Essentials of Programming Languages, Third Edition

by Daniel P. Friedman and Mitchell Wand

Chapter 5: Continuation-Passing Interpreters

Overview

In chapter 3, we used the concept of environments to explore the behavior of bindings, which establish the data context in which each portion of a program is executed. Here we will do the same for the *control context* in which each portion of a program is executed. We will introduce the concept of a *continuation* as an abstraction of the control context, and we will write interpreters that take a continuation as an argument, thus making the control context explicit.

Consider the following definition of the factorial function in Scheme.

```
(define fact
  (lambda (n)
    (if (zero? n) 1 (* n (fact (- n 1))))))
```

We can use a derivation to model a calculation with fact:

```
  (fact 4)
= (* 4 (fact 3))
= (* 4 (* 3 (fact 2)))
= (* 4 (* 3(* 2 (fact 1))))
= (* 4 (* 3(* 2(* 1 (fact 0)))))
= (* 4 (* 3(* 2(* 1 1))))
= (* 4 (* 3(* 2 1)))
= (* 4 (* 3 2))
= (* 4 6)
= 24
```

This is the natural recursive definition of factorial. Each call of fact is made with a promise that the value returned will be multiplied by the value of n at the time of the call. Thus fact is invoked in larger and larger *control contexts* as the calculation proceeds. Compare this behavior to that of the following procedures.

```
(define fact-iter
  (lambda (n)
    (fact-iter-acc n 1)))

(define fact-iter-acc
  (lambda (n a)
    (if (zero? n) a (fact-iter-acc (- n 1) (* n a)))))
```

With these definitions, we calculate:

```
  (fact-iter 4)
= (fact-iter-acc 4 1)
= (fact-iter-acc 3 4)
= (fact-iter-acc 2 12)
= (fact-iter-acc 1 24)
= (fact-iter-acc 0 24)
= 24
```

Here, fact-iter-acc is always invoked in the same control context: in this case, no context at all. When fact-iter-acc calls itself, it does so at the “tail end” of an execution of fact-iter-acc. No promise is made to do anything with the returned value other than to return it as the result of the call to fact-iter-acc. We call this a *tail call*. Thus each step in the derivation above has the form (fact-iter-acc *n a*).

When a procedure such as fact executes, additional control information is recorded with each recursive call, and this information is retained until the call returns. This reflects growth of the control context in the first derivation above. Such a process is said to exhibit *recursive control behavior*.

By contrast, no additional control information need be recorded when fact-iter-acc calls itself. This is reflected in the derivation by recursive calls occurring at the same level within the expression (on the outside in the derivation above). In such cases the system does not need an ever-increasing amount of memory for control contexts as the depth of recursion (the number of recursive calls without corresponding returns) increases. A process that uses a bounded amount of memory for control information is said to exhibit *iterative control behavior*.

Why do these programs exhibit different control behavior? In the recursive definition of factorial, the procedure fact is called *in an operand position*. We need to save context around this call because we need to remember that after the evaluation of the

procedure call, we still need to finish evaluating the operands and executing the outer call, in this case to the waiting multiplication. This leads us to an important principle:

> It is evaluation of operands, not the calling of procedures, that makes the control context grow.

In this chapter we will learn how to track and manipulate control contexts. Our central tool will be the data type of *continuations*. Continuations are an abstraction of the notion of control context, much as environments are an abstraction of data contexts. We will explore continuations by writing an interpreter that explicitly passes a continuation parameter, just as our previous interpreters explicitly passed an environment parameter. Once we do this for the simple cases, we can see how to add to our language facilities that manipulate control contexts in more complicated ways, such as exceptions and threads.

In chapter 6 we show how the same techniques we used to transform the interpreter can be applied to any program. We say that a program transformed in this manner is in *continuation-passing style*. Chapter 6 also shows several other important uses of continuations.

5.1 A Continuation-Passing Interpreter

In our new interpreter, the major procedures such as value-of will take a third parameter. This new parameter, the *continuation*, is intended to be an abstraction of the control context in which each expression is evaluated.

We begin with an interpreter in figure 5.1 of the language LETREC of section 3.4. We refer to the result of value-of-program as a *FinalAnswer* to emphasize that this expressed value is the final value of the program.

```
FinalAnswer = ExpVal

value-of-program : Program → FinalAnswer
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (value-of exp1 (init-env))))))

value-of : Exp × Env → ExpVal
(define value-of
  (lambda (exp env)
    (cases expression exp
      (const-exp (num) (num-val num))
      (var-exp (var) (apply-env env var))
      (diff-exp (exp1 exp2)
        (let ((num1 (expval->num (value-of exp1 env)))
              (num2 (expval->num (value-of exp2 env))))
          (num-val (- num1 num2))))
      (zero?-exp (exp1)
        (let ((num1 (expval->num (value-of exp1 env))))
          (if (zero? num1) (bool-val #t) (bool-val #f))))
      (if-exp (exp1 exp2 exp3)
        (if (expval->bool (value-of exp1 env))
          (value-of exp2 env)
          (value-of exp3 env)))
      (let-exp (var exp1 body)
        (let ((val1 (value-of exp1 env)))
          (value-of body (extend-env var val1 env))))
      (proc-exp (var body)
        (proc-val (procedure var body env)))
      (call-exp (rator rand)
        (let ((proc1 (expval->proc (value-of rator env)))
              (arg (value-of rand env)))
          (apply-procedure proc1 arg)))
      (letrec-exp (p-name b-var p-body letrec-body)
        (value-of letrec-body
          (extend-env-rec p-name b-var p-body env))))))

apply-procedure : Proc × ExpVal → ExpVal
(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of body (extend-env var val saved-env))))))
```

Figure 5.1: Environment-passing Interpreter

Our goal is to rewrite the interpreter so that no call to value-of builds control context. When the control context needs to grow, we extend the continuation parameter, much as we extended the environment in the interpreters of chapter 3 as the program builds up data context. By making the control context explicit, we can see how it grows and shrinks, and later, in sections 5.4–5.5 we will use it to add new control behavior to our language.

Now, we know that an environment is a representation of a function from symbols to denoted values. What does a continuation represent? The continuation of an expression represents a procedure that takes the result of the expression and completes the computation. So our interface must include a procedure apply-cont that takes a continuation cont and an expressed value val and finishes the computation as specified by cont. The contract for apply-cont will be

```
FinalAnswer = ExpVal
apply-cont : Cont × ExpVal → FinalAnswer
```

We call the result of apply-cont a *FinalAnswer* to remind ourselves that it is the final value of the computation: it will not be used by any other part of our program.

What kind of continuation-builders will be included in the interface? We will discover these continuation-builders as we analyze the interpreter. To begin, we will need a continuation-builder for the context that says there is nothing more to do with the value of the computation. We call this continuation (end-cont), and we will specify it by

```
(apply-cont (end-cont) val)
= (begin
    (eopl:printf "End of computation.~%")
    val)
```

Invoking (end-cont) prints out an end-of-computation message and returns the value of the program. Because (end-cont) prints out a message, we can tell how many times it has been invoked. In a correct completed computation, it should be invoked exactly once.

We rewrite value-of-program as:

```
value-of-program : Program → FinalAnswer
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (value-of/k exp1 (init-env) (end-cont))))))
```

We can now begin to write value-of/k. We consider each of the alternatives in value-of in turn. The first few lines of value-of simply calculate a value and return it, without calling value-of again. In the continuation-passing interpreter, these same lines send the same value to the continuation by calling apply-cont:

```
value-of/k : Exp × Env × Cont → FinalAnswer
(define value-of/k
  (lambda (exp env cont)
    (cases expression exp
      (const-exp (num) (apply-cont cont (num-val num)))
      (var-exp (var) (apply-cont cont (apply-env env var)))
      (proc-exp (var body)
        (apply-cont cont
          (proc-val (procedure var body env))))
      ...)))
```

Up to now the only possible value of cont has been the end continuation, but that will change momentarily. It is easy to check that if the program consists of an expression of one of these forms, the value of the expression will be supplied to end-cont (through apply-cont).

The behavior of letrec is almost as simple: it creates a new environment without calling value-of, and then evaluates the body in the new environment. The value of the body becomes the value of the entire expression. That means that the body is performed in the same control context as the entire expression. Therefore the value of the body should be returned to the continuation of the entire expression. Therefore we write

```
(letrec-exp (p-name b-var p-body letrec-body)
  (value-of/k letrec-body
    (extend-env-rec p-name b-var p-body env)
    cont))
```

This illustrates a general principle:

Tail Calls Don't Grow the Continuation

If the value of exp_1 is returned as the value of exp_2, then exp_1 and exp_2 should run in the same continuation.

It would not be correct to write

```
(letrec-exp (p-name b-var p-body letrec-body)
  (apply-cont cont
    (value-of/k letrec-body
      (extend-env-rec p-name b-var p-body env)
      (end-cont))))
```

because the call to value-of/k is in an operand position: it appears as an operand to apply-cont. In addition, using the continuation (end-cont) causes the end-of-computation message to be printed before the computation is finished, so an error like this is easy to detect.

Let us next consider a zero? expression. In a zero? expression, we want to evaluate the argument, and then return a value to the continuation depending on the value of the argument. So we evaluate the argument in a new continuation that will look at the returned value and do the right thing.

