Part 5

Control

When we think about the dining-out procedure discussed in the introduction to Part 1, we can begin to understand the power of abstracting control. Imagine that there is a genie photographing us while we dine. Here is a photo of us just about to order. Do you see the waiter standing by our table? Now, here is one of us polishing off dessert. The genie saves these photographs. When a meal has been particularly good and we long to go back to that little café in Paris whose name we have long since forgotten, there is one way we can relive the experience. We may ask the genie to rub a magic liquid on a photograph. When that happens, we escape to the same café where we were long ago. We will have the same waiter and perhaps order the same food. Whether the waiter aged or not, or whether we are heavier, will depend on whether changes have occurred. If not, then we are the same. If so, then some aspects may be the same, like the café, but other aspects may have changed. Perhaps the genie rubbed the wrong photograph, and instead of rubbing the photograph to get us to the café, he rubbed the photograph of us paying the waiter. What a shame, thrust back to that delicious café and not reliving the meal. What happens after the meal is over? You have two choices. You can stay in Paris and enjoy the night life, as you did long ago, or you can ask the genie to rub another photograph. Each time one is rubbed, you are escaping to another point in your past but with possible changes.

A computer is like a genie. While computing, it takes a snapshot of where you are in the computation. However, rather than keep every photograph around, it keeps only the ones that you tell it are worth saving. The photographs correspond to what are called escape procedures, and invoking an escape procedure corresponds to rubbing the photograph. The point of Part 5 is to show you how to reason with the power of escape procedures.

16 Introduction to Continuations

16.1 Overview

Did you ever lie in bed early in the morning and think about what you were going to do that day? Your thinking probably led to something like this: "I've got to shower, then brush my teeth, eat breakfast, find my way to campus, and get to my first class. After I get to my first class, I'll think about what I have left to do for the rest of the day." You packaged *the rest of the day* into a single concept, relative to some point in the morning. You did not consciously figure out what you would do with the rest of the day; you formed an abstraction of the rest of the day. This notion can carry over to computations as well. In Scheme *the rest of the computation* relative to some point in an evaluation can also be packaged in the same way that we packaged *the rest of the day* in our real-world experiences. The rest of a computation is a *continuation*. This chapter is an introduction to the use of continuations in Scheme. It shows what they are, how they work, and when to use them.

When we learn to deal with continuations, we shall be able to do all sorts of interesting things. For example, we shall be able to exit with a result from within a deep recursion. In addition, we shall be able to design *break packages* and *coroutines*, new concepts introduced in this and the next chapter.

In order to understand continuations, two new concepts—*contexts* and *escape procedures*—must be acquired. The first concept formalizes the creation of a procedure with respect to a subexpression of an expression. The second characterizes a procedure that upon invocation does not return to the point of its invocation. A continuation is a context that has been made into an escape procedure. Such continuations are created by invocations of `call-with-current-continuation`.

We have already encountered an escape procedure, **error**. When **error** gets invoked, its context, a procedure that represents the rest of the computation, is abandoned. Consider the very simple expression:

```
(cons (if (zero? divisor)
          (error "/:" dividend "divided by zero")
          (/ dividend divisor))
      '(a b c))
```

The result of invoking this expression is either an invocation of **error** or a list of length four, whose first element is a number. If **error** were a conventional procedure, then when it returned, we would do the cons and get a list of length four, whose first element would not likely be a number. But we know that that is not what happens, so **error** is not a conventional procedure. We describe how to construct such escape procedures in Section 16.3, but for now we observe that if **error** gets invoked, no consing occurs. In the next section we develop contexts, procedures that describe what does *not* happen when such escape procedures get invoked.

16.2 Contexts

A *context* is a procedure of one variable, □. We use the symbol □, pronounced "hole," to distinguish contexts from other procedures. If e is a subexpression of E, then we use the terminology that "the procedure c is a context of e in E." In the absence of side effects, the procedure c applied to the value of e is the value of E.

Consider the following expression that evaluates to `47`:

```
(+ 3 (* 4 (+ 5 6)))
```

The expression is evaluated using the following scheme. First, add `5` and `6` and get `11`. Next multiply `11` by `4`, yielding `44`, and then increase that result by `3`. Now, what is the context of `(+ 5 6)` in that expression? We must find a procedure that, if passed the value `11`, will produce `47`. There are lots of such procedures, but we will find one by using a simple two-step technique. In the first step we replace e, that is, `(+ 5 6)`, by □. In the second step, we form a procedure from the value of the result of the first step wrapped within `(lambda (□) ...)`. The context of `(+ 5 6)` in

```
(+ 3 (* 4 (+ 5 6)))
```

is the procedure, which is the value of:

```
(lambda (□)
  (+ 3 (* 4 □)))
```

Then applying this context to 11 results in 47.

Let's look at another example. What is the context of (* 3 4) in

```
(* (+ (* 3 4) 5) 2)
```

To form this context, we simply replace (* 3 4) by □ and then wrap what remains by (lambda (□) ...) leading to the procedure, which is the value of:

```
(lambda (□)
  (* (+ □ 5) 2))
```

Applying this context to 12 results in (* (+ 12 5) 2), which evaluates to 34. But we can apply it to other values. Applying it to 3 yields 16. What does applying it to 24 yield?

Let us next extend the mechanism for creating contexts. The second step remains the same, but the first step does more. Before, all we did in the first step was replace a subexpression by □. Now we extend the first step by evaluating the expression with the hole. When evaluation can no longer proceed because of the hole, we have finished the first step. Thus contexts are procedures created at the point in the computation where we can no longer compute because of the existence of □. The previous examples were correct because no evaluation was possible. To demonstrate this way to form contexts, consider the slightly more complicated expression:

```
(if (zero? 5)
    (+ 3 (* 4 (+ 5 6)))
    (* (+ (* 3 4) 5) 2))
```

In finding the context of (* 3 4), the result of the first step is what is left after evaluating

```
(if (zero? 5)
    (+ 3 (* 4 (+ 5 6)))
    (* (+ □ 5) 2))
```

(zero? 5) is false, so we choose the alternative of the if expression, which leads to (* (+ □ 5) 2). No more computation can take place. Thus, the procedure formed as a result of the second step is the value of

```
(lambda (□)
  (* (+ □ 5) 2))
```

Consider the context of (* 3 4) in:

```
(let ((n 1))
  (if (zero? n)
      (writeln (+ 3 (* 4 (+ 5 6))))
      (writeln (* (+ (* 3 4) 5) 2)))
  n)
```

The result of the first step is:

```
(begin
  (writeln (* (+ □ 5) 2))
  n)
```

The begin is needed because it is a sequence of expressions. We cannot do the addition because of the hole. We cannot do the multiplication because we cannot do the addition, we cannot do the displaying because we cannot do the multiplication, and we cannot return the value of n because we cannot determine the value of the expression that precedes it. In figuring out the value of expressions, we work from the inside and try to work outward. The procedure formed as the result of the second step is responsible for remembering the value of the free variable n. Thus we observe that contexts are procedures and must respect *free variables*. We do not need to worry about the let expression, and we do not need to worry about the if expression. Evaluation proceeds until the presence of □ makes it impossible to continue and then we do the second step that forms the context, which is the value of:

```
(lambda (□)
  (begin
    (writeln (* (+ □ 5) 2))
    n))
```

Applying it to 6 leads to (begin (writeln (* (+ 6 5) 2)) n), and with n bound to 1 the value displayed is 22 with the result 1. Applying it to 8, 26 is displayed.

The let expression is just a procedure invocation. We can reformulate the last example with a global procedure:

```
(define tester
  (lambda (n)
    (if (zero? n)
        (writeln (+ 3 (* 4 (+ 5 6))))
        (writeln (* (+ (* 3 4) 5) 2)))
    n))
```

Then we can determine the context of `(* 3 4)` in the expression `(tester 1)`. Although `(* 3 4)` does not appear physically within `(tester 1)`, we know that the computation will eventually get to that point, so the same context will be formed. If we were looking for the context within the expression `(* 10 (tester 1))`, then the context would be formed from the value of:

```
(lambda (□)
  (* 10 (begin
          (writeln (* (+ □ 5) 2))
          n)))
```

Let us apply these rules to a begin expression:

```
(begin
  (writeln 0)
  (let ((n 1))
    (if (zero? n)
        (writeln (+ 3 (* 4 (+ 5 6))))
        (writeln (* (+ (* 3 4) 5) 2)))
    n))
```

We are still forming the context of `(* 3 4)`. At the first step, `(* 3 4)` is replaced by □ just prior to evaluation:

```
(begin
  (writeln 0)
  (let ((n 1))
    (if (zero? n)
        (writeln (+ 3 (* 4 (+ 5 6))))
        (writeln (* (+ □ 5) 2)))
    n))
```

First, a 0 is displayed. Then the context is determined as the procedure, which is the value of:

```
(lambda (□)
  (begin
    (writeln (* (+ □ 5) 2))
    n))
```

Invoking it with 9 causes the displaying of 28 and then returns 1.

A context might involve the use of set!. The example below is similar to the last one, except that within the scope of the let expression is an assignment to the local variable n. The context of (* 3 4) in

```
(begin
  (writeln 0)
  (let ((n 1))
    (if (zero? n)
        (writeln (+ 3 (* 4 (+ 5 6))))
        (writeln (* (+ (* 3 4) 5) 2)))
    (set! n (+ n 2))
    n))
```

is the value of

```
(lambda (□)
  (begin
    (writeln (* (+ □ 5) 2))
    (set! n (+ n 2))
    n))
```

The free variable n, initially 1, is taken from the let expression. Each time the context is invoked, the variable n is incremented to the next positive odd integer, and what gets subsequently returned is also increased. If <*c*> is this context, then the first invocation of <*c*> assigns 3 to n, and the second invocation assigns 5 to n. From the way in which n changes upon each invocation of <*c*>, it follows that contexts are *procedures* that may even maintain state.

In the next example, we look at the terminating condition of a recursive procedure invocation. Consider the definition of the procedure map-add1, which adds one to each element of a list, but instead of returning the empty list, it returns (23) as the result of the terminating condition:

```
(define map-add1
 (lambda (ls)
  (if (null? ls)
    (cons (+ 3 (* 4 5)) '())
    (let ((val (add1 (car ls))))
     (cons val (map-add1 (cdr ls)))))))
```

For example, (map-add1 '(1 3 5)) is (2 4 6 23). What is the context of (* 4 5) in (cons 0 (map-add1 '(1 3 5)))? This is the same as "run this until the existence of □ stops the computation, and what is left is the context." We compute the expression looking for □:

```
(cons 0 (map-add1 '(1 3 5))) ⟹
(cons 0 (cons 2 (map-add1 (cdr '(1 3 5))))) ⟹
(cons 0 (cons 2 (map-add1 '(3 5)))) ⟹
(cons 0 (cons 2 (cons 4 (map-add1 '(5))))) ⟹
(cons 0 (cons 2 (cons 4 (cons 6 (map-add1 '()))))) ⟹
(cons 0 (cons 2 (cons 4 (cons 6 (cons (+ 3 □) '())))))
```

Because of the hole, no additional computation can be performed, so the context is the procedure formed from

```
(lambda (□)
 (cons 0 (cons 2 (cons 4 (cons 6 (cons (+ 3 □) '()))))))
```

If we invoke this context on 5, we create the list (0 2 4 6 8), and if we invoke it on 13, we get (0 2 4 6 16). What makes this a bit unusual is the fact that the hole does not show up in the expression right away, and in this case, it shows up just as the termination condition is considered.

