The difference between a *interpreter* and a *compiler* is really two points on a *spectrum* of possible computations that occur either “now” (compile time) or “later” (run time). The mechanics of *variable lookup* is a case in point.

When the metacircular evaluator needs to find the value of a variable \( x \), it searches through the entire environment. Our Scheme evaluator uses *static binding*: every user-defined procedure has a *fixed* place where, no matter when and where the procedure is used, all frames (created by procedure application) are attached. It turns out that *without running the program*, we can figure out *where* in the environment every binding will occur, even if we cannot know what *value* will occur in that binding.

“Knowing where” can speed up variable lookup at runtime. If variable \( x \) occurs four frames up, fifth variable in the frame, we don’t need to search the frames in between: we march up the list of frames (using `cdr`), enter the frame, march over four bindings, and—*voilà*. Even if the environment is just one long list of bindings, lookup gets faster—we don’t need to check each binding.
For simplification, all procedure take exactly one argument—then the compile-time environment is just a list of variable names. (Why don’t we need frames?)

```
(compile '5 '())
;Value: 5

(compile 'x '(u v w x y))
;Value: (var 3)

(The variable x is three “steps” from the front of the list.)

(compile 'v '())
;Value: (var v)

(Since v doesn’t appear in the compile-time environment, we’ll leave it unchanged.)

(compile '(lambda (x) x) '(u v w x y))
;Value: (lambda (var 0))

(When the body of the procedure is evaluated, the compile-time environment will be x u v w x y.)

(compile '(lambda (x) (x w)) '(u v w x y))
;Value: (lambda ((var 0) (var 3)))

(compile '(lambda (x) (lambda (y) (x y))) '())
;Value: (lambda (lambda ((var 1) (var 0))))

(compile '(if (zero? n) 0 y) '(n y))
;Value: (if ((var zero?) (var 0)) 0 (var 1))

(compile '(enter ((+ 5) (exit z))) '(x y z u v w))
;Value: (enter (((var +) 5) (exit (var 3))))
(compile 'letrec ((inc (lambda (x) (1+ x))))
  (inc 10))
  '())
;Value: (letrec ((inc (lambda ((var 1+) (var 0))))
  ((var 0) 10))

(When the body of a letrec (a recursive let) is evaluated, the bindings are already in the environment.)

(compile 'letrec ((square (lambda (x) ((* x) x)))
  (decrease (lambda (y) ((- y) 1))))
  (decrease (square 5)))
  '))
;Value:
(letrec ((square (lambda (((var *) (var 0)) (var 0))))
  (decrease (lambda (((var -) (var 0)) 1))))
  ((var 1) ((var 0) 5)))
;No value

initial-names
;Value: (cons car cdr null? pair? zero? true false - * + = < 1+)

(compile 'letrec ((square (lambda (x) ((* x) x)))
  (decrease (lambda (y) ((- y) 1))))
  (decrease (square 5)))
  initial-names)
;Value:
(letrec ((square (lambda (((var 12) (var 0)) (var 0))))
  (decrease (lambda (((var 11) (var 0)) 1))))
  ((var 1) ((var 0) 5)))
;No value
(define (compile exp env-names)
  (cond ((constant? exp) exp)
        ((variable? exp)
         (list 'var (lookup-variable exp env-names)))
        ((letrec? exp)
         (compile-letrec (letrec-vars exp)
                         (letrec-vals exp)
                         (letrec-body exp)
                         env-names))
        ((lambda? exp)
         (list 'lambda
               (compile (body exp)
                         (append (binders exp) env-names)))
        ((if? exp)
         (cons 'if (compile-list (cdr exp) env-names)))
        ((sequence? exp)
         (cons 'begin (compile-list (cdr exp) env-names)))
        ((enter? exp)
         (cons 'enter (compile-list (cdr exp)
                               (cons '*EXIT* env-names))))
        ((exit? exp)
         (cons 'exit (compile-list (cdr exp) env-names)))
        (else ; it's an application!
         (compile-list exp env-names))))

(define (compile-list exp env-names)
  (map (lambda (e) (compile e env-names)) exp))
(define (compile-letrec vars vals body env-names)
  (let ((new-env-names (append vars env-names)))
    (let ((compiled-vals
           (map (lambda (val) (compile val new-env-names)) vals))
           (body-code (compile body new-env-names)))
      (list 'letrec
            (zip-together vars compiled-vals)
            body-code))))

(define (zip-together vars vals)
  (if (null? vars)
      ()
      (cons (cons (car vars) (list (car vals))
               (zip-together (cdr vars) (cdr vals)))))))

(define (lookup-variable v env-names)
  (if (null? env-names)
      v
      (if (eq? v (car env-names))
          0
          (let ((a (lookup-variable v (cdr env-names)))
                (if (number? a) (1+ a) a))))))

(define initial-names
  '(cons car cdr null? pair? zero? true false - * + = < 1+))
Notice the weird coding of `lookup-variable`—why?

```
(define (lookup-variable v env-names)
  (if (null? env-names)
      v
      (if (eq? v (car env-names))
          0
          (let ((a (lookup-variable v (cdr env-names))))
              (if (number? a) (1+ a) a)))))
```

Here is something better—but we need `enter` and `exit` in the underlying Scheme system!

```
(define (lookup-variable v env-names)
  (define (loop names)
    (if (null? names)
        (exit v)
        (if (eq? v (car names))
            0
            (1+ (loop (cdr names))))))
  (enter (loop env-names)))
```