Chapter 9

Control

I shall be telling this with a sigh
Somewhere ages and ages hence:
Two roads diverged in a wood, and I —
I took the one less traveled by,
And that has made all the difference.

— The Road Not Taken, st. 4, Robert Frost

"Did he ever return, no he never returned
And his fate is still unlearned"

— MTA, performed by the Kingston Trio,
written by Bess Hawes & Jacqueline Steiner

9.1 Motivation: Control Contexts and Continuations

So far, we have studied two different kinds of contexts important in the evaluation of programming language expressions:

- A naming context that determines the meaning of free variable names within an expression.
- A state context that specifies the time-dependent behavior of mutable entities.

By objectifying both of these contexts as mathematical entities — environments and stores — the denotational approach provides significant leverage for us to investigate the space of language features that depend on these contexts. In the case of naming, environments help us to understand issues like parameter passing, scoping, and inheritance. In the case of state, stores help us to understand issues involving mutable variables and data structures.

There is a third major context that is still missing from our toolbox: a **control** context. Informally, control describes the path taken by a programmer's eyes and fingertips when hand-simulating the code in a listing. For example, when simulating a `while` or `for` loop in an imperative language, it is often necessary to refocus attention on the beginning of the loop after the end of the loop code is reached. Conditional expressions and procedure calls are other simple examples of control constructs that we have seen.

What does it mean for expressions to have a control *context*? As an example, consider the following FL! expression:

```
(let ((square (lambda (x) (* x x))))
  (+ (square 5) (* (+ 1 2) (square 5))))
```

There are two different occurrences of the `(square 5)` expression. What is the difference between them? Both are evaluated in the same environment and the same store, so they are guaranteed to yield the same value. What distinguishes them is how their value is used by the rest of the program. Reading from left to right, the first `(square 5)` returns 25 to a process that is collecting the first of two arguments to the procedural value of `+`. The second `(square 5)` yields its result to a process that is collecting the second of two arguments to the procedural value of `*`; this, in turn, is a subtask of the process that is collecting the second of two arguments to `+`, which itself is a subtask of the process that is waiting for the answer to the entire `let` expression. What distinguishes the occurrences of `(square 5)` is their control context: the part of the computation that remains to be done after the expression is evaluated.

The denotational descriptions we have employed so far have not explicitly represented the notion of "the rest of the computation." A denotational semantics without an explicit control model is said to be a **direct semantics**. A direct semantics for a programming language cannot deal elegantly with interruptions of the normal flow of control of a program. As long as valuation clauses are recursive in the obvious way, the flow of control in the clauses has no choice but to follow the structure of the program's parse tree.

A simple example of the limitation of direct semantics can be seen in its clumsy handling of error conditions in the languages that we have already encountered. An error is detected in one part of the semantics, and every other

part of the semantics must be able to cope with the possibility that some subexpression has produced an error instead of a normal result. This approach to error checking does not capture the intuition that a computation encountering an error immediately aborts without further processing. Abstractions like *with-value* help to hide this error checking, but they do not remove it. Indeed, interpreters based on the direct semantics of FL and its variants expend considerable effort performing such checks.

More generally, a direct semantics cannot easily explain constructs that interrupt the "normal" flow of control:

- *non-local exits* as provided by C's `break` and `continue` or COMMON LISP's `throw` and `catch`.
- *unrestricted jumps* permitted in numerous languages via `goto`.
- sophisticated *exception handling* as seen in CLU, ML, COMMON LISP, DYLAN, and JAVA.
- *coroutines* such as iterators in CLU and communicating sequential processes in many languages, notably OCCAM and even JAVA (JCSP).
- *backtracking,* which is used to model nondeterminism, e.g., to search a tree of possibilities, as in PROLOG and other logic programming languages.

In each of these cases, a program phrase does not simply return some value and/or an updated store, but instead bypasses the control context that invoked it and transfers control to some other place in the program.

The notion of **continuation** addresses this problem and provides a mathematical model of such transfers of control. A continuation is an entity that explicitly represents the "rest" of some computation. In implementation terms, it corresponds to the part of the machine state that comprises the current configuration of the runtime stack, together with a return address that specifies what code to run when the current computation returns a value. The continuation corresponding to the textually subsequent code in a program is usually referred to as the **normal continuation**. Many control constructs achieve their effect by substituting some other continuation for the normal one.

This chapter shows how continuations simplify the descriptions of the languages we have studied so far and allow the modeling of advanced control features in these languages. Be forewarned that control constructs are notoriously hard to think about. Even though many of the formal descriptions of control constructs are surprisingly concise, this does not imply that they are proportionately easier to understand. The often convoluted nature of control can lead

the reader into mental gymnastics that are likely to leave the brain a little bit sore at first. Luckily, with sufficient practice, the concepts can begin to seem natural.

To help build up some intuitions about continuations, we will first discuss how to achieve some sophisticated control behavior using only first class procedures. Then we will be better prepared to understand the use of continuations in denotational definitions.

9.2 Using Procedures to Model Control

We said before that continuations represent the rest of a computation. In a functional language, the continuation for an expression E is "waiting for" the value of E. It is therefore natural to think of continuations in a functional language as being procedures of one argument. For example, in the FL expression

```
(let ((square (lambda (x) (* x x))))
  (+ (square 5) (* (+ 1 2) (square 5))))
```

the continuation of the first `(square 5)` might be thought of as

```
(lambda (v1) (+ v1 (* (+ 1 2) (square 5))))
```

and the continuation for the second `(square 5)` might be thought of as

```
(lambda (v2) (+ 25 (* 3 v2)))
```

The above approximations indicate that operands to an FL application are evaluated in left-to-right order. When the first call to `square` is being evaluated, the second argument to `+` is the unevaluated `(* (+ 1 2) (square 5))`. But by the time the second call to `square` is evaluated, the first `(square 5)` has been evaluated to `25` and the `(+ 1 2)` argument has been evaluated to `3`.

Even in languages that do not support mutation, continuations require a computation to be viewed in a purely sequential way; some expressions are evaluated "before" other expressions. In fact, it is really control, not state, that must be linearly threaded through a sequential computation. State is just a piece of information carried along by the control in its linear walk. This separation of control and state makes it easier to think about sophisticated control constructs like backtracking, where a computation may revert to a previous state even though it is progressing in time.

First-class procedures are powerful enough to implement some fancy control behavior. In this section, we show how first-class procedures can be used to implement procedures returning multiple values, non-local exits, and coroutines.

9.2.1 Multiple-value Returns

It is often useful for a procedure to return more than one result. A classic example of the utility of multiple-value returns concerns integer division and remainder. Languages often provide two primitives for these operations even though the same algorithm computes both. It would make more sense to have a single operation that returns two values.