In the next example, we cannot initially find a place to insert □. However, we know that □ will occur, so we can compute until it occurs and eventually stops the computation. Consider the simple procedure sum+n, which adds n to the sum of the numbers from 1 to n:

```
(define sum+n
 (lambda (n)
  (if (zero? n)
    0
    (+ (add1 n) (sum+n (sub1 n))))))
```

What is the context of (add1 n), just when n is 3, in (* 10 (sum+n 5))? As in the previous example, we are looking for a context associated with a recursive procedure invocation. However, this differs from the previous example by the additional detail used in its description. Stepping through the computation leads eventually to an occurrence of □:[1]

```
(* 10 (sum+n 5)) ⟹
(* 10 (if (zero? 5) 0 (+ (add1 5) (sum+n (sub1 5))))) ⟹
(* 10 (+ 6 (sum+n 4))) ⟹
(* 10 (+ 6 (if (zero? 4) 0 (+ (add1 4) (sum+n (sub1 4)))))) ⟹
(* 10 (+ 6 (+ 5 (sum+n 3)))) ⟹
(* 10 (+ 6 (+ 5 (if (zero 3) 0 (+ □ (sum+n (sub1 3))))))) ⟹
(* 10 (+ 6 (+ 5 (+ □ (sum+n 2)))))
```

Thus, the context is the procedure formed from:

```
(lambda (□)
  (* 10 (+ 6 (+ 5 (+ □ (sum+n 2))))))
```

The final example uses the predicate of an if expression. Consider the context of `(* 3 4)` in `(if (zero? (* 3 4)) 8 9)`. First, determining the expression prior to evaluation results in `(if (zero? □) 8 9)`. There is no evaluation possible, so the context is the value of

```
(lambda (□)
  (if (zero? □) 8 9)).
```

When this context is applied, its value will be 8 or 9, depending on what value gets bound to □.

In order to understand continuations, you will need to have lots of experience forming contexts. The exercises below should give you enough practice.

Exercises

Exercise 16.1

What is the context of `(cons 3 '())` in `(cons 1 (cons 2 (cons 3 '())))`? What results when we apply this context to `'(a b c)`, `'(x y)`, and `'(3)`?

[1] The trace that follows assumes a left to right order of evaluation of the operands to +. The procedure `map-add1` imposed a left to right order of evaluation of the operands to cons by using a let expression.

Exercise 16.2
For the following exercises assume these bindings: a is 1, b is 2, c is 3, d is 4, n is 5, x is 6, y is 7, and z is 8. Each answer will be in two parts. In the first part, describe the context of each expression; in the second part, determine the resultant values found by sequentially applying the context to each of 5, 6, and 7.

a. `(+ a b)` in `(* c (+ a b))`.

b. `x` in `(+ x y)`.

c. `y` in `(- x y)`.

d. `x` in `(let ((a 4)) (+ a x))`.

e. `(* c (+ a b))` in `(+ d (* c (+ a b)))`.

f. `(zero? n)` in `(if (zero? n) a b)`.

g. `x` in `(if x y z)`.

h. `a` in `(let ((x 3)) (set! x (+ a x)) x)`.

Exercise 16.3
For each expression below, determine the context of `(cons 3 '(4))` and the result of applying that context to `(1 2 3)`.

```
a. (letrec ((f (lambda (n)
                 (if (zero? n)
                     (car (cons 3 '(4)))
                     (* n (f (sub1 n)))))))
     (f 3))
```

```
b. (letrec ((f (lambda (n)
                 (if (zero? n)
                     (car (cons 3 '(4)))
                     (* n (f (sub1 n)))))))
     (+ 1000 (f 3)))
```

16.3 Escape Procedures

We now introduce a new procedure type, called *escape* procedures. An escape procedure upon invocation yields a value but never passes that value to others. When an escape procedure is invoked, its result *is* the result of the entire computation. Anything awaiting the result is ignored. Let us assume the existence of a procedure, escape-*, which is an escape multiply:

```
(+ (escape-* 5 2) 3)
```

This expression evaluates to 10. The waiting + is abandoned. It is as if (* 5 2) were the entire expression.

At this point we do not have a mechanism for creating escape procedures such as escape-*. Let us further assume there is a procedure escaper that takes any procedure as an argument and returns a similarly defined escape procedure. Then with escaper we can define escape-*

```
(define escape-* (escaper *))
```

and

```
(+ ((escaper *) 5 2) 3)
```

evaluates to 10.

Consider the invocation:

```
(+ ((escaper
      (lambda (x)
        (- (* x 3) 7)))
     5)
   4)
```

Here the addition cannot happen, so this is the same as

```
((lambda (x)
   (- (* x 3) 7))
 5)
```

so the answer is 8. Consider the following expression with an escape subtraction procedure:

```
(+ ((escaper
      (lambda (x)
        ((escaper -) (* x 3) 7)))
     5)
   4)
```

This is also 8, because once (escaper -) is invoked, the result is determined, and + is abandoned. But consider what happens with the following escape multiplication procedure:

```
(+ ((escaper
      (lambda (x)
        ((escaper -) ((escaper *) x 3)
                      7)))
    5)
   4)
```

The invocation of `(escaper *)` results in `15`. The `(escaper -)` is never invoked, so the subtraction never occurs. The following four expressions have the same value. Why?

```
1. ((lambda (x)
      (* x 3))
    5)
2. (+ ((escaper
         (lambda (x)
           (- ((escaper *) x 3)
              7)))
       5)
      4)
3. (+ ((lambda (x)
         ((escaper -) ((escaper *) x 3)
                      7))
       5)
      17)
4. (+ ((lambda (x)
         (- ((escaper *) x 3)
            7))
       5)
      2000)
```

Does this fully characterize the behavior of escape procedures? Not quite. Consider the following:

```
(/ (+ ((escaper
         (lambda (x)
           (- (* x 3) 7)))
       5)
      4)
   2)
```

The awaiting addition is abandoned. Is the division, which awaits the addition, also abandoned? Yes. Since the division awaits the addition and since the addition has been abandoned by the escape invocation, the division has also been abandoned. This behavior can be characterized by an equation: if *e* is an escape procedure and *f* is any procedure, then (**compose** *f* *e*) = *e*. That is, (*f* (*e* **expr**)) is the same as (*e* **expr**) for all expressions **expr**. The context of (*e* **expr**) in (*f* (*e* **expr**)) is (**lambda** (□) (*f* □)), which is the same as *f*. Since the result of (*f* (*e* **expr**)) is the result of (*e* **expr**), we say that an escape invocation *abandons its context*. In our last example, the context of the escape invocation included the awaiting addition and the awaiting division. We discuss special escape procedures in the next section where we characterize **call-with-current-continuation**.

Exercises

Exercise 16.4
Evaluate each of the following:

```
a. ((escaper add1) ((escaper sub1) 0))
b. (let ((es-cons (escaper cons)))
     (es-cons 1 (es-cons 2 (es-cons 3 '()))))
```

Exercise 16.5
Using the definition of **es-cons** from the previous exercise, determine the context of (**es-cons 3 '()**) in (**es-cons 1 (es-cons 2 (es-cons 3 '()))**).

Exercise 16.6: **reset**
Consider the definition of **reset**:

```
(define reset
  (lambda ()
    ((escaper
       (lambda ()
         (writeln "reset invoked"))))))
```

Determine the value of (**cons 1 (reset)**).

Exercise 16.7
Let *e* be an escape procedure, and let *f* and *g* be any procedures. To what is (**compose** *g* (**compose** *f* *e*)) equivalent? Can this be generalized to an arbitrary number of procedure compositions?

Exercise 16.8
Let *f* be any procedure. When can *f* be replaced by (**escaper** *f*) and still produce the same value as *f*?

16.4 Continuations from Contexts and Escape Procedures

We are about to discuss `call-with-current-continuation` (or `call/cc`). If `call/cc` is not available on your Scheme, define it as follows:

Program 16.1 call/cc

```
(define call/cc call-with-current-continuation)
```

`call/cc` is a procedure of one argument; we call the argument a *receiver*. The receiver is a procedure of one argument. Its argument is called a *continuation*. The continuation is also a procedure of one argument. Regardless of how we form the continuation, `(call/cc receiver)` is the same as `(receiver` *continuation*`)`. What is left is to understand how *continuation* is formed. To form *continuation*, we first form the context, c, of `(call/cc receiver)` in some expression E. We then invoke `(escaper` c`)`, which forms *continuation*. We have now completely characterized `call/cc`. All we have left to do is see how our understanding of how to form continuations leads us to determine correctly the evaluation of expressions using `call/cc`.

Consider the following expression:

```
(+ 3 (* 4 (call/cc r)))
```

The context of `(call/cc r)` is the procedure, which is the value of

```
(lambda (□) (+ 3 (* 4 □)))
```

so our original expression means the same as:

```
(+ 3 (* 4 (r (escaper (lambda (□) (+ 3 (* 4 □)))))))
```

That is, after the *system* forms the context of `(call/cc r)`, the *system* passes it as an escape procedure to `r`. Since this is now just a simple invocation, *all the rules for procedure invocation apply*. A little practice is helpful. Let us consider `r` to be the value of `(lambda (continuation) 6)`. What is the value of the expression derived from the call/cc expression above?

```
(+ 3 (* 4 ((lambda (continuation) 6)
           (escaper (lambda (□) (+ 3 (* 4 □)))))))
```

The value of

```
((lambda (continuation) 6)
 (escaper (lambda (□) (+ 3 (* 4 □)))))
```

is 6; it does not use `continuation`, so the result is 27 (i.e., $3 + 4*6$). What about this one?

```
(+ 3 (* 4 ((lambda (continuation) (continuation 6))
           (escaper (lambda (□) (+ 3 (* 4 □)))))))
```

The explicit invocation of `continuation` on 6 leads to

```
((escaper (lambda (□) (+ 3 (* 4 □)))) 6)
```

and then the result is 27. Is this one any different?

```
(+ 3 (* 4 ((lambda (continuation) (+ 2 (continuation 6)))
           (escaper (lambda (□) (+ 3 (* 4 □)))))))
```

The explicit invocation of `continuation` on 6 leads to

```
((escaper (lambda (□) (+ 3 (* 4 □)))) 6)
```

and then the result is 27. Remember, an escape invocation abandons its context, so `(lambda (□) (+ 3 (* 4 (+ 2 □))))` is abandoned. `continuation` has the value `(escaper (lambda (□) ... □ ...))`. Because the context of a `call/cc` invocation is turned into an escape procedure, we use the notation <*ep*> for procedures that get passed to `r`.

Scheme supports procedures as values, and since <*ep*> is a procedure, it is possible to invoke the same continuation more than once. In the next section there are three experiments with `call/cc`, and in the last experiment the same continuation is invoked twice. The `countdown` example of Chapter 17 shows what happens when the same continuation is invoked many times.

