- creating a (potentially) infinite `while` loop that repeatedly invokes the `.next` method of `i` until the iterator raises the `StopIteration` exception.

There are many such patterns in modern programming languages.

14 Control Operations

The term *control* refers to any programming language instruction that causes evaluation to proceed, because it "controls" the program counter of the machine. In that sense, even a simple arithmetic expression should qualify as "control", and operations such as sequential program execution, or function calls and returns, most certainly do. However, in practice we use the term to refer primarily to those operations that cause *non-local* transfer of control, especially beyond that of mere functions and procedures, and the next step up, namely exceptions. We will study such operations in this chapter.

As we study the following control operators, it's worth remembering that even without them, we still have languages that are Turing-complete, and therefore have no more "power". Therefore, what control operators do is change and potentially improve the way we express our intent, and therefore enhance the structure of programs. Thus, it pays to being our study by focusing on program structure.

14.1 Control on the Web

Let us begin our study by examining the structure of Web programs. Consider the following program:

```
(display
  (+ (read-number "First number")
     (read-number "Second number")))
```

Henceforth, we'll call this our "addition server". You should, of course, understand this as a stand-in for more sophisticated applications. For instance, the two prompts might ask for starting and ending points for a trip, and in place of addition we might compute a route or compute airfares. There might even be computation between the two steps: e.g., after entering the first city, the airline might prompt us with choices of where it flies from there.

To test these ideas, here's an implementation of `read-number`:

```
(define (read-number [prompt : string]) : number
  (begin
    (display prompt)
    (let ([v (read)])
      (if (s-exp-number? v)
          (s-exp->number v)
          (read-number prompt)))))
```

When run at the console or in DrRacket, this program prompts us for one number, then another, and then displays their sum.

Now suppose we want to run this on a Web server. We immediately encounter a difficulty: the structure of server-side Web programs is such that they generate a single Web page—such as the one asking for the first number—and then *halt*. As a result, the *rest of the program*—which in this case prompts for the second number, then adds them, and then prints that result, is lost.

Do Now!

Why do Web servers behave in such a strange way?

There are at least two reasons for this behavior: one perhaps historical, and the other technical. The historical reason is that Web servers were initially designed to serve *pages*, i.e., static content. Any program that ran had to generate its output to a file, from which a server could offer it. Naturally, developers wondered why that same program couldn't run on demand. This made Web content *dynamic*. Terminating the program after generating a single piece of output was the simplest incremental step towards programs, not pages, on the Web.

The more important reason—and the one that has stayed with us—is technical. Imagine our addition server has generated its first prompt. Recall that there is considerable pending computation: the second prompt, the addition, and the display of the result. This computation must suspend waiting for the user's input. If there are millions of users, then millions of computations must be suspended, creating an enormous performance problem. Furthermore, suppose a user does not actually complete the computation—analogous to searching at an on-line bookstore or airline site, but not completing the purchase. How does the server know when or even whether to terminate the computation? And until it does, the resources associated with that computation remain in use.

Conceptually, therefore, the Web protocol was designed to be *stateless*: it would not store state on the server associated with intermediate computations. Instead, Web program developers would be forced to maintain all necessary state elsewhere, and each request would need to be able to resume the computation in full. In practice, the Web has not proven to be stateless at all, but it still hews largely in this direction, and studying the structure of such programs is very instructive.

Now consider client-side Web programs: those that run inside the browser, written in or compiled to JavaScript. Suppose such a computation needs to communicate with a server. The primitive for this is called `XMLHttpRequest`. The user makes an instance of this primitive and invokes its `send` method to send a message to the server. Communicating with a server is not, however, instantaneous (and indeed may never complete, depending on the state of the network). This leaves the sending process suspended.

The designers of JavaScript decided to make the language *single-threaded*: i.e., there would be only one thread of execution at a time. This avoids the various perils that arise from combining mutation with threads. As a result, however, the JavaScript process locks up awaiting the response, and nothing else can happen: e.g., other handlers on the page no longer respond.

Due to the structuring problems this causes, there are now various proposals to, in effect, add "safe" threads to JavaScript. The ideas described in this chapter can be viewed as an alternative that offer similar structuring benefits.

To avoid this problem, the design of `XMLHttpRequest` demands that the developer provide a procedure that responds to the request if and when it arrives. This callback procedure is registered with the system. It needs to embody the *rest of the processing* of that request. Thus, for entirely different reasons—not performance, but avoiding the problems of synchronization, non-atomicity, and deadlocks—the client-side Web has evolved to demand the same pattern of developers. Let us now better understand that pattern.

14.1.1 Program Decomposition into Now and Later

Let us consider what it takes to make our above program work in a stateless setting, such as on a Web server. First we have to determine the *first* interaction. This is the prompt for the first number, because Racket evaluates arguments from left to right. It is instructive to divide the program into two parts: what happens to generate the first interaction (which can all run now), and what needs to happen after it (which must be "remembered" somehow). The former is easy:

```
(read-number "First number")
```

We've already explained in prose what's left, but now it's time to write it *as a program*. It seems to be something like

We're intentionally ignoring `read-number` for now, but we'll return to it. For now, let's pretend it's built-in.

```
(display
  (+ <the result from the first interaction>
     (read-number "Second number")))
```

A Web server can't execute the above, however, because it evidently isn't a *program*. We instead need some way of writing this as one.

Let's observe a few characteristics of this computation:

- It needs to be a legitimate program.
- It needs to stay suspended until the request comes in.
- It needs a way—such as a parameter—to refer to the value from the first interaction.

Put together these characteristics and we have a clear representation—a *function*:

```
(lambda (v1)
  (display
    (+ v1
       (read-number "Second number"))))
```

14.1.2 A Partial Solution

On the Web, there is an additional wrinkle: each Web page with input elements needs to refer to a program stored on the Web, which will receive the data from the form and process it. This program is named in the `action` field of a form. Thus, imagine that the server generates a fresh label, stores the above function in a table associated with that label, and refers to the table in the `action` field. If and when the client actually submits the form, the server extracts the associated function, supplies it with the form's values, and thus resumes execution.

Do Now!

Is the solution above stateless?

Let's imagine that we have a custom Web server that maintains the above table. In such a server, we might have a special version of `read-number`—call it `read-number/suspend`—that records the rest of the program:

```
(read-number/suspend "First number"
                     (lambda (v1)
                       (display
                        (+ v1
                           (read-number "Second number")))))
```

To test this, let's implement such a procedure. First, we need a representation for labels; numbers are an easy substitute:

```
(define-type-alias label number)
```

Let's say `new-label` generates a fresh label on each invocation.

Exercise

Define `new-label`. You might use `new-loc` for inspiration.