As another example, suppose that we want to write an FL program that, given a binary tree with integers as leaves, computes both the depth and the sum of the leaves in the tree and returns their product. One approach is to apply two different procedures to the tree and combine the results as in Figure 9.1. Notice that `depth*sum`$_1$ requires two walks over the given tree.

```
(define depth*sum1
  (lambda (tr)
    (letrec ((depth (lambda (tree)
                      (if (leaf? tree)
                          0
                          (+ 1 (max (depth (tree-left tree))
                                    (depth (tree-right tree)))))))
             (sum (lambda (tree)
                    (if (leaf? tree)
                        tree
                        (+ (sum (tree-left tree))
                           (sum (tree-right tree)))))))
      (* (depth tr) (sum tr)))))
```

Figure 9.1: The first version of `depth*sum` performs two tree traversals.

A procedure that returns multiple values allows one to perform the computation in a single tree walk. A simple method of doing this is to return a pair at each node of the tree as in Figure 9.2. However, the bundling and unbundling of values makes this approach to multiple values messy and hard to read.

An alternate approach to returning multiple values is to use first-class procedures. If procedure M is supposed to return multiple values, we can modify it to take an extra argument R[1], called the **receiver**. The receiver is a procedure that expects the multiple values as its arguments and will combine them into some result. M returns its results by calling R on them. We have already seen numerous examples of this strategy in metalanguage functions and interpreter procedures (e.g., *with-* functions have this form). Figure 9.3 shows how to apply this idea to our example.

[1]By convention, the extra argument usually comes last.

```
(define depth*sum2
  (lambda (tr)
    (letrec ((inner
              (lambda (tree)
                (if (leaf? tree)
                    (cons 0 tree)
                    (let ((depth&sum1 (inner (tree-left tree)))
                          (depth&sum2 (inner (tree-right tree))))
                      (cons (+ 1 (max (car depth&sum1)
                                      (car depth&sum2)))
                            (+ (cdr depth&sum1) (cdr depth&sum2)))))))))
      (let ((depth&sum (inner tr)))
        (* (car depth&sum) (cdr depth&sum))))))
```

Figure 9.2: The second version of `depth*sum` uses pairs to return multiple values.

```
(define depth*sum3
  (lambda (tr)
    (letrec ((inner
              (lambda (tree receiver)
                (if (leaf? tree)
                    (receiver 0 tree)
                    (inner
                      (tree-left tree)
                      (lambda (depth1 sum1)
                        (inner (tree-right tree)
                               (lambda (depth2 sum2)
                                 (receiver (+ 1 (max depth1 depth2))
                                           (+ sum1 sum2))))))))))
      (inner tr *))))
```

Figure 9.3: The third version of `depth*sum` passes multiple values to procedural continuations.

This style of code can be difficult to read. The `receiver` argument to the `inner` procedure acts as a continuation encoding what computation needs to be performed on the two values that `inner` "returns." For example, the call `(inner tr *)` starts off the process by applying `inner` to the tree `tr` with a receiver `*` that will take the two results and return their product.

Even though the receiver is an argument, it is typical to ignore its argument status and view it as a different entity when reading a call like `inner`. So `(inner` E_1 `(lambda (`I_1 I_2`)` E_2`))` can be read as "Call `inner` on E_1 and apply the procedure `(lambda (`I_1 I_2`)` E_2`)` to the results" or "Evaluate E_2 in an environment where I_1 and I_2 are bound to the two results of applying `inner` to E_1." Note that these readings treat `inner` as a procedure of one argument that returns two results, not a procedure of two arguments. Viewing continuation argument(s) as different entities from other arguments is important for getting a better working understanding of them.

Unlike the other two approaches, using a receiver forces us to choose a particular order for examining the branches of the binary tree. The main advantage of a receiver is that it allows the multiple returned values to be named using the standard naming construct, `lambda`. It is not necessary to invent a new syntax for naming intermediate values: `lambda` suffices.

As a concrete example, consider the following application of `depth*sum`$_3$:

```
(depth*sum3 '((5 7) (11 (13 17))))
```

An operational trace of the evaluation of this expression appears in Figures 9.4 and 9.5. Here, a tree node is represented by a list of the left and right subtrees, while a leaf is represented by an integer. Note how the continuation argument to `inner` acts like a stack that keeps track of the pending operations.

9.2.2 Non-local Exits

A continuation represents all the pending operations that are waiting to be done after the current operation. When continuations are implicit, the computation can only terminate successfully when all of the pending operations have been done. Yet we sometimes want a computation buried deep in pending operations to terminate immediately with a result or at least circumvent a number of pending operations. We can achieve these so-called **non-local exits** by using explicit procedure objects representing continuations.

For example, consider the task of multiplying a list of numbers. Figure 9.6 shows the natural recursive solution to this problem. E.g.,

```
(product-of-list1 (list 2 4 8)) ⟶eval 64
```

```
(depth*sum3 ((5 7) (11 (13 17))))
⇒ (inner ((5 7) (11 (13 17))) *)
⇒ (inner (5 7) (lambda (d1 s1)
                 (inner (11 (13 17)) (lambda (d2 s2)
                                       (* (+ 1 (max d1 d2))
                                          (+ s1 s2))))))
⇒ (inner 5 (lambda (d3 s3)
              (inner 7 (lambda (d4 s4)
                         ((lambda (d1 s1)
                            (inner (11 (13 17))
                              (lambda (d2 s2)
                                (* (+ 1 (max d1 d2)) (+ s1 s2)))))
                          (+ 1 (max d3 d4))
                          (+ s3 s4))))))
⇒ (inner 7 (lambda (d4 s4)
              ((lambda (d1 s1)
                 (inner (11 (13 17)) (lambda (d2 s2)
                                       (* (+ 1 (max d1 d2))
                                          (+ s1 s2)))))
               (+ 1 (max 0 d4))
               (+ 5 s4))))
⇒ ((lambda (d1 s1)
     (inner (11 (13 17)) (lambda (d2 s2)
                           (* (+ 1 (max d1 d2)) (+ s1 s2)))))
   (+ 1 (max 0 0))
   (+ 5 7))
⇒ (inner (11 (13 17)) (lambda (d2 s2)
                        (* (+ 1 (max 1 d2)) (+ 12 s2))))
⇒ (inner 11 (lambda (d5 s5)
              (inner (13 17) (lambda (d6 s6)
                               ((lambda (d2 s2)
                                  (* (+ 1 (max 1 d2)) (+ 12 s2)))
                                (+ 1 (max d5 d6))
                                (+ s5 s6))))))
```