So in value-of/k we write

```
(zero?-exp (exp1)
  (value-of/k exp1 env
    (zero1-cont cont)))
```

where (zero1-cont cont) is a continuation with the property that

```
(apply-cont (zero1-cont cont) val)
= (apply-cont cont
    (bool-val
      (zero? (expval->num val))))
```

Just as with letrec, we could not write in value-of/k

```
(zero?-exp (exp1)
  (let ((val (value-of/k exp1 env (end-cont))))
    (apply-cont cont
      (bool-val
        (zero? (expval->num val))))))
```

because the call to value-of/k is in operand position. The right-hand side of a let is in operand position, because (let ((*var* exp_1)) exp_2) is equivalent to ((lambda (*var*) exp_2) exp_1). The value of the call to value-of/k eventually becomes the operand of expval->num. As before, if we ran this code, the end-of-computation message would appear twice: once in the middle of the computation and once at the real end.

A let expression is just slightly more complicated than a zero? expression: after evaluating the right-hand side, we evaluate the body in a suitably extended environment. The original code for let was

```
(let-exp (var exp1 body)
  (let ((val1 (value-of exp1 env)))
    (value-of body
      (extend-env var val1 env))))
```

In the continuation-passing interpreter, we need to evaluate exp1 in a context that will finish the computation. So in value-of/k we write

```
(let-exp (var exp1 body)
  (value-of/k exp1 env
    (let-exp-cont var body env cont)))
```

and we add to our continuations interface the specification

```
(apply-cont (let-exp-cont var body env cont) val)
= (value-of/k body (extend-env var val env) cont)
```

The value of the body of the let expression becomes the value of the let expression, so the body of the let expression is evaluated in the same continuation as the entire let expression. This is another instance of the Tail Calls Don't Grow the

Continuation principle.

Let us move on to if expressions. In an if expression, the first thing evaluated is the test, but the result of the test is not the value of the entire expression. We need to build a new continuation that will see if the result of the test expression is a true value, and evaluate either the true expression or the false expression. So in value-of/k we write

```
(if-exp (exp1 exp2 exp3)
  (value-of/k exp1 env
    (if-test-cont exp2 exp3 env cont)))
```

where if-test-cont is a new continuation-builder subject to the specification

```
(apply-cont (if-test-cont exp2 exp3 env cont) val)
= (if (expval->bool val)
    (value-of/k exp2 env cont)
    (value-of/k exp3 env cont))
```

So far, we have four continuation-builders. We can implement them using either a procedural representation or a data structure representation. The procedural representation is in figure 5.2 and the data structure representation, using define-datatype, is in figure 5.3.

```
Cont = ExpVal → FinalAnswer

end-cont : () → Cont
(define end-cont
  (lambda ()
    (lambda (val)
      (begin
        (eopl:printf "End of computation.~%")
        val))))

zero1-cont : Cont → Cont
(define zero1-cont
  (lambda (cont)
    (lambda (val)
      (apply-cont cont
        (bool-val
          (zero? (expval->num val)))))))

let-exp-cont : Var × Exp × Env × Cont → Cont
(define let-exp-cont
  (lambda (var body env cont)
    (lambda (val)
      (value-of/k body (extend-env var val env) cont))))

if-test-cont : Exp × Exp × Env × Cont → Cont
(define if-test-cont
  (lambda (exp2 exp3 env cont)
    (lambda (val)
      (if (expval->bool val)
        (value-of/k exp2 env cont)
        (value-of/k exp3 env cont)))))

apply-cont : Cont × ExpVal → FinalAnswer
(define apply-cont
  (lambda (cont v)
    (cont v)))
```

Figure 5.2: Procedural representation of continuations

```
(define-datatype continuation continuation?
  (end-cont)
  (zero1-cont
    (cont continuation?))
  (let-exp-cont
    (var identifier?)
    (body expression?)
    (env environment?)
    (cont continuation?))
  (if-test-cont
    (exp2 expression?)
    (exp3 expression?)
```

```
    (env environment?)
    (cont continuation?)))

apply-cont : Cont × ExpVal → FinalAnswer
(define apply-cont
  (lambda (cont val)
    (cases continuation cont
      (end-cont ()
        (begin
          (eopl:printf "End of computation.~%")
          val))
      (zero1-cont (saved-cont)
        (apply-cont saved-cont
          (bool-val
            (zero? (expval->num val)))))
      (let-exp-cont (var body saved-env saved-cont)
        (value-of/k body
          (extend-env var val saved-env) saved-cont))
      (if-test-cont (exp2 exp3 saved-env saved-cont)
        (if (expval->bool val)
          (value-of/k exp2 saved-env saved-cont)
          (value-of/k exp3 saved-env saved-cont))))))
```

Figure 5.3: Data structure representation of continuations

Here is a sample calculation that shows how these pieces fit together. As we did in section 3.3, we write «*exp*» to denote the abstract syntax tree associated with the expression *exp*. Assume ρ_0 is an environment in which b is bound to (bool-val #t) and assume $cont_0$ is the initial continuation, which is the value of (end-cont). The commentary is informal and should be checked against the definition of value-of/k and the specification of apply-cont. This example is contrived because we have letrec to introduce procedures but we do not yet have a way to invoke them.

```
(value-of/k <<letrec p(x) = x in if b then 3 else 4>>
  ρ0 cont0)
= letting ρ1 be (extend-env-rec ... ρ0)
(value-of/k <<if b then 3 else 4>> ρ1 cont0)
= next, evaluate the test expression
(value-of/k <<b>> ρ1 (test-cont <<3>> <<4>> ρ1 cont0))
= send the value of b to the continuation
(apply-cont (test-cont <<3>> <<4>> ρ1 cont0)
            (bool-val #t))
= evaluate the then-expression
(value-of/k <<3>> ρ1 cont0)
= send the value of the expression to the continuation
(apply-cont cont0 (num-val 3))
= invoke the final continuation with the final answer
(begin (eopl:printf ...) (num-val 3))
```

Difference expressions add a new wrinkle to our interpreter because they must evaluate both operands. We begin as we did with if, evaluating the first argument:

```
(diff-exp (exp1 exp2)
  (value-of/k exp1 env
    (diff1-cont exp2 env cont)))
```

When (diff1-cont exp2 env cont) receives a value, it should evaluate exp2 in a context that saves the value of exp1. We specify this by writing

```
(apply-cont (diff1-cont exp2 env cont) val1)
= (value-of/k exp2 env
    (diff2-cont val1 cont))
```

When a (diff2-cont val1 cont) receives a value, we know the values of both operands so we can proceed to send their difference to cont, which has been waiting to receive it. The specification is

```
(apply-cont (diff2-cont val1 cont) val2)
= (let ((num1 (expval->num val1))
        (num2 (expval->num val2)))
    (apply-cont cont
      (num-val (- num1 num2))))
```

Let's watch this system do an example.

```
(value-of/k
  <<-(-(44,11),3)>>
  ρ0
   #(struct:end-cont))
= start working on first operand
(value-of/k
  <<-(44,11)>>
  ρ0
  #(struct:diff1-cont <<3>> ρ0
     #(struct:end-cont)))
= start working on first operand
(value-of/k
  <<44>>
  ρ0
  #(struct:diff1-cont <<11>> ρ0
     #(struct:diff1-cont <<3>> ρ0
        #(struct:end-cont))))
= send value of <<44>> to continuation
(apply-cont
  #(struct:diff1-cont <<11>> ρ0
     #(struct:diff1-cont <<3>> ρ0
        #(struct:end-cont)))
  (num-val 44))
= now start working on second operand
(value-of/k
  <<11>>
  ρ0
  #(struct:diff2-cont (num-val 44)
     #(struct:diff1-cont <<3>> ρ0
        #(struct:end-cont))))
= send value to continuation
(apply-cont
  #(struct:diff2-cont (num-val 44)
     #(struct:diff1-cont <<3>> ρ0
        #(struct:end-cont)))
  (num-val 11))
= 44 - 11 is 33, send that to the continuation
(apply-cont
  #(struct:diff1-cont <<3>> ρ0
     #(struct:end-cont))
  (num-val 33))
= start working on second operand <<3>>
(value-of/k
  <<3>>
  ρ0
  #(struct:diff2-cont (num-val 33)
     #(struct:end-cont)))
= send value to continuation
(apply-cont
  #(struct:diff2-cont (num-val 33)
     #(struct:end-cont))
  (num-val 3))
= 33 - 3 is 30, send that to the continuation
(apply-cont
  #(struct:end-cont)
  (num-val 30))
```

apply-cont prints out the completion message "End of computation" and returns (num-val 30) as the final answer of the computation.

The last thing in our language is procedure application. In the environment-passing interpreter, we wrote

```
(call-exp (rator rand)
  (let ((proc1 (expval->proc (value-of rator env)))
        (val (value-of rand env)))
    (apply-procedure proc1 val)))
```

Here we have two calls to consider, as we did in diff-exp. So we must choose one of them to be first, and then we must transform the remainder to handle the second. Furthermore, we will have to pass the continuation to apply-procedure, because apply-procedure contains a call to value-of/k.