We need a table to store the procedures representing the rest of the program.

```
(define table (make-hash empty))
```

Now we can store these procedures:

```
(define (read-number/suspend [prompt : string] rest)
  (let ([g (new-label)])
    (begin
      (hash-set! table g rest)
      (display prompt)
      (display " To enter it, use the action field label ")
      (display g))))
```

If we now run the above invocation of `read-number/suspend`, the system prints
`First number To enter it, use the action field label 1`
This is tantamount to printing the prompt in a Web page, and putting the label `1` in the `action` field. Because we're simulating it, we need something to represent the browser's submission process. This needs both the label (from the `action` field) and the value entered in the form. Given these two values, this procedure needs to extract the relevant procedure from the table, and apply it to the form value.

```
(define (resume [g : label] [n : number])
  ((some-v (hash-ref table g)) n))
```

With this, we can now simulate the act of entering `3` and clicking on a "Submit" button by running:
`> (resume 1 3)`

where 1 is the label and 3 is the user's input. Unfortunately, this simply produces another prompt, because we haven't fully converted the program. If we delete `read-number`, we're forced to convert the entire program:

```
(read-number/suspend "First number"
                     (lambda (v1)
                       (read-number/suspend "Second number"
                                            (lambda (v2)
                                              (display
                                               (+ v1 v2))))))
```

Just to be safe, we can also make sure the computation terminates after each output by adding an `error` invocation at the end of `read-number/suspend` (to truly ensure the most extreme form of "suspension").

When we execute this program, we have to use `resume` twice:

```
First number To enter it, use the action field label 1
halting: Program shut down
> (resume 1 3)
Second number To enter it, use the action field label 2
halting: Program shut down
> (resume 2 10)
13
```

where the two user inputs are 3 and 10, giving a total of 13, and the `halting` messages are generated by the `error` command we inserted.

We've purposely played a little coy with the types of the interesting parts of our program. Let's examine what these types should be. The second argument to `read-number/suspend` needs to be a procedure that consumes numbers and returns whatever the computation eventually produces: `(number -> 'a)`. Similarly, the return type of `resume` is the same `'a`. How do these `'a`s communicate with one another? This is done by `table`, which maps labels to `(number -> 'a)`. That is, at every step the computation makes progress towards the same outcome. `read-number/suspend` writes into this table, and `resume` reads from it.

14.1.3 Achieving Statelessness

We haven't actually achieved statelessness yet, because we have this large table residing on the server, with no clear means to remove entries from it. It would be better if we could avoid the server state entirely. This means we have to move the relevant state to the client.

There are actually two ways in which the server holds state. One is that we have reserved the right to create as many entries in the hash table as we wish, rather than a constant number (i.e., linear in the size of the program itself). The other is what we're storing in the table: honest-to-goodness closures, which might be holding on to an arbitrary amount of state. We'll see this more clearly soon.

Let's start by eliminating the closure. Instead, let's have each of the funtion arguments to be named, top-level functions (which immediately forces us to have only a fixed number of them, because the program's size cannot be unbounded):

```
(read-number/stateless "First number" prog1)

(define (prog1 v1)
  (read-number/stateless "Second number" prog2))

(define (prog2 v2)
  (display (+ v1 v2)))
```

Observe how each code block refers only to the *name* of the next, rather than to a real closure. The value of the argument comes from the form. There's just one problem: `v1` in `prog2` is a free identifier!

The way to fix this problem is, instead of creating a closure after one step, to send `v1` to the client to be stored there. Where do we store this? The browser offers two mechanisms for doing this: *cookies* and *hidden fields*. Which one do we use?

14.1.4 Interaction with State

The fundamental difference between cookies and hidden fields is that *all pages share the same cookie, but each page has its own hidden fields*.

First, let's consider a sequence of interactions with the existing program that uses `read-number/suspend` (at both interaction points). It looks like this:

```
First number To enter it, use the action field label 1
> (resume 1 3)
Second number To enter it, use the action field label 2
> (resume 2 10)
13
```

Thus, resuming with label 2 appears to represent adding 3 to the given argument (i.e., form field value). To be sure,

```
> (resume 2 15)
18
```

So far, so good. Now suppose we use label 1 again:

```
> (resume 1 5)
Second number To enter it, use the action field label 3
```

Observe that this new program execution needs to be resumed by using label 3, not 1. Indeed,

```
> (resume 3 10)
15
```

But we ought to ask, what happens if we reuse label 2?

Do Now!

Try `(resume 2 10)`.

Doing this is tantamount to resuming the old computation. We therefore expect it produce the same answer as before:

```
> (resume 2 10)
13
```

Now let's create a stateful implementation. We can simulate this by observing that each closure has its own environment, but all closures share the same mutable state. We can simulate this using our existing `read-number/suspend` by making sure we don't rely on the closure behavior of `lambda`, i.e., by not having any free identifiers in the body.

```
(define cookie '-100)

(read-number/suspend "First number"
                     (lambda (v1)
                       (begin
                         (set! cookie v1)
                         (read-number/suspend "Second number"
                                              (lambda (v2)
                                                (display
                                                 (+ cookie v2)))))))
```

Exercise

What do we *expect* for the same sequence as before?

Do Now!

What happens?

Initially, nothing seems different:

```
First number To enter it, use the action field label 1
> (resume 1 3)
Second number To enter it, use the action field label 2
> (resume 2 10)
13
```

When we reuse the initial computation, we indeed get a new resumption label:

```
> (resume 1 5)
Second number To enter it, use the action field label 3
```

which, when used, computes what we'd expect:

```
> (resume 3 10)
15
```

Now we come to the critical step:

```
> (resume 2 10)
15
```

It is unsurprising that the two resumptions of label 2 would produce different answers, given that they rely on mutable state. The reason it's problematic is because of what happens when we translate the same behavior to the Web.

Imagine visiting a hotel reservation Web site and searching for hotels in a city. In return, you are shown a list of hotels and the label 1. You explore one of them in a new tab or window; this produces information on that hotel, and label 2 to make the

reservation. You decide, however, to return to the hotel listing and explore another hotel in a fresh tab or window. This produces the second hotel's information, with label `3` to reserve at that hotel. You decide, however, to choose the first hotel, return to the first hotel's page, and choose its reservation button—i.e., submit label `2`. Which hotel did you expect to be booked into? Though you expected a reservation at the *first* hotel, on most travel sites, this will either reserve at the *second* hotel—i.e., the one you last viewed, but not the one on the page whose reservation button you clicked—or produce an error. This is because of the pervasive use of cookies on Web sites, a practice encouraged by most Web APIs.

14.2 Continuation-Passing Style

The functions we've been writing have a name. Though we've presented ideas in terms of the Web, we're relying on a much older idea: the functions are called *continuations*, and this style of programs is called *continuation-passing style* (CPS). This is worth studying in its own right, because it is the basis for studying a variety of other non-trivial control operations—such as generators.

We will take the liberty of using CPS as both a noun and verb: a particular structure of code and the process that converts code into it.