Figure 9.4: Stylized operational trace of a procedural implementation of multiple-value return, part 1.

```
⇒ (inner (13 17) (lambda (d6 s6)
                   ((lambda (d2 s2)
                      (* (+ 1 (max 1 d2)) (+ 12 s2)))
                    (+ 1 (max 0 d6))
                    (+ 11 s6))))
⇒ (inner 13 (lambda (d7 s7)
               (inner 17 (lambda (d8 s8)
                           ((lambda (d6 s6)
                              ((lambda (d2 s2)
                                 (* (+ 1 (max 1 d2)) (+ 12 s2)))
                               (+ 1 (max 0 d6))
                               (+ 11 s6)))
                            (+ 1 (max d7 d8))
                            (+ s7 s8))))))
⇒ (inner 17 (lambda (d8 s8)
               ((lambda (d6 s6)
                  ((lambda (d2 s2)
                     (* (+ 1 (max 1 d2)) (+ 12 s2)))
                   (+ 1 (max 0 d6))
                   (+ 11 s6)))
                (+ 1 (max 0 d8))
                (+ 13 s8))))))
⇒ ((lambda (d6 s6)
      ((lambda (d2 s2)
         (* (+ 1 (max 1 d2)) (+ 12 s2)))
       (+ 1 (max 0 d6))
       (+ 11 s6)))
    (+ 1 (max 0 0))
    (+ 13 17))))))
⇒ ((lambda (d2 s2)
      (* (+ 1 (max 1 d2)) (+ 12 s2)))
    (+ 1 (max 0 1))
    (+ 11 30))
⇒ (* (+ 1 (max 1 2))
      (+ 12 41))
⇒ 159
```

Figure 9.5: Stylized operational trace of a procedural implementation of multiple-value return, part 2.

```
(define product-of-list1
    (lambda (nums)
        (if (null? nums)
            1
            (* (car nums) (product-of-list1 (cdr nums))))))
```

Figure 9.6: A first cut at the product of a list.

Figure 9.7 shows a continuized version of `product-of-list`$_1$. The behavior is exactly the same; we have just made the continuations explicit. For an empty list, `product-of-list`$_1$ continues with the value 1. For a non-empty list, we compute the product of the tail of the list passing along a new continuation that passes the product of the list tail and the current element to the current continuation.

```
(define product-of-list2
  (lambda (nums cont)
    (if (null? nums)
        (cont 1)
        (product-of-list2 (cdr nums)
                          (lambda (v)
                            (cont (* v (car nums))))))))
```

Figure 9.7: A continuized procedure for computing the product of a list.

Notice that `product-of-list`$_1$ and `product-of-list`$_2$ dutifully multiply all the elements of the list even if it contains a zero element. This is a waste since the answer is known to be 0 the moment a 0 is encountered. There is no need to look at any other list elements or to perform any more multiplications. `product-of-list`$_3$ in Figure 9.8 performs this optimization.

To accomplish a non-local exit, `product-of-list`$_3$ distinguishes the continuation passed to the initial call from continuations generated by recursive calls. The escape continuation is kept in `final-cont`. The local recursive procedure `prod` behaves like `product-of-list`$_2$ except that it jumps immediately to `final-cont` upon encountering a 0, thus avoiding unnecessary recursive calls by-passing all pending multiplications.

As a more complicated example, consider the pattern matching program for FL presented in Section 6.2.4.3. The core of the program is the `match-with-dict` procedure in Figure 9.9, where we have unraveled the failure abstractions to make the present discussion more concrete.

As written, `match-with-dict` performs a left-to-right depth-first walk simul-

```
(define product-of-list3
  (lambda (nums final-cont)
    (letrec ((prod
              (lambda (nums normal-cont)
                (if (null? nums)
                    (normal-cont 1)
                    (let ((thisnum (car nums)))
                      (if (= thisnum 0)
                          (final-cont 0)
                          (prod (cdr nums)
                                (lambda (val)
                                  (normal-cont
                                    (* val thisnum))))))))))
      (prod nums final-cont))))
```

Figure 9.8: Computing the product of list, exiting as soon as the answer is apparent.

```
(define match-sexp
  (lambda (pat sexp)
    (match-with-dict pat sexp (dict-empty))))

(define match-with-dict
  (lambda (pat sexp dict)
    (cond ((eq? dict '*failed*)       ; Propagate failures
           '*failed*)
          ((null? pat)
           (if (null? sexp)
               dict                   ; Pat and sexp both ended
               '*failed*))            ; Pat ended but sexp didn't
          ((null? sexp) '*failed*)    ; Sexp ended but pat didn't
          ((pattern-constant? pat)
           (if (sym=? pat sexp) dict '*failed*))
          ((pattern-variable? pat)
           (dict-bind (pattern-variable-name pat) sexp dict))
          (else (match-with-dict (cdr pat)
                                 (cdr sexp)
                                 (match-with-dict (car pat)
                                                  (car sexp)
                                                  dict))))))
```

Figure 9.9: Core of the pattern matching program.

taneously over the `pat` and `sexp` trees. It carries along a dictionary representing bindings for variables that have already been matched. Failure is represented in a rather ad hoc manner by replacing the dictionary with the symbol `*failed*`. Since failure may occur deep in the tree where many pending matches are waiting to be performed, each call of `match-with-dict` must check for and propagate failure tokens that appear as the dictionary argument.

It would be more desirable to handle failures by by-passing all the pending activations and simply returning the symbol `*failed*` as the value of `match-sexp`. This effect can be achieved by passing two extra arguments to `match-with-dict`: a success continuation and a failure continuation. A success continuation is a procedure of one argument, a dictionary, that continues a thus-far successful match with the given dictionary. A failure continuation is a procedure of no arguments that effectively returns `*failed*` for the initial call to `match-with-dict`. It is necessary to package up both continuations so that the program has the option of ignoring one. This strategy is implemented in Figure 9.10. Note in the final clause of the `cond`, `match-inner` works from the outside in while `match-with-dict` works from the inside out. This explains why the calls to `car` and `cdr` appear differently in the two programs, even though both walk the tree in a left-to-right depth-first manner.

In the modified version of the pattern matcher, the interface to `match-sexp` would be cleaner if it took success and failure continuations as well. Then we could more easily specify the behavior we want in these cases.

```
(define match-sexp
  (lambda (pat sexp succeed fail)
    (match-inner pat sexp (dict-empty) succeed fail)))

(match-sexp '((? a) (? a)) '(x x)
            (lambda (dict) #t)
            (lambda () #f))
```

$\xrightarrow[FL!]{}$ *true*

```
(match-sexp '((? a) (? a)) '(x y)
            (lambda (dict) #t)
            (lambda () #f))
```

$\xrightarrow[FL!]{}$ *false*

▷ **Exercise 9.1**

a. Modify the code in `product-of-list`$_3$ to return an error symbol if there is a non-integer element in the list.