Exercises

Exercise 16.9

For each expression below, there are four parts. In Part a, determine the expression's value. In Part b, define `r` locally using `let`, and form the original application of `(call/cc r)`, which leads to this expression. In Part c, define `r` globally, and in Part d, using the global `r`, form the original application of `(call/cc r)`, which leads to this expression. The solution to problem `[1]` is given below:

```
[1] (- 3 (* 5 ((lambda (continuation) (continuation 5))
               (escaper (lambda (□) (- 3 (* 5 □)))))))
```

a. -22

```
b. (let ((r (lambda (continuation)
              (continuation 5))))
     (- 3 (* 5 (call/cc r))))
```

```
c. (define r
     (lambda (continuation)
       (continuation 5)))
```

```
d. (- 3 (* 5 (call/cc r)))
```

```
[2] (- 3 (* 5 ((lambda (continuation) 5)
               (escaper (lambda (□) (- 3 (* 5 □)))))))
```

```
[3] (- 3 (* 5 ((lambda (continuation) (+ 1000 (continuation 5)))
               (escaper (lambda (□) (- 3 (* 5 □)))))))
```

Exercise 16.10

If `r` is

```
(lambda (continuation) (continuation body))
```

in `(... (call/cc r) ...)`, why can `r` be rewritten as

```
(lambda (continuation) body)
```

Exercise 16.11
If `r` is

```
(escaper (lambda (continuation) (continuation body)))
```

in `(... (call/cc r) ...)`, when can `r` be rewritten as

```
(lambda (continuation) body)
```

16.5 Experimenting with call/cc

We next consider three simple experiments. Each experiment includes one use of a receiver (remember that a receiver is just a single-parameter procedure) without using `call/cc` and one that uses `call/cc`. The point of these experiments is to show the simple behavioral characteristics of `call/cc` expressions. Although the differences may seem minor in the first two experiments, their differences are important. In the last experiment, however, the differences demonstrate the unusual behavior of continuations. The receivers we use to demonstrate these properties are presented in Program 16.2.

Program 16.2 `receiver-1`, `receiver-2`, `receiver-3`

```
(define receiver-1
  (lambda (proc)
    (proc (list 1))))

(define receiver-2
  (lambda (proc)
    (proc (list (proc (list 2))))))

(define receiver-3
  (lambda (proc)
    (proc (list (proc (list 3 proc))))))
```

Each receiver consumes a procedure (possibly a continuation) that is invoked at least once. In `receiver-3`, not only is the procedure invoked at least once, but it is also used as an argument. We consider the behavior of each of these receivers using two global variables, `result` and `resultcc`, given

Program 16.3 `result`, `resultcc`

```
(define result "any value")

(define resultcc "any value")
```

Program 16.4 `writeln/return`, `answer-maker`, `call`

```
(define writeln/return
  (lambda (x)
    (writeln x)
    x))

(define answer-maker
  (lambda (x)
    (cons 'answer-is (writeln/return x))))

(define call
  (lambda (receiver)
    (receiver writeln/return)))
```

in Program 16.3, and three simple procedures, **`writeln/return`**, **`answer-maker`**, and **`call`**, given in Program 16.4. The procedure **`writeln/return`** displays and returns its argument. The procedure **`answer-maker`** is like **`writeln/return`**, but instead of returning its argument, it returns the consing of **`answer-is`** to its argument. Thus, **`(receiver-1 answer-maker)`** displays **`(1)`** and returns **`(answer-is 1)`**. The procedure **`call`** invokes its argument on **`writeln/return`**.

For reasons that are not yet clear but will be by the end of this section, we use **`set!`** to hold the results of each experiment. Recall that **`receiver-1`** is the value of

```
(lambda (proc)
  (proc (list 1)))
```

Experiment 1:

A.

```
[1] (set! result (answer-maker (call receiver-1)))
(1)
(1)
[2] result
(answer-is 1)
```

B.

```
[3] (set! resultcc (answer-maker (call/cc receiver-1)))
(1)
[4] resultcc
(answer-is 1)
```

These results are identical except that in Part A `writeln/return` is invoked in `call` so there is an additional `(1)`. The continuation formed in Part B is the value of:

```
(escaper
  (lambda (□)
    (set! resultcc (answer-maker □))))
```

Then this continuation is invoked on `(list 1)`, and since it is an escape procedure, that is all that happens. The procedure `answer-maker` is invoked on `(list 1)`, causing `(1)` to appear, and its result, `(answer-is 1)`, is assigned to `resultcc`. At `[4]` we verify that `resultcc` is indeed `(answer-is 1)`.

For Experiment 2, recall that `receiver-2` is the value of:

```
(lambda (proc)
  (proc (list (proc (list 2)))))
```

Experiment 2:

A.

```
[1] (set! result (answer-maker (call receiver-2)))
(2)
((2))
((2))
[2] result
(answer-is (2))
```

B.

```
[3] (set! resultcc (answer-maker (call/cc receiver-2)))
(2)
[4] resultcc
(answer-is 2)
```

In Part A the main difference is the extra set of parentheses around the value, which is the result passed to `answer-maker`. Both invocations of `proc` do a `writeln/return`. The first time is with `(2)` as its argument. When this returns, its argument is passed to `list`, resulting in `((2))`. Now we are ready for the second invocation of `writeln/return`. It displays its argument `((2))` and returns it to `answer-maker`, which displays its argument by invoking `writeln/return` and returns the result `(answer-is (2))`. This is the value assigned to `result`. In Part B, why is there just one displaying of `(2)`, and where did the extra set of parentheses go? Recall that the continuation built from the context of `(call/cc receiver-2)` is an escape procedure. Thus, once invoked, it abandons its context, the value of

```
(lambda (□)
  (set! resultcc (answer-maker (proc (list □)))))
```

The `list` invocation and the `proc` invocation waiting for the result of `list` are abandoned. The `list` invocation not occurring accounts for the missing set of parentheses, and the `proc` invocation not occurring accounts for why only one `(2)` is displayed. In Part B when `proc`, the continuation, is invoked, its argument is passed to the waiting `answer-maker`. The value `(2)` is displayed, and the result `(answer-is 2)` is sent to the waiting `set!`. The `set!` causes the value `(answer-is 2)` to be associated with `resultcc`. The result of the experiment is verified at `[4]`.

We have come to our last experiment. This one is slightly trickier than the earlier ones. Because of this, we discuss all of Part A before we look at Part B. We recall that `receiver-3` is the value of

```
(lambda (proc)
  (proc (list (proc (list 3 proc)))))
```

Experiment 3:[2]

---

[2] To denote the procedure that is the value of the variable procedure-name, we use the notation <*procedure-name*>.

A.

```
[1] (set! result (answer-maker (call receiver-3)))
(3 <writeln/return>)
((3 <writeln/return>))
((3 <writeln/return>))
[2] result
(answer-is (3 <writeln/return>))
[3] ((2nd (2nd result)) (list 1000))
(1000)
(1000)
[4] result
(answer-is (3 <writeln/return>))
```

The result of `(call receiver-3)` to be passed to **`answer-maker`** is

```
(<writeln/return>
  (list (<writeln/return>
          (list 3 <writeln/return>))))
```

First, the list `(3` *<writeln/return>*`)` is passed to *<writeln/return>*. It dutifully displays its argument. Then a set of parentheses is wrapped around it, and that result, `((3` *<writeln/return>*`))`, is displayed and passed to **`answer-maker`**. The procedure **`answer-maker`** displays that list and passes `(`**`answer-is`** `(3` *<writeln/return>*`))` to the waiting **`set!`**. The **`set!`** does the appropriate assignment. At `[2]` the experiment is verified. At `[3]` the procedure *<writeln/return>* is extracted using `(2nd (2nd result))`. That procedure is then invoked on `(1000)`. As expected *<writeln/return>* displays its argument `(1000)` and returns `(1000)`. At `[4]` nothing has changed **`result`**. Although this is a contrived experiment, only simple procedures are used to do simple things. We are now ready to consider Part B.

B.

```
[5] (set! resultcc (answer-maker (call/cc receiver-3)))
(3 <ep>)
[6] resultcc
(answer-is 3 <ep>)
[7] ((3rd resultcc) (list 1000))
(1000)
[8] resultcc
(answer-is 1000)
```

The result of `(call/cc receiver-3)` to be passed to **`answer-maker`** is

```
(<ep>
   (list (<ep>
           (list 3 <ep>))))
```

where *<ep>* is the continuation, which is the value of

```
(escaper
  (lambda (□)
    (set! resultcc (answer-maker □))))
```

but since *<ep>* is invoked, the outer list, *<ep>*, and answer-maker invocations are abandoned, as well as the set! expression. Therefore, the result of (call/cc receiver-3) is the result of invoking (*<ep>* (list 3 *<ep>*)). The escape procedure *<ep>* is invoked giving the value (3 *<ep>*) as the value that is passed to answer-maker, which displays the list (3 *<ep>*). Next answer-is is consed to the front of (3 *<ep>*), which yields (answer-is 3 *<ep>*). Then the set! is done, which changes the value of resultcc. At [6], we verify that what was expected has indeed occurred. We are about to execute the code at [7]. The expression (3rd resultcc) yields the escape procedure *<ep>* that was saved earlier. It is passed the list (1000). What is (*<ep>* (list 1000))? Recall that *<ep>* is an escape procedure that passes its argument to answer-maker and then assigns to resultcc the result of the answer-maker invocation. The procedure answer-maker displays its argument and then returns (answer-is 1000). The list (answer-is 1000) is for the waiting set! and so the set! happens again. This time resultcc gets the value (answer-is 1000), and the role of the escape procedure has ended. Was resultcc really changed? How do we find out? At [8], we check the value of resultcc. This time it has been changed to (answer-is 1000)! Although the set! was done back at [5], the escape procedure *<ep>* included doing everything again once it was invoked.

Exercises

Exercise 16.12
Rewrite answer-maker using call.