Earlier, we converted programs so that no Web input operation was nested inside another. The motivation was simple: when the program terminates, all nested computations are lost. A similar argument applies, in a more local sense, in the case of `XMLHttpRequest`: any computation depending on the result of a response from a Web server needs to reside in the callback associated with the request to the server.

In fact, we don't need to transform *every* expression. We only care about expressions that involve actual Web interaction. For example, if we computed a more complex mathematical expression than just addition, we wouldn't need to transform it. If, however, we had a function call, we'd either have to be absolutely certain the function didn't have any Web invocations either inside it, or in the functions in invokes, or the ones *they* invoke...or else, to be defensive, we should transform them all. Therefore, we have to transform every expression that we can't be sure performs no Web interactions.

The heart of our transformation is therefore to turn every one-argument function, `f`, into one with an extra argument. This extra argument is the continuation, which represents the rest of the computation. The continuation is itself a function of one argument. This argument takes the value that *would have been returned* by `f` and passes it to the rest of the computation. `f`, instead of *returning* a value, instead *passes* the value it would have returned to its continuation.

CPS is a general transformation, which we can apply to any program. Because it's a program transformation, we can think of it as a special kind of desugaring: in particular, instead of transforming programs from a larger language to a smaller one (as macros do), or from one language to entirely another (as compilers do), it transforms programs *within* the same language: from the full language to a more restricted version that obeys the pattern we've been discussing. As a result, we can reuse an evaluator for the full language to also evaluate programs in the CPS subset.

14.2.1 Implementation by Desugaring

Because we already have good support for desguaring, let's use to define the CPS transform. Concretely, we'll implement a CPS macro [REF]. To more cleanly separate the source language from the target, we'll use slightly different names for most of the language constructs: a one-armed `with` and `rec` instead of `let` and `letrec`; `lam` instead of `lambda`; `cnd` instead of `if`; `seq` for `begin`; and `set` for `set!`. We'll also give ourselves a sufficiently rich language to write some interesting programs!

<*cps-macro*> **::=**

```
(define-syntax (cps e)
  (syntax-case e (with rec lam cnd seq set quote display read-number)
    <cps-macro-with-case>
    <cps-macro-rec-case>
    <cps-macro-lam-case>
    <cps-macro-cnd-case>
    <cps-macro-display-case>
    <cps-macro-read-number-case>
    <cps-macro-seq-case>
    <cps-macro-set-case>
    <cps-macro-quote-case>
    <cps-macro-app-1-case>
    <cps-macro-app-2-case>
    <cps-macro-atomic-case>))
```

The presentation that follows orders the cases of the macro from what I believe are easiest to hardest. However, the code in the macro must avoid non-overlapping patterns, and hence follows a diffent order.

Our representation in CPS will be to turn *every* expression into a procedure of one argument, the continuation. The converted expression will eventually either supply a value to the continuation or will pass the continuation on to some other expression that will—by preserving this invariant inductively—supply it with a value. Thus, all output from CPS will look like `(lambda (k) ...)` (and we will rely on hygiene [REF] to keep all these introduced `k`'s from clashing with one another).

First let's dispatch with the easy case, which is atomic values. Though conceptually easiest, we have written this last because otherwise this pattern would shadow all the other cases. (Ideally, we should have written it first and provided a guard expression that precisely defines the syntactic cases we want to treat as atomic. We're playing loose here becuase our focus is on more interesting cases.) In the atomic case, we already have a value, so we simply need to supply it to the continuation:

<*cps-macro-atomic-case*> **::=**

```
[(_ atomic)
 #'(lambda (k)
     (k atomic))]
```

Similarly for quoted constants:

<*cps-macro-quote-case*> **::=**

```
[(_ 'e)
 #'(lambda (k) (k 'e))]
```

Also, we already know, from [REF] and [REF], that we can treat `with` and `rec` as macros, respectively:

<*cps-macro-with-case*> **::=**

```
[(_ (with (v e) b))
 #'(cps ((lam (v) b) e))]
```

<*cps-macro-rec-case*> **::=**

```
[(_ (rec (v f) b))
 #'(cps (with (v (lam (arg) (error 'dummy "nothing")))
              (seq
               (set v f)
               b)))]
```

Mutation is easy: we have to evaluate the new value, and then perform the actual update:

<*cps-macro-set-case*> **::=**

```
[(_ (set v e))
 #'(lambda (k)
     ((cps e) (lambda (ev)
                (k (set! v ev)))))]
```

Sequencing is also straightforward: we perform each operation in turn. Observe how this preserves the semantics of sequencing: not only does it obey the order of operations, the value of the first sub-term (`e1`) is not mentioned anywhere in the body of the second (`e2`), so the name given to the identifier holding its value is irrelevant.

<*cps-macro-seq-case*> **::=**

```
[(_ (seq e1 e2))
 #'(lambda (k)
     ((cps e1) (lambda (_)
                 ((cps e2) k))))]
```

When handling conditionals, we need to create a new continuation to remember that we are waiting for the test expression to evaluate. Once we have its value, however, we can dispatch on the result and return to the existing continuations:

<*cps-macro-cnd-case*> **::=**

```
[(_ (cnd tst thn els))
 #'(lambda (k)
     ((cps tst) (lambda (tstv)
                  (if tstv
                      ((cps thn) k)
                      ((cps els) k)))))]
```

When we get to applications, we have two cases to consider. We absolutely need to handle the treatment of procedures created in the language: those with one argument. For the purposes of writing example programs, however, it is useful to be able to employ primitives such as `+` and `*`. Thus, we will *assume for simplicity* that one-argument procedures are written by the user, and hence need conversion to CPS, while two-argument ones are primitives that will not perform any Web or other control operations and hence can be invoked directly; we will *also* assume that the primitive will be written in-line (i.e., the application position will not be a complex expression that can itself, say, perform a Web interaction).

For an application we have to evaluate both the function and argument expressions. Once we've obtained these, we are ready to apply the function. Therefore, it is tempting to write

<***cps-macro-app-1-case-take-1***> **::=**

```
[(_ (f a))
 #'(lambda (k)
     ((cps f) (lambda (fv)
                ((cps a) (lambda (av)
                           (k (fv av))))))))]
```

Do Now!

Do you see why this is wrong?

The problem is that, though the function is now a value, that value is a closure with a potentially complicated body: evaluating the body can, for example, result in further Web interactions, at which point the rest of the function's body, as well as the pending `(k ...)` (i.e., the rest of the program), will all be lost. To avoid this, we have to supply `k` to the function's value, and let the inductive invariant ensure that `k` will eventually be invoked with the value of applying `fv` to `av`:

<***cps-macro-app-1-case***> **::=**

```
[(_ (f a))
 #'(lambda (k)
     ((cps f) (lambda (fv)
                ((cps a) (lambda (av)
                           (fv av k))))))]
```

Treating the special case of built-in binary operations is easier:

<***cps-macro-app-2-case***> **::=**

```
[(_ (f a b))
 #'(lambda (k)
     ((cps a) (lambda (av)
                ((cps b) (lambda (bv)
                           (k (f av bv))))))))]
```

The very pattern we could not use for user-defined procedures we employ here, because we assume that the application of `f` will always return without any unusual transfers of control.