We choose the evaluation of the operator to be first, so in value-of/k we write

```
(call-exp (rator rand)
  (value-of/k rator env
    (rator-cont rand env cont)))
```

As with diff-exp, a rator-cont will evaluate the operand in a suitable continuation:

```
(apply-cont (rator-cont rand env cont) val1)
= (value-of/k rand env
    (rand-cont val1 cont))
```

When a rand-cont receives a value, it is ready to call the procedure:

```
(apply-cont (rand-cont val1 cont) val2)
= (let ((proc1 (expval->proc val1)))
    (apply-procedure/k proc1 val2 cont))
```

Last, we must modify apply-procedure to fit in this continuation-passing style:

```
apply-procedure/k : Proc × ExpVal × Cont → FinalAnswer
(define apply-procedure/k
  (lambda (proc1 val cont)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of/k body
          (extend-env var val saved-env)
          cont)))))
```

This completes the presentation of the continuation-passing interpreter. The complete interpreter is shown in figures 5.4 and 5.5. The complete specification of the continuations is shown in figure 5.6.

```
value-of-program : Program → FinalAnswer
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (value-of/k exp1 (init-env) (end-cont))))))

value-of/k : Exp × Env × Cont → FinalAnswer
(define value-of/k
  (lambda (exp env cont)
    (cases expression exp
      (const-exp (num) (apply-cont cont (num-val num)))
      (var-exp (var) (apply-cont cont (apply-env env var)))
      (proc-exp (var body)
        (apply-cont cont
          (proc-val
            (procedure var body env))))
      (letrec-exp (p-name b-var p-body letrec-body)
        (value-of/k letrec-body
          (extend-env-rec p-name b-var p-body env)
          cont))
      (zero?-exp (exp1)
        (value-of/k exp1 env
          (zero1-cont cont)))
      (if-exp (exp1 exp2 exp3)
        (value-of/k exp1 env
          (if-test-cont exp2 exp3 env cont)))
      (let-exp (var exp1 body)
        (value-of/k exp1 env
          (let-exp-cont var body env cont)))
      (diff-exp (exp1 exp2)
        (value-of/k exp1 env
          (diff1-cont exp2 env cont)))
      (call-exp (rator rand)
        (value-of/k rator env
          (rator-cont rand env cont))))))
```

Figure 5.4: Continuation-passing interpreter (part 1)

```
apply-procedure/k : Proc × ExpVal × Cont → FinalAnswer
```

```
(define apply-procedure/k
  (lambda (proc1 val cont)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of/k body
          (extend-env var val saved-env)
          cont)))))
```

Figure 5.5: Continuation-passing interpreter (part 2)

```
(apply-cont (end-cont) val)
= (begin
    (eopl:printf
      "End of computation.~%")
    val)

(apply-cont (diff1-cont exp2 env cont) val1)
= (value-of/k exp2 env (diff2-cont val1 cont))

(apply-cont (diff2-cont val1 cont) val2)
= (let ((num1 (expval->num val1))
        (num2 (expval->num val2)))
    (apply-cont cont (num-val (- num1 num2))))

(apply-cont (rator-cont rand env cont) val1)
= (value-of/k rand env (rand-cont val1 cont))

(apply-cont (rand-cont val1 cont) val2)
= (let ((proc1 (expval->proc val1)))
    (apply-procedure/k proc1 val2 cont))

(apply-cont (zero1-cont cont) val)
= (apply-cont cont (bool-val (zero? (expval->num val))))

(apply-cont (if-test-cont exp2 exp3 env cont) val)
= (if (expval->bool val)
    (value-of/k exp2 env cont)
    (value-of/k exp3 env cont))

(apply-cont (let-exp-cont var body env cont) val1)
= (value-of/k body (extend-env var val1 env) cont)
```

Figure 5.6: Specification of continuations for figure 5.4

Now we can check the assertion that it is evaluation of actual parameters, not the calling of procedures, that requires growing the control context. In particular, if we evaluate a procedure call (exp_1 exp_2) in some continuation $cont_1$, the body of the procedure to which exp_1 evaluates will also be evaluated in the continuation $cont_1$.

But procedure calls do not themselves grow control contexts. Consider the evaluation of (exp_1 exp_2), where the value of exp_1 is some procedure $proc_1$ and the value of exp_2 is some expressed value val_2.

```
(value-of/k <<(exp1 exp2)>> ρ1 cont1)
= evaluate operator
(value-of/k <<exp1>> ρ1
  (rator-cont <<exp2>> ρ1 cont1))
= send the procedure to the continuation
(apply-cont
  (rator-cont <<exp2>> ρ1 cont1)
  proc1)
= evaluate the operand
(value-of/k <<exp2>> ρ1
  (rand-cont proc1 cont1))
= send the argument to the continuation
(apply-cont
  (rand-cont proc1 cont1)
  val2)
= apply the procedure
(apply-procedure/k proc1 val2 cont1)
```

So the procedure is applied, and its body is evaluated, in the same continuation in which it was called. It is the evaluation of operands, not the entry into a procedure body, that requires control context.

Exercise 5.1 [*] Implement this data type of continuations using the procedural representation.

Exercise 5.2 [*] Implement this data type of continuations using a data-structure representation.

Exercise 5.3 [*] Add let2 to this interpreter. A let2 expression is like a let expression, except that it defines exactly two variables.

Exercise 5.4 [*] Add let3 to this interpreter. A let3 expression is like a let expression, except that it defines exactly three variables.

Exercise 5.5 [*] Add lists to the language, as in exercise 3.9.

Exercise 5.6 [**] Add a list expression to the language, as in exercise 3.10. As a hint, consider adding two new continuation-builders, one for evaluating the first element of the list and one for evaluating the rest of the list.

Exercise 5.7 [**] Add multideclaration let (exercise 3.16) to this interpreter.

Exercise 5.8 [**] Add multiargument procedures (exercise 3.21) to this interpreter.

Exercise 5.9 [**] Modify this interpreter to implement the IMPLICIT-REFS language. As a hint, consider including a new continuation-builder (set-rhs-cont env var cont).

Exercise 5.10 [**] Modify the solution to the previous exercise so that the environment is not kept in the continuation.

Exercise 5.11 [**] Add the begin expression of exercise 4.4 to the continuation-passing interpreter. Be sure that no call to value-of or value-of-rands occurs in a position that would build control context.

Exercise 5.12 [*] Instrument the interpreter of figures 5.4–5.6 to produce output similar to that of the calculation on page 150.

Exercise 5.13 [*] Translate the definitions of fact and fact-iter into the LETREC language. You may add a multiplication operator to the language. Then, using the instrumented interpreter of the previous exercise, compute (fact 4) and (fact-iter 4). Compare them to the calculations at the beginning of this chapter. Find (* 4(* 3(* 2 (fact 1)))) in the trace of (fact 4). What is the continuation of apply-procedure/k for this call of (fact 1)?

Exercise 5.14 [*] The instrumentation of the preceding exercise produces voluminous output. Modify the instrumentation to track instead only the *size* of the largest continuation used during the calculation. We measure the size of a continuation by the number of continuation-builders employed in its construction, so the size of the largest continuation in the calculation on page 150 is 3. Then calculate the values of fact and fact-iter applied to several operands. Confirm that the size of the largest continuation used by fact grows linearly with its argument, but the size of the largest continuation used by fact-iter is a constant.

Exercise 5.15 [*] Our continuation data type contains just the single constant, end-cont, and all the other continuation-builders have a single continuation argument. Implement continuations by representing them as lists, where (end-cont) is represented by the empty list, and each other continuation is represented by a non-empty list whose car contains a distinctive data structure (called *frame* or *activation record*) and whose cdr contains the embedded continuation. Observe that the interpreter treats these lists like a stack (of frames).

Exercise 5.16 [**] Extend the continuation-passing interpreter to the language of exercise 4.22. Pass a continuation argument to result-of, and make sure that no call to result-of occurs in a position that grows a control context. Since a statement does not return a value, distinguish between ordinary continuations and continuations for statements; the latter are usually called *command continuations*. The interface should include a procedure apply-command-cont that takes a command continuation and invokes it. Implement command continuations both as data structures and as zero-argument procedures.

5.2 A Trampolined Interpreter

One might now be tempted to transcribe the interpreter into an ordinary procedural language, using a data structure representation of continuations to avoid the need for higher-order procedures. Most procedural languages, however, make it difficult to do this translation: instead of growing control context only when necessary, they add to the control context (the stack!) on every procedure call. Since the procedure calls in our system never return until the very end of the computation, the stack in these systems continues to grow until that time.

This behavior is not entirely irrational: in such languages almost every procedure call occurs on the right-hand side of an assignment statement, so that almost every procedure call must grow the control context to keep track of the pending assignment. Hence the architecture is optimized for this most common case. Furthermore, most languages store environment information on the stack, so every procedure call must generate a control context that remembers to remove the environment information from the stack.

In such languages, one solution is to use a technique called *trampolining*. To avoid having an unbounded chain of procedure calls, we break the chain by having one of the procedures in the interpreter actually return a zero-argument procedure. This procedure, when called, will continue the computation. The entire computation is driven by a procedure called a *trampoline* that bounces from one procedure call to the next. For example, we can insert a (lambda () …) around the body of apply-procedure/k, since in our language no expression would run more than a bounded amount of time without performing a procedure call.

The resulting code is shown in figure 5.7, which also shows all the tail calls in the interpreter. Since we have modified apply-procedure/k to return a procedure, rather than an *ExpVal*, we must rewrite its contract and also the contracts of all the procedures that call it. We must therefore review the contracts of all the procedures in the interpreter.

```
Bounce = ExpVal ∪ (() → Bounce)

value-of-program : Program → FinalAnswer
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp)
        (trampoline
          (value-of/k exp (init-env) (end-cont)))))))

trampoline : Bounce → FinalAnswer
(define trampoline
  (lambda (bounce)
    (if (expval? bounce)
      bounce
      (trampoline (bounce)))))

value-of/k : Exp × Env × Cont → Bounce
(define value-of/k
  (lambda (exp env cont)
    (cases expression exp
      (... (value-of/k ...))
      (... (apply-cont ...))))))

apply-cont : Cont × ExpVal → Bounce
(define apply-cont
  (lambda (cont val)
    (cases continuation cont
      (... val)
      (... (value-of/k ...))
      (... (apply-cont ...))
      (... (apply-procedure/k ...))))))

apply-procedure/k : Proc × ExpVal × Cont → Bounce
(define apply-procedure/k
  (lambda (proc1 val cont)
    (lambda ()
      (cases procedure proc1
        (... (value-of/k ...)))))))
```

Figure 5.7: Procedural representation of trampolining

We begin with value-of-program. Since this is the procedure that is used to invoke the interpreter, its contract is unchanged. It calls value-of/k and passes the result to trampoline. Since we are now doing something with the result of value-of/k, that result is something other than a *FinalAnswer*. How can that be, since we have not changed the code of value-of/k? The procedure value-of/k calls apply-cont tail-recursively, and apply-cont calls apply-procedure/k tail-recursively, so any result of apply-procedure/k could appear as the result of value-of/k. And, of course, we have modified apply-procedure/k to return something different than it did before.

