Brandeis University

Computer Science (1)21b (Spring Term, 2016)
Structure and Interpretation of Computer Programs

Problem Set 6: A Scheme-to-Scheme Compiler

Due Monday, May 2

Reading Assignment: Chapter 5, Sections 5.1, 5.4, 5.5, and class handouts. (The textbook reading is supplementary, but read the handouts *very carefully.*)

Bad news: this is a difficult assignment. Good news: like the last problem set, you need to understand a lot, but in the end, you don't have to code that much.

This problem set builds on our discussion of the metacircular evaluator, in particular the evaluator with fully analyzed syntax (see Section 4.1.7 of the textbook), explicit description of both the environment and the control (see handout on the metacircular explicit-control evaluator), and lexical analysis of variables (see that handout too). You've seen all the pieces in class lectures; now we'll put them together to make a Scheme-to-Scheme *compiler.*

Without fully analyzed syntax, as in the first version of the metacircular evaluator, every time we run `factorial` in computing `(factorial 5)`, we check whether the body is an `if` expression. The "vanilla" metacircular evaluator tells everything there is to know about handling environments—but (most mysteriously!) adding an explicit *continuation* to the evaluator creates an *explicit-control* evaluator: not only does the tail-recursive code of the evaluator tell everything there is to know about the inherently recursive control, it lets us put gadgets in the interpreted language that can actually hack the control. Finally, lexical analysis trades "compile-time" analysis of code for a faster "run-time" variable lookup when we need to access those values.

Here is what you need to do in this problem set: you are given, as a model, a Scheme-to-Scheme compiler[1] that has explicit handling of both the environment and the control, lexical addressing, `letrec`, `enter` and `exit`, but *only* `lambda`-expressions that have exactly one bound variable.

- First, you have to modify the compiler so `lambda`-expressions can have an arbitrary number of bound variables. In fact, you will be given such a compiler, minus certain specific pieces that you need to code: how to compile variables (lexical addressing), and how to compile `lambda`-expressions and applications.
- Second, you have to implement a refined cousin of `enter` and `exit` called `call/cc`, which means "call with current continuation." The `call/cc` construct takes the current continuation and packages it up as a Scheme *function* that can be applied to a value. It is a "named" version of `enter` and `exit`. More explanations follow below.
- Third, you are asked to code a simple (very, very short!) *linking loader* that lets compiled Scheme target code interact normally with the underlying Scheme system.

All of the code for this problem set can be found on the home page for the course. The code that is presented here is only included for expository purposes, along with explanatory text.

[1]The virtue of a Scheme-to-Scheme compiler is that the "source language" and the "target language" are the same: we don't have to describe an assembly language or machine language! You may be used to thinking of a compiler as a translator from a "high level" to "low level" language—why then compile Scheme into itself? The "low level" is supposed to expose in more detail *how* the high-level code is to be run. In introducing lexical addressing, fully analyzed syntax, environments, and explicit control, we do exactly that.

1 A "unary" compiler

The compiler is given a Scheme expression, and a *compile-time environment* that lists the positions of the *free* variable values in a *run-time environment.* The compiler returns a Scheme procedure that can be executed with a run-time environment of values, and a run-time continuation:

```
(compile '(lambda (x) (lambda (y) (x (z (y w))))) '(z w))
;Value: #[compound-procedure 1]

(define code
  (compile '(letrec ((square (lambda (n) ((* n) n))))
             ((+ free) (square 5)))
           (cons 'free initial-names)))
;Value: code

(code (cons 10 initial-values) initial-continuation)
;Value: 35
```

(For now, think of `letrec` as synonymous with `let`. More later.) [2]