A function is itself a value, so it should be returned to the pending computation. The application case above, however, shows that we have to transform functions to take an extra argument, namely the continuation at the point of invocation. This leaves us with a quandary: which continuation do we supply to the body?

<*cps-macro-lam-case-take-1*> **::=**

```
[(_ (lam (a) b))
 (identifier? #'a)
 #'(lambda (k)
     (k (lambda (a dyn-k)
          ((cps b) ...))))]
```

That is, in place of `...`, which continuation do we supply: `k` or `dyn-k`?

Do Now!

Which continuation should we supply?

The former is the continuation *at the point of closure creation*. The latter is the continuation *at the point of closure invocation*. In other words, the former is "static" and the latter is "dynamic". In this case, we need to use the dynamic continuation, otherwise something very strange would happen: the program would return to the point where the closure was created, rather than where it is being used! This would result in seemingly very strange program behavior, so we wish to avoid it. Observe that we are consciously choosing the dynamic continuation just as, where scope was concerned, we chose the static environment.

<*cps-macro-lam-case*> **::=**

```
[(_ (lam (a) b))
 (identifier? #'a)
 #'(lambda (k)
     (k (lambda (a dyn-k)
          ((cps b) dyn-k))))]
```

Finally, for the purpose of modeling Web programming, we can add our input and output procedures. Output follows the application pattern we've already seen:

<*cps-macro-display-case*> **::=**

```
[(_ (display output))
 #'(lambda (k)
     ((cps output) (lambda (ov)
                     (k (display ov)))))]
```

Finally, for input, we can use the pre-existing `read-number/suspend`, but this time *generate* its uses rather than force the programmer to construct them:

<*cps-macro-read-number-case*> **::=**

```
[(_ (read-number prompt))
 #'(lambda (k)
     ((cps prompt) (lambda (pv)
                     (read-number/suspend pv k))))]
```

Notice that the continuation bound to `k` is precisely the continuation that we need to stash at the point of a Web interaction.

Testing any code converted to CPS is slightly annoying because all CPS terms expect a continuation. The initial continuation is one that simply either (a) consumes a value and returns it, or (b) consumes a value and prints it, or (c) consumes a value, prints it, and gets ready for another computation (as the prompt in the DrRacket Interactions window does). All three of these are effectively just the identity function in various guises. Thus, the following definition is helpful for testing:

```
(define (run c) (c identity))
```

For instance,

```
(test (run (cps 3))                           3)
(test (run (cps ((lam ()    5)       )))      5)
(test (run (cps ((lam (x)   (* x x)) 5)))     25)
(test (run (cps (+ 5 ((lam (x) (* x x)) 5)))) 30)
```

We can also test our old Web program:

```
(run (cps (display (+ (read-number "First")
                      (read-number "Second")))))
```

Lest you get lost in the myriad of code, let me highlight the important lesson here: *We've recovered our code structure*. That is, we can write the program in *direct style*, with properly nested expressions, and a compiler—in this case, the CPS converter—takes care of making it work with a suitable underlying API. This is what good programming languages ought to do!

14.2.2 Converting the Example

Let's consider the example above and see what it converts to. You can either do this by hand, or take the easy way out and employ the Macro Stepper of DrRacket. Assuming we include the application to `identity` contained in `run`, we get:

For now, you need to put the code in `#lang racket` to get the full force of the Macro Stepper.

```
(lambda (k)
  ((lambda (k)
     ((lambda (k)
        ((lambda (k)
           (k "First")) (lambda (pv)
                          (read-number/suspend pv k))))
      (lambda (lv)
        ((lambda (k)
```

```
          ((lambda (k)
             (k "Second")) (lambda (pv)
                             (read-number/suspend pv k))))
        (lambda (rv)
          (k (+ lv rv)))))))
  (lambda (ov)
    (k (display ov)))))
```

What! This isn't at all the version we wrote by hand!

In fact, this program is full of so-called *administrative* `lambda`s that were introduced by the particular CPS algorithm we used. Fear not! If we stepwise apply each of these `lambda`s and substitute, however—

Designing better CPS algorithms, that eliminate needless administrative `lambda`s, is therefore an ongoing and open research question.

Do Now!

Do it!

—the program reduces to

```
(read-number/suspend "First"
                     (lambda (lv)
                       (read-number/suspend "Second"
                                            (lambda (rv)
                                              (identity
                                               (display (+ lv rv)))))))
```

which is precisely what we wanted.

14.2.3 Implementation in the Core

Now that we've seen how CPS can be implemented through desguaring, we should ask whether it can be put in the core instead.

Recall that we've said that CPS applies to all programs. We have one program we are especially interested in: the interpreter. Sure enough, we can apply the CPS transformation to it, making available what are effectively the same continuations.

First, we'll find it convenient to use a procedural representation of closures [REF]. We'll have the interpreter take an extra argument, which consumes values (those given to the continuation) and eventually returns them:

<***cps-interp***> **::=**

```
(define (interp/k [expr : ExprC] [env : Env] [k : (Value -> Value)]) : Value
  <cps-interp-body>)
```

In the easy cases, instead of returning a value we need to simply pass it to the continuation argument:

<***cps-interp-body***> **::=**

```
(type-case ExprC expr
  [numC (n) (k (numV n))]
```

```
  [idC (n) (k (lookup n env))]
  <cps-interp-plusC-case>
  <cps-interp-appC-case>
  <cps-interp-lamC-case>)
```

(Note that `multC` is handled entirely analogous to `plusC`.)

Let's start with the easy case, `plusC`. First we interpret the left sub-expression. The continuation for this evaluation interprets the right sub-expression. The continuation for that adds the result. What should happen to the result of addition? In `interp`, it was returned to whichever computation caused the `plusC` to be interpreted. Now, remember, we no longer return values; instead we pass them to the continuation:

<***cps-interp-plusC-case***> **::=**

```
[plusC (l r) (interp/k l env
                       (lambda (lv)
                         (interp/k r env
                                   (lambda (rv)
                                     (k (num+ lv rv))))))]
```

Exercise

Implement the code for `multC`.

This leaves the two difficult, and related, pieces.