We introduce the set *Bounce* for the possible results of the value-of/k. (We call it *Bounce* because it is the input to trampoline.) What kind of values could appear in this set? value-of/k calls itself and apply-cont tail-recursively, and these are the only tail-

recursive calls it makes. So the only values that could appear as results of value-of/k are those that appear as results of apply-cont. Also, apply-procedure/k calls value-of/k tail-recursively, so whatever *Bounce* is, it is the set of results of value-of/k, apply-cont, and apply-procedure/k.

The procedures value-of/k and apply-cont just call other procedures tail-recursively. The only procedure that actually puts values in *Bounce* is apply-procedure/k. What kind of values are these? Let's look at the code.

```
(define apply-procedure/k
  (lambda (proc1 val cont)
    (lambda ()
      (cases procedure proc1
        (... (value-of/k ...))))))
```

We see that apply-procedure/k returns a procedure of no arguments, which when called returns either an *ExpVal* or the result of a call to one of value-of/k, apply-cont, or apply-procedure/k, that is, a *Bounce*. So the possible values of apply-procedure/k are described by the set

$$ExpVal \cup (() \to Bounce)$$

These are the same as the possible results of value-of/k, so we conclude that

$$Bounce = ExpVal \cup (() \to Bounce)$$

and that the contracts should be

```
value-of-program  : Program→ FinalAnswer
trampoline          : Bounce → FinalAnswer
value-of/k            : Exp× Env × Cont → Bounce
apply-cont          : Cont × ExpVal→ Bounce
apply-procedure/k : Proc × ExpVal× Cont → Bounce
```

The procedure trampoline satisfies its contract: it is initially passed a *Bounce*. If its argument is an *ExpVal* (and hence a *FinalAnswer*), then it returns it. Otherwise the argument must be a procedure that returns a *Bounce*. So it invokes the procedure on no arguments, and calls itself with the resulting value, which will always be a *Bounce*. (We will see in section 7.4 how to automate reasoning like this.)

Each zero-argument procedure returned by apply-procedure/k represents a snapshot of the computation in progress. We could choose to return such a snapshot at different places in the computation; we see in section 5.5 how this idea can be utilized to simulate atomic actions in multithreaded programs.

Exercise 5.17 [*] Modify the trampolined interpreter to wrap (lambda () ...) around each call (there's only one) to apply-procedure/k. Does this modification require changing the contracts?

Exercise 5.18 [*] The trampoline system in figure 5.7 uses a procedural representation of a *Bounce*. Replace this by a data structure representation.

Exercise 5.19 [*] Instead of placing the (lambda () ...) around the body of apply-procedure/k, place it around the body of apply-cont. Modify the contracts to match this change. Does the definition of *Bounce* need to change? Then replace the procedural representation of *Bounce* with a data-structure representation, as in exercise 5.18.

Exercise 5.20 [*] In exercise 5.19, the last bounce before trampoline returns a *FinalAnswer* is always something like (apply-cont (end-cont) *val*), where *val* is some *ExpVal*. Optimize your representation of bounces in exercise 5.19 to take advantage of this fact.

Exercise 5.21 [**] Implement a trampolining interpreter in an ordinary procedural language. Use a data structure representation of the snapshots as in exercise 5.18, and replace the recursive call to trampoline in its own body by an ordinary while or other looping construct.

Exercise 5.22 [***] One could also attempt to transcribe the environment-passing interpreters of chapter 3 in an ordinary procedural language. Such a transcription would fail in all but the simplest cases, for the same reasons as suggested above. Can the technique of trampolining be used in this situation as well?

5.3 An Imperative Interpreter

In chapter 4, we saw how assignment to shared variables could sometimes be used in place of binding. Consider the familiar

example of even and odd at the top of figure 5.8. It could be replaced by the program below it in figure 5.8. There the shared variable x allows communication between the two procedures. In the top example, the procedure bodies look for the relevant data in the environment; in the other program, they look for the relevant data in the store.

```
letrec
 even(x) = if zero?(x)
           then 1
           else (odd sub1(x))
 odd(x) = if zero?(x)
          then 0
          else (even sub1(x))
in (odd 13)

let x = 0
in letrec
    even() = if zero?(x)
             then 1
             else let d = set x = sub1(x)
                  in (odd)
    odd() = if zero?(x)
            then 0
            else let d = set x = sub1(x)
                 in (even)
   in let d = set x = 13
      in (odd)

      x = 13;
      goto odd;
even: if (x=0) then return(1)
               else {x = x-1;
                     goto odd;}
odd:  if (x=0) then return(0)
               else {x = x-1;
                     goto even;}

  (odd  13)
= (even 12)
= (odd  11)
...
= (odd   1)
= (even  0)
= 1
```

Figure 5.8: Three programs with a common trace

Consider a trace of the computation at the bottom of figure 5.8. This could be a trace of either computation. It could be a trace of the first computation, in which we keep track of the procedure being called and its argument, or it could be a trace of the second, in which we keep track of the procedure being called and the contents of the register x.

Yet a third interpretation of this trace would be as the trace of *goto*s (called a flowchart program), in which we keep track of the location of the program counter and the contents of the register x.

But this works only because in the original code the calls to even and odd do not grow any control context: they are tail calls. We could not carry out this transformation for fact, because the trace of fact grows unboundedly: the “program counter” appears not at the outside of the trace, as it does here, but inside a control context.

We can carry out this transformation for any procedure that does not require control context. This leads us to an important principle:

> A 0-argument tail call is the same as a jump.

If a group of procedures call each other only by tail calls, then we can translate the calls to use assignment instead of binding, and then we can translate such an assignment program into a flowchart program, as we did in figure 5.8.

In this section, we will use this principle to translate the continuation-passing interpreter into a form suitable for transcription into a language without higher-order procedures.

We begin with the interpreter of figures 5.4 and 5.5, using a data structure representation of continuations. The data structure representation is shown in figures 5.9 and 5.10.

```
(define-datatype continuation continuation?
  (end-cont)
  (zero1-cont
    (saved-cont continuation?))
  (let-exp-cont
    (var identifier?)
    (body expression?)
    (saved-env environment?)
    (saved-cont continuation?))
  (if-test-cont
    (exp2 expression?)
    (exp3 expression?)
    (saved-env environment?)
    (saved-cont continuation?))
  (diff1-cont
    (exp2 expression?)
    (saved-env environment?)
    (saved-cont continuation?))
  (diff2-cont
    (val1 expval?)
    (saved-cont continuation?))
  (rator-cont
    (rand expression?)
    (saved-env environment?)
    (saved-cont continuation?))
  (rand-cont
    (val1 expval?)
    (saved-cont continuation?)))
```

Figure 5.9: Data structure implementation of continuations (part 1)

```
apply-cont : Cont × ExpVal → FinalAnswer
(define apply-cont
  (lambda (cont val)
    (cases continuation cont
      (end-cont ()
        (begin
          (eopl:printf
            "End of computation.~%")
          val))
      (zero1-cont (saved-cont)
        (apply-cont saved-cont
          (bool-val
            (zero? (expval->num val)))))
      (let-exp-cont (var body saved-env saved-cont)
        (value-of/k body
          (extend-env var val saved-env) saved-cont))
      (if-test-cont (exp2 exp3 saved-env saved-cont)
        (if (expval->bool val)
          (value-of/k exp2 saved-env saved-cont)
          (value-of/k exp3 saved-env saved-cont)))
      (diff1-cont (exp2 saved-env saved-cont)
        (value-of/k exp2
          saved-env (diff2-cont val saved-cont)))
      (diff2-cont (val1 saved-cont)
        (let ((num1 (expval->num val1))
              (num2 (expval->num val)))
          (apply-cont saved-cont
            (num-val (- num1 num2)))))
      (rator-cont (rand saved-env saved-cont)
        (value-of/k rand saved-env
          (rand-cont val saved-cont)))
      (rand-cont (val1 saved-cont)
        (let ((proc (expval->proc val1)))
          (apply-procedure/k proc val saved-cont))))))
```

Figure 5.10: Data structure implementation of continuations (part 2)

Our first task is to list the procedures that will communicate via shared registers. These procedures, with their formal parameters, are:

```
(value-of/k exp env cont)
```

```
(apply-cont cont val)
(apply-procedure/k proc1 val cont)
```

So we will need five global registers: exp, env, cont, val, and proc1. Each of the three procedures above will be replaced by a zero-argument procedure, and each call to one of these procedures will be replaced by code that stores the value of each actual parameter in the corresponding register and then invokes the new zero-argument procedure. So the fragment

```
(define value-of/k
  (lambda (exp env cont)
    (cases expression exp
      (const-exp (num) (apply-cont cont (num-val num)))
      ...)))
```

can be replaced by

```
(define value-of/k
  (lambda ()
    (cases expression exp
      (const-exp (num)
        (set! cont cont)
        (set! val (num-val num))
        (apply-cont))
      ...)))
```

We can now systematically go through each of our four procedures and perform this transformation. We will also have to transform the body of

value-of-program, since that is where value-of/k is initially called. There are just three easy-to-resolve complications:

1. Often a register is unchanged from one procedure call to another. This yields an assignment like (set! cont cont) in the example above. We can safely omit such assignments.