Exercise 16.13
Run the experiment with exer-receiver.

```
(define exer-receiver
  (lambda (proc)
    (list (proc (list 'exer proc)))))
```

Exercise 16.14

For each expression below, describe the binding that `continuation` gets, and give the value(s) of the expression. Each expression must be tested more than once. We include the solution for Part a.

```
a. (let ((r (lambda (continuation)
              (continuation 6))))
     (* (+ (call/cc r) 3) 8))
```

The value of `(escaper (lambda (□) (* (+ □ 3) 8)))`, 72.

```
b. (let ((r (lambda (continuation)
              (+ 1000 (continuation 6)))))
     (* (+ (call/cc r) 3) 8))
```

```
c. (let ((r (lambda (continuation)
              (+ 1000 6))))
     (* (+ (call/cc r) 3) 8))
```

```
d. (let ((r (lambda (continuation)
              (if (zero? (random 2))
                  (+ 1000 6)
                  (continuation 6)))))
     (* (+ (call/cc r) 3) 8))
```

```
e. (let ((r (lambda (continuation)
              (if (zero? (random 2))
                  (+ 1000 6)
                  (continuation 6)))))
     (+ (* (+ (call/cc r) 3) 8)
        (* (+ (call/cc r) 3) 8)))
```

```
f. (let ((r (lambda (continuation)
              (continuation
                (if (zero? (continuation (random 2)))
                    (+ 1000 6)
                    6)))))
     (+ (* (+ (call/cc r) 3) 8)
        (* (+ (call/cc r) 3) 8)))
```

Exercise 16.15

Determine the outcome of Experiment 3 with [1] and [5] replaced by the expressions below.

```
[1] (begin
      (set! result (answer-maker (call receiver-3)))
      'done)
```

```
[5] (begin
      (set! resultcc (answer-maker (call/cc receiver-3)))
      'done)
```

Exercise 16.16

We define a procedure **map-sub1** that takes a list of numbers and returns a list with each element of the list decremented by one. In addition to doing the work of **map-sub1**, it also sets the global variable **deep** to a continuation.

```
(define deep "any continuation")

(define map-sub1
  (lambda (ls)
    (if (null? ls)
        (let ((receiver (lambda (k)
                          (set! deep k)
                          '())))
          (call/cc receiver))
        (cons (sub1 (car ls)) (map-sub1 (cdr ls))))))
```

Consider the following experiment:

```
[1] (cons 1000 (map-sub1 '()))
(1000)
[2] (cons 2000 (deep '(a b c)))
?___________
[3] (cons 1000 (map-sub1 '(0)))
(1000 -1)
[4] (cons 2000 (deep '(a b c)))
?___________
[5] (cons 1000 (map-sub1 '(1 0)))
(1000 0 -1)
[6] (cons 2000 (deep '(a b c)))
?___________
[7] (cons 1000 (map-sub1 '(5 4 3 2 1 0)))
(1000 4 3 2 1 0 -1)
[8] (cons 2000 (deep '(a b c)))
?___________
```

After each invocation of **map-sub1**, **deep** is reset. The first continuation formed at [1] is:

```
(escaper
  (lambda (□)
    (cons 1000 □)))
```

At [3], the next continuation formed is:

```
(escaper
  (lambda (□)
    (cons 1000 (cons -1 □))))
```

The third continuation formed at [5] is:

```
(escaper
  (lambda (□)
    (cons 1000 (cons 0 (cons -1 □)))))
```

At [7], a fourth continuation is formed and bound to **deep**. Write an expression that characterizes that continuation, and then fill in the four blanks of the experiment.

16.6 Defining escaper

We now have all the tools we need to define **escaper**:

Program 16.5 **escaper**

```
(define *escape/thunk* "any continuation")

(define escaper
  (lambda (proc)
    (lambda (x)
      (*escape/thunk* (lambda () (proc x))))))
```

Although ***escape/thunk*** is defined as a global variable, it does not yet have the right value. To remedy this, one more experiment must be performed. For this experiment, a receiver is used to assign a value to ***escape/thunk***.

Program 16.6 `receiver-4`

```
(define receiver-4
  (lambda (continuation)
    (set! *escape/thunk* continuation)
    (*escape/thunk* (lambda () (writeln "escaper is defined")))))
```

We then have:

```
[1] ((call/cc receiver-4))
escaper is defined
[2] (*escape/thunk* (lambda () (add1 6)))
7
[3] (+ 5 (*escape/thunk* (lambda () (add1 6))))
7
```

At `[1]`, the continuation `(escaper (lambda (□) (□)))` is formed by the system. It becomes the value of `continuation` and, in turn, the value of `*escape/thunk*`, indirectly changing the definition of `escaper` in Program 16.5. This escape procedure takes as its argument a procedure of zero arguments and immediately invokes it. Next `*escape/thunk*` is passed the procedure

```
(lambda () (writeln "escaper is defined"))
```

This escapes while binding □ to

```
(lambda () (writeln "escaper is defined"))
```

Finally,

```
((lambda () (writeln "escaper is defined")))
```

displays `escaper is defined`. At `[2]`, invoking `*escape/thunk*` on

```
(lambda () (add1 6))
```

yields 7; at `[3]`, invoking it on

```
(lambda () (add1 6))
```

once again yields 7. Because `*escape/thunk*` is an escape procedure, the context

```
(lambda (□) (+ 5 □))
```

is abandoned. Earlier we hypothesized escaper's existence in order to explain the continuations formed from invocations of call/cc. Now we have defined escaper using call/cc, which *is* in Scheme. The procedure call/cc is not built with escaper, as we suggested earlier, but it behaves as though it were. On some systems, it may be necessary to determine the value of *escape/thunk* at the prompt by invoking ((call/cc receiver-4)).

Using *escape/thunk* we can redefine escaper so that it accepts procedures of any number of arguments:

Program 16.7 escaper

```
(define escaper
  (lambda (proc)
    (lambda args
      (*escape/thunk*
        (lambda ()
          (apply proc args))))))
```

This definition of escaper can be used to test all the results and exercises of this chapter.

Exercises

Exercise 16.17

Assume the existence of escaper and then define *escape/thunk* with escaper. You may not use call/cc.

Exercise 16.18

Determine the value of (/ 5 (*escape/thunk* (lambda () 0))).

Exercise 16.19: reset

Use call/cc to define a zero-argument procedure reset that upon invocation abandons its context and causes the string "reset invoked" to be displayed. In Chapter 7, when we defined error, we assumed the existence of reset. For example,

```
[1] (cons 1 (reset))
reset invoked
```

Exercise 16.20
Explain why (*escape/thunk* *escape/thunk*) causes an error.

Exercise 16.21
Determine the value of the following expressions:

```
[1] (let ((r (escaper
               (lambda (proc)
                 (cons 'c (proc (cons 'd '())))))))
      (cons 'a (cons 'b (call/cc r))))

[2] (let ((r (escaper
               (lambda (proc)
                 (cons 'c (cons 'd '()))))))
      (cons 'a (cons 'b (call/cc r))))
```

Exercise 16.22
Consider the procedure `new-escaper` below.

```
(define new-escaper "any procedure")
(let ((receiver (lambda (continuation)
                  (set! new-escaper
                    (lambda (proc)
                      (lambda args
                        (continuation
                          (lambda ()
                            (apply proc args))))))
                  (lambda () (writeln "new-escaper is defined")))))
  ((call/cc receiver))) displays new-escaper is defined
```

Are `new-escaper` and `escaper` the same? Why is `new-escaper` *better* than `escaper`?

16.7 Escaping from Infinite Loops

Suppose we would like to separate some code into control and action. To be a bit more specific, consider a piece of program that we want to run forever:

```
(let ((r (random n)))
  (if (= r target)
      (begin (writeln count) (set! count 0))
      (set! count (+ count 1))))
```

Program 16.8 `how-many-till`

```
(define how-many-till
  (lambda (n target)
    (let ((count 0))
      (cycle-proc
        (lambda ()
          (let ((r (random n)))
            (if (= r target)
                (begin (writeln count) (set! count 0))
                (set! count (+ count 1)))))))))
```

Then using `cycle-proc` (see Program 14.11), which runs a zero-argument procedure forever, we can write Program 16.8. The procedure `how-many-till` continuously reports how many values are unequal to the target. If the number displayed is always the same, then we ought to question the randomness of the random number generator. Each time it displays a count, the counter is reset. The only way to stop this program is by some kind of keyboard interrupt mechanism. However, we can build into `how-many-till` an exit facility using `call/cc`. Instead of looping indefinitely, we exit whenever the sum of the counts is greater than some threshold. We need an additional local variable that maintains the sum. We invoke the procedure `how-many-till` with the threshold as an additional argument. This version of `how-many-till` is given in Program 16.9. If `exit-above-threshold` is ever invoked, then we come out of the invocation of `(how-many-till n target thresh)`; otherwise we stay within `cycle-proc`. What is interesting about this example is that it is possible to exit an infinite loop without changing the definition of `cycle-proc`.

An example of the use of `how-many-till` is given in Program 16.10, where we can invoke `(random-data 10 20)`.

The first continuation formed (by the `call/cc` in `how-many-till`) is the value of

Program 16.9 `how-many-till`

```
(define how-many-till
  (lambda (n target thresh)
    (let ((receiver
            (lambda (exit-above-threshold)
              (let ((count 0) (sum 0))
                (cycle-proc
                  (lambda ()
                    (if (= (random n) target)
                        (begin
                          (writeln "target " target
                                   " required " count " trials")
                          (set! sum (+ sum count))
                          (set! count 0)
                          (if (> sum thresh)
                              (exit-above-threshold sum)))
                        (set! count (+ count 1)))))))))
      (call/cc receiver))))
```

Program 16.10 `random-data`

```
(define random-data
  (lambda (n thresh)
    (letrec ((loop (lambda (target)
                     (cond
                       ((negative? target) '())
                       (else (cons (how-many-till n target thresh)
                                   (loop (sub1 target))))))))
      (loop (sub1 n)))))
```

```
(escaper
  (lambda (□)
    (cons □ (loop (sub1 target)))))
```

where `loop` is as it is defined in `random-data` and `target` is 9.

Exercise

Exercise 16.23
Explain why the test for termination within **random-data** is **(negative? target)**.

16.8 Escaping from Flat Recursions

The **call/cc** operator gives the ability to escape from recursive computations while basically throwing out all the work that has stacked up. A simple example clarifies in what sense the mechanism avoids doing pending computations. We look at the problem of taking the product of a list of numbers and adding the number **n** to the product if the result is nonzero:

```
(product+ 5 '(3 6 2 7)) ⟹ (+ 5 252) ⟹ 257
(product+ 7 '(2 3 0 8)) ⟹ 0
```

Here is the solution in a functional style:

Program 16.11 **product+**

```
(define product+
  (lambda (n nums)
    (letrec
      ((product (lambda (nums)
                  (cond
                    ((null? nums) 1)
                    (else (* (car nums) (product (cdr nums))))))))
      (let ((prod (product nums)))
        (if (zero? prod) 0 (+ n prod))))))
```

This solution can be improved by adding a test to determine if one of the values in the list is zero. This stops the recursion upon encountering the first zero. This version is in Program 16.12. Consider the following subtle fact: *Finding a zero in the list does not stop the computation of* **product**. In fact, what happens is that if the first zero is in the kth position, then there are $k-1$ multiplications using zero. This is because the context of the **product**

Program 16.12 **product+**

```
(define product+
  (lambda (n nums)
    (letrec
      ((product (lambda (nums)
                  (cond
                    ((null? nums) 1)
                    ((zero? (car nums)) 0)
                    (else (* (car nums) (product (cdr nums))))))))
      (let ((prod (product nums)))
        (if (zero? prod) 0 (+ n prod))))))
```

invocations includes $k-1$ multiplications. When the zero is found, each of the $k-1$ waiting multiplications must still be done.

Is it possible to exit the invocation of `product` so that the result causes no waiting multiplications to occur? A solution is in Program 16.13. Consider the invocation `(+ 100 (product+ 10 '(2 3 4 0 6 7)))`. Since the list of numbers contains a `0`, the continuation, which is the value of

```
(escaper
  (lambda (□)
    (+ 100 □)))
```

is invoked, and the result is 100. This follows because the continuation is being invoked on `0`. If, however, no zero is found, then `(product nums)` terminates normally, and `(+ n prod)` is returned as the value of `(receiver <`*ep*`>)`. Since `prod` cannot be zero, the result returned is `(+ n prod)`. The let expression can be shortened to `(+ n (product nums))`. This version is in Program 16.14.

