatten '((a (b) ((c (d))) e)))
```

returns the list `(a b c d e)`.

The usual way to define flatten is by a tree recursive procedure:

```
(define (flatten tree)
  (cond ((null? tree) '())
        ((not (pair? tree)) (list tree))
        (else (append (flatten (car tree))
                      (flatten (cdr tree))))))
```

It isn't obvious how such a procedure could be made tail recursive. Here's how: we will write a continuation style flatten procedure `cs-flatten` that takes *two* arguments, a tree, `T`, to be flattened and a one argument continuation procedure, `receiver`.

Evaluating `(flatten T receiver)` leads to `receiver` being applied to the flattened list of leaves of `T`. For example,

```
(define identity (lambda (l) l))
;Value: "identity --> #[compound-procedure 17 identity]"

(cs-flatten '(((a) b) (((c)))) identity)
;Value: (a b c)

(cs-flatten '(((a) b) (((c)))) second)
;Value: b

(define (announce-null leaves)
  (if (null? leaves)
      (begin (display "empty tree")
             nil)
      leaves))
;Value: "announce-null --> #[compound-procedure 18 announce-null]"

(cs-flatten '(((a) b) (((c)))) announce-null)
;Value: (a b c)

(cs-flatten '() announce-null)
empty tree
;Value: #f
```

To define `cs-flatten`, we begin with the observation that the usual recursion for flattening a tree, T, consists of three main steps:

1. flatten the car of T,
2. flatten the cdr of T,
3. append the results of these two flattenings.

Let's assume for the moment that we can take `append` as a primitive. We can accomplish the first step, flattening the car of T, by calling `(cs-flatten (car T) identity)`. But this call `returns` the list of leaves instead of passing it on. Continuation passing strategy is to pass the leaves of the car to a continuation that will go on to perform the remaining two steps.

Now the second step is to flatten the cdr of T, so a continuation to accomplish the second step is

```
(lambda (flattened-car) (cs-flatten (cdr T) ...)
```

What goes in the ellipsis (three dots) above? The answer is a continuation that receives the leaves of the cdr and performs the third step of appending the two lists. This final list must then be passed on to `receiver`. So this final continuation is:

```
(lambda (cdr-leaves) (receiver (append car-leaves cdr-leaves)))
```

Here is the final definition:

```
(define (cs-flatten T receiver)
  (cond ((pair? T)
         (cs-flatten                   ;(1.) flatten
          (car T)                      ;the car of T, and pass the result to
          (lambda (car-leaves)         ;a procedure that will
            (cs-flatten                ;(2.) flatten
             (cdr T)                   ;the cdr of T, and pass the result to
             (lambda (cdr-leaves)      ;a procedure that will
                              ;(3.) append the two flattened lists
                              ;and pass the resulting list to RECEIVER:
               (receiver
                (append car-leaves cdr-leaves)))))))
        ((null? T) (receiver nil))
        (else (receiver (list T)))))
```

It may seem like cheating to treat `append` as primitive. But we can also define `append` in continuation passing style: break the usual recursion for appending two lists into two steps:

1. append the `cdr` of the first list to the second list, and
2. cons the car of the first list onto the resulting list.

This leads to

```
(define (cs-append l1 l2 list-receiver)
    (if (pair? l1)
        (cs-append              ;(1.) append
         (cdr l1) l2            ;the CDR of l1 to l2, and pass the result to
         (lambda (appended-cdr)  ;a procedure that will
                            ;(2.) CONS the CAR of l1 to that result,
                            ;and pass the final list to LIST-RECEIVER:
           (list-receiver
            (cons (car l1) appended-cdr))))
        (list-receiver l2)))
```

Putting `cs-append` together with `cs-flatten` yields an entirely tail-recursive, continuation passing procedure:

```
(define (cs-flatten T receiver)
  (define (cs-append l1 l2 list-receiver)
    (if (pair? l1)
        (cs-append
         (cdr l1) l2
         (lambda (appended-cdr)
           (list-receiver
            (cons (car l1) appended-cdr))))
        (list-receiver l2)))
  (cond ((pair? T)
         (cs-flatten
          (car T)
          (lambda (car-leaves)
            (cs-flatten
             (cdr T)
             (lambda (cdr-leaves)
               (cs-append car-leaves cdr-leaves receiver)))))) ;NOTE THE CHANGE HERE
        ((null? T) (receiver nil))
        (else (receiver (list T)))))