2. We must make sure that no field name in a cases expression happens to be the same as a register name. In this situation, the field shadows the register, so the register becomes inaccessible. For example, if in value-of-program we had written

   ```
   (cases program pgm
     (a-program (exp)
       (value-of/k exp (init-env) (end-cont))))
   ```

   then exp would be locally bound, so we could not assign to the global register exp. The solution is to rename the local variable to avoid the conflict:

   ```
   (cases program pgm
     (a-program (exp1)
       (value-of/k exp1 (init-env) (end-cont))))
   ```

   Then we can write

   ```
   (cases program pgm
     (a-program (exp1)
       (set! cont (end-cont))
       (set! exp exp1)
       (set! env (init-env))
       (value-of/k)))
   ```

   We have already carefully chosen the field names in our data types to avoid such conflicts.

3. An additional complication may arise if a register is used twice in a single call. Consider transforming a first call in (cons (f (car x)) (f (cdr x))), where x is the formal parameter of f. A naive transformation of this call would be:

   ```
   (begin
     (set! x (car x))
     (set! cont (arg1-cont x cont))
     (f))
   ```

   But this is incorrect, because it loads the register x with the new value of x, when the old value of x was intended. The solution is either to reorder the assignments so the right values are loaded into the registers, or to use temporary variables. Most occurrences of this bug can be avoided by assigning to the continuation variable first:

   ```
   (begin
     (set! cont (arg1-cont x cont))
   ```

```
      (set! x (car x))
      (f))
```

Occasionally, temporary variables are unavoidable; consider (f y x) where x and y are the formal parameters of f. Again, this complication does not arise in our example.

The result of performing this translation on our interpreter is shown in figures 5.11–5.14. This process is called *registerization*. It is an easy process to translate this into an imperative language that supports gotos.

```
(define exp 'uninitialized)
(define env 'uninitialized)
(define cont 'uninitialized)
(define val 'uninitialized)
(define proc1 'uninitialized)

value-of-program : Program → FinalAnswer
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (set! cont (end-cont))
        (set! exp exp1)
        (set! env (init-env))
        (value-of/k)))))

value-of/k : () → FinalAnswer
usage: : relies on registers
      exp  : Exp
      env  : Env
      cont : Cont
(define value-of/k
  (lambda ()
    (cases expression exp
      (const-exp (num)
        (set! val (num-val num))
        (apply-cont))
      (var-exp (var)
        (set! val (apply-env env var))
        (apply-cont))
      (proc-exp (var body)
        (set! val (proc-val (procedure var body env)))
        (apply-cont))
      (letrec-exp (p-name b-var p-body letrec-body)
        (set! exp letrec-body)
        (set! env (extend-env-rec p-name b-var p-body env))
        (value-of/k))
```

Figure 5.11: Imperative interpreter (part 1)

```
(zero?-exp (exp1)
  (set! cont (zero1-cont cont))
  (set! exp exp1)
  (value-of/k))
(let-exp (var exp1 body)
  (set! cont (let-exp-cont var body env cont))
  (set! exp exp1)
  (value-of/k))
(if-exp (exp1 exp2 exp3)
  (set! cont (if-test-cont exp2 exp3 env cont))
  (set! exp exp1)
  (value-of/k))
(diff-exp (exp1 exp2)
  (set! cont (diff1-cont exp2 env cont))
  (set! exp exp1)
  (value-of/k))
(call-exp (rator rand)
  (set! cont (rator-cont rand env cont))
  (set! exp rator)
  (value-of/k)))))
```

Figure 5.12: Imperative interpreter (part 2)

```
apply-cont : () → FinalAnswer
usage: : reads registers
      cont : Cont
      val  : ExpVal
(define apply-cont
  (lambda ()
    (cases continuation cont
      (end-cont ()
        (eopl:printf "End of computation.~%")
        val)
      (zero1-cont (saved-cont)
        (set! cont saved-cont)
        (set! val (bool-val (zero? (expval->num val))))
        (apply-cont))
      (let-exp-cont (var body saved-env saved-cont)
        (set! cont saved-cont)
        (set! exp body)
        (set! env (extend-env var val saved-env))
        (value-of/k))
      (if-test-cont (exp2 exp3 saved-env saved-cont)
        (set! cont saved-cont)
        (if (expval->bool val)
          (set! exp exp2)
          (set! exp exp3))
        (set! env saved-env)
        (value-of/k))
```

Figure 5.13: Imperative interpreter (part 3)

```
(diff1-cont (exp2 saved-env saved-cont)
  (set! cont (diff2-cont val saved-cont))
  (set! exp exp2)
  (set! env saved-env)
  (value-of/k))
(diff2-cont (val1 saved-cont)
  (let ((num1 (expval->num val1))
        (num2 (expval->num val)))
    (set! cont saved-cont)
    (set! val (num-val (- num1 num2)))
    (apply-cont)))
(rator-cont (rand saved-env saved-cont)
  (set! cont (rand-cont val saved-cont))
  (set! exp rand)
  (set! env saved-env)
  (value-of/k))
(rand-cont (rator-val saved-cont)
  (let ((rator-proc (expval->proc rator-val)))
    (set! cont saved-cont)
    (set! proc1 rator-proc)
    (set! val val)
    (apply-procedure/k))))))

apply-procedure/k : () → FinalAnswer
usage: : relies on registers
      proc1 : Proc
      val   : ExpVal
      cont  : Cont
(define apply-procedure/k
  (lambda ()
    (cases proc proc1
      (procedure (var body saved-env)
        (set! exp body)
        (set! env (extend-env var val saved-env))
        (value-of/k)))))
```

Figure 5.14: Imperative interpreter (part 4)

Exercise 5.23 [*] What happens if you remove the “goto” line in one of the branches of this interpreter? Exactly how does the interpreter fail?

Exercise 5.24 [*] Devise examples to illustrate each of the complications mentioned above.

Exercise 5.25 [**] Registerize the interpreter for multiargument procedures (exercise 3.21).

Exercise 5.26 [*] Convert this interpreter to a trampoline by replacing each call to apply-procedure/k with (set! pc apply-procedure/k) and using a driver that looks like

```
(define trampoline
  (lambda (pc)
    (if pc (trampoline (pc)) val)))
```

Exercise 5.27 [*] Invent a language feature for which setting the cont variable last requires a temporary variable.

Exercise 5.28 [*] Instrument this interpreter as in exercise 5.12. Since continuations are represented the same way, reuse that code. Verify that the imperative interpreter of this section generates *exactly* the same traces as the interpreter in exercise 5.12.

Exercise 5.29 [*] Apply the transformation of this section to fact-iter (page 139).

Exercise 5.30 [**] Modify the interpreter of this section so that procedures rely on dynamic binding, as in exercise 3.28. As a hint, consider transforming the interpreter of exercise 3.28 as we did in this chapter; it will differ from the interpreter of this section only for those portions of the original interpreter that are different. Instrument the interpreter as in exercise 5.28. Observe that just as there is only one continuation in the state, there is only one environment that is pushed and popped, and furthermore, it is pushed and popped in parallel with the continuation. We can conclude that dynamic bindings have *dynamic extent*: that is, a binding to a formal parameter lasts exactly until that procedure returns. This is different from lexical bindings, which can persist indefinitely if they wind up in a closure.

Exercise 5.31 [*] Eliminate the remaining let expressions in this code by using additional global registers.

Exercise 5.32 [**] Improve your solution to the preceding exercise by minimizing the number of global registers used. You can get away with fewer than 5. You may use no data structures other than those already used by the interpreter.

Exercise 5.33 [**] Translate the interpreter of this section into an imperative language. Do this twice: once using zero-argument procedure calls in the host language, and once replacing each zero-argument procedure call by a goto. How do these alternatives perform as the computation gets longer?

Exercise 5.34 [**] As noted on page 157, most imperative languages make it difficult to do this translation, because they use the stack for all procedure calls, even tail calls. Furthermore, for large interpreters, the pieces of code linked by goto's may be too large for some compilers to handle. Translate the interpreter of this section into an imperative language, circumventing this difficulty by using the technique of trampolining, as in exercise 5.26.

5.4 Exceptions

So far we have used continuations only to manage the ordinary flow of control in our languages. But continuations allow us to alter the control context as well. Let us consider adding *exception handling* to our defined language. We add to the language two new productions:

Expression ::= `try` *Expression* `catch (`*Identifier*`)` *Expression*

```
try-exp (exp1 var handler-exp)
```

Expression ::= `raise` *Expression*

```
raise-exp (exp)
```

A try expression evaluates its first argument in the context of the exception handler described by the catch clause. If this expression returns normally, its value becomes the value of the entire try expression, and the exception handler is removed.

A raise expression evaluates its single expression and raises an exception with that value. The value is sent to the most recently installed exception handler and is bound to the variable of the handler. The body of the handler is then evaluated. The handler body can either return a value, which becomes the value of the associated try expression, or it can *propagate* the exception by raising another exception; in this case the exception would be sent to the next most recently installed exception handler.

Here's an example, where we assume for the moment that we have added strings to the language.

```
let list-index =
     proc (str)
      letrec inner (lst)
       = if null?(lst)
         then raise("ListIndexFailed")
         else if string-equal?(car(lst), str)
              then 0
              else -((inner cdr(lst)), -1)
```

The procedure list-index is a Curried procedure that takes a string and list of strings, and returns the position of the string in the list. If the desired list element is not found, inner raises an exception and passes "ListIndexFailed" to the most recently installed handler, skipping over all the pending subtractions.