We see that finding a zero in the list produces a value to pass to the continuation formed from the invocation of `(call/cc receiver)` and finishes the computation of `product+`. Moreover, we observe the rather surprising fact that *if there is a zero in the list, then no multiplications occur regardless of where in the list that zero occurs.*

Program 16.13 product+

```
(define product+
  (lambda (n nums)
    (let ((receiver
            (lambda (exit-on-zero)
              (letrec
                ((product (lambda (nums)
                            (cond
                              ((null? nums) 1)
                              ((zero? (car nums)) (exit-on-zero 0))
                              (else (* (car nums)
                                       (product (cdr nums))))))))
                (let ((prod (product nums)))
                  (if (zero? prod) 0 (+ n prod))))))))
      (call/cc receiver))))
```

Program 16.14 product+

```
(define product+
  (lambda (n nums)
    (let ((receiver
            (lambda (exit-on-zero)
              (letrec
                ((product (lambda (nums)
                            (cond
                              ((null? nums) 1)
                              ((zero? (car nums)) (exit-on-zero 0))
                              (else (* (car nums)
                                       (product (cdr nums))))))))
                (+ n (product nums))))))
      (call/cc receiver))))
```

16.9 Escaping from Deep Recursions

Let us take a look at a slightly more complicated example. The problem is to redefine **product+** for a larger class of lists. Specifically, we allow deep lists of numbers. Thus we can invoke

```
(product+ 5 '((1 2) (1 1 (3 1 1)) (((((1 1 0) 1) 4) 1) 1)))
```

Program 16.15 `product+`

```
(define product+
  (lambda (n nums)
    (let ((receiver
            (lambda (exit-on-zero)
              (letrec
                ((product
                   (lambda (nums)
                     (cond
                       ((null? nums) 1)
                       ((number? (car nums))
                        (cond
                          ((zero? (car nums)) (exit-on-zero 0))
                          (else (* (car nums)
                                   (product (cdr nums))))))
                       (else (* (product (car nums))
                                (product (cdr nums))))))))
                (+ n (product nums))))))
      (call/cc receiver))))
```

Program 16.16 `*-and-count-maker`

```
(define *-and-count-maker
  (lambda ()
    (let ((local-counter 0))
      (lambda (n1 n2)
        (set! local-counter (+ local-counter 1))
        (writeln "Number of multiplications = " local-counter)
        (* n1 n2)))))
```

which results in 0. However, if the 0 had been a 3, then the result would have been 77. The new definition of **product+** is given in Program 16.15.

Some, but not all, multiplications are avoidable. By counting the number of multiplications, we can discover how many can be avoided. This can be done by invoking a special multiplication procedure ***-and-count-maker**, given in Program 16.16, and then passing the result of its invocation as an argument to **product+**. The procedure **product+** in a functional style would have once again introduced all those multiplications by zero. (See Program 16.17.) Thus we can invoke:

Program 16.17 `product+`

```
(define product+
  (lambda (n nums *-proc)
    (letrec
      ((product
         (lambda (nums)
           (cond
             ((null? nums) 1)
             ((number? (car nums))
              (cond
                ((zero? (car nums)) 0)
                (else (*-proc (car nums) (product (cdr nums))))))
             (else
               (let ((val (product (car nums))))
                 (cond
                   ((zero? val) 0)
                   (else (*-proc val (product (cdr nums))))))))))))
      (let ((prod (product nums)))
        (if (zero? prod) 0 (+ n prod))))))
```

```
(let ((counter (*-and-count-maker))
      (num-list '((1 2) (1 1 (3 1 1)) (((((1 1 0) 1) 4) 1) 1))))
  (product+ 5 num-list counter))
```

When product+ of Program 16.17 is used on the given tree, there are 12 multiplications, and when product+ of Program 16.15 is used there are fewer than 12 multiplications. There is, of course, a way to avoid all multiplications, but it involves walking through the entire list looking for 0's before starting the multiplication process. This makes the algorithm two-pass (it would require two passes through the list).

Exercises

Exercise 16.24

Run product+ of Program 16.14 with the *-proc argument over a list of numbers to verify the claim that no multiplications occur if the list contains a 0. Run product+ of Program 16.12 with the *-proc argument over the same list to compare with the first part of this exercise.

Exercise 16.25

Run **product+** of Program 16.15 with the `*-proc` argument over the nested list of numbers given above. Run **product+** of Program 16.17 with the `*-proc` argument over the same list to compare with the first part of this exercise.

Exercise 16.26

Rewrite **product+** of Program 16.15 where **n** is always `0`.

Exercise 16.27

Rewrite **product+** of Programs 16.14 and 16.15 using a local variable to maintain the accumulating product. Can this be done without using `call/cc`?

17 Using Continuations

17.1 Overview

In this chapter we discover some unusual properties of continuations. We demonstrate how to build a *break* facility. This allows computations to halt and then restart an indefinite number of times. Each time the computation halts, the user will be able to interact with the system. In addition, we show how to build a *coroutine* system. In such systems, multiple procedures can interact with each other without actually returning control from within each process. Before we begin this development, we review the fundamental rules concerning `call/cc`.

17.2 Review of call/cc

1. `call/cc`'s argument is called a *receiver*.
2. A receiver's argument is called a *continuation*. It is an escape procedure <*ep*> of one argument formed from the context of the call/cc invocation.
3.¡ A continuation's argument is passed to the context from which <*ep*> was formed by invoking <*ep*> on that value.
4. If the escape procedure <*ep*> is formed from the call/cc invocation and is then ignored, the following hold, where the use of ellipses surrounding an expression indicates that the expression may be embedded:

```
(let ((receiver (lambda (continuation) body)))
  ... (call/cc receiver) ...)
  =
(let ((receiver (lambda (continuation) body)))
  ... (receiver 'anything) ...)
  =
... body ...
```

and

```
...
  (let ((receiver (escaper (lambda (continuation) body))))
    ... (call/cc receiver) ...)
...
  =
...
  (let ((receiver (escaper (lambda (continuation) body))))
    ... (receiver 'anything) ...)
...
  = (receiver 'anything)
  = body
```

where the next to the last equality holds since `receiver` is an escape procedure, and the last equality holds since `continuation` is ignored.

5. In all circumstances the following hold:

```
(let ((receiver (lambda (continuation) (continuation body))))
  ... (call/cc receiver) ...)
  =
(let ((receiver (lambda (continuation) body)))
  ... (call/cc receiver) ...)
```

and

```
(let ((receiver (escaper (lambda (continuation) (continuation body)))))
  ... (call/cc receiver) ...)
  =
(let ((receiver (lambda (continuation) body)))
  ... (call/cc receiver) ...)
```

Program 17.1 `countdown`

```
(define countdown
  (lambda (n)
    (writeln "This only appears once")
    (let ((pair (message "Exit" (attempt (message "Enter" n)))))
      (let ((v (1st pair))
            (returner (2nd pair)))
        (writeln "    The non-negative-number: " v)
        (if (positive? v)
            (returner (list (sub1 v) returner))
            (writeln "Blastoff"))))))
```

17.3 Making Loops with One Continuation

In the previous chapter we introduced continuations. We noted that continuations were escape procedures and could be the value returned by any procedure or could be stored in data structures; however, our examples (except for the third experiment and **escaper**) ignored that feature. Each example shared the property that once a receiver was exited, the continuation was useless. Each receiver's continuation was always invoked; it was never passed as an argument or considered as the value of any procedure invocation. This property led us to refer to the continuations with such names as **exit-above-threshold** and **exit-on-zero**, because each was invoked only once for each invocation of its associated receiver. Now we abandon this property so that a continuation survives beyond giving a value to its associated receiver's invocation.

Earlier we used a continuation to exit deep recursions with the various definitions of **product+**. However, we have not yet developed an interesting use of a continuation, other than ***escape/thunk***, that can be returned as a value and stored in a data structure. To illustrate such a continuation, we define a procedure **countdown** that counts a positive integer down until it reaches zero. This is a very simple loop. We use two different definitions of the auxiliary procedure **attempt**. The first does not create any continuations and does not perform a loop. The second does create a single continuation and with this continuation is able to perform a loop. The definition of **countdown** uses a trivial displaying procedure **message** for tracking the flow of the computation. The definitions are given in Programs 17.1, 17.2, and 17.3.

The value of **proc** is just the identity procedure we denote as <*proc*>. Here is what appears when **(countdown 3)** is invoked:

Program 17.2 `message`

```
(define message
  (lambda (direction value)
    (writeln "   " direction "ing attempt with value: " value)
    value))
```

Program 17.3 `attempt`

```
(define attempt
  (lambda (n)
    (let ((receiver (lambda (proc) (list n proc))))
      (receiver (lambda (x) x)))))
```

```
This only appears once
   Entering attempt with value: 3
   Exiting attempt with value: (3 <proc>)
    The non-negative-number: 3
(2 <proc>)
```

"**`This only appears once`**" appears once. The next event is an attempt to find the value of the expression `(message "Enter" 3)`. This produces the message, "**`Entering attempt with value: 3`**" and **`message`** returns its second argument, 3. So now we attempt to find the value of the invocation `(attempt 3)`. This invocation yields the list `(3 <`*proc*`>)` because once the list `(list n proc)` is constructed, **`attempt`** is exited. Next we attempt to find the value of the expression `(message "Exit" (3 <`*proc*`>))`. Once again the message is displayed, but this time it is an exiting message, "**`Exiting attempt with value:`** `(3 <`*proc*`>)`." The invocation's value is `(3 <`*proc*`>)`. Now we bind **`pair`** to this list, take the pair apart, bind **`v`** to 3, and bind **`returner`** to <*proc*>. We display a message that acknowledges where we are and that we do indeed have the correct value. The message is, "**`The non-negative number:`** 3." We then check to see if the number is positive. In this case it is. We invoke `(returner (list (sub1 v) returner))`. We form the list `(2 <`*proc*`>)` and hand this list to <*proc*>, which returns `(2 <`*proc*`>)`. With the definition of `attempt` in Program 17.3, we did not create a loop nor did the result end with **`Blastoff`**.

We now redefine `attempt` (see Program 17.4) to create a continuation <*ep*> that we return in place of <*proc*>. In the discussion that follows, we explain

Program 17.4 `attempt`

```
(define attempt
  (lambda (n)
    (let ((receiver (lambda (proc) (list n proc))))
      (call/cc receiver))))
```

how that continuation is powerful enough to build a looping construct.