b. Suppose the final continuation of `product-of-list`$_3$ must receive an integer value. How would you handle errors? Rewrite `product-of-list`$_3$ to incorpo-

```
(define match-sexp
  (lambda (pat sexp)
    (match-inner pat
                 sexp
                 (dict-empty)
                 (lambda (dict) dict)
                 (lambda () '*failed*))))

(define match-inner
  (lambda (pat sexp dict succeed fail)
    (cond ((null? pat)
           (if (null? sexp)
               (succeed dict)         ; Pat and sexp both ended
               (fail)))               ; Pat ended but sexp didn't
          ((null? sexp) (fail))       ; Sexp ended but pat didn't
          ((pattern-constant? pat)
           (if (sym=? pat sexp)
               (succeed dict)
               (fail)))
          ((pattern-variable? pat)
           (succeed
             (dict-bind (pattern-variable-name pat) sexp dict)))
          (else (match-inner (car pat)
                             (car sexp)
                             dict
                             (lambda (car-dict)
                               (match-inner (cdr pat)
                                            (cdr sexp)
                                            car-dict
                                            succeed
                                            fail))
                             fail)))))
```

Figure 9.10: A version of the pattern matcher that uses success and failure continuations.

rate your changes and show a sample call. ◁

9.2.3 Coroutines

Coroutining is a situation in which control jumps back and forth between conceptually independent processes. The most common version is producer/consumer coroutines, where a consumer process transfers control to a producer process when it wants the next value generated by the producer, and the producer returns control to the consumer along with the value. The standard example of this kind of coroutine is a compiler front end in which a parser requests tokens from the lexical scanner.

Here, we will show how simple producer/consumer coroutines can be implemented by using first-class procedures to represent control. The stream notion we will see in Chapter 10 (beginning on page 431) is an alternate technique for implementing such coroutines.

We represent a producer as a procedure that takes a consumer as its argument and hands that consumer the requested value along with the next producer. We represent a consumer as a procedure that takes a value and producer, and either returns or calls the producer on the next consumer.

For example, suppose `(count-from` n`)` makes a producer which generates the (conceptually infinite) increasing sequence of integers beginning with n, and `(add-first` m`)` makes a consumer that adds up the first m elements of the producer it's attached to. Then `((count-from 3) (add-first 5))` should return the sum of the integers from 3 to 7, inclusive. This example,coded in FL, is in Figure 9.11.

9.3 A Standard Semantics of FL!

To handle state in our semantics, we took the idiom of single-threading a store through a computation and made it part of the computational model. Similarly, we will handle control in our semantics by embedding in our computational model the idiom of passing continuations through a computation. The strategy of capturing common programming idioms in a semantic framework — or any language — is a powerful idea that lies at the foundation of programming language design. Indeed, languages can be considered expressive to the extent that they relieve the programmer of managing the details of common programming idioms.

Together, environments, stores, and continuations are sufficiently powerful to model most programming languages. As noted earlier, a semantics that uses

```
(define (count-from num)
  (letrec ((new-producer
            (lambda (n)
              (lambda (consumer)
                (consumer n (new-producer (+ n 1)))))))
    (new-producer num)))

(define (add-first count)
  (letrec ((new-consumer
            (lambda (c)
              (lambda (value next-producer)
                (if (= c 0)
                    0
                    (+ value (next-producer
                              (new-consumer (- c 1)))))))))
    (new-consumer count)))

;; Add up the 5 consecutive integers starting at 3
((count-from 3) (add-first 5)) ⟶FL 25
```

Figure 9.11: An example producer/consumer example.

only environments and stores is called a **direct semantics**. A semantics that adds continuations to a direct semantics is called a **standard semantics**, since most denotational definitions are written in this style. A standard semantics also implies particular conventions about the signatures of valuation functions. One advantage of standard semantics is that following a set of conventions simplifies the comparison of different programming languages defined by standard semantics. We already saw this kind of advantage when we studied parameter passing and scoping. Comparing different approaches was facilitated by the fact that the styles of the denotational definitions we were comparing were similar.

Now that we've built up some intuitions about continuations, it's time to model continuations explicitly in our denotational definitions. Figures 9.12–9.14 present the standard semantics for FLK!. *The definition given in the figures is just an alternate way to write the same semantics that we gave before.* In fact, there are mechanical transformations that could transform the denotational definition from the previous chapter into the definition in Figures 9.12–9.14.[2]

We introduce a standard semantics for FLK! because it is a much more powerful tool for studying control features than the direct semantics. In fact,

[2]Section 17.9 presents a mechanical transformation of FLAVAR! programs into **continuation passing style**.

$\gamma \in \mathit{Cmdcont} = \mathit{Store} \to \mathit{Answer}$
$k \in \mathit{Expcont} = \mathit{Value} \to \mathit{Cmdcont}$
$j \in \mathit{Explistcont} = \mathit{Value}^* \to \mathit{Cmdcont}$
$\mathit{Answer} =$ language dependent ; *Typically Expressible*
$p \in \mathit{Procedure} = \mathit{Denotable} \to \mathit{Expcont} \to \mathit{Cmdcont}$
$x \in \mathit{Expressible} = (\mathit{Value} + \mathit{Error})_\perp$; *As before*
$v \in \mathit{Value} =$ language dependent
$y \in \mathit{Error} = \mathit{Sym}$; *Modified*

New auxiliaries:

top-level-cont : *Expcont*
$= \lambda v\,.\ \lambda s\,.\ (\mathit{Value} \mapsto \mathit{Expressible}\ v)$; *Assume Answer = Expressible*

error-cont : *Error* $\to$ *Cmdcont*
$= \lambda I\,.\ \lambda s\,.\ (\mathit{Error} \mapsto \mathit{Expressible}\ I)$; *Assume Answer = Expressible*

test-boolean : (*Bool* $\to$ *Cmdcont*) $\to$ *Expcont*
$= \lambda f\ .\ (\lambda v\,.$ **matching** v
$\triangleright\, (\mathit{Bool} \mapsto \mathit{Value}\ b) \llbracket\ (f\ \ b)$
$\triangleright$ **else** (*error-cont* `non-boolean`)
endmatching)

Similarly for:
test-procedure : (*Procedure* $\to$ *Cmdcont*) $\to$ *Expcont*
test-location : (*Location* $\to$ *Cmdcont*) $\to$ *Expcont*
etc.

ensure-bound : *Binding* $\to$ *Expcont* $\to$ *Cmdcont*
$= \lambda \beta k\,.$ **matching** β
$\triangleright\, (\mathit{Denotable} \mapsto \mathit{Binding}\ v) \llbracket\ (k\ \ v)$; *Assume CBV*
$\triangleright\, (\mathit{Unbound} \mapsto \mathit{Binding}\ \mathit{unbound}) \llbracket$ (*error-cont* `unbound-variable`)
endmatching

Similarly for:
ensure-assigned : *Assignment* $\to$ *Expcont* $\to$ *Cmdcont*
ensure-value : *Expressible* $\to$ *Expcont* $\to$ *Cmdcont*

Figure 9.12: Semantic algebras for standard semantics of FLK!.