The handler can take advantage of knowledge at the call site to handle the exception appropriately.

```
let find-member-number =
     proc (member-name)
      ... try ((list-index member-name) member-list)
            catch (exn)
             raise("CantFindMemberNumber")
```

The procedure find-member-number takes a string and uses list-index to find the position of the string in the list member-list. The caller of find-member-number has no reason to know about list-index, so find-member-number translates the error message into an exception that its caller can understand.

Yet another possibility, depending on the purpose of the program, is that find-member-number might return some default number if the member's name is not in the list.

```
let find-member-number =
     proc (member-name)
      ... try ((list-index member-name) member-list)
            catch (exn)
             the-default-member-number
```

In both these programs, we have ignored the value of the exception. In other situations, the value passed by raise might include some partial information that the caller could utilize.

Implementing this exception-handling mechanism using the continuation-passing interpreter is straightforward. We begin with the try expression. In the data-structure representation, we add two continuation-builders:

```
(try-cont
  (var identifier?)
  (handler-exp expression?)
  (env environment?)
  (cont continuation?))
(raise1-cont
  (saved-cont continuation?))
```

and we add to value-of/k the following clause for try:

```
(try-exp (exp1 var handler-exp)
  (value-of/k exp1 env
    (try-cont var handler-exp env cont)))
```

What happens when the body of the try expression is evaluated? If the body returns normally, then that value should be sent to the continuation of the try expression, in this case cont:

```
(apply-cont (try-cont var handler-exp env cont) val)
= (apply-cont cont val)
```

What happens if an exception is raised? First, of course, we need to evaluate the argument to raise.

```
(raise-exp (exp1)
  (value-of/k exp1 env
    (raise1-cont cont)))
```

When the value of exp1 is returned to raise1-cont, we need to search through the continuation for the nearest handler, which may be found in the topmost try-cont continuation. So in the specification of continuations we write

```
(apply-cont (raise1-cont cont) val)
= (apply-handler val cont)
```

where apply-handler is a procedure that finds the closest exception handler and applies it (figure 5.15).

```
apply-handler : ExpVal × Cont → FinalAnswer
(define apply-handler
  (lambda (val cont)
    (cases continuation cont
      (try-cont (var handler-exp saved-env saved-cont)
        (value-of/k handler-exp
          (extend-env var val saved-env)
          saved-cont))
      (end-cont ()
        (report-uncaught-exception))
      (diff1-cont (exp2 saved-env saved-cont)
        (apply-handler val saved-cont))
      (diff2-cont (val1 saved-cont)
        (apply-handler val saved-cont))
      ...)))
```

Figure 5.15: The procedure apply-handler

To show how all this fits together, let us consider a calculation using a defined language implementation of index. Let exp_0 denote the expression

```
let index
     = proc (n)
        letrec inner (lst)
         = if null?(lst)
           then raise 99
           else if zero?(-(car(lst),n))
                then 0
                else -((inner cdr(lst)), -1)
        in proc (lst)
            try (inner lst)
             catch (x) -1
in ((index 5) list(2, 3))
```

We start exp_0 in an arbitrary environment ρ_0 and an arbitrary continuation $cont_0$. We will show only the highlights of the calculation, with comments interspersed.

```
(value-of/k
  <<let index = ... in ((index 5) list(2, 3))>>
  ρ0 cont0)
= execute the body of the let
(value-of/k
  <<((index 5) list(2, 3))>>
  ((index                                          call this ρ1
     #(struct:proc-val
        #(struct:procedure n <<letrec ...>> ρ0)))
   (i #(struct:num-val 1))
   (v #(struct:num-val 5))
   (x #(struct:num-val 10)))
  #(struct:end-cont))
= eventually we evaluate the try
(value-of/k
  <<try (inner2 lst) catch (x) -1>>
  ((lst                                            call this ρlst=(2 3)
     #(struct:list-val
        (#(struct:num-val 2) #(struct:num-val 3))))
   (inner2 ...)
   (n #(struct:num-val 5))
   ρ0)
  #(struct:end-cont))
= evaluate the body of the try in a try-cont continuation
(value-of/k
  <<(inner2 lst)>>
  ρlst=(2 3)
  #(struct:try-cont x <<-1>> ρlst=(2 3)
     #(struct:end-cont)))
= evaluate the body of inner2 with lst bound to (2 3)
```

```
(value-of/k
  <<if null?(lst) ... >>
  ρlst=(2 3)
  #(struct:try-cont x <<-1>> ρlst=(2 3)
     #(struct:end-cont)))
= evaluate the conditional, getting to the recursion line
(value-of/k
  <<-((inner2 cdr(lst)), -1)>>
  ρlst=(2 3)
  #(struct:try-cont x <<-1>> ρlst=(2 3)
     #(struct:end-cont)))
= evaluate the first argument of the diff-exp
(value-of/k
  <<(inner2 cdr(lst))>>
  ρlst=(2 3)
  #(struct:diff1-cont <<-1>> ρlst=(2 3)
     #(struct:try-cont x <<-1>> ρlst=(2 3)
        #(struct:end-cont))))
= evaluate the body of inner2 with lst bound to (3)
(value-of/k
  <<if null?(lst) ...>>
  ((lst #(struct:list-val (#(struct:num-val 3)))) call this ρlst=(3)
   (inner2 ...)
   ρ0)
  #(struct:diff1-cont <<-1>> ρlst=(2 3)
     #(struct:try-cont x <<-1>> ρlst=(2 3)
        #(struct:end-cont))))
= evaluate the conditional, getting to the recursion line again
(value-of/k
  <<-((inner2 cdr(lst)), -1)>>
  ρlst=(3)
  #(struct:diff1-cont <<-1>> ρlst=(2 3)
     #(struct:try-cont x <<-1>> ρlst=(2 3)
        #(struct:end-cont))))
= evaluate the first argument of the diff-exp
(value-of/k
  <<(inner2 cdr(lst))>>
  ρlst=(3)
  #(struct:diff1-cont <<-1>> ρlst=(3)
     #(struct:diff1-cont <<-1>> ρlst=(2 3)
        #(struct:try-cont x <<-1>> ρlst=(2 3)
           #(struct:end-cont)))))
= evaluate the body of inner2 with lst bound to ()
(value-of/k
  <<if null?(lst) ... >>
  ((lst #(struct:list-val ()))                    call this ρlst=()
   (inner2 ...)
   (n #(struct:num-val 5))
   ...)
  #(struct:diff1-cont <<-1>> ρlst=(3)
     #(struct:diff1-cont <<-1>> ρlst=(2 3)
        #(struct:try-cont x <<-1>> ρlst=(2 3)
           #(struct:end-cont)))))
= evaluate the raise expression
(value-of/k
  <<raise 99>>
  ρlst=(())
  #(struct:diff1-cont <<-1>> ρlst=(3)
     #(struct:diff1-cont <<-1>> ρlst=(2 3)
        #(struct:try-cont x <<-1>> ρlst=(2 3)
           #(struct:end-cont)))))
= evaluate the argument of the raise expression
(value-of/k
  <<99>>
  ρlst=(())
  #(struct:raise1-cont
     #(struct:diff1-cont <<-1>> ρlst=(3)
        #(struct:diff1-cont <<-1>> ρlst=(2 3)
           #(struct:try-cont x <<-1>> ρlst=(2 3)
              #(struct:end-cont))))))
= use apply-handler to unwind the continuation until we find a handler
```

```
(apply-handler
  #(struct:num-val 99)
  #(struct:diff1-cont <<-1>> ρlst=(3)
     #(struct:diff1-cont <<-1>> ρlst=(2 3)
        #(struct:try-cont x <<-1>> ρlst=(2 3)
           #(struct:end-cont)))))
=
(apply-handler
  #(struct:num-val 99)
  #(struct:diff1-cont <<-1>> ρlst=(2 3)
     #(struct:try-cont x <<-1>> ρlst=(2 3)
        #(struct:end-cont))))
=
(apply-handler
  #(struct:num-val 99)
  #(struct:try-cont x <<-1>> ρlst=(2 3)
     #(struct:end-cont)))
= Handler found; bind the value of the exception to x
(value-of/k
  #(struct:const-exp -1)
  ((x #(struct:num-val 99))
  ρlst=(2 3)...)
  #(struct:end-cont))
=
(apply-cont #(struct:end-cont) #(struct:num-val -1))
=
#(struct:num-val -1)
```

If the list had contained the desired element, then we would have called apply-cont instead of apply-handler, and we would have executed all the pending diffs in the continuation.

Exercise 5.35 [**] This implementation is inefficient, because when an exception is raised, apply-handler must search linearly through the continuation to find a handler. Avoid this search by making the try-cont continuation available directly in each continuation.

Exercise 5.36 [*] An alternative design that also avoids the linear search in apply-handler is to use two continuations, a normal continuation and an exception continuation. Achieve this goal by modifying the interpreter of this section to take two continuations instead of one.

Exercise 5.37 [*] Modify the defined language to raise an exception when a procedure is called with the wrong number of arguments.

Exercise 5.38 [*] Modify the defined language to add a division expression. Raise an exception on division by zero.

Exercise 5.39 [**] So far, an exception handler can propagate the exception by re-raising it, or it can return a value that becomes the value of the try expression. One might instead design the language to allow the computation to resume from the point at which the exception was raised. Modify the interpreter of this section to accomplish this by running the body of the handler with the continuation from the point at which the raise was invoked.

Exercise 5.40 [***] Give the exception handlers in the defined language the ability to either return or resume. Do this by passing the continuation from the raise exception as a second argument. This may require adding continuations as a new kind of expressed value. Devise suitable syntax for invoking a continuation on a value.