First, we give a brief tour of the "unary" compiler so you get some intuitions about the code you are to modify. Here's the top level (yawn!):

```
(define (compile exp env-names)
  (cond ((constant? exp)
         (compile-constant exp))
        ((variable? exp)
         (compile-variable exp env-names))
        ((letrec? exp)
         (compile-letrec (letrec-var exp)
                         (letrec-val exp)
                         (letrec-body exp)
                         env-names))
        ((lambda? exp)
         (compile-lambda (binder exp) (body exp) env-names))
        ((if? exp)
         (compile-if (predicate exp)
                     (then-part exp)
                     (else-part exp)
                     env-names))
        ((sequence? exp)
         (compile-sequence (cdr exp) env-names))
        ((enter? exp)
         (compile-enter (enter-body exp) env-names))
        ((exit? exp)
         (compile-exit (exit-body exp) env-names))
        (else ; it's an application!
         (compile-application (function-of exp)
                              (argument-of exp)
                              env-names))))
```

OK. *Now what is compiled code supposed to look like?* Here is the simplest example: a compiled constant. For example, to compile the constant 10, we generate the Scheme target code

[2] Since all procedures have only *one* argument, primitive procedures with more than one argument have been *curried*, so that we write `((* 3) 5)` instead of `(* 3 5)`, for example.

```
  (lambda (env-values cont)
    (cont 10)))
```

A compiled piece of code takes two arguments: an *environment* `env-values` of values, which supplies the *run-time environment* when the code is actually run, and a *continuation function* `cont` that says what to do with the value (in this case, a constant) that is returned by running the code.

In this simple compiler, a run-time environment is just a list of data, like `(3 1 4 5 9)`. Think of this as a list of 5 frames, but because `lambda`-expressions only have one bound variable, every frame has exactly one binding, so we just collect them in a simple list.

Analogous to a run-time environment—which we can only know when the compiled code is run—there is a *compile-time environment* that is known by the compiler. It is just another list, but of variable *names*, like `(x y z u v)`. The compiler cannot know what *values* will be paired with these names at runtime, but it *can* know *where* these values will be stored in the environment.

To repeat this essential point, think about *interpreting* integer addition as opposed to *compiling* integer addition. To interpret addition, you have to actually *add* two numbers—you need to know where they are, and what their *values* are. But to *compile* addition, you only need to generate machine (or Scheme) instructions that say, “Get the two items. Add them. Put the answer over here. Now go do something else.” To generate these instructions, you only need to know *where* in the environment the relevant integers are stored—you don’t need their values. The values can wait until when the code is run!

So here is how to compile a *variable* `v` in a compile-time environment `env-names`. Find the *lexical address* of the variable in the environment, and generate code that says to retrieve the datum at that address in the run-time environment:

```
(define (compile-variable v env-names)
  (let ((i (lookup-variable v env-names)))
    (lambda (env-values cont)
      (cont (list-ref env-values i)))))
```

Thus when the code is run, the `i`-*th* value is picked out of the run-time environment, and that value is passed to the continuation function. The procedure implementing variable lookup is quite ordinary:

```
(define (lookup-variable v env-names)
  (cond ((null? env-names) (error "Unbound variable"))
        ((eq? v (car env-names)) 0)
        (else (1+ (lookup-variable v (cdr env-names))))))
```

In reading the above code, it is important to separate in your mind what happens at “compile time” and what happens at “run time.” The above code `(lookup-variable v env-names)` is executed at compile time, and a lexical address is generated.[3] In contrast, the procedure `(lambda (env-values cont) ...)` encapsulates what happens at run time.

[3]For example, `(lookup-variable 'z '(x y z u v))` evaluates to the lexical address 2.

To compile a conditional, compile the predicate, consequent and alternative, and put them together appropriately:

```
(define (compile-if test then else env-names)
  (let ((test-code (compile test env-names))
        (then-code (compile then env-names))
        (else-code (compile else env-names)))
    (lambda (env-values cont)
      (test-code env-values
                 (lambda (p)
                   ((if p then-code else-code) env-values cont))))))
```

In the compiled code, the test code—also of the form `(lambda (env-values cont) ...)`— is executed using the run-time environment, with the following continuation: if evaluating the predicate returns a value `p`, check to see whether it is true or false, and run the appropriate ("then" or "else") code on the environment with the *original* continuation that specified what computation followed the evaluation of the `if` expression.