$\mathcal{TL} : \text{Exp} \to \mathit{Answer}$; *Assume Answer = Expressible*
$\mathcal{E} : \text{Exp} \to \mathit{Environment} \to \mathit{Expcont} \to \mathit{Cmdcont}$
$\mathcal{E}^* : \text{Exp}^* \to \mathit{Environment} \to \mathit{Explistcont} \to \mathit{Cmdcont}$
$\mathcal{L} : \text{Lit} \to \mathit{Value}$; *Defined as usual*
$\mathcal{Y} : \text{Symlit} \to \mathit{Sym}$; $Y \in$Symlit are symbolic literals

$$\mathcal{TL}[\![E]\!] = (\mathcal{E}[\![E]\!]\ \mathit{empty\text{-}env}\ \mathit{top\text{-}level\text{-}cont}\ \mathit{empty\text{-}store})$$

$$\mathcal{E}[\![L]\!] = \lambda ek\,.\ (k\ \mathcal{L}[\![L]\!])$$

$$\mathcal{E}[\![I]\!] = \lambda ek\,.\ (\mathit{ensure\text{-}bound}\ (\mathit{lookup}\ e\ I)\ k)$$

$$\mathcal{E}[\![\texttt{(proc}\ I\ E\texttt{)}]\!] = \lambda ek_1\,.\ (k_1\ (\mathit{Procedure} \mapsto \mathit{Value}\ (\lambda \delta k_2\,.\ (\mathcal{E}[\![E]\!]\ [I:\delta]e\ k_2))))$$

$$\mathcal{E}[\![\texttt{(call}\ E_1\ E_2\texttt{)}]\!] = \lambda ek\,.\ (\mathcal{E}[\![E_1]\!]\ e\ (\mathit{test\text{-}procedure}\ (\lambda p\,.\ (\mathcal{E}[\![E_2]\!]\ e\ (\lambda v\,.\ (p\ v\ k))))))$$

$$\mathcal{E}[\![\texttt{(if}\ E_1\ E_2\ E_3\texttt{)}]\!] = \lambda ek\,.\ (\mathcal{E}[\![E_1]\!]\ e\ (\mathit{test\text{-}boolean}\ (\lambda b\,.\ \textbf{if}\ b\ \textbf{then}\ (\mathcal{E}[\![E_2]\!]\ e\ k)\ \textbf{else}\ (\mathcal{E}[\![E_3]\!]\ e\ k)\ \textbf{fi})))$$

$$\mathcal{E}[\![\texttt{(pair}\ E_1\ E_2\texttt{)}]\!] = \lambda ek\,.\ (\mathcal{E}[\![E_1]\!]\ e\ (\lambda v_1\,.\ (\mathcal{E}[\![E_2]\!]\ e\ (\lambda v_2\,.\ (k\ (\mathit{Pair} \mapsto \mathit{Value}\ \langle v_1, v_2\rangle))))))$$

$$\mathcal{E}[\![\texttt{(cell}\ E\texttt{)}]\!] = \lambda ek\,.\ (\mathcal{E}[\![E]\!]\ e\ (\lambda vs\,.\ (k\ (\mathit{Location} \mapsto \mathit{Value}\ (\mathit{fresh\text{-}loc}\ s))\ (\mathit{assign}\ (\mathit{fresh\text{-}loc}\ s)\ v\ s))))$$

$$\mathcal{E}[\![\texttt{(begin}\ E_1\ E_2\texttt{)}]\!] = \lambda ek\,.\ (\mathcal{E}[\![E_1]\!]\ e\ (\lambda v_{ignore}\,.\ (\mathcal{E}[\![E_2]\!]\ e\ k)))$$

$$\mathcal{E}[\![\texttt{(primop}\ O\ E^*\texttt{)}]\!] = \lambda ek\,.\ (\mathcal{E}^*[\![E^*]\!]\ e\ (\lambda v^*\,.\ (\mathcal{P}_{FLK!}[\![O]\!]\ v^*\ k)))$$

$$\mathcal{E}[\![\texttt{(error}\ I\texttt{)}]\!] = \lambda ek\,.\ (\mathit{error\text{-}cont}\ I)$$

$$\mathcal{E}^*[\![[\,]]\!] = \lambda ej\,.\ (j\ [\,]_{\mathit{Value}})$$

$$\mathcal{E}^*[\![E_{\mathit{first}}\ .\ E_{\mathit{rest}}{}^*]\!] = \lambda ej\,.\ (\mathcal{E}[\![E_{\mathit{first}}]\!]\ e\ (\lambda v\,.\ (\mathcal{E}^*[\![E_{\mathit{rest}}]\!]\ e\ (\lambda v^*\,.\ (j\ v\ .\ v^*)))))$$

Figure 9.13: Valuation clauses for standard semantics of FLK!, Part I.

$\mathcal{P}_{FLK!}$: Primop $\rightarrow$ *Value** $\rightarrow$ *Expcont* $\rightarrow$ *Cmdcont*
$\mathcal{P}_{FLK}$: Primop $\rightarrow$ *Value** $\rightarrow$ *Expressible* ; *Defined as usual*

$\mathcal{P}_{FLK!}[\![\texttt{cell-ref}]\!]$
$= \lambda[v]ks\,.\ ((\textit{test-location}\ (\lambda ls'\,.\ ((\textit{ensure-assigned}\ (\textit{fetch}\ l\ s')\ k)\ s')))$
$v\ s)$

$\mathcal{P}_{FLK!}[\![\texttt{cell-set!}]\!]$
$= \lambda[v_1, v_2]ks\,.\ ((\textit{test-location}\ (\lambda ls'\,.\ (k\ (\textit{Unit} \mapsto \textit{Value}\ \textit{unit})\ (\textit{assign}\ l\ v_2\ s'))))$
$v_1\ s)$

$\mathcal{P}_{FLK!}[\![O_{FLK}]\!] = \lambda v^* k\,.\ (\textit{ensure-value}\ (\mathcal{P}_{FLK}[\![O_{FLK}]\!]\ v^*)\ k)$
where $O_{FLK} \in$Primop $- \{\texttt{cell-ref}, \texttt{cell-set!}\}$

Figure 9.14: Valuation clauses for standard semantics of FLK!, Part II.

the area of control is the big payoff for our investment in denotational semantics; many control constructs that have succinct denotational descriptions are difficult to describe in an operational framework.

The standard semantics for FLK! differs from the direct semantics for FLK! in the following ways:

- The *Expressible* domain has been replaced by the *Answer* domain. In a standard semantics, the *Answer* domain is used to represent the "final" value of a program. Not all standard semantics actually return a value for an expression. For example, the initial continuation might be an interpreter's read-eval-print loop, which never returns. In this case, the behavior of the program could be modeled as a mapping from a sequence of input s-expressions to a sequence of output s-expressions. Nevertheless, in the particular case of FLK!, *Answer* is the same as *Expressible*.