In an application, we again have to interpret the two sub-expressions, and then apply the resulting closure to the argument. But we've already agreed that every application needs a continuation argument. Therefore, we have to update our definition of a value:

```
(define-type Value
  [numV (n : number)]
  [closV (f : (Value (Value -> Value) -> Value))])
```

Now we have to decide what continuation to pass. In an application, it's the continuation given to the interpreter:

<***cps-interp-appC-case***> **::=**

```
[appC (f a) (interp/k f env
                      (lambda (fv)
                        (interp/k a env
                                  (lambda (av)
                                    ((closV-f fv) av k)))))]
```

Finally, the `lamC` case. We have to create a `closV` using a `lambda`, as before. However, this procedure needs to take two arguments: the actual value of the argument, and the continuation of the application. The critical question is, what is this latter value?

We have essentially two choices. `k` represents the *static* continuation: the one active at the point of closure *construction*. However, what we want is the continuation at the point of closure *invocation*: the *dynamic* continuation.

<***cps-interp-lamC-case***> **::=**

```
[lamC (a b) (k (closV (lambda (arg-val dyn-k)
                        (interp/k b
                                  (extend-env (bind a arg-val)
                                              env)
                                  dyn-k))))]
```

To test this revised interpreter, we need to invoke `interp/k` with some kind of initial continuation value. This needs to be a procedure that represents nothing remaining in the computation. A natural representation for this is the identity function:

```
(define (interp [expr : ExprC]) : Value
  (interp/k expr mt-env
            (lambda (ans)
              ans)))
```

To signify that this is strictly a top-level interface to `interp/k`, we've dropped the environment parameter and pass the empty environment automatically. If we want to be especially sure we haven't accidentally used this procedure recursively, we could insert a call to `error` at its end to prevent it from returning and its return value being used.

14.3 Generators

Many programming languages now have a notion of *generators*. A generator is like a procedure, in that one can invoke it in an application. Whereas a regular procedure always begins execution at the beginning, a generator *resumes* from where it last left off. Of course, that means a generator needs a notion of "exiting before it's done". This is known as *yielding*, namely returning control to whatever called it.

14.3.1 Design Variations

There are many variations between generators. The points of variation, predictably, have to do with how to enter and exit a generator:

- In some languages a generator is an object that is instantiated like any other object, and its execution is resumed by invoking a method (such as `next` in Python). In others it is just like a procedure, and indeed it is re-entered by applying it like a function.
- In some languages the yielding operation—such as Python's `yield`—is available only inside the syntactic body of the generator. In others, such as Racket, `yield` is an applicable value bound in the body, but by virtue of being a value, it can be passed to abstractions, stored in data structures, and so on.

In languages where values in addition to regular procedures can be used in an application, all such values are collectively called *applicables*.

Python's design represents an extreme point in that a generator is simply *any function that contains the keyword* `yield` *in its body*. In addition, Python's `yield` cannot be passed as a parameter to another function that performs the yielding on behalf of the generator.

There is also a small issue of naming. In many languages with generators, the yielder is *automatically* called word `yield`: either as a keyword (as in Python) or as an identifier bound to an applicable value (as in Racket). Another possibility is that the user of the generator must indicate in the generator expression what name to give the yielder. That is, a use might look like

Curiously, Python expects users to determine what to call `self` or `this` in objects, but it does not provide the same flexibility for `yield`, because it has no other way to determine which functions are generators!

```
(generator (yield) (from)
           (rec (f (lam (n)
                     (seq
                       (yield n)
                       (f (+ n 1)))))
             (f from)))
```

but it might equivalently be

```
(generator (y) (from)
           (rec (f (lam (n)
                     (seq
                       (y n)
                       (f (+ n 1)))))
             (f from)))
```

and if the yielder is an actual value, a user can also abstract over yielding:

```
(generator (y) (from)
           (rec (f (lam (n)
                     (seq
                       ((yield-helper y) n)
                       (f (+ n 1)))))
             (f from)))
```

where `yield-helper` will presumably perform the actual yielding.

There are actually two more design decisions:

1. Is `yield` a statement or expression? In many languages it is actually an expression, meaning it has a value: the one supplied when resuming the generator. This makes the generator more flexible because the user of a generator can use the parameter(s) to alter the generator's behavior, rather than being *forced* to use state to communicate desired changes.

2. What happens at the end of the generator's execution? In many languages, a generator raises an exception to signal its completion.

14.3.2 Implementing Generators

To implement generators, it will be especially useful to employ our CPS macro language. Let's first decide where we stand regarding the above design decisions. We will use the applicative representation of generators: that is, asking for the next value from the generator is done by applying it to any necessary arguments. Similarly, the yielder will also be an applicable value and will in turn be an expression. Though we have already seen how macros can automatically capture a name [REF], let's make the yielder's name explicit to keep the macro simpler. Finally, we'll raise an error when the generator is done executing.

How do generators work? To yield, a generator must

- remember where in its execution it currently is, and
- know where in its caller it should return to.

while, when invoked, it should

- remember where in its execution its caller currently is, and
- know where in its body it should return to.

Observe the duality between invocation and yielding.

As you might guess, these "where"s correspond to continuations.

Let's build up the generator rule of the `cps` macro incrementally. First a header pattern:

<***cps-macro-generator-case***> **::=**

```
[(_ (generator (yield) (v) b))
 (and (identifier? #'v) (identifier? #'yield))
 <generator-body>]
```

The beginning of the body is easy: all code in CPS needs to consume a continuation, and because a generator is a value, this value should be supplied to the continuation:

<***generator-body***> **::=**

```
#'(lambda (k)
    (k <generator-value>))
```

Now we're ready to tackle the heart of the generator.

Recall that a generator is an applicable value. That means it can occur in an application position, and must therefore have the same "interface" as a procedure: a procedure of two arguments, the first a value and the second the continuation at the point of application. What should this procedure do? We've described just this above. First the generator must remember where the caller is in its execution, which is precisely the continuation at the point of application; "remember" here most simply means "must be stored in state". Then, the generator should return to where it previously was, i.e., its *own* continuation, which must clearly have been stored. Therefore the core of the applicable value is:

<***generator-core***> **::=**

```
(lambda (v dyn-k)
  (begin
    (set! where-to-go dyn-k)
    (resumer v)))
```

Here, `where-to-go` records the continuation of the caller, to resume it upon yielding; `resumer` is the local continuation of the generator. Let's think about what their initial values must be:

- `where-to-go` has no initial value (because the generator has yet to be invoked), so it needs to throw an error if ever used. Fortunately this error will never occur, because `where-to-go` is mutated on the first entry into the generator, so the error is just a safeguard against bugs in the implementation.
- Initially, the rest of the generator is the whole generator, so `resumer` should be bound to the (CPS of) `b`. What is its continuation? This is the continuation of the entire generator, i.e., what to do when the generator finishes. We've agreed that this should also signal an error (except in this case the error truly can occur, in case the generator is asked to produce more values than it's equipped to).

We still need to bind `yield`. It is, as we've pointed out, symmetric to generator resumption: save the local continuation in `resumer` and return by applying `where-to-go`.