Exercise 5.41 [***] We have shown how to implement exceptions using a data-structure representation of continuations. We can't immediately apply the recipe of section 2.2.3 to get a procedural representation, because we now have two observers: apply-handler and apply-cont. Implement the continuations of this section as a pair of procedures: a one-argument procedure representing the action of the continuation under apply-cont, and a zero-argument procedure representing its action under apply-handler.

Exercise 5.42 [**] The preceding exercise captures the continuation only when an exception is raised. Add to the language the ability to capture a continuation anywhere by adding the form letcc *Identifier* in *Expression* with the specification

```
(value-of/k (letcc var body) ρ cont)
= (value-of/k body (extend-env var cont ρ) cont)
```

Such a captured continuation may be invoked with throw: the expression throw *Expression* to *Expression* evaluates the two subexpressions. The second expression should return a continuation, which is applied to the value of the first expression. The current continuation of the throw expression is ignored.

Exercise 5.43 [**] Modify letcc as defined in the preceding exercise so that the captured continuation becomes a new kind of procedure, so instead of writing throw exp_1 to exp_2, one would write (exp_2 exp_1).

Exercise 5.44 [**] An alternative to letcc and throw of the preceding exercises is to add a single procedure to the language. This procedure, which in Scheme is called call-with-current-continuation, takes a one-argument procedure, p, and passes to p a procedure that when invoked with one argument, passes that argument to the current continuation, cont. We could define call-with-current-continuation in terms of letcc and throw as follows:

```
let call-with-current-continuation
      = proc (p)
          letcc cont
          in (p proc (v) throw v to cont)
in ...
```

Add call-with-current-continuation to the language. Then write a translator that takes the language with letcc and throw and translates it into the language without letcc and throw, but with call-with-current-continuation.

5.5 Threads

In many programming tasks, one may wish to have multiple computations proceeding at once. When these computations are run in the same address space as part of the same process, they are usually called *threads*. In this section, we will discover how to modify our interpreter to simulate the execution of multithreaded programs.

Rather than having a single thread of computation, our multithreaded interpreter will maintain several threads. Each thread consists of a computation in progress, like those shown earlier in this chapter. Threads communicate through a single shared memory, using assignment as in chapter 4.

In our system, the entire computation consists of a *pool* of threads. Each thread may be either *running*, *runnable*, or *blocked*. In our system, exactly one thread is running at a time. In a multi-CPU system, one might have several running threads. The runnable threads will be kept on a queue called the *ready queue*. There may be other threads that are not ready to be run, for one reason or another. We say that these threads are *blocked*. Blocked threads will be introduced later in this section.

Threads are scheduled for execution by a *scheduler*, which keeps the ready queue as part of its state. In addition, it keeps a timer, so that when a thread has completed a certain number of steps (its *time slice* or *quantum*), it is interrupted and put back on the ready queue, and a new thread is selected from the ready queue to run. This is called *pre-emptive scheduling*.

Our new language is based on IMPLICIT-REFS and is called THREADS. In THREADS, new threads are created by a construct called spawn. spawn takes one argument, which should evaluate to a procedure. A new thread is created, which, when run, passes an unspecified argument to that procedure. This thread is not run immediately, but is placed on the ready queue to be run when its turn arrives. spawn is executed for effect; in our system we have arbitrarily decided to have it return the number 73.

Let's look at two examples of programs in this language. Figure 5.16 defines a procedure noisy that takes a list, prints its first element and then recurs on the rest of the list. Here the main expression creates two threads, which compete to print out the lists [1, 2, 3, 4, 5] and [6, 7, 8, 9, 10]. The exact way in which the lists are interleaved depends on the scheduler; in this example each thread prints out two elements of its list before the scheduler interrupts it.

```
test: two-non-cooperating-threads

letrec
  noisy (l) = if null?(l)
              then 0
              else begin print(car(l)); (noisy cdr(l)) end
in
   begin
    spawn(proc (d) (noisy [1,2,3,4,5])) ;
    spawn(proc (d) (noisy [6,7,8,9,10])) ;
    print(100);
    33
   end

100
1
2
6
7
```

```
3
4
8
9
5
10
correct outcome: 33
actual outcome:  #(struct:num-val 33)
correct
```

Figure 5.16: Two threads showing interleaved computation

Figure 5.17 shows a producer and a consumer, connected by a buffer initialized to 0. The producer takes an argument n, goes around the wait loop

```
let buffer = 0
in let producer = proc (n)
        letrec
         wait(k) = if zero?(k)
                   then set buffer = n
                   else begin
                         print(-(k,-200));
                         (wait -(k,1))
                        end
        in (wait 5)
   in let consumer = proc (d)
           letrec busywait (k) = if zero?(buffer)
                                 then begin
                                       print(-(k,-100));
                                       (busywait -(k,-1))
                                      end
                                 else buffer
            in (busywait 0)
      in begin
          spawn(proc (d) (producer 44));
          print(300);
          (consumer 86)
         end

300
205
100
101
204
203
102
103
202
201
104
105
correct outcome: 44
actual outcome: #(struct:num-val 44)
correct
```

Figure 5.17: A producer and consumer, linked by a buffer

5 times, and then puts n in the buffer. Each time through the wait loop, it prints the countdown timer (expressed in 200s). The consumer takes an argument (which it ignores) and goes into a loop, waiting for the buffer to become non-zero. Each time through this loop, it prints a counter (expressed in 100s) to show how long it has waited for its result. The main thread puts the producer on the ready queue, prints 300, and starts the consumer in the main thread. So the first two items, 300 and 205, are printed by the main thread. As in the preceding example, the consumer thread and the producer thread each go around their loop about twice before being interrupted.

The implementation starts with a continuation-passing interpreter for the language IMPLICIT-REFS. This is similar to the one in section 5.1, with the addition of a store like the one in IMPLICIT-REFS (of course!) and a set-rhs-cont continuation builder like the one in exercise 5.9.

To this interpreter we add a scheduler. The scheduler keeps a state consisting of four values and provides six procedures in its interface for manipulating those values. These are shown in figure 5.18.

Internal State of the Scheduler	
the-ready-queue	the ready queue
the-final-answer	the value of the main thread, if done
the-max-time-slice	the number of steps that each thread may run
the-time-remaining	the number of steps remaining for the currently running thread.
Scheduler Interface	
initialize-scheduler!	: *Int* → *Unspecified*
	initializes the scheduler state
place-on-ready-queue!	: *Thread* → *Unspecified*
	places thread on the ready queue
run-next-thread	: () → *FinalAnswer*
	runs next thread. If no ready threads, returns the final answer.
set-final-answer!	: *ExpVal*→ *Unspecified*
	sets the final answer
time-expired?	:()→ *Bool*
	tests whether timer is 0
decrement-timer!	:()→ *Unspecified*
	decrements time-remaining

Figure 5.18: State and interface of the scheduler

Figure 5.19 shows the implementation of this interface. Here (enqueue *qval*) returns a queue like *q*, except that *val* has been placed at the end. (dequeue *q f*) takes the head of the queue and the rest of the queue and passes them to *f* as arguments.

```
initialize-scheduler! : Int → Unspecified
(define initialize-scheduler!
  (lambda (ticks)
    (set! the-ready-queue (empty-queue))
    (set! the-final-answer 'uninitialized)
    (set! the-max-time-slice ticks)
    (set! the-time-remaining the-max-time-slice)))

place-on-ready-queue! : Thread → Unspecified
(define place-on-ready-queue!
  (lambda (th)
    (set! the-ready-queue
      (enqueue the-ready-queue th))))

run-next-thread : () → FinalAnswer
(define run-next-thread
  (lambda ()
    (if (empty? the-ready-queue)
      the-final-answer
      (dequeue the-ready-queue
        (lambda (first-ready-thread other-ready-threads)
          (set! the-ready-queue other-ready-threads)
          (set! the-time-remaining the-max-time-slice)
          (first-ready-thread))))))

set-final-answer! : ExpVal → Unspecified
(define set-final-answer!
  (lambda (val)
    (set! the-final-answer val)))

time-expired? : () → Bool
(define time-expired?
  (lambda ()
    (zero? the-time-remaining)))

decrement-timer! : () → Unspecified
(define decrement-timer!
  (lambda ()
    (set! the-time-remaining (- the-time-remaining 1))))
```

Figure 5.19: The scheduler

We represent a thread as a Scheme procedure of no arguments that returns an expressed value:

Thread = () → *ExpVal*

If the ready queue is non-empty, then the procedure run-next-thread takes the first thread from the ready queue and runs it, giving it a new time slice of size the-max-time-slice. It also sets the-ready-queue so that it consists of the remaining threads, if any. If the ready queue is empty, then run-next-thread returns the contents of the-final-answer. This is how the computation eventually terminates.

We next turn to the interpreter. A spawn expression evaluates its argument in a continuation which, when executed, places a new thread on the ready queue and continues by returning 73 to the caller of the spawn. The new thread, when executed, passes an arbitrary value (here 28) to the procedure that was the value of the spawn's argument. To accomplish this, we add to value-of/k the clause

```
(spawn-exp (exp)
  (value-of/k exp env
    (spawn-cont cont)))
```

and to apply-cont the clause

```
(spawn-cont (saved-cont)
  (let ((proc1 (expval->proc val)))
    (place-on-ready-queue!
      (lambda ()
        (apply-procedure/k proc1
          (num-val 28)
          (end-subthread-cont))))
    (apply-cont saved-cont (num-val 73))))
```

This is what the trampolined interpreter did when it created a snapshot: it packaged up a computation (here (lambda () (apply-procedure/k ...))) and passed it to another procedure for processing. In the trampoline example, we passed the thread to the trampoline, which simply ran it. Here we place the new thread on the ready queue and continue our own computation.