- The standard semantics introduces two continuation domains, *Expcont* and *Cmdcont*:

  $k \in \textit{Expcont} = \textit{Expressible} \rightarrow \textit{Cmdcont}$
  $\gamma \in \textit{Cmdcont} = \textit{Store} \rightarrow \textit{Answer}$

  Expcont is the domain of expression continuations; *Cmdcont* is the domain of command continuations. These types of continuations reflect a common distinction in programming languages between **commands** and **expressions**. An expression yields a value in addition to any modifications it

might make to the store. The example languages in this book are *expression languages* because all program constructs are expressions that return a value. Many languages, e.g., PASCAL, have syntactically distinct expressions and commands as well as contexts that require one or the other.[3]

A command, on the other hand, is executed for its effect(s) and does not produce a meaningful value. Program output is the classic example of a command: `writeln` in PASCAL, for example, prints its arguments as a line of output on the standard output device. Variable assignment is also naturally thought of as a command. In FLAVAR!, `set!` expressions return the uninteresting value `#u` simply because they are required to return something, but the reason to execute an assignment is to modify the store. Sequencing using `begin` is a natural command context: it exists to enforce an order of state transformations.

Since expressions return a value and modify the store, the continuation for an expression expects both the value and store produced by that expression. A command continuation expects only a store. Note that because $Cmdcont = Store \rightarrow Answer$, we can also view *Expcont* as:

$k \in Expcont = Expressible \rightarrow Store \rightarrow Answer$

That is, we can think of an expression continuation as taking an expressible value and returning a command continuation; or we may think of it as taking an expressible value and a store and returning an answer. Which perspective is more fruitful depends on the situation.

- The signature of $\mathcal{E}$ has been modified:

  $\mathcal{E} : \text{Exp} \rightarrow Environment \rightarrow Expcont \rightarrow Cmdcont$

  Recall that since $Cmdcont = Store \rightarrow Answer$ we can also view $\mathcal{E}$ as:

  $\mathcal{E} : \text{Exp} \rightarrow Environment \rightarrow Expcont \rightarrow Store \rightarrow Answer$

  That is, $\mathcal{E}$ takes a syntactic expression and representations of the naming (*Environment*), control (*Expcont*), and state (*Store*) contexts, and finds the meaning of the expression (an answer) with respect to these contexts.

[3] It is possible to coerce an expression to a command by ignoring its return value. This is what FL does with all but the final subexpression in a `begin`.

An expression of the form

$$(\mathcal{E}[\![E]\!]\ e\ (\lambda v\ .\ \Box))$$

can be read as "Find the value of E in e and name the result v in $\Box$."

Since evaluating an expression requires a store in FLK!, why doesn't a store appear in the above expression? The reason is that the order of arguments to $\mathcal{E}$ has been chosen to take the store as its final argument, rather than the continuation. This argument order is one of the conventions of a standard semantics; it is used because it hides the store when it is threaded through an expression untouched. In essence, *Cmdcont* fulfills the role that the *Computation* domain did when we introduced state into FL. To specify that an expression takes in one store, say s_0, and returns another, s_1, we write:

$$(\mathcal{E}[\![E]\!]\ e\ (\lambda v s_1\ .\ \Box)\ s_0)$$

(Observe the explicit store parameters in the denotations for constructs involving cells.)

- The definition of the *Procedure* domain is changed to take an expression continuation:

  $p \in \mathit{Procedure} = \mathit{Denotable} \to \mathit{Expcont} \to \mathit{Cmdcont}$

  Again, we can also view this definition as:

  $p \in \mathit{Procedure} = \mathit{Denotable} \to \mathit{Expcont} \to \mathit{Store} \to \mathit{Answer}$

  The *Procedure* domain in the standard semantics differs from the *Procedure* domain in the direct semantics in that procedures take an additional argument from the *Expcont* domain. Intuitively, this argument is the "return address" that a procedure returns to when it returns a value.

- The new *test-xxx* auxiliary functions are used to convert continuations expecting arguments of type *xxx* into expression continuations. Like the *with-xxx* functions from previous semantics, the *test-xxx* functions hide details of error generation. However, unlike the *with-xxx* functions, the *test-xxx* functions do not propagate errors.

Even though FL! does not have any advanced control features (we'll add quite a few in the remainder of this chapter), the standard semantics still has an advantage over a direct semantics: the modelling of errors. A valuation clause in a standard semantics generates an error by ignoring the current continuation and directly returning an error. See, for example, the valuation clause for `error`. This captures the intuition that an error immediately aborts the computation.

The standard semantics valuation clauses in Figures 9.13 and 9.14 do not employ the computation abstraction that we have been using in our denotational definitions. We presented them in a concrete manner to help build intuitions about continuations. However, it is not difficult to recast standard semantics into the computation framework. Figures 9.15 and 9.16 show an implementation of the computation abstraction that defines *Computation* as *Expcont* $\rightarrow$ *Store* $\rightarrow$ *Expressible*. (We assume that *Answer* is *Expressible*, but we could readily redefine *Answer* to be another domain.) With this implementation of the computation abstraction, the FLK! valuation clauses from the previous chapter still hold, except for a minor tweak in the handling of CBV `rec`:

$\mathcal{E}[\![$`(rec` I E`)`$]\!]$
$= \lambda e\,.\ \mathbf{fix}_{Computation}\ (\lambda c\,.\ \lambda ks_0\,.\ (\mathcal{E}[\![E]\!]\ [I :: (\textit{extract-value}\ c\ s_0)]e\ k\ s_0))$

$\textit{extract-value} : \textit{Computation} \rightarrow \textit{Store} \rightarrow \textit{Binding}$
$= \lambda cs_0\,.\ \mathbf{matching}\,(c\ (\lambda vs\,.\,(\textit{Value} \mapsto \textit{Expressible}\ v))\ s_0)$
$\quad \triangleright\,(\textit{Value} \mapsto \textit{Expressible}\ v)\ [\!]\ (\textit{Denotable} \mapsto \textit{Binding}\ v)$
$\quad \triangleright\,(\textit{Error} \mapsto \textit{Expressible}\ y)\ [\!]\ \bot_{Binding}$
$\quad$ **endmatching**

There are two important changes in the valuation clause for `rec`:

- *extract-value* must account for the fact that the *Answer* domain is *Expressible* rather than *Expressible* $\times$ *Store*.
- The new $\mathcal{E}$ function requires a continuation argument, which we use to hijack the value used in the binding for I. Notice that this continuation is rather like the top level continuation in Figure 9.12.

Figure 9.17 introduces two continuation-specific auxiliary functions along with their associated reasoning laws. Since no FLK! construct does anything interesting with a continuation, these auxiliaries would not appear in valuation clauses for FLK!. However, they will be useful when we extend FLK! with advanced control features.