Next, we consider function abstraction and application:

```
(define (compile-lambda v exp env-names)
  (let ((body-code (compile exp (cons v env-names))))
    (lambda (env-values cont)
      (cont (lambda (x k)
              (body-code (cons x env-values) k))))))
```

At compile time, the body of the `lambda`-abstraction is compiled in the *extended* compile-time environment `(cons v env-names)`, where `v` is the formal parameter (variable binder) in the `lambda`-expression.

A one-argument `lambda`-expression in the source language becomes a *two*-argument procedure in the target language, also expressed as a `lambda`-expression (but in the underlying Scheme system!):

```
      (lambda (x k) (body-code (cons x env-values) k))
```

The arguments are the single parameter `x` of the source language expression, with an additional parameter `k` that is the continuation from which the procedure is *called* at run time. The code is: run the compiled body in the extended environment `(cons x env-values)` with continuation `k`. Stated otherwise: at run time, push datum `x` on the environment stack, run the code, and at the conclusion, resume computation at the place specified by the continuation `k` (a sort of "return address" in machine language jargon).

You may ask: what is the difference between the continuation `cont` and the continuation `k` in the code of `compile-lambda`? The first is the continuation when the `lambda`-expression is *evaluated*, producing a procedure; the second is the continuation when that procedure is *used.*

Dually, what happens when a function is applied to a single argument?

```
(define (compile-application fun arg env-names)
  (let ((fun-code (compile fun env-names))
        (arg-code (compile arg env-names)))
    (lambda (env-values cont)
      (fun-code env-values
                (lambda (f)
                  (arg-code env-values
                            (lambda (a)
                              (f a cont))))))))
```

Translation: run the compiled code `fun-code` for the function in the current run-time environment; a procedure `f` is then returned. Next, run the compiled code `arg-code` producing a value `a`. Recall `f` has *two* arguments: an input and a continuation, so use the original continuation `cont`, telling what to with the value returned by the function application.

Now to compile a sequence, the individual pieces are compiled separately, and then combined appropriately:

```
(define (compile-sequence sequence env-names)
  (let ((compiled-first
         (compile (car sequence) env-names)))
    (if (null? (cdr sequence))
        compiled-first
        (let ((compiled-rest
               (compile-sequence (cdr sequence) env-names)))
          (lambda (env-values cont)
            (compiled-first
              env-values
              (lambda (a) (compiled-rest env-values cont))))))))
```

Again, think about what happens at compile time and what happens at run time: at compile time, each of the expressions in the sequence is compiled separately and then integrated into a single piece of code. The compiled code that is returned, namely

```
          (lambda (env-values cont)
            (compiled-first
              env-values
              (lambda (a) (compiled-rest env-values cont))))
```

is executed at run time: run the first (compiled) expression in the run-time environment, and pass the value `a` returned to the continuation function. (Notice that this value is just thrown away—the input to the continuation is never used!)

Finally, we have replaced `define` with an equivalent syntactic form called `letrec`, denoting a “recursive” `let`-expression. For example, we can define the factorial function and compute 5! as:

```
(letrec ((fact
         (lambda (n) (if ((= n) 0) 1 ((* n) (fact ((- n) 1)))))))
  (fact 5))
```

How awful!—since all procedures have only *one* argument, primitive procedures with more than one argument have been *curried*, so that we write `((* 3) 5)` instead of `(* 3 5)`, for example. Question: why will such a definition not work if `letrec` was instead replaced by `let`?