Putting together these pieces, we get:

<*generator-value*> **::=**

```
(let ([where-to-go (lambda (v) (error 'where-to-go "nothing"))])
  (letrec([resumer (lambda (v)
                     ((cps b) (lambda (k)
                                (error 'generator "fell through"))))]
          [yield (lambda (v gen-k)
                   (begin
                     (set! resumer gen-k)
                     (where-to-go v)))])
    <generator-core>))
```

Do Now!

Why this pattern of `let` and `letrec` instead of `let`?

Observe the dependencies between these code fragments. `where-to-go` doesn't depend on either of `resumer` or `yield`. `yield` clearly depends on both `where-to-go` and `resumer`. But why are `resumer` and `yield` mutually referential?

Do Now!

Try the alternative!

The subtle dependency you may be missing is that `resumer` contains `b`, the body of the generator, which may contain references to `yield`. Therefore, it needs to be closed over the binding of the yielder.

Exercise

How do generators differ from coroutines and threads? Implement coroutines and threads using a similar strategy.

14.4 Continuations and Stacks

Surprising as it may seem, CPS conversion actually provides tremendous insight into the nature of the program execution *stack*. The first thing to understand is that every continuation is actually *the stack itself*. This might seem odd, given that stacks are low-level machine primitives while continuations are seemingly complex procedures. But what is the stack, really?

- It's a record of what remains to be done in the computation. So is the continuation.
- It's traditionally thought of as a list of stack *frames*. That is, each frame has a reference to the frames remaining after it finishes. Similarly, each continuation is a small procedure that refers to—and hence closes over—its own continuation. If we had chosen a different representation for program instructions, combining this with the data structure representation of closures, we would obtain a continuation representation that is essentially the same as the machine stack.
- Each stack frame also stores procedure parameters. This is implicitly managed by the procedural representation of continuations, whereas this was done explicitly in the data stucture representation (using `bind`).
- Each frame also has space for "local variables". In principle so does the continuation, though by using the macro implementation of local binding, we've effectively reduced everything to procedure parameters. Conceptually, however, some of these are "true" procedure parameters while others are local bindings turned into procedure parameters by a macro.
- The stack has references to, but does not close over, the heap. Thus changes to the heap are visible across stack frames. In precisely the same way, closures refer to, but do not close over, the store, so changes to the store are visible across closures.

Therefore, traditionally the stack is responsible for maintaining lexical scope, which we get automatically because we are using closures in a statically-scoped language.

Now we can study the conversion of various terms to understand the mapping to stacks. For instance, consider the conversion of a function application [REF]:

```
[(_ (f a))
 #'(lambda (k)
     ((cps f) (lambda (fv)
                ((cps a) (lambda (av)
                           (fv av k))))))]
```

How do we "read" this? As follows:

- Let's use `k` to refer to the stack present before the function application begins to evaluate.
- When we begin to evaluate the function position (`f`), create a new stack frame (`(lambda (fv) ...)`). This frame has one free identifier: `k`. Thus its closure needs to record one element of the environment, namely the rest of the stack.
- The code portion of the stack frame represents what is left to be done once we obtain a value for the function: evaluate the argument, and perform the application, and return the result to the stack expecting the result of the application: `k`.
- When evaluation of `f` completes, we begin to evaluate `a`, which also creates a stack frame: `(lambda (av) ...)`. This frame has *two* free identifiers: `k` and `fv`. This tells us:
  - We no longer need the stack frame for evaluating the function position, but
  - we now need a *temporary* that records the value—hopefully a function value—of evaluating the function position.
- The code portion of this second frame also represents what is left to be done: invoke the function value with the argument, in the stack expecting the value of the application.

Let us apply similar reasoning to conditionals:

```
[(_ (cnd tst thn els))
 #'(lambda (k)
     ((cps tst) (lambda (tstv)
                  (if tstv
                      ((cps thn) k)
                      ((cps els) k))))))]
```

It says that to evaluate the conditional expression we have to create a new stack frame. This frame closes over the stack expecting the value of the entire conditional. This frame makes a decision based on the value of the conditional expression, and invokes one of the other expressions. Once we have examined this value the frame created to evaluate the conditional expression is no longer necessary, so evaluation can proceed in `k`.

Viewed through this lens, we can more easily provide an operational explanation for generators. Each generator has its own private stack, and when execution attempts to return past its end, our implementation raises an error. On invocation, a generator stores a reference to the stack of the "rest of the program" in `where-to-go`, and resumes its own stack, which is referred to by `resumer`. On yielding, the system swaps references to stacks. Coroutines, threads, and generators are all conceptually similar: they are all mechanisms to create "many little stacks" instead of having a single, global stack.

14.5 Tail Calls

Observe that the stack patterns above add a frame to the current stack, perform some evaluation, and eventually always return to the current stack. In particular, observe that in an application, we need stack space to evaluate the function position and then the arguments, but once all these are evaluated, we resume computation using the stack we started out with before the application. In other words, *function calls do not themselves need to consume stack space*: we only need space to compute the arguments.

However, not all languages observe or respect this property. In languages that do, programmers can use *recursion* to obtain *iterative behavior*: i.e., a sequence of function calls can consume no more stack space than no function calls at all. This removes the need to create special looping constructs; indeed, loops can simply be expressed as a syntactic sugar.

Of course, this property does not apply in general. If a call to `f` is performed to compute an argument to a call to `g`, the call to `f` is still consuming space relative to the context surrounding `g`. Thus, we should really speak of a relationship between expressions: one expression is in *tail position* relative to another if its evaluation requires no additional stack space beyond the other. In our CPS macro, every expression that uses `k` as its continuation—such as a function application after all the sub-expressions have been evaluated, or the then- and else-branches of a conditional—are all in tail position relative to the enclosing application (and perhaps recursively further up). In contrast, every expression that has to create a new stack frame is not in tail position.

Some languages have special support for tail *recursion*: when a procedure calls itself in tail position relative to its body. This is obviously useful, because it enables recursion to efficiently implement loops. However, it hurts "loops" that cannot be squeezed into a single recursive function. For instance, when implementing a scanner or other state machine, it is most convenient to have a set of functions each representing one state, and transitioning to other states by making (tail) function calls. It is onerous (and misses the point) to turn these into a single recursive function. If, however, a language recognizes tail calls as such, it can optimize these cross-function calls just as much as it does intra-function ones.

Racket, in particular, promises to implement tail calls without allocating additional stack space. Though some people refer to this as "tail call optimization", this term is misleading: an optimization is optional, whereas whether or not a language promises to properly implement tail calls is a *semantic* feature. Developers need to know how the language will behave because it affects how they program.

Because of this feature, observe something interesting about the program after CPS transformation: all of its function applications are themselves tail calls! You can see this starting with the `read-number/suspend` example that began this chapter: any pending computation was put into the continuation argument. Assuming the program might terminate at any call is tantamount to not using any stack space at all (because the stack would get wiped out).