(define (flatten T)
   (cs-flatten T identity))
```

Problem 10: Define a tail-recursive, continuation passing style procedure `matchup-var-vals` taking three arguments: a list of variables (represented as Scheme symbols), a corresponding list of values, and a receiver procedure. Applying `matchup-var-vals` will split the list of variables into two lists—those that do *not* satisfy a `dynamic?` predicate and those that do—and likewise split the corresponding list of values into two lists. These four lists will be passed as arguments to the receiver procedure.

The `dynamic?` predicate is included in `c-eval.scm`; it is satisfied precisely by symbols that start with `%d-`. For example,

```
(matchup-var-vals
 '(aa dd %d-xx %d %dd %d-aa)
 '(1  2  3     4  5   6)
 list)
;Value: ((aa dd %d %dd) (1 2 4 5) (%d-xx %d-aa) (3 6))
```

You may assume that the first two arguments to `matchup-var-vals` are of equal length.

Also, remember that you are defining this in the normal Scheme evaluator.

Hint: think about how many arguments the continuation procedure should take.

Continuations in The Interpreter

Now we can turn to the continuation evaluator. Let's look at how continuations work in the explicit continuation interpreter. The `c-eval` procedure, like `m-eval`, takes an expression `<expr>` and an environment `<env>` as arguments, but it also takes as a third argument some procedure `<cont>`. Applying `c-eval` causes the result of evaluating `<expr>` in `<env>` to be passed on to `<cont>`. For example, evaluating

```
(c-eval expr env identity)
```

returns the same value and has exactly the same effect as evaluating

```
(m-eval expr env)
```

We have already remarked that, like the other interpreters we have considered, the definition of `c-eval` is a conditional branch on the type of expression being evaluated. The simplest case is when the expression is a number or other self-evaluating value. Then `c-eval` simply passes the expression directly to its continuation:

```
(define (c-eval expr env cont)
  (cond ((self-evaluating? expr) (cont expr))
        ...))
```

A more interesting case is for `if` expressions;

```
(define (c-eval expr env cont)
  (cond ...
        ((if? expr) (eval-if expr env cont))))
```

The strategy for defining `eval-if` is to remember the two steps in the recursion for evaluating if-expressions:

1. evaluate the predicate of the if-expression, and then

2. if the value of the predicate is true, evaluate the consequent of the if-expression, otherwise evaluate the alternative of the if-expression.

The first step can be accomplished with a recursive call to `c-eval` on the predicate of the if-expression:

```
(define (eval-if expr env cont)
   (define choose-branch-eval-and-continue ...)
   (c-eval (if-predicate expr) env choose-branch-eval-and-continue))
```

Here `choose-branch-eval-and-continue` should be a continuation that will perform the second step and pass the resulting value to `cont`:

```
(define choose-branch-eval-and-continue
  (lambda (pred-val)
    (c-eval (if pred-val
                (if-consequent expr)
                (if-alternative expr))
            env cont)))
```

Notice that in addition to the predicate value it receives as an argument, `choose-branch-eval-and-continue` needs values for `expr`, `env` and `cont`. It gets these by being defined in the scope of `eval-if`.

Problem 11: You should now be able to read and understand the procedure definitions in `c-eval.scm` well enough to write some similar ones of your own.

In particular, extend `c-eval` so that the `or` special form is handled directly instead of requiring prior desugaring. Namely, define a continuation style `eval-or` procedure, add it to the `c-eval.scm` file, and add an `or?` clause to the `c-eval` procedure. Handling `or` as a special form avoids some of the technical problems with accidental variable capture we noted can come up when `or` is handled as syntactic sugar.

Execute `(trace-entry eval-or)` and then test your extended `c-eval` on