The code works by building a new run-time environment with a dummy value `*UNDEFINED*` inserted where the recursively-defined object is supposed to go. At run time, when the definition is to be executed, the defined value is swapped for the dummy value. (Why? Think about how this environment structure then implements a recursive definition via a circle of pointers.)

```
(define (compile-letrec var val body env-names)
  (let ((new-env-names (cons var env-names)))
    (let ((val-code (compile val new-env-names))
          (body-code (compile body new-env-names)))
      (lambda (env-values cont)
        (let ((new-env-values (cons '*UNDEFINED* env-values)))
          (val-code new-env-values
                    (lambda (x)
                      (set-car! new-env-values x)
                      (body-code new-env-values cont))))))))
```

One problem with the above coding of `compile-letrec` is that it only allows one binding (definition)—how do we compile *mutual recursion* in the following style?

```
(letrec
  ((odd (lambda (n) (if (= n 0) 0 (even (- n 1)))))
   (even (lambda (n) (if (= n 0) 1 (odd (- n 1))))))
  (even 11)))
```

Elaborating the above coding of `compile-letrec`, we use instead

```
(define (compile-letrec vars vals body env-names)
  (let ((new-env-names (add-frame vars env-names)))
    (let ((val-codes (map (lambda (val) (compile val new-env-names))
                          vals))
          (body-code (compile body new-env-names)))
      (lambda (env-values cont)
        (let ((new-env-values
               (add-frame (map (lambda (v) '*UNDEFINED*) vals) env-values)))
          (bind-values val-codes (car new-env-values) new-env-values)
          (body-code new-env-values cont))))))
```

At this point, we have completed the "bare bones" of a Scheme compiler. Now we can add some funny hacks to modify control, similar to a "goto" or "jump" in an imperative language. We introduce two new constructs `enter` and `exit`, whose meanings are first given by example:

```
(enter ((+ 5) (exit 3)))
;Value: 3

((lambda (x) (lambda (y) ((* ((+ x) y)) (enter ((- x) (exit y)))))
   10) 2)
;Value: 24
```

In an expression `(enter` $\langle exp \rangle$`)`, the continuation ("what to do next with the returned value") is stored in the run-time environment. Calling `(exit` $\langle exp \rangle$`)` evaluates $\langle exp \rangle$ and passes it to the continuation that was stored by `enter`, rather than to the current continuation (which is thrown away).

In the first example, the continuation for `(enter ...)` is the *initial continuation* `(lambda (v) v)`. This continuation is thus passed the value 3, which is returned.

In the second, more complicated example, the continuation for `(enter ((- x) (exit y)))` is something like `(lambda (v) (* 12 v))`—recall $10 + 2 = 12$. Rather than perform the subtraction, `(exit y)` throws away the current continuation (question: what is it?), and passes the value of `y` to the earlier-stored continuation.

In compiling an `enter`-expression, the body is compiled in a larger environment with a new variable `*EXIT*`, bound at run time to the current continuation:

```
(define (compile-enter exp env-names)
  (let ((body-code (compile exp (cons '*EXIT* env-names))))
    (lambda (env-values cont)
      (body-code (cons cont env-values) cont))))
```

To compile an exit-expression, the body is compiled, and a lexical address for the earlier-stored continuation is generated. At run time, the compiled body is executed with the continuation stored at that address, and the current continuation cont is discarded:

```
(define (compile-exit exp env-names)
  (let ((body-code (compile exp env-names))
        (i (lookup-variable '*EXIT* env-names)))
    (lambda (env-values cont)
      (body-code env-values (list-ref env-values i)))))
```