$\triangleright$ **Exercise 9.2** Imperative Languages Inc. was impressed with the simplicity and power of FL!. Noticing that it lacks a looping construct and not willing to support a product not in consonance with the company's programming language philosophy, the company calls Ben Bitdiddle to extend the language. Instead of a myriad of different constructs (e.g. `for`, `while`, `do-while`, etc.) Ben designs a single loop expression that embodies all forms of looping. The syntax of FLK! was extended as follows:

$E ::= \ldots$	[As before]
| `(loop` E`)`	[Evaluate E repeatedly]
| `(break` E`)`	[End lexically enclosing loop with value of E]
| `(continue)`	[Restart evaluation of enclosing loop expression]

$c \in \mathit{Computation} = \mathit{Expcont} \to \mathit{Store} \to \mathit{Expressible}$
$k \in \mathit{Expcont} = \mathit{Value} \to \mathit{Store} \to \mathit{Expressible}$
$x \in \mathit{Expressible} = (\mathit{Value} + \mathit{Error})_\perp$; *As before*
$v \in \mathit{Value} =$ *language dependent*
$y \in \mathit{Error} = \mathit{Sym}$; *Modified*

expr-to-comp : $\mathit{Expressible} \to \mathit{Computation}$
$= \lambda x\,.$ **matching** x
$\quad \triangleright\, (\mathit{Value} \mapsto \mathit{Expressible}\ v) \,[\!]\, (\textit{val-to-comp}\ v)$
$\quad \triangleright\, (\mathit{Error} \mapsto \mathit{Expressible}\ y) \,[\!]\, (\textit{err-to-comp}\ y)$
$\quad$ **endmatching**

val-to-comp : $\mathit{Value} \to \mathit{Computation}\ = \lambda v\,.\ \lambda k\,.\ (k\ v)$

err-to-comp : $\mathit{Error} \to \mathit{Computation}\ = \lambda I\,.\ \lambda ks\,.\ (\mathit{Error} \mapsto \mathit{Expressible}\ I)$

with-value : $\mathit{Computation} \to (\mathit{Value} \to \mathit{Computation}) \to \mathit{Computation}$
$= \lambda cf\ .\ \lambda k\,.\ (c\ (\lambda v\,.\,(f\ \ v\ k)))$
with-values, *with-boolean*, *with-procedure*, etc. can be written in terms of *with-value*.

check-location : $\mathit{Value} \to (\mathit{Location} \to \mathit{Computation}) \to \mathit{Computation}$
$= \lambda vf\ .$ **matching** v
$\quad \triangleright\, (\mathit{Location} \mapsto \mathit{Value}\ l) \,[\!]\, (f\ \ l)$
$\quad \triangleright$ **else** (*err-to-comp* `non-location`)
$\quad$ **endmatching**
check-boolean, *check-procedure*, etc. are similar.

Figure 9.15: Continuation-based computation abstraction, Part I.

State-based auxiliaries:

allocating : *Storable* → (*Location* → *Computation*) → *Computation*
$= \lambda\sigma f\ .\ \lambda ks\ .\ (f\ \ (\textit{fresh-loc}\ s)\ \ k\ \ (\textit{assign}\ \ (\textit{fresh-loc}\ s)\ \ \sigma\ \ s))$

fetching : *Location* → (*Storable* → *Computation*) → *Computation*
$= \lambda lf\ .\ \lambda ks\ .$ **matching** (*fetch* $l\ s$)
▷ (*Storable* ↦ *Assignment* σ) ‖ $(f\ \ \sigma\ \ k\ \ s)$
▷ **else** ((*err-to-comp* `unassigned-location`) $k\ s$)
endmatching

update : *Location* → *Storable* → *Computation*
$= \lambda l\sigma\ .\ \lambda ks\ .\ (k\ \ (\textit{Value} \mapsto \textit{Expressible}\ \ (\textit{Unit} \mapsto \textit{Value}\ \ \textit{unit}))\ \ (\textit{assign}\ \ l\ \ \sigma\ \ s))$

sequence : *Computation* → *Computation* → *Computation*
$= \lambda c_1\, c_2\ .\ (\textit{with-value}\ \ c_1\ \ (\lambda v\ .\ c_2))$; *Unchanged from before*

Figure 9.16: Continuation-based computation abstraction, Part II.

New auxiliary functions for control:

capturing-cont : (*Expcont* → *Computation*) → *Computation*
$= \lambda f\ .\ \lambda k\ .\ ((f\ \ k)\ \ k)$

install-cont : *Expcont* → *Computation* → *Computation*
$= \lambda k_{new}\, c\ .\ \lambda k_{old}\ .\ (c\ \ k_{new})$

New Reasoning Laws:

$(\textit{with-value}\ \ (\textit{install-cont}\ \ k\ \ c)\ \ f) = (\textit{install-cont}\ \ k\ \ c)$

$(\textit{with-value}\ \ (\textit{capturing-cont}\ \ f)\ \ g) = (\textit{capturing-cont}\ \ (\lambda k\ .\ (\textit{with-value}\ \ (f\ \ k)\ \ g)))$
where k is not free in f or g.

$(\textit{capturing-cont}\ \ (\lambda k\ .\ (\textit{install-cont}\ \ k\ \ c))) = c$
where k is not free in c.

Figure 9.17: Continuation-specific auxiliary functions on computations.

Here's the informal semantics of the new constructs:

a. `(loop` *E*`)`: Evaluates *E* (the "looping expression") repeatedly forever.

b. `(break` *E*`)`: Ends the nearest lexically enclosing `loop`, which then returns the value of *E*.

c. `(continue)`: Restarts the evaluation of the looping expression for the nearest lexically enclosing `loop`.

It is an error to evaluate either a `break` or a `continue` expression outside a lexically enclosing `loop` expression.

Here's an example of Ben's looping constructs in action:

```
; Ben's iterative factorial procedure
(lambda (n)
  (let ((fact 1))
    (loop
      (if (= n 0)
        (break fact)
        (begin
          (set! fact (* fact n))
          (set! n (- n 1)))))))
```

In addition to extending the language, Ben has been asked to extend its standard denotational semantics. It is here that Ben has subcontracted you.

a. Give the new signature of the meaning function $\mathcal{E}$.

b. Give the meaning function clauses for `(loop` *E*`)`, `(break` *E*`)`, and `(continue)`.

◁

▷ **Exercise 9.3** Ben Bitdiddle is very excited about the power of the standard semantics to describe complicated flows of control. Wanting to practice more with this wonderful tool, he started churning out a lot of FL! extensions (not all of them useful). Most recently, Ben added a construct `(self` *E*`)` in order to allow a procedure to call itself, without using `rec` or `letrec`. Ben modified the FLK! grammar as follows:

E ::= ... `existing` FLK! `constructs` ...
| `(self` *E*`)`

Informally, `(self` *E*`)` recursively calls the current procedure with an actual argument that is the result of evaluating *E*. Here is a small example:

```
(let ((fact (lambda (n) (if (= n 0) 1 (* n (self (- n 1)))))))
     (fact 4)) ⟶ 24
                eval
```

When `(self` *E*`)` is used outside a procedure, it causes the program to terminate immediately with a value that is the result of evaluating *E*.