The result of `(countdown 3)` using `attempt` of Program 17.4 follows:

```
This only appears once
   Entering attempt with value: 3
   Exiting attempt with value: (3 <ep>)
    The non-negative-number: 3
   Exiting attempt with value: (2 <ep>)
    The non-negative-number: 2
   Exiting attempt with value: (1 <ep>)
    The non-negative-number: 1
   Exiting attempt with value: (0 <ep>)
    The non-negative-number: 0
Blastoff
```

"`This only appears once`" appears once. The next event is an attempt to find the value of the expression `(message "Enter" 3)`. This produces the message, "`Entering attempt with value: 3`" and `message` returns its second argument, `3`. So now we attempt to find the value of the invocation `(attempt 3)`. This invocation yields the list `(3 <`*ep*`>)` because once the list `(list n proc)` is constructed, `attempt` is exited. Next we attempt to find the value of the expression `(message "Exit" (3 <`*ep*`>))`. Once again the message is displayed, but this time it is an exiting message, "`Exiting attempt with value: (3 <`*ep*`>)`." The invocation's value is `(3 <`*ep*`>)`. Now we bind `pair` to this list, take the pair apart, bind `v` to `3`, and bind `returner` to `<`*ep*`>`. We display a message that acknowledges where we are and that we do indeed have the correct value. The message is, "`The non-negative number: 3`." We then check to see if the number is positive. In this case it is. We invoke `(returner (list (sub1 v) returner))`. We form the list `(2 <`*ep*`>)` and hand this list to `<`*ep*`>`.

To this point, everything has been the same as in the analysis of `attempt` of Program 17.3. In fact, all we did to write the above paragraph was change instances of `<`*proc*`>` to `<`*ep*`>`. Now we are doing something new. Instead of invoking `<`*proc*`>`, we are invoking `<`*ep*`>`. The continuation `<`*ep*`>` is the value of

```
(escaper
  (lambda (□)
    (let ((pair (message "Exit" □)))
      (let ((v (1st pair))
            (returner (2nd pair)))
        (writeln "    The non-negative-number: " v)
        (if (positive? v)
            (returner (list (sub1 v) returner))
            (writeln "Blastoff"))))))
```

This continuation is formed as the result of the first and only invocation of `attempt`. That is, the value passed as an argument to <*ep*> becomes the second argument to **`message`** in the let expression that binds **`pair`**. The next event is the displaying of the message, "**`Exiting attempt with value:`** `(2 <`*ep*`>)`." To go a bit further, the value of this message invocation is `(2 <`*ep*`>)`. We bind **`pair`** to this list, take the pair apart binding `v` to `2` and binding `returner` to the same <*ep*>. Once again we display a message that acknowledges where we are and that we do indeed have the correct value. The message is, "**`The non-negative number:`** 2." We then check to see if the number is still positive. In this case it is. We invoke

```
(returner (list (sub1 v) returner))
```

Clearly we are in a loop, with v replaced by `(sub1 v)`. The loop terminates when v is no longer positive. An important point is that `call/cc` is invoked only once. Therefore, we know for certain that <*ep*> is always the same continuation. The procedure `attempt` of Program 17.4 is invoked only once and its body is never reentered. This follows because the sentence "**`Entering attempt with value:`** *n*" appears only when *n* is 3.

Exercise

Exercise 17.1: `cycle-proc`
Rewrite `cycle-proc` using continuations instead of recursion as presented in Program 14.11.

17.4 Experimenting with Multiple Continuations

In this section we consider an experiment where we use more than one continuation. Everything until now has worked with just one continuation. Now we shall use several continuations. To keep track of the full meaning of each continuation, we shall plug in values for variables that will not change. This frees us from having to remember their values for use later.

In this experiment we need a receiver and a testing procedure. The receiver returns <*ep*>, which it receives as an argument. There are several continuations formed in this one example, so it is easy to get confused.

Program 17.5 `receiver`

```
(define receiver
  (lambda (continuation)
    (continuation continuation)))
```

Program 17.6 `tester`

```
(define tester
  (lambda (continuation)
    (writeln "beginning")
    (call/cc continuation)
    (writeln "middle")
    (call/cc continuation)
    (writeln "end")))
```

Experiment:

```
[1] (tester (call/cc receiver))
beginning
beginning
middle
beginning
end
[2]
```

The first event is to form <*ep*>, which, if it ever gets an argument, passes that argument to `tester`. <*ep*> is the value of:

```
(escaper
  (lambda (□)
    (tester □)))
```

We can think of <*ep*> as `(escaper tester)`. We invoke `(tester <ep>)`. Now `continuation` is bound to <*ep*>. We write `beginning`. We next invoke `(call/cc <ep>)`. This causes us to create <*epa*>. Before we figure out anything about what <*ep*> does with <*epa*>, we must understand what <*epa*> does if it ever gets invoked. <*epa*> is the value of:

```
(escaper
  (lambda (□)
    □
    (writeln "middle")
    (call/cc <ep>)
    (writeln "end")))
```

The continuation <*epa*> ignores its argument, □, displays `middle`, then invokes `(call/cc <ep>)`, and when that returns, it displays `end`. Now recall that <*ep*> takes its argument and invokes `(escaper tester)` on its argument, so `continuation` is bound to <*epa*>. We write `beginning`. We next invoke `(call/cc <epa>)`. This causes us to create <*epb*>. Before we figure out anything about what <*epa*> does with <*epb*>, we must understand what <*epb*> does if it ever gets invoked. <*epb*> is the value of:

```
(escaper
  (lambda (□)
    □
    (writeln "middle")
    (call/cc <epa>)
    (writeln "end")))
```

The continuation <*epb*> ignores its argument, displays `middle`, then invokes `(call/cc <epa>)`, and when that returns, it displays `end`. Now recall that <*epa*> takes its argument (ignores it) and displays `middle`, which we do now, and then invokes `(call/cc <ep>)`. Once again we form the new continuation <*epc*>, which is the value of:

```
(escaper
  (lambda (□)
    □
    (writeln "end")))
```

This continuation ignores its argument and displays end, so now we invoke ((escaper tester) <*epc*>). First, we display beginning. Next we invoke (call/cc <*epc*>). This causes the creation of the new continuation <*epd*>, which is the value of:

```
(escaper
  (lambda (□)
    □
    (writeln "middle")
    (call/cc <epc>)
    (writeln "end")))
```

This continuation displays middle, invokes (call/cc <*epc*>), and when that returns, it displays end. What is (<*epc*> <*epd*>)? The continuation <*epc*> is an escape procedure that ignores its argument and displays end. So we ignore <*epd*>, after having gone to all the trouble of constructing it, and display end.

Exercises

Exercise 17.2
During the experiment, how many more continuations were formed than were invoked?

Exercise 17.3
Determine what this expression represents:

```
(let ((receiver (lambda (continuation)
                  (call/cc continuation))))
  (call/cc receiver))
```

What is (call/cc call/cc)?

17.5 Escaping from and Returning to Deep Recursions

In product+ of Section 16.9, we demonstrated how to escape from deep recursions. Sometimes we want to escape from deep recursions but jump right back in when we so desire. In this section, we present a use of continuations that allows such behavior. We leave the deep recursion, but we give ourselves the ability to get right back where we were at the time we left. We assume

Program 17.7 `flatten-number-list`

```
(define flatten-number-list
  (lambda (s)
    (cond
      ((null? s) '())
      ((number? s) (list (break s)))
      (else
        (let ((flatcar
                (flatten-number-list (car s))))
          (append flatcar
                  (flatten-number-list (cdr s))))))))
```

Program 17.8 `break`

```
(define break
  (lambda (x)
    x))
```

Program 17.9 `break`

```
(define break
  (lambda (x)
    (let ((break-receiver
            (lambda (continuation)
              (continuation x))))
      (call/cc break-receiver))))
```

that the data for the example are the same as those of `product+`: a deep list of numbers. (See Section 16.9.)

Consider the definition of `flatten-number-list` in Program 17.7, where the first version of `break` is the identity procedure given in Program 17.8. Hence:

```
(flatten-number-list '((1 2 3) ((4 5)) (6))) ⟹ (1 2 3 4 5 6)
```

Another way to write `break`, which uses continuations but has the same meaning, is given in Program 17.9. This follows because we return as a value the argument to `break`. Since that value is `x`, we get the equivalent of `(lambda`

Program 17.10 `break`

```
(define get-back "any procedure")

(define break
  (lambda (x)
    (let ((break-receiver
            (lambda (continuation)
              (set! get-back (lambda () (continuation x)))
              (any-action x))))
      (call/cc break-receiver))))
```

Program 17.11 `any-action`

```
(define any-action
  (lambda (x)
    (writeln x)
    (get-back)))
```

Program 17.12 `any-action`

```
(define any-action
  (lambda (x)
    ((escaper (lambda () x))
    (get-back)))
```

`(x) x)`. But now we have access to `continuation`, and, moreover, we can characterize its behavior. Whenever `break` is invoked, we can think about the call as temporarily halting the computation; by invoking `continuation` on the same argument, we can continue the computation where it left off. We do not notice anything about the *pause* taking place because the `continuation` invocation happens immediately. But that is not required. For example, in Program 17.10, we display the value of the argument to `break`, using `any-action`, which is defined in Program 17.11. But since `any-action` is any action whatsoever, we may rewrite it as shown in Program 17.12 instead of explicitly writing the value of `x`.

Does the invocation of `(get-back)` in `any-action` of Program 17.12 ever happen? Because we are invoking an escape procedure prior to invoking

Program 17.13 break

```
(define get-back "any escape procedure")

(define break
  (lambda (x)
    (let ((break-receiver
            (lambda (continuation)
              (set! get-back continuation)
              (any-action x))))
      (call/cc break-receiver))))
```

(get-back), the answer is no. Is there a way to get back into the original computation? The answer is yes. Since get-back is bound globally, we can invoke it at the prompt. Below is an experiment using these tools.

```
[1] (flatten-number-list '((1 2) 3))
1
[2] (get-back)
2
[3] (get-back)
3
[4] (get-back)
(1 2 3)
```

The procedure break has a limitation. There is no control over what value is sent back. Unfortunately, that is determined by the definition of get-back. We can soften the definition by allowing get-back to accept an argument. Then get-back becomes

```
(lambda (v) (continuation v))
```

which is the same as continuation and gives us Program 17.13. We can still use any-action defined in Program 17.12 since the escaper invocation guarantees that (get-back) will never be invoked. Whenever get-back is invoked, it must be passed an argument.

Then the experiment could produce different results:

```
[1] (flatten-number-list '((1 2) 3))
1
[2] (get-back 4)
2
```

Program 17.14 `any-action`

```
(define break-argument "any value")

(define any-action
  (lambda (x)
    (set! break-argument x)
    ((escaper (lambda () x)))))
```

```
[3] (get-back 5)
3
[4] (get-back 6)
(4 5 6)
```

Why is the result `(4 5 6)` in this experiment, whereas it was `(1 2 3)` in the previous experiment? By returning the values `4`, `5`, and `6` to the `get-back` continuation, we are returning a different value each time. The computation was repeatedly suspended waiting for a value, which we supplied interactively at the prompt.

We might want to make public the value of the argument to `break`. We can do this in `any-action`, as shown in Program 17.14.