Here are various relevant hacks for the environment. (Needed predicates and destructors to implement syntax can be found on-line—let's save paper!)

```
(define (unop op)
  (lambda (x k) (k (op x))))

(define (binop op)
  (lambda (x k) (k (lambda (y kk) (kk (op x y))))))

(define (initial-continuation v) v)

(define (extend var val bindings)
  (cons (cons var val) bindings))

(define initial-global-environment
  (extend 'cons (binop cons)
  (extend 'car (unop car)
  (extend 'cdr (unop cdr)
  (extend 'null? (unop null?)
  (extend 'pair? (unop pair?)
  (extend 'zero? (unop zero?)
  (extend 'true #t
  (extend 'false #f
  (extend '- (binop -)
  (extend  '* (binop *)
  (extend '+ (binop +)
  (extend '= (binop =)
  (extend '< (binop <)
  (extend '1+ (unop 1+) '() )))))))))))))))

(define initial-names (map car initial-global-environment))
(define initial-values (map cdr initial-global-environment))
```

If you want to try running your compiled code, the following little procedure comes in handy:

```
(define (try exp)
  ((compile exp initial-names) initial-values initial-continuation))
```

2 No More Mr. Nice Guy...

Now that the exposition is over, you have to do something. First, we want to modify the compiler so lambda-expressions can have any number of arguments.

Much of the work has already been done: see file meta-compiler.scm accessible from the home page. This is the code you are to modify. It is similar in spirit to the compiler we have discussed above, but ready for the modifications you are to implement.

Problem 1. In this fancier compiler, we represent environments as lists of frames, rather than just lists of binders and bindings. For example, a compile-time environment might look like this:

```
(define sample
 '((x y z)
   (bob carol ted alice)
   (s p q r)
   (cons car cdr null? pair? zero? true false - * + = < 1+)))
;Value: sample
```

Code a procedure `lookup-variable` that finds variable names in the compile-time environment. For example, calling the procedure with the environment above, searching for variable `ted`, should return the list `(1 2)`—frame 1, item 2. (Notice we start counting from zero!)

Dually, define a procedure `fetch` to be used at run-time: given a run-time environment `env-values` with the structure of the compile-time environment given above, `(fetch env-values (list 1 2))` should return the value bound to `ted`.

Here are examples of their use:

```
(lookup-variable 'ted sample)
;Value: (1 2)

(fetch sample (list 1 2))
;Value: ted
```

Problem 2. Code the procedures `compile-lambda` and `compile-application`. You may find the procedure `compile-arguments` (included in the code) useful, but feel free to do things another way.

Try your implementation on the examples `test1` and `test2` given in the code:

```
(define test1
'(letrec
   ((neg (lambda (x) (- 1 x)))
    (square (lambda (x) (* x x)))
    (fun (lambda (a b) (neg (+ (square a) (square b))))))
   (fun (fun 10 20) 30)))

(define test2
'(letrec
   ((odd (lambda (n) (if (= n 0) 0 (even (- n 1)))))
    (even (lambda (n) (if (= n 0) 1 (odd (- n 1))))))
   (even 11)))
```

Just turn in your code—no example testing needs to be handed in.

Problem 3. Next, we want to add a new feature to the source language called `call/cc`, similar to `enter` and `exit`. For example:

```
(call/cc (lambda (k) (+ 5 (k 3))))
;Value: 3

((lambda (x y) (* (+ x y) (call/cc (lambda (c) (- x (c y))))))
 10 2)
;Value: 24
```

Evaluating `(call/cc (lambda (v)` ⟨*exp*⟩`))` causes ⟨*exp*⟩ to be evaluated in a run-time environment where the current continuation—implemented as a function—is bound to variable `v`. Thus a subexpression `(v` ⟨*e*⟩`)` in ⟨*exp*⟩ will "jump out" and cause the value returned by evaluating ⟨*e*⟩ to be passed to the continuation `v`.

To implement `call/cc`, you need to change the definition of `compile` to recognize such expressions, with relevant predicates and destructors, and define a procedure

```
(define (compile-call/cc exp env-names) ...)
```

Pay special attention to how primitive procedures are stored in the run-time environment using `apply`—they are a good hint how the stored continuation should be represented.

Your implementation of `compile-call/cc` should compile its argument `(lambda (k) ...)` in the ordinary way—just as an everyday, vanilla `lambda`-expression.