Exercise

How is the program able to function in the absence of a stack?

14.6 Continuations as a Language Feature

With this insight into the connection between continuations and stacks, we can now return to the treatment of procedures: we ignored the continuation at the point of closure *creation* and instead only used the one at the point of closure *invocation*. This of course corresponds to normal procedure behavior. But now we can ask, what if we use the creation-time continuation instead? This would correspond to maintaining a reference to (a copy of) the stack at the point of "procedure" creation, and when the procedure is applied, ignoring the dynamic evaluation and going back to the point of procedure creation.

In principle, we are trying to leave `lambda` intact and instead give ourselves a language construct that corresponds to this behavior:

cc = "current continuation"

<***cps-macro-let/cc-case***> **::=**

```
[(_ (let/cc kont b))
 (identifier? #'kont)
 #'(lambda (k)
     (let ([kont (lambda (v dyn-k)
                   (k v))])
       ((cps b) k)))]
```

What this says is that either way, control will return to the expression that immediately surrounds the `let/cc`: either by falling through (because the continuation of the body, `b`, is `k`) or—more interestingly—by invoking the continuation, which discards the dynamic continuation `dyn/k` by simply ignoring it and returning to `k` instead.

Here's the simplest test:

```
(test (run (cps (let/cc esc 3)))
      3)
```

This confirms that if we never use the continuation, evaluation of the body proceeds as if the `let/cc` weren't there at all (because of the `((cps b) k)`). If we use it, the value given to the continuation returns to the point of creation:

```
(test (run (cps (let/cc esc (esc 3))))
      3)
```

This example, of course, isn't revealing, but consider this one:

```
(test (run (cps (+ 1 (let/cc esc (esc 3)))))
      4)
```

This confirms that the addition actually happens. But what about the dynamic continuation?

```
(test (run (cps (let/cc esc (+ 2 (esc 3)))))
      3)
```

This shows that the addition by 2 never happens, i.e., the dynamic continuation is indeed ignored. And just to be sure that the continuation at the point of creation is respected, observe:

```
(test (run (cps (+ 1 (let/cc esc (+ 2 (esc 3))))))
      4)
```

From these examples, you have probably noticed a familiar pattern: `esc` here is behaving like an *exception*. That is, if you do not throw an exception (in this case, invoke a continuation) it's as if it's not there, but if you do throw it, all pending intermediate computation is ignored and computation returns to the point of exception creation.

Exercise

Using `let/cc` and macros, create a `throw/catch` mechanism.

However, these examples only scratch the surface of available power, because the continuation at the point of invocation is always an extension of one at the point of creation: i.e., the latter is just earlier in the stack than the former. However, nothing actually demands that `k` and `dyn-k` be at all related. That means they are in fact free to be *un*related, which means each can be a distinct stack, so we can in fact easily implement stack-switching procedures with them.

Exercise

To be properly analogous to `lambda`, we should have introduced a construct called, say, `cont-lambda` with the following expansion:

```
[(_ (cont-lambda (a) b))
 (identifier? #'a)
 #'(lambda (k)
     (k (lambda (a dyn-k)
          ((cps b) k))))]
```

Why didn't we? Consider both the static typing implications, and also how we might construct the above exception-like behaviors using this construct instead.

14.6.1 Presentation in the Language

Writing programs in our little toy languages can soon become frustrating. Fortunately, Racket already provides a construct called `call/cc` that reifies continuations. `call/cc` is a procedure of one argument, which is itself a procedure of one argument, which Racket applies to the current continuation—which is a procedure of one argument. Got that?

Fortunately, we can easily write `let/cc` as a macro over `call/cc` and program with that instead. Here it is:

```
(define-syntax let/cc
  (syntax-rules ()
    [(let/cc k b)
     (call/cc (lambda (k) b))]))
```

To be sure, all our old tests still pass:

```
(test (let/cc esc 3) 3)
(test (let/cc esc (esc 3)) 3)
(test (+ 1 (let/cc esc (esc 3))) 4)
(test (let/cc esc (+ 2 (esc 3))) 3)
(test (+ 1 (let/cc esc (+ 2 (esc 3)))) 4)
```

14.6.2 Defining Generators

Now we can start to create interesting abstractions. For instance, let's build generators. Whereas previously we needed to both CPS expressions and pass around continuations, now this is done for us automatically by `call/cc`. Therefore, whenever we need the current continuation, we simply conjure it up without having to transform the program. Thus the extra `...-k` parameters can disappear with a `let/cc` in the same place to capture the same continuation:

```
(define-syntax (generator e)
  (syntax-case e ()
    [(generator (yield) (v) b)
     #'(let ([where-to-go (lambda (v) (error 'where-to-go "nothing"))])
         (letrec ([resumer (lambda (v)
                             (begin b
                                    (error 'generator "fell through")))]
                  [yield (lambda (v)
                           (let/cc gen-k
                             (begin
                               (set! resumer gen-k)
                               (where-to-go v))))])
           (lambda (v)
             (let/cc dyn-k
               (begin
                 (set! where-to-go dyn-k)
                 (resumer v)))))))]))
```

Observe the close similarity between this code and the implementation of generators by desugaring into CPS code. Specifically, we can drop the extra continuation arguments, and replace them with invocations of `let/cc` that will capture precisely the same continuations. The rest of the code is fundamentally unchanged.

Exercise

What happens if we move the `let/cc`s and mutation to be the first statement inside the `begin`s instead?

We can, for instance, write a generator that iterates from the initial value upward:

```
(define g1 (generator (yield) (v)
                      (letrec ([loop (lambda (n)
                                       (begin
                                         (yield n)
                                         (loop (+ n 1))))])
                        (loop v))))
```

whose behavior is:

```
> (g1 10)
10
> (g1 10)
11
> (g1 0)
12
>
```

Because the body refers only to the initial value, ignoring that returned by invoking `yield`, the values we pass on subsequent invocations are irrelevant. In contrast, consider this generator:

```
(define g2 (generator (yield) (v)
                      (letrec ([loop (lambda (n)
                                       (loop (+ (yield n) n)))])
                        (loop v))))
```

On its first invocation, it returns whatever value it was supplied. On subsequent invocations, this value is added to that provided on re-entry into the generator. In other words, this generator additively accumulates all values given to it:

```
> (g2 10)
10
> (g2 15)
25
> (g2 5)
30
```

Exercise

Now that you've seen how to implement generators using `call/cc` and `let/cc`, implement coroutines and threads as well.