Finally, we note in Program 17.15 that `any-action` is not strictly necessary and can be included in the definition of `break-receiver`. The procedure `break` is an interesting program. It is very useful for interactive debugging. For example, by changing the argument to `break`, we can construct a mechanism for accessing and modifying part of the local state at the point of the invocation of `break`. This can be accomplished by passing to `break` procedures such as

`(lambda () x)` or `(lambda (v) (set! x v))`

In this case, if `x` is locally bound at the time of invocation of `break`, the list composed of these two procedures gives a lot of power to affect the internal state of a computation. Program 17.16 shows how `flatten-number-list` changes to support `break`. Whenever `break` occurs, `break-argument` gets bound to a two-element list and we define the extract and store procedures as shown in Programs 17.17 and 17.18.

This is just the tip of an iceberg. We are concerned only about one variable. This idea for debugging can be generalized to lists of arbitrarily many variables, but its utility diminishes as the number of variables increases. If

Program 17.15 break

```
(define get-back "any escape procedure")

(define break-argument "any value")

(define break
  (lambda (x)
    (let ((break-receiver
            (lambda (continuation)
              (set! get-back continuation)
              (set! break-argument x)
              ((escaper (lambda () x))))))
      (call/cc break-receiver))))
```

Program 17.16 flatten-number-list

```
(define flatten-number-list
  (lambda (s)
    (cond
      ((null? s) '())
      ((number? s) (list
                     (break
                       (list (lambda () s)
                             (lambda (v) (set! s v))))))
      (else
        (let ((flatcar
                (flatten-number-list (car s))))
          (append flatcar
                  (flatten-number-list (cdr s))))))))
```

Program 17.17 extract

```
(define extract
  (lambda ()
    ((1st break-argument))))
```

there are too many variables, it may be time to redesign the procedures. With break we have seen how there are many continuations coming from one pro-

Program 17.18 `store`

```
(define store
  (lambda (value)
    ((2nd break-argument) value)))
```

cedure invocation of **`flatten-number-list`**. Each of these continuations is eventually invoked after escaping to the prompt. The escape to the prompt is not very exciting. In the next section we allow far more interesting behavior to balance each continuation. Because the behavior of such uses of continuations is balanced, these continuations are called *co*routines.

Exercises

Exercise 17.4: **`flatten-number-list`**
Consider the new definition of **`flatten-number-list`** below. What changes are needed to make the sequence of invocations to **`get-back`** in the first experiment produce the same result? How about for the second experiment?

```
(define flatten-number-list
  (lambda (s)
    (letrec
      ((flatten
         (lambda (s)
           (cond
             ((null? s) '())
             ((number? s) (break (list s)))
             (else (let ((flatcar (flatten (car s))))
                     (append flatcar (flatten (cdr s)))))))))
      (flatten s))))
```

Exercise 17.5
Consider how we can repeat the results of the first experiment using **`flatten-number-list`** of Program 17.16. A condition imposed on this exercise is that no number may be input from [2] to the end of the experiment. *Hint*: Do not use **`store`**.

Exercise 17.6: **`product+`**
Consider **`product+`** below and define **`break-on-zero`**, which displays a `0` and escapes to the prompt. Each time it displays a `0`, resume the computation

as if the 0 had been a 1. This can be done by typing (get-back 1) at the prompt. If more than three zeros are found, then the result is "error: too many zeros." This is actually a form of exception handling where finding the 0 corresponds to an *exception* and finding the fourth 0 corresponds to an *error*. Experiment with different models of user interaction.

```
(define product+
  (lambda (n ls)
    (letrec ((product
               (lambda (ls)
                 (cond
                   ((null? ls) 1)
                   ((number? (car ls))
                    (* (if (zero? (car ls)) (break-on-zero) (car ls))
                       (product (cdr ls))))
                   (else (* (product (car ls))
                            (product (cdr ls))))))))
      (+ n (product ls)))))
```

Experiment with

```
(product+ 5 '((1 2) (3 4) (0 6) (7 0)))

(product+ 5 '((1 2) (0 3) (2 ((0 0 5) 0) 0)))
```

Exercise 17.7: break-var
A syntax table entry for break-var can be written so that:

(break-var *var*)
≡
(break (list (lambda () *var*) (lambda (v) (set! *var* v))))

Test flatten-number-list of Program 17.16 using break-var.
Bonus: This works for all variable names except one. Why is the variable name, for which it does not work, a bad choice?

Exercise 17.8
Consider the following experiment:

```
[1] (flatten-number-list '((1 2) 3))
1
[2] (get-back 4)
2
[3] (flatten-number-list '((5 6 7) 8))
5
```

In this experiment, (flatten-number-list '((1 2) 3)) never gets a value. Why? Generalize break to maintain a list (as a stack) of get-back continuations so that no information is lost. Then continue the experiment to get these results.

```
[4] (get-back 7)
6
[5] (get-back 8)
7
[6] (get-back 9)
8
[7] (get-back 10)
(7 8 9 10)
[8] (get-back 5)
3
[9] (get-back 6)
(4 5 6)
```

Exercise 17.9
Consider the results of the experiment from the previous exercise. How would the results differ if the list of continuations were treated like a queue instead of a stack?

17.6 Coroutines: Continuations in Action

There are lots of ways to package control information. We next look at a famous problem along with a well-known control mechanism. The problem is Grune's problem, and the control mechanism is called coroutines. Before we look at Grune's problem, we consider a simplified version of the use of coroutines. It is sometimes legitimate to imagine that several procedures are running at the same time, sending information among themselves. In this model, only one procedure is running at any given time. When information is sent from an active procedure to a dormant procedure, the active procedure becomes dormant, and the dormant procedure, the one receiving the information, becomes the active one.

One of the best examples for thinking about coroutines comes from game playing. Imagine a typical board game with three players. Each player is modeled by a coroutine, so there are three coroutines. Let us name these coroutines A, B, and C. Let us further assume that A plays first, hands the dice to B, B then plays and hands the dice to C, and then C plays and hands the dice back to A, and so on. In translating this game into a computer program, the code for A indicates a transfer of control by *resuming* B, and it indicates a transfer of the dice by passing them as an operand with the *resume* operation. This is accomplished by including in the code for A an instance of (resume B dice). Similarly, the code for B includes (resume C dice), and the code for C includes (resume A dice). The act of resuming means that the coroutine stops processing, and the entity that is the first argument to resume continues processing where it left off.

The board game's control flow is very regular. A plays, then B plays, then C plays, then A plays, and so on. As a result, not enough of the generality of coroutines can be seen through a board game simulation. If each player determined randomly which opposing player was to play next, this would require much of the generality of coroutines. Rather than using random numbers we simply picked an unnatural ordering that is illustrated in Program 17.19. Remember that nothing is displayed in a writeln expression until *all* of its operands have a value. Now if we invoke (A '*) we get the following output:

```
[1] (A '*)
This is A
              This is B
                            This is C
Came from C
Back in A
                            Came from A
                            Back in C
              Came from C
              Back in B
Came from B
```

Let us see what it takes to make these programs work. We need the procedure coroutine-maker, which takes a procedure as an argument. This argument is a procedure that obtains a meaning for resume and v when it is invoked. The variable v is of little concern. We focus on the variable resume. From these examples, we see that resume necessarily must look like a procedure of two arguments. When resume is invoked, it does not immediately return a value. In fact, it gives up control to whomever it is resuming and eventually gets an answer when someone else resumes it. (Since coroutines are first class, not only can they be passed as the required first argument to resume, but they can also be included in the second argument to resume.) Program 17.20 contains coroutine-maker.

The first thing that coroutine-maker does is create a local variable that will only hold continuations. Next, a procedure update-continuation! is

Program 17.19 Coroutines for a simple board game

```
(define A
  (let ((A-proc (lambda (resume v)
                  (writeln "This is A")
                  (writeln "Came from " (resume B "A"))
                  (writeln "Back in A")
                  (writeln "Came from " (resume C "A")))))
    (coroutine-maker A-proc)))

(define B
  (let ((B-proc (lambda (resume v)
                  (writeln (blanks 14) "This is B")
                  (writeln (blanks 14)
                           "Came from " (resume C "B"))
                  (writeln (blanks 14) "Back in B")
                  (writeln (blanks 14)
                           "Came from " (resume A "B")))))
    (coroutine-maker B-proc)))

(define C
  (let ((C-proc (lambda (resume v)
                  (writeln (blanks 28) "This is C")
                  (writeln (blanks 28)
                           "Came from " (resume A "C"))
                  (writeln (blanks 28) "Back in C")
                  (writeln (blanks 28)
                           "Came from " (resume B "C")))))
    (coroutine-maker C-proc)))
```

formed so that local side effects to **saved-continuation** can be done within other procedures. This is reminiscent of some of the techniques we presented in Chapter 12 when we showed how objects were built. The procedure **resumer**, having the properties of **resume** we discussed above, is next defined using **resume-maker**, whose code is given in Program 17.21. A boolean flag, **first-time**, is initially true. Then a procedure is returned. The first time this procedure is invoked, **(proc resumer value)** is evaluated. This is where the binding of **resume** and **v** in the programs above takes place. Subsequent invocations of this procedure invoke a continuation that was stored as a result of an earlier invocation of a **resume** to some other coroutine. Basically, the structure of **resumer** is

Program 17.20 `coroutine-maker`

```
(define coroutine-maker
  (lambda (proc)
    (let ((saved-continuation "any continuation"))
      (let ((update-continuation!
              (lambda (v)
                (set! saved-continuation v))))
        (let ((resumer (resume-maker update-continuation!))
              (first-time #t))
          (lambda (value)
            (if first-time
                (begin
                  (set! first-time #f)
                  (proc resumer value))
                (saved-continuation value))))))))
```

Program 17.21 `resume-maker`

```
(define resume-maker
  (lambda (update-proc!)
    (lambda (next-coroutine value)
      (let ((receiver (lambda (continuation)
                        (update-proc! continuation)
                        (next-coroutine value))))
        (call/cc receiver)))))
```

```
(lambda (next-coroutine value)
  (let ((receiver (lambda (continuation)
                    (<update-continuation!> continuation)
                    (next-coroutine value))))
    (call/cc receiver)))
```

Thus far the code has not shown us where the continuations are being created. In `coroutine-maker`, this is done in the procedure formed by invoking `(resume-maker update-continuation!)`.

When `resumer` is invoked with a coroutine, say `B`, and a value, say `"V"`, a continuation is bound to `continuation`. That continuation is stored in the `saved-continuation` associated with the code of the invoker of `resumer`. For example, if the code `(resume B "V")` is invoked from within `A`, then the updating takes place in the **saved-continuation** associated with coroutine **A**. When the updating is finished, the value "V" is sent to coroutine B. B then causes the invocation of the **saved-continuation**, which was stored as a result of an earlier invocation of its resumer.

Exercises

Exercise 17.10
To clarify the behavior of **coroutine-maker** and **resume-maker**, we used many variables. Very few are required. Furthermore, **resume-maker** itself is not necessary. Using this knowledge, rewrite **coroutine-maker** with as few variables as possible.

Exercise 17.11
Look at the results of the previous exercise. If there is a **first-time** flag, rewrite **coroutine-maker** so that it no longer requires such a variable.

Exercise 17.12
Study the definitions of **ping** and **pong** below:

```
(define ping
  (let ((ping-proc (lambda (resume v)
                     (display "ping-")
                     (resume pong 'ignored-ping))))
    (coroutine-maker ping-proc)))

(define pong
  (let ((pong-proc (lambda (resume v)
                     (display "pong")
                     (newline)
                     (resume ping 'ignored-pong))))
    (coroutine-maker pong-proc)))
```

What happens when we evaluate **(begin (ping '*) (pong '*))**?