Turn in a new version of `compile` along with a syntactic recognizer `call/cc?` and the related procedure `compile-call/cc`.

Problem 4. How can you get compiled code to "interact" with Scheme code you write in the underlying system? (*Think about this question*—don't turn in an answer!) For example, suppose we compile the factorial and exponential functions:

```
(define ff
  (try '(letrec ((fact
                  (lambda (n) (if (= n 0) 1 (* n (fact (- n 1))))))
          fact)))
;Value: ff

(define ee
  (try '(letrec ((expt
                  (lambda (b e) (if (= e 0) 1 (* b (expt b (- e 1))))))
          expt)))
;Value: ee
```

Define and turn in a procedure `link` that will work as follows:

```
((link ff) 5)
;Value: 120

((link ee) 3 4)
;Value: 81
```

You may find the following useful:

- The Scheme pretty-printer: try evaluating `(pp ff)` and `(pp ee)` to see what happens—this should give you some hints about the *functionality* of the compiled objects.
- The Scheme `apply` will apply a function to a list of arguments, for example `(apply + '(1 2 3 4 5))`. Here is a nice extension to the syntax of `lambda`: `(lambda w ...  )` will take an *arbitrary list* of arguments and bind it to `w`. For example, `((lambda w (apply + w)) 1 2 3 4 5)` evaluates to 15. This lets you define a procedure when you "don't know" how many arguments it has.