```
(or (begin (display "hello ") #f)
    (or (begin "good-bye " (= 1 2))
        (not #t))
    (let ((x "only once")) (display x) #f)
    'done)
```

(Be sure to redefine the `desugar` procedure so it does not desugar `or`'s. Otherwise, your `eval-or` procedure will never be called.)

Then exit the `c-eval` interpreter, evaluate `(untrace eval-or)`, `(trace-exit eval-or)`, and repeat the test. If your `eval-or` is in proper continuation passing style, there should be no returned values displayed by `trace-exit`. Turn in the code and results with your solutions to Part 2.

Errors and Exits

Now we can explain how the `error` and `exit` procedures work. For procedure application, `c-eval` calls `c-apply`. Like `m-apply`, the procedure `c-apply` takes an operator and operand values as arguments. It also takes an explicit continuation argument.

The normal task of `c-apply` is to apply the operator to the operands and pass the resulting value to its continuation argument. If the operator is a special-procedure, it dispatches to an `apply-special` procedure. When `apply-special` detects that the operator to be applied is `error` or `exit`[2], then `apply-special` winds up ignoring its continuation. Instead of passing a value to its continuation, it displays a message and *actually returns* some unspecified error or exit value:

```
(define (apply-special op vals cont)
  (let ((name (special-op-name op)))
    (cond
       ...
     ((eq? '*error name)                  ;CONT IS IGNORED HERE
      (apply fatal-c-eval-error vals))
     ((eq? '*exit name)
      "exited C-EVAL")               ;AND HERE
     ...)))
```

Since all the evaluation procedures are tail-recursive, returning a value causes execution of the interpreter to stop, with the value going directly to Scheme's read-eval-print loop. After that, the user can continue his Scheme session. You've seen `exit` and `error` behave this way in Part I.

Another feature of the explicit continuation interpreter is that it need not bounce all errors back to the Scheme read-eval-print loop. For example, suppose `c-eval` calls for looking up a variable in an environment, and the variable is not there. Here too, the explicit continuation gets ignored. But instead of *returning* an error message, the lookup procedure *displays* an error message, and then restarts the interpreter read-eval-print loop:

```
(define (lookup-variable-value var env cont)
  ....
  (if (eq? env the-empty-environment)
      (begin
       (display "C-EVAL INTERPETATION ERROR: Unbound variable: ")
       (display var)
       (newline)
       (restart-c-eval))
       ...))
```

Problem 12: We observed that when the operator is `error`, the `apply-special` procedure ignores its continuation argument and returns an error message directly to Scheme's read-eval-print loop

[2]More precisely, `error` and `exit` are variables. Their values are special procedures, each of which is represented as a tagged name.

(or if it is running inside itself, it returns up one level). The environment where `error` was applied is saved, so the user can call `(restart-c-eval)` and proceed to interact with the interpreter in the saved environment.

It would be useful for debugging, among other things, if after such an `error` return the user has another option: by calling `restart-c-eval` with an argument, the user could continue the evaluation where it left off when the error occurred. This can be accomplished simply by *storing* the continuation argument inside the `restart-c-eval` procedure instead of ignoring it. (Hint: Think about what this means. To capture the continuation, you need to redefine `restart-c-eval` so that it has access to a local frame in which some variable is bound to that continuation, so that it can access it.)

In particular, if the user calls `restart-c-eval` in Scheme with an argument, say `(restart-c-eval <expr>`, then `<expr>` gets evaluated by `c-eval`, and the resulting value is passed on if there was no error. For example,

```
;;C-Eval1>>
(list (begin (display 'a) (newline) 1)
      (error 'jump-out-here)
      (begin (display 'z) (newline) 3))
a                                         ;displayed 'a here
FATAL-C-EVAL-ERROR: jump-out-here
;Value: (unspecified)

(restart-c-eval 'middle-value)
z                                         ;displayed 'z here

;;C-Eval1 value: (1 middle-value 3)

;;C-Eval1>>
```

Extend `c-eval` so that `restart-c-eval` can take an optional argument and will behave in this way. (This means restart-c-eval must be redefined to take zero or one argument.)

Handling `call-with-current-continuation`

Flow of control gets really powerful when there is a procedure like the restartable `error` of the previous problem, but which allows the computation to proceed—with the option to restart where the "error" occurred—instead of bouncing out of the interpreter with an error message. Scheme's procedure `call-with-current-continuation`—`call-cc` for short—does just that.

Instead of an error message, the argument to `call-cc` will be a procedure, and instead of storing the "error" continuation in the `restart-c-eval` procedure, `call-cc` passes the continuation to its argument.

For example, we can use `call-cc` instead of `error` to "automatically" restart the aborted evaluation in the above, instead of asking the user to invoke `restart-c-eval`:

```
;;C-Eval1>>
(list (begin (display 'a) (newline) 1)
      (call-cc (lambda (restart)
                 (display "<let c-eval continue without going to Scheme REPL> ")
                 (newline)
                 (restart 'middle-value)))
      (begin (display 'z) (newline) 3))
a
<let c-eval continue without going to Scheme REPL>
z

;;C-Eval1 value: (1 middle-value 3)

;;C-Eval1>>
```

Problem 13: Add a `call-cc` special procedure to `c-eval`. (Hint: This can be accomplished by adding a total of about a half dozen lines to the definitions of `apply-special` and `make-initial-environment`. Here's how it works: to evaluate `(call-cc <expr>)`, expression `<expr>` gets evaluated and should return a procedure value `<f>`. Then the `apply-special` packages up its continuation argument—the "current" continuation—into a new kind of special procedure with a "continuation" tag, and the value of `<f>` is applied to the newly created special procedure. Later if one of these special tagged procedures `<p>` is applied to an argument `<a>`, the `apply-special` procedure ignores its given continuation argument. Instead it extracts the continuation packaged inside `<p>` and applies it `<a>`.)

OPEN-ENDED Problem 14:

This final problem is meant to give you an opportunity to do some original language design or implementation of your own.

Your modification need not be extensive. Here are some examples of things you might do.

- Implement the Scheme special form `do` that defines a program loop, and then you could add some `break` procedure to abort the loop and immediately return a value to the procedure that called the `do`. You could do this in `c-eval`, or if you prefer, do it in the traditional `m-eval`. We have made a copy of `m-eval` available on the project web page for you to download. Note in doing this, you get to define the syntax for a "do" loop, and the semantics of how it should proceed. Typically such a loop has some counter, that it tests against an end test at each stage. If the end test is true, the loop exits with some return value, otherwise, the body of the loop is evaluated, the loop counter is incremented, and the process is repeated.

- Add lazy parameters, with some different rules for when a thunk gets forced.

- Revise the code in `c-eval.scm` to define an extension of the `c-eval` interpreter that treats variables satisfying the `dynamic?` predicate as dynamically scoped. Namely, with this extension, all the procedures like `eval-if` and `eval-assignment` will take *two* environment parameters, the usual static environment `env`, and an additional dynamic environment `denv`. Applying a compound procedure to arguments now may cause frames to be hung onto each of the environments: the usual frame with the static variables bound to their values gets hung onto the static environment `env`, and another frame with the dynamic variables bound to *their* values gets hung onto `denv`. Also, dynamic variables get looked up in `denv`, while non-dynamic variables are treated as usual and looked up in the static environment `env`.

Your extended interpreter should remain in continuation passing style. This revision affects all the important procedures in `c-eval.scm`, but the required changes are minimal, consisting mainly of adding a `denv` parameter to the procedures. The interesting part of the revision concerns code for application of compound procedures. Here the `matchup-var-vals` procedure of II.2 will come in handy.

- Revise the intepreter to be an "analyzing" interpreter as described in SICP. (If you do rewrite it as an analyzing interpreter, you should notice a dramatic improvement in speed; see how many levels deep in self-evaluating interpreters you can get before the innermost evaluation slows to a crawl.)
- Something else (though if you do this, we strongly suggest that you send a note describing what you plan to do to your TA **EARLY** so that you can get some guidance on whether your idea is doable in a reasonable amount of time and effort).

Whatever extension you choose to make, be sure you describe it clearly and hand in your description, *new* code for the extended interpreter (don't include in your submission any procedures in the downloaded files that you have not changed), and a selection of examples with comments.

By the end of this project, we hope that you will have mastered much of the art of intepretation.

Collaboration Statement

We encourage you to work with others on problem sets as long as you acknowledge it (see the 6.001 General Information handout). If you cooperated with other students, LA's, or others, indicate who they were and explicitly label all text and code you are submitting which is the same as that being submitted by one of your collaborators. Otherwise, write below "I worked alone using only the course reference materials."