17.7 Grune's Problem

Now we are ready to look at Grune's problem (Grune 1977). The problem is described as follows:

> We have a process A that copies symbols from input to output in such a way that where the input has *aa*, the output will have *b* instead. And we have a similar process B that converts *bb* into *c*. Now we want to connect these processes in series by feeding the output of A into B. Input with *aab* yields *c*, as does *baa*.

If we line the processes up as:

```
Input ⇄ A ⇄ B ⇄ Output
```

we can think of the flow of requests emanating at `Output`. *Requests for values* flow from right to left and *values*, themselves, flow from left to right. This is reminiscent of streams. The coroutine `Output` requests of `B` to find a symbol for `Output` to display. The coroutine `B` requests of `A` to find a symbol for `B` to consider in its analysis of a "possible *c*." The coroutine `A` requests of `Input` to find a symbol for `A` to consider in its analysis of a "possible *b*." Having made these requests, control now lies within `Input`. It does a read by first prompting the user. It responds by resuming `A` with that symbol. It does this with the following code, `(resume right (prompt-read "in> "))`, where `A` is bound to `right`. `A` is now in control. If the symbol is an *a*, `A` cannot pass it along to B because the next symbol read might be an *a*. The only possible alternative for `A` is to give control back to `Input`. Once again `Input` prompts for the next symbol. This symbol is also sent to `A`. Now `A` has enough information to send something to `B`. Here are the conditions under which information flows to the right. In these rules, x and y are the symbols in question, q is not the same as x, where x is *a* (respectively, *b*) and y is *b* (respectively, *c*):

1. x x $\Longrightarrow$ send y to the right.
2. x q $\Longrightarrow$ send x to the right, saving q for the next request.
3. q $\Longrightarrow$ send q to the right.

The code segment for this characterization follows:

```
(let ((symbol-1 (resume left 'ok)))
  (if (eq? symbol-1 x)
      (let ((symbol-2 (resume left 'more)))
        (if (eq? symbol-2 x)
            (resume right y)
            (begin
              (resume right symbol-1)
              (resume right symbol-2))))
      (resume right symbol-1)))
```

Program 17.22 `reader`

```
(define reader
  (lambda (right)
    (let ((co-proc (lambda (resume v)
                     (cycle-proc
                       (lambda ()
                         (resume right (prompt-read "in> ")))))))
      (coroutine-maker co-proc))))
```

In order to replace *aa* by *b*, **`x`** is *a*, **`y`** is *b*, **`left`** is `Input`, and **`right`** is `B`; in order to replace *bb* by *c*, **`x`** is *b*, **`y`** is *c*, **`left`** is `A`, and **`right`** is `Output`. Here is a description of the code segment. Get a symbol from **`left`**. If that symbol differs from **`x`**, send it along to **`right`**. If not, get another symbol from **`left`**. If that symbol is the same as the first, send **`y`** to **`right`**. If it differs, send both symbols, one at a time, to **`right`**.

The action of `Output` is simple. It makes a request from its left neighbor (i.e., `B`). If it finds a symbol matching **`end`**, it invokes an escape procedure, and the computation halts. If not, it writes the symbol. The code segment for this `Output` action is:

```
(let ((symbol (resume left 'ok)))
  (if (eq? symbol 'end)
      (escape-on-end symbol)
      (writeln "out> " symbol)))
```

The action of `Input` sends to its right neighbor whatever it read after first displaying a prompt:

```
(resume right (prompt-read "in> "))
```

Given that these are the basic actions, it is a relatively simple task to make sure all free variables have the correct values and that each code segment is run as a nonterminating loop with **`cycle-proc`**. The three procedures for forming the coroutines are given in Programs 17.22, 17.23, and 17.24.

We still have the task of building the wires into the communication channels. We are now going to use **`letrec`** to create the mutually recursive coroutines `Input`, `A`, `B`, and `Output`. One might expect the following letrec expression to work:

Program 17.23 `writer`

```
(define writer
  (lambda (left escape-on-end)
    (let ((co-proc (lambda (resume v)
                     (cycle-proc
                       (lambda ()
                         (let ((symbol (resume left 'ok)))
                           (if (eq? symbol 'end)
                               (escape-on-end 'end)
                               (writeln "out> " symbol))))))))
      (coroutine-maker co-proc))))
```

Program 17.24 `x->y`

```
(define x->y
  (lambda (x y left right)
    (let ((co-proc (lambda (resume v)
                     (cycle-proc
                       (lambda ()
                         (let ((symbol-1 (resume left 'ok)))
                           (if (eq? symbol-1 x)
                               (let ((symbol-2 (resume left 'more)))
                                 (if (eq? symbol-2 x)
                                     (resume right y)
                                     (begin
                                       (resume right symbol-1)
                                       (resume right symbol-2))))
                               (resume right symbol-1))))))))
      (coroutine-maker co-proc))))
```

```
(letrec
  ((Input (reader A))
   (A (x->y 'a 'b Input B))
   (B (x->y 'b 'c A Output))
   (Output (writer B escape-grune)))
  (Output 'ok))
```

Each of `Input`, `A`, `B`, and `Output` is built by invoking the procedures `reader`, `x->y`, `x->y`, and `writer`, respectively. The procedures `reader`, `x->y`, and

Program 17.25 grune

```
(define grune
  (lambda ()
    (let ((grune-receiver
            (lambda (escape-grune)
              (letrec
                ((Input (reader (lambda (v) (A v))))
                 (A (x->y 'a 'b (lambda (v) (Input v)) (lambda (v) (B v))))
                 (B (x->y 'b 'c (lambda (v) (A v)) (lambda (v) (Output v))))
                 (Output (writer (lambda (v) (B v)) escape-grune)))
                (Output 'ok)))))
      (call/cc grune-receiver))))
```

writer have been carefully designed to avoid invoking any of their coroutine arguments: **Input**, **A**, **B**, and **Output**. Here is the problem. All of the procedures are being created at the same time as they are being passed as arguments. For example, to create **Input**, we need **A**, and to create **A**, we need **Input**. To solve this problem, we must *freeze* the coroutines that are arguments in the right-hand sides of definitions. This has the effect of postponing the evaluation of the variables that refer to the coroutines. Unfortunately, if we freeze these variables, we get the wrong arity; that is, coroutines take one argument, but frozen objects (i.e., thunks) take no arguments. The code that follows, however, works:

```
(letrec
  ((Input (reader (lambda (v) (A v))))
   (A (x->y 'a 'b (lambda (v) (Input v)) (lambda (v) (B v))))
   (B (x->y 'b 'c (lambda (v) (A v)) (lambda (v) (Output v))))
   (Output (writer (lambda (v) (B v)) escape-grune)))
  (Output 'ok))
```

Program 17.25 shows the final definition of **grune** with all the necessary uses of `(lambda (v) (--- v))`. In the exercises, we develop a more natural way to think about this unusual behavior.

Exercises

Exercise 17.13: `wrap`
Consider the special form `wrap`, which has the following syntax table entry:

```
(wrap proc) ≡ (lambda args (apply proc args))
```

This works in all cases but one: when `args` is a free variable in the `proc` expression. Rewrite `wrap` using thunks to avoid this potential free variable capture.

Exercise 17.14
Using the results of the previous exercise, write the syntax table entry for `wrap` when `proc` is known to refer to a procedure of just one argument. This is the case for the coroutines used in `grune`. Is free variable capture still a problem?

Exercise 17.15
Using the results of the previous exercise, redefine `grune` using `wrap`.

Exercise 17.16: `safe-letrec`
Another way to implement `grune` is with a special form `safe-letrec`. This special form is like `letrec` except that each right-hand side variable is wrapped if it also appears as a left-hand side variable. Using the results of the previous exercise, create the syntax table entry for `safe-letrec` so that the following definition of `grune` works. (*Hint*: Use `let` to bind `proc` to `(wrap proc)` to avoid processing each right-hand side.)

```
(define grune
  (lambda ()
    (let ((grune-receiver (lambda (escape-grune)
                            (safe-letrec
                              ((Input (reader A))
                               (A (x->y 'a 'b Input B))
                               (B (x->y 'b 'c A Output))
                               (Output (writer B escape-grune)))
                              (Output 'ok)))))
      (call/cc grune-receiver))))
```

Exercise 17.17: `process-maker`

Sometimes processes are perceived as automatically being in an infinite loop. Use the following variation of `coroutine-maker`, called `process-maker`, and rewrite the solution to the Grune problem using processes.

```
(define process-maker
  (lambda (f)
    (let ((saved-continuation "any continuation"))
      (let ((update-continuation!
              (lambda (v)
                (set! saved-continuation v))))
        (let ((resumer (resume-maker update-continuation!))
              (first-time #t))
          (lambda (value)
            (if first-time
                (begin
                  (set! first-time #f)
                  (cycle-proc
                    (lambda ()
                      (f resumer value))))
                (saved-continuation value))))))))
```

Exercise 17.18

Using the results of the previous exercise, explain how `process-maker` differs from `coroutine-maker` by constructing an appropriate example.

Exercise 17.19

Redesign Towers of Hanoi using `coroutine-maker` so that each disk is a coroutine. Can `process-maker` be used?

Exercise 17.20

Redesign the solution of the Eight Queens problem using `coroutine-maker` so that each queen is a coroutine. Can `process-maker` be used?

Exercise 17.21

Implement Grune's problem using streams instead of coroutines.

Exercise 17.22

Extend `grune` to any number of x->y pairs. *Hint*: This can be accomplished by rewriting the procedure `grune` leaving everything else unchanged.

Exercise 17.23

Rework the previous exercise using streams.

17.8 Final Thoughts

We have not shown you all the interesting things you can think about with continuations, but we have tried to show you some of the ways that continuations can be used. Most of the time, you should be content to solve problems with conventional procedural techniques. Occasionally you will be tempted to use state changing operations like those we used when we worked with object-oriented programming. And even less frequently you will run across a need for continuations. This is your basic bag of tricks.

The existence of the computer has been incidental to the understanding of the concepts conveyed in this book. The computer's role has been much like that of a chemist's laboratory, used primarily for experimentation. What would happen if you added two parts hydrogen to one part oxygen? If you are curious about what happens when you compose two procedures, use the computer as your laboratory. What happens when you compose the procedure `(lambda (x) (+ x 1))` with the procedure `(lambda (x) (- x 1))`? This book has been about ideas and how we can combine separate categories of ideas to create procedures that do our computing. Although some emphasis has been placed on how fast the computer determines the value of a computation, we have tried to approach the ideas in this book more in terms of capturing the essence of a computation. Subtle issues of efficiency can come much later. We have challenged you at every turn. Each piece of the computational puzzle fits together and is described in terms of simple ideas. Under our guidance, you have entered the universe of computer science. It was our goal to cause you to look forward to future explorations into this fascinating field.

PROBLEMS

Problems worthy
of attack
prove their worth
by hitting back.

Piet Hein, *Grooks*