3 Code to be modified

Here is the Scheme code you are to modify in Problems 1 to 4 above.

```
;; Scheme-to-Scheme compiler
;; with lexical addressing, multiple arguments, letrec, call/cc, enter, exit

(define (compile exp env-names)
  (cond ((constant? exp)
         (compile-constant exp))
        ((variable? exp)
         (compile-variable exp env-names))
        ((letrec? exp)
         (compile-letrec (letrec-vars exp)
                         (letrec-vals exp)
                         (letrec-body exp)
                         env-names))
        ((lambda? exp)
         (compile-lambda (binders exp) (body exp) env-names))
        ((if? exp)
         (compile-if (predicate exp)
                     (then-part exp)
                     (else-part exp)
                     env-names))
        ((sequence? exp)
         (compile-sequence (cdr exp) env-names))
        ((enter? exp)
         (compile-enter (enter-body exp) env-names))
        ((exit? exp)
         (compile-exit (exit-body exp) env-names))
        (else ; it's an application!
         (compile-application (function-of exp)
                              (arguments-of exp)
                              env-names))))

(define (compile-constant c)
  (lambda (env-values cont)
    (cont c)))

(define (compile-variable v env-names)
  (let ((a (lookup-variable v env-names)))
    (lambda (env-values cont)
      (cont (fetch env-values a)))))
```

```
(define (compile-sequence sequence env-names)
  (let ((compiled-first
        (compile (car sequence) env-names)))
    (if (null? (cdr sequence))
        compiled-first
        (let ((compiled-rest
              (compile-sequence (cdr sequence) env-names)))
          (lambda (env-values cont)
            (compiled-first
               env-values
               (lambda (a) (compiled-rest env-values cont))))))))

(define (compile-if test then else env-names)
  (let ((test-code (compile test env-names))
        (then-code (compile then env-names))
        (else-code (compile else env-names)))
    (lambda (env-values cont)
      (test-code env-values
                 (lambda (p)
                   ((if p then-code else-code) env-values cont))))))

(define (compile-arguments args env-names)
  (if (null? args)
      (lambda (env-values cont)
         (cont '()))
      (let ((first-code (compile (car args) env-names))
            (rest-code (compile-arguments (cdr args) env-names)))
        (lambda (env-values cont)
          (first-code env-values
                      (lambda (first-value)
                        (rest-code env-values
                                   (lambda (rest-values)
                                     (cont (cons first-value
                                                 rest-values)))))))))))

(define (compile-application fun args env-names) ... )

(define (compile-lambda binders exp env-names) ... )
```

```
(define (compile-letrec vars vals body env-names)
  (let ((new-env-names (add-frame vars env-names)))
    (let ((val-codes (map (lambda (val) (compile val new-env-names))
                          vals))
          (body-code (compile body new-env-names)))
      (lambda (env-values cont)
        (let ((new-env-values
               (add-frame (map (lambda (v) '*UNDEFINED*) vals) env-values)))
          (bind-values val-codes (car new-env-values) new-env-values)
          (body-code new-env-values cont))))))

(define (bind-values compiled-bindings frame-values new-env-values)
  (if (null? frame-values)
      'done
      ((car compiled-bindings)
       new-env-values
       (lambda (b)
         (set-car! frame-values b)
         (bind-values (cdr compiled-bindings)
                      (cdr frame-values)
                      new-env-values)))))

(define (compile-enter exp env-names)
  (let ((body-code (compile exp (add-frame (list '*EXIT*) env-names))))
    (lambda (env-values cont)
      (body-code (add-frame (list cont) env-values) cont))))

(define (compile-exit exp env-names)
  (let ((body-code (compile exp env-names))
        (i (lookup-variable '*EXIT* env-names)))
    (lambda (env-values cont)
      (body-code env-values (fetch env-values i)))))

(define (compile-call/cc exp env-names) ...)

; Syntax stuff

(define (begins-with atom)
  (lambda (exp)
    (if (pair? exp) (equal? (car exp) atom) #f)))

(define constant? integer?)
(define (variable? v) (not (pair? v)))

(define letrec? (begins-with 'letrec))
(define (letrec-vars exp) (map car (cadr exp)))
(define (letrec-vals exp) (map cadr (cadr exp)))
(define letrec-body caddr)

(define lambda? (begins-with 'lambda))
(define binders cadr)
(define body caddr)

(define if? (begins-with 'if))
(define predicate cadr)
```

```
(define else-part cadddr)
(define then-part caddr)

(define sequence? (begins-with 'begin))

(define enter? (begins-with 'enter))
(define exit? (begins-with 'exit))
(define enter-body cadr)
(define exit-body cadr)

(define function-of car)
(define arguments-of cdr)


; Utilities

(define (add-frame frame env)
  (cons frame env))

(define (initial-continuation v) v)

(define (extend var val bindings)
  (cons (cons var val) bindings))

(define (prim-op op)
  (lambda (x k) (k (apply op x))))

(define initial-global-environment
  (extend 'cons (prim-op cons)
  (extend 'car (prim-op car)
  (extend 'cdr (prim-op cdr)
  (extend 'null? (prim-op null?)
  (extend 'pair? (prim-op pair?)
  (extend 'zero? (prim-op zero?)
  (extend 'true #t
  (extend 'false #f
  (extend '- (prim-op -)
  (extend  '* (prim-op *)
  (extend '+ (prim-op +)
  (extend '= (prim-op =)
  (extend '< (prim-op <)
  (extend '1+ (prim-op 1+) '() ))))))))))))))



(define (lookup-variable v env-names) ...)

(define (fetch env-values a) ...)

(define initial-names (cons (map car initial-global-environment) '()))
(define initial-values (cons (map cdr initial-global-environment) '()))

(define (try exp)
  ((compile exp initial-names) initial-values initial-continuation))
```

```
(define test1
'(letrec
   ((neg (lambda (x) (- 1 x)))
    (square (lambda (x) (* x x)))
    (fun (lambda (a b) (neg (+ (square a) (square b))))))
   (fun (fun 10 20) 30)))

(define test2
'(letrec
   ((neg (lambda (x) (- 1 x)))
    (odd (lambda (n) (if (= n 0) 0 (even (- n 1)))))
    (even (lambda (n) (if (= n 0) 1 (odd (- n 1))))))
   (even 11)))
```