This leads us to the key question: what continuation should we run each thread in?

- The main thread should be run with a continuation that records the value of the main thread as the final answer, and then runs any remaining ready threads.
- When the subthread finishes, there is no way to report its value, so we run it in a continuation that ignores the value and simply runs any remaining ready threads.

This gives us two new continuations, whose behavior is implemented by the following lines in apply-cont:

```
(end-main-thread-cont ()
  (set-final-answer! val)
  (run-next-thread))

(end-subthread-cont ()
  (run-next-thread))
```

We start the entire system with value-of-program:

```
value-of-program : Int × Program → FinalAnswer
(define value-of-program
  (lambda (timeslice pgm)
    (initialize-store!)
    (initialize-scheduler! timeslice)
    (cases program pgm
      (a-program (exp1)
        (value-of/k
          exp1
          (init-env)
          (end-main-thread-cont))))))
```

Last, we modify apply-cont to decrement the timer each time it is called. If the timer has expired, then the current computation is suspended. We do this by putting on the ready queue a thread that will try the apply-cont again, with the timer restored by some call to run-next-thread.

```
apply-cont : Cont × ExpVal → FinalAnswer
(define apply-cont
  (lambda (cont val)
    (if (time-expired?)
      (begin
        (place-on-ready-queue!
          (lambda () (apply-cont cont val)))
        (run-next-thread))
      (begin
        (decrement-timer!)
        (cases continuation cont
          ...)))))
```

Shared variables are an unreliable method of communication because several threads may try to write to the same variable. For example, consider the program in figure 5.20. Here we create three threads, each of which tries to increment the same counter x. If one thread reads the counter, but is interrupted before it can update it, then both threads will change the counter to the same number. Hence the counter may become 2, or even 1, rather than 3.

```
let x = 0
in let incr_x = proc (id)
                  proc (dummy)
                    set x = -(x,-1)
in begin
    spawn((incr_x 100));
    spawn((incr_x 200));
    spawn((incr_x 300))
   end
```

Figure 5.20: An unsafe counter

We would like to be able to ensure that interferences like this do not occur. Similarly, we would like to be able to organize our program so that the consumer in figure 5.17 doesn't have to busy-wait. Instead, it should be able to put itself to sleep and be awakened when the producer has inserted a value in the shared buffer.

There are many ways to design such a synchronization facility. A simple one is the *mutex* (short for *mutual exclusion*) or *binary semaphore*.

A mutex may either be *open* or *closed*. It also contains a queue of threads that are *waiting* for the mutex to become open. There are three operations on mutexes:

- mutex is an operation that takes no arguments and creates an initially open mutex.
- wait is a unary operation by which a thread indicates that it wants access to a mutex. Its argument must be a mutex. Its behavior depends on the state of the mutex.
    - If the mutex is closed, then the current thread is placed on the mutex's wait queue, and is suspended. We say that the current thread is *blocked* waiting for this mutex.
    - If the mutex is open, it becomes closed and the current thread continues to run.

  A wait is executed for effect only; its return value is unspecified.
- signal is a unary operation by which a thread indicates that it is ready to release a mutex. Its argument must be a mutex.
    - If the mutex is closed, and there are no threads waiting on its wait queue, then mutex becomes open and the current thread proceeds.
    - If the mutex is closed and there are threads in its wait queue, then one of the threads from the wait queue is put on the scheduler's ready queue, and the mutex remains closed. The thread that executed the signal continues to compute.
    - If the mutex is open, then the thread leaves it open and proceeds.

  A signal is executed for effect only; its return value is unspecified. A signal operation always succeeds: the thread that executes it remains the running thread.

These properties guarantee that only one thread can execute between a successive pair of calls to wait and signal. This portion of the program is called a *critical region*. It is impossible for two different threads to be concurrently excecuting code in a critical region. For example, figure 5.21 shows

```
let x = 0
in let mut = mutex()
in let incr_x = proc (id)
                  proc (dummy)
                   begin
                    wait(mut);
                    set x = -(x,-1);
                    signal(mut)
                   end
in begin
    spawn((incr_x 100));
    spawn((incr_x 200));
    spawn((incr_x 300))
   end
```

Figure 5.21: A safe counter using a mutex

our previous example, with a mutex inserted around the critical line. In this program, only one thread can execute the set x = -(x,-1) at a time, so the counter is guaranteed to reach the final value of 3.

We model a mutex as two references: one to its state (either open or closed) and one to a list of threads waiting for this mutex. We also make mutexes expressed values.

```
(define-datatype mutex mutex?
  (a-mutex
    (ref-to-closed? reference?)
    (ref-to-wait-queue reference?)))
```

We add the appropriate line to value-of/k

```
(mutex-exp ()
  (apply-cont cont (mutex-val (new-mutex))))
```

where

```
new-mutex : () → Mutex
(define new-mutex
  (lambda ()
    (a-mutex
      (newref #f)
      (newref '()))))
```

wait and signal will be new unary operations, which simply call the procedures wait-for-mutex and signal-mutex. wait and signal both evaluate their single argument, so in apply-cont we write

```
(wait-cont (saved-cont)
  (wait-for-mutex
    (expval->mutex val)
    (lambda () (apply-cont saved-cont (num-val 52)))))

(signal-cont (saved-cont)
  (signal-mutex
    (expval->mutex val)
    (lambda () (apply-cont saved-cont (num-val 53)))))
```

Now we can write wait-for-mutex and signal-mutex. These procedures take two arguments: a mutex and a thread, and they work as described in the text above (figure 5.22).

```
wait-for-mutex : Mutex × Thread → FinalAnswer
usage: waits for mutex to be open, then closes it.
(define wait-for-mutex
  (lambda (m th)
    (cases mutex m
      (a-mutex (ref-to-closed? ref-to-wait-queue)
        (cond
          ((deref ref-to-closed?)
```

```
            (setref! ref-to-wait-queue
              (enqueue (deref ref-to-wait-queue) th))
            (run-next-thread))
           (else
             (setref! ref-to-closed? #t)
             (th)))))))

signal-mutex : Mutex × Thread → FinalAnswer
(define signal-mutex
  (lambda (m th)
    (cases mutex m
      (a-mutex (ref-to-closed? ref-to-wait-queue)
        (let ((closed? (deref ref-to-closed?))
              (wait-queue (deref ref-to-wait-queue)))
          (if closed?
            (if (empty? wait-queue)
              (setref! ref-to-closed? #f)
              (dequeue wait-queue
                (lambda (first-waiting-th other-waiting-ths)
                  (place-on-ready-queue!
                    first-waiting-th)
                  (setref!
                    ref-to-wait-queue
                    other-waiting-ths)))))
          (th))))))
```

Figure 5.22: `wait-for-mutex` and `signal-mutex`

Exercise 5.45 [*] Add to the language of this section a construct called yield. Whenever a thread executes a yield, it is placed on the ready queue, and the thread at the head of the ready queue is run. When the thread is resumed, it should appear as if the call to yield had returned the number 99.

Exercise 5.46 [**] In the system of exercise 5.45, a thread may be placed on the ready queue either because its time slot has been exhausted or because it chose to yield. In the latter case, it will be restarted with a full time slice. Modify the system so that the ready queue keeps track of the remaining time slice (if any) of each thread, and restarts the thread only with the time it has remaining.

Exercise 5.47 [*] What happens if we are left with two subthreads, each waiting for a mutex held by the other subthread?

Exercise 5.48 [*] We have used a procedural representation of threads. Replace this by a data-structure representation.

Exercise 5.49 [*] Do exercise 5.15 (continuations as a stack of frames) for THREADS.

Exercise 5.50 [**] Registerize the interpreter of this section. What is the set of mutually tail-recursive procedures that must be registerized?

Exercise 5.51 [***] We would like to be able to organize our program so that the consumer in figure 5.17 doesn't have to busy-wait. Instead, it should be able to put itself to sleep and be awakened when the producer has put a value in the buffer. Either write a program with mutexes to do this, or implement a synchronization operator that makes this possible.

Exercise 5.52 [***] Write a program using mutexes that will be like the program in figure 5.21, except that the main thread waits for all three of the subthreads to terminate, and then returns the value of x.

Exercise 5.53 [***] Modify the thread package to include *thread identifiers*. Each new thread is associated with a fresh thread identifier. When the child thread is spawned, it is passed its thread identifier as a value, rather than the arbitrary value 28 used in this section. The child's number is also returned to the parent as the value of the spawn expression. Instrument the interpreter to trace the creation of thread identifiers. Check to see that the ready queue contains at most one thread for each thread identifier. How can a child thread know its parent's identifier? What should be done about the thread identifier of the original program?

Exercise 5.54 [**] Add to the interpreter of exercise 5.53 a kill facility. The kill construct, when given a thread number, finds the corresponding thread on the ready queue or any of the waiting queues and removes it. In addition, kill should return a true value if the target thread is found and false if the thread number is not found on any queue.

Exercise 5.55 [**] Add to the interpreter of exercise 5.53 an interthread communication facility, in which each thread can send a value to another thread using its thread identifier. A thread can receive messages when it chooses, blocking if no message has been sent to it.

Exercise 5.56 [**] Modify the interpreter of exercise 5.55 so that rather than sharing a store, each thread has its own store. In such a language, mutexes can almost always be avoided. Rewrite the example of this section in this language, without using mutexes.

Exercise 5.57 [***] There are lots of different synchronization mechanisms in your favorite OS book. Pick three and implement them in this framework.

Exercise 5.58 [definitely *] Go off with your friends and have some pizza, but make sure only one person at a time grabs a piece!