14.6.3 Defining Threads

Having done generators, let's do another, similar primitive: threads. That is, let's assume we want to be able to write a program such as this:

```
(define d display) ;; a useful shorthand in what follows
```

```
(scheduler-loop-0
 (list
  (thread-0 (y) (d "t1-1  ") (y) (d "t1-2  ") (y) (d "t1-3 "))
  (thread-0 (y) (d "t2-1  ") (y) (d "t2-2  ") (y) (d "t2-3 "))
  (thread-0 (y) (d "t3-1  ") (y) (d "t3-2  ") (y) (d "t3-3 "))))
```

and expect the output

```
t1-1  t2-1  t3-1  t1-2  t2-2  t3-2  t1-3 t2-3 t3-3
```

We'll build all the pieces necessary to achieve this.

Let's start by defining the thread scheduler. It consumes a list of "threads", whose interface we assume will be a procedure that consumes a continuation to which it eventually yields control. Each time the scheduler reactivates the thread, it supplies it with a continuation. The scheduler might be choose between threads in a simple *round-robin* manner, or it might use some more complex algorithm; the details of how it chooses don't concern us here.

As with generators, we'll assume that yielding is done by invoking a procedure named by the user: `y`, above. We could use name capture [REF] to automatically bind a name like `yield`.

More importantly, notice that the user of the thread system manually yields control. This is called *cooperative multitasking*. Instead, we could have chosen to have a timer or other intrinsic mechanism automtically yield without the user's permission; this is called *preemptive multitasking* (because the system "pre-empts"—i.e., wrests control from—the thread). While the distinction is important for buildling systems, it is not interesting from the perspective of setting up the continuations.

Exercise

> After we are done building cooperative multitasking, implement preemptive multitasking. What changes?

With our stated constraints, we can write a first scheduler. It consumes a lists of threads and continues executing so long as there are threads remaining. Each time, it applies the thread procedure to a continuation that represents returning to the scheduler and proceeding to the next thread:

```
(define (scheduler-loop-0 threads)
  (cond
    [(empty? threads) 'done]
    [(cons? threads)
     (begin
       (let/cc after-thread ((first threads) after-thread))
       (scheduler-loop-0 (append (rest threads)
                                 (list (first threads)))))]))
```

When the recipient thread invokes the continuation bound to `after-thread`, control returns to the end of the first statement in the `begin` sequence. As a result, the value supplied to the continuation is ignored (and can hence be any dummy value; we'll chose `'dummy`, so that we can easily spot it if it shows up in undesired places). Control

then proceeds with the rest of the scheduler loop after appending the most recently invoked thread to the end of the list of threads (i.e., treating the list as a circular queue).

Now let's define a thread. As we've said, it will be a procedure of one argument, the scheduler's continuation. Because the thread needs to *resume*, i.e., continue where it left off, presumably it must store where it last was: we'll call this `thread-resumer`. Initially this is the entire thread body, but on subsequent instances it will be a continuation: specifically, the continuation of invoking `yield`. Thus, we obtain the following skeleton:

```
(define-syntax thread-0
  (syntax-rules ()
    [(thread (yielder) b ...)
     (letrec ([thread-resumer (lambda (_)
                                (begin b ...))])
       (lambda (sched-k)
         (thread-resumer 'dummy)))]))
```

That still leaves the yielder. It needs to be a procedure of no arguments that stores the thread's continuation in `thread-resumer`, and then invoke the scheduler continuation with `'dummy`. However, *which* scheduler continuation does it need to invoke? Not the one provided on thread initiation, but rather the most recent one. Thus, we must somehow "thread" the value in `sched-k` to the yielder. There are many ways to accomplish it, but the simplest, perhaps most brutal, way is to simply reconstruct the yielder on each thread resumption, always closed over the most recent value of `sched-k`:

```
(define-syntax thread-0
  (syntax-rules ()
    [(thread (yielder) b ...)
     (letrec ([thread-resumer (lambda (_)
                                (begin b ...))]
              [yielder (lambda () (error 'yielder "nothing here"))])
       (lambda (sched-k)
         (begin
           (set! yielder
                 (lambda ()
                   (let/cc thread-k
                     (begin
                       (set! thread-resumer thread-k)
                       (sched-k 'dummy)))))
           (thread-resumer 'tres))))]))
```

When we run this ensemble, we get:
`t1-1 t2-1 t3-1 t1-2 t2-2 t3-2 t1-3 t2-3 t3-3`
Hey, that's what we wanted! But it continues:
`t1-3 t2-3 t3-3 t1-3 t2-3 t3-3 t1-3 t2-3 t3-3`
Hmmm.

What's happening? Well, we've been quiet all along about what needs to happen when a thread reaches its end. In fact, control just returns to the thread scheduler, which appends the thread to the end of the queue, and when the thread comes to the head of the queue again, control resumes from the same previously stored continuation: the one corresponding to printing the third value. This prints, control returns, the thread is appended...ad infinitum.

Clearly, we need the thread scheduler to be notified when a thread has terminated, so that the scheduler can remove it from the thread queue. We'll create a simple datatype to represent this signal:

```
(define-type ThreadStatus
  [Tsuspended]
  [Tdone])
```

(In a real system, of course, these status messages might also carry informative values from the computation.) We must now modify our scheduler to actually check for and use these values:

```
(define (scheduler-loop-1 threads)
  (cond
    [(empty? threads) 'done]
    [(cons? threads)
     (type-case ThreadStatus (let/cc after-thread ((first threads) after-
thread))
       [Tsuspended () (scheduler-loop-1 (append (rest threads)
                                                (list (first threads))))]
       [Tdone () (scheduler-loop-1 (rest threads))])]))
```

We have to now modify our thread representation in two ways: it must provide `Tsuspended` to the scheduler's continuation on intermediate returns, and provide `Tdone` when it terminates. Where does it terminate? After executing the code in the body, `b ...`. Observe, finally, that the termination process must be sure to use the latest scheduler continuation, just as yielding does. Thus:

```
(define-syntax thread-1
  (syntax-rules ()
    [(thread (yielder) b ...)
     (letrec ([thread-resumer (lambda (_)
                                (begin b ...
                                       (finisher)))]
              [finisher (lambda () (error 'finisher "nothing here"))]
              [yielder (lambda () (error 'yielder "nothing here"))])
       (lambda (sched-k)
         (begin
           (set! finisher
                 (lambda ()
                   (let/cc thread-k
```

```
          (sched-k (Tdone)))))
(set! yielder
      (lambda ()
              (let/cc thread-k
                (begin
                  (set! thread-resumer thread-k)
                  (sched-k (Tsuspended))))))
(thread-resumer 'tres))))]))
```

If we now replace `scheduler-loop-0` with `scheduler-loop-1` and `thread-0` with `thread-1` and re-run our example program above, we get just the output we desire.

14.6.4 Better Primitives for Web Programming

Finally, to tie the knot back to where we began, let's return to `read-number`: observe that, if the language running the server program has `call/cc`, instead of having to CPS the entire program, we can simply capture the current continuation and save it in the hash table, leaving the program structure again intact.

15 Checking Program Invariants Statically: Types