Ben started describing the formal semantics of `(self` *E*`)` by modifying the signature of the meaning function $\mathcal{E}$ as follows:

$\mathcal{E}$: Exp → *Environment* → *SelfProc* → *Expcont* → *Cmdcont*
where *SelfProc* = *Procedure*

In spite of his enthusiasm, Ben is still inexperienced with standard semantics. It is your task to help him specify the formal semantics of the `self` construct.

a. Give the revised definition of the top level meaning function $\mathcal{TL}[\![E]\!]$.

b. Give the meaning function clause for `(call` E_1 E_2`)`, `(self` E`)` and `(proc` I E`)`.

c. Prove that `(self (self 1))` evaluates to (*Value* ↦ *Expressible* (*Int* ↦ *Value* 1)).

d. Prove that `(proc x (self 1))` evaluates to a procedure that, no matter what input it is called with, loops forever. ◁

▷ **Exercise 9.4** Ben Bitdiddle is now working in a major research university where he's investigating a new approach to programming based on coroutines. Of course, he bases his research on FLK! and its standard semantics. He adds the following expressions to the FLK! grammar:

E ::= `...`| `(coroutine (`I E_1`)` E_2`)`| `(yield` E`)`

The meaning of the expression `(coroutine (`I E_1`)` E_2`)` is simply E_2, unless E_2 performs a `(yield` E_3`)`. If E_2 performs a `(yield` E_3`)`, then the value of the coroutine expression is simply E_1, except that I is bound to E_3. However, if E_1 performs a `(yield` E_4`)`, then control transfers back to E_2, with the value of the original `(yield` E_3`)` — the point where control was originally transferred to E_1 — being replaced by E_4. Thus, `yield` transfers control back and forth between the two expressions, passing a value between them. A `(yield` E`)` in one expression transfers control to the other expression; that expression resumes at the point of its last `yield`, whose value is set to E.

The following example coroutine expressions may help to make things clear. The underline mark shows the active expression; the series of dots ····· shows a `yield` expression that has already yielded control to the other expression.

```
(coroutine (x 1) 2)
⇒ 2

(coroutine (x 1) (yield 2))
⇒ (coroutine (x 1) ·····)
⇒ 1

(coroutine (x x) (yield 2))
⇒ (coroutine (x x) ·····)
⇒ (coroutine (x 2) ·····)
⇒ 2
```

```
(coroutine (x (yield (+ 1 x))) (yield 2))
⇒ (coroutine (x (yield (+ 1 x))) ·····)
⇒ (coroutine (x (yield (+ 1 2))) ·····)
⇒ (coroutine (x (yield 3)) ·····)
⇒ (coroutine (x ·····) 3)
⇒ 3

(coroutine (x (yield (+ 1 x))) (+ 2 (yield 3)))
⇒ (coroutine (x (yield (+ 1 x))) (+ 2 ·····))
⇒ (coroutine (x (yield (+ 1 3))) (+ 2 ·····))
⇒ (coroutine (x (yield 4)) (+ 2 ·····))
⇒ (coroutine (x ·····) (+ 2 4))
⇒ 6
```

As one of Ben's students, your job is to write the denotational semantics for FLK! + coroutines. Ben has already revised the domain equations to include both normal and yield continuations, as follows:

$k \in \mathit{Normal\text{-}Cont} = \mathit{Expcont}$
$y \in \mathit{Yield\text{-}Cont} = \mathit{Expcont}$
$\mathit{Expcont} = \mathit{Value} \to \mathit{Cmdcont}$
$\mathit{Cmdcont} = \mathit{Yield\text{-}Cont} \to \mathit{Store} \to \mathit{Expressible}$
$p \in \mathit{Procedure} = \mathit{Denotable} \to \mathit{Normal\text{-}Cont} \to \mathit{Cmdcont}$

He's also changed the signature of the $\mathcal{E}$ meaning function so that every expression is evaluated with both a normal and a yield continuation:

$$\begin{aligned}\mathcal{E} \;&:\; \mathrm{Exp} \to \mathit{Environment} \to \mathit{Normal\text{-}Cont} \to \mathit{Cmdcont}\\ &= \mathrm{Exp} \to \mathit{Environment} \to \mathit{Normal\text{-}Cont} \to \mathit{Yield\text{-}Cont} \to \mathit{Store} \to \mathit{Expressible}\end{aligned}$$

He didn't get that far when defining $\mathcal{E}$, but he did give you the meaning function clause for `(if` E_1 E_2 E_3`)` for reference.

$$\begin{aligned}&\mathcal{E}[\![\texttt{(if } E_1\ E_2\ E_3\texttt{)}]\!] =\\ &\quad \lambda e k y\,.\ (\mathcal{E}[\![E_1]\!]\ e\\ &\qquad\qquad (\mathit{test\text{-}boolean}\ \ (\lambda b\,.\,\mathbf{if}\ b\ \ \mathbf{then}\ (\mathcal{E}[\![E_2]\!]\ e\ k)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathbf{else}\ \ (\mathcal{E}[\![E_3]\!]\ e\ k)))\\ &\qquad\qquad y)\end{aligned}$$

a. Give the meaning function clause for L, given the new domains.

b. Give the meaning function clause for `(yield` E`)`.

c. Give the meaning function clause for `(coroutine (`I E_1`)` E_2`)`.

d. Compute the meaning of `(yield (yield 3))` according to your semantics. ◁

▷ **Exercise 9.5** Alyssa P. Hacker thinks coroutines (see Section 9.2.3) make programs too hard to reason about. She suggests a simplified version of coroutines: the producer/consumer paradigm. Informally, a producer generates values one at a time, and

the values are used by a consumer in the order they are produced. Alyssa modifies the FLK! grammar as follows:

E ::= ... *normal* FLK! *constructs* ...
| `(producer` I_{yield} E_{body}`)`
| `(consume` $E_{producer}$ $I_{current}$ E_{body}`)`

The informal semantics of these two newly added constructs are:

- `(producer` I_{yield} E_{body}`)` creates a new kind of first-class object called a *producer*. When a producer is invoked (by `consume`) the identifier I_{yield} is bound to a yielding procedure. Calling I_{yield} in E_{body} with a value passes control and the yielded value to the consumer. When the consumer is done processing the value, the I_{yield} procedure returns `#u` to the producer. The value of E_{body} is the value returned by the producer when there are no more values to yield.
- `(consume` $E_{producer}$ $I_{current}$ E_{body}`)` invokes the producer that $E_{producer}$ evaluates to (it is an error if $E_{producer}$ doesn't evaluate to a producer). Whenever the producer yields a value, $I_{current}$ is bound to that value and E_{body} is evaluated. The result of evaluating E_{body} is then discarded, and control returns to the producer. The result returned by `consume` is the result returned by the producer.

For example, `up-to` is a procedure that takes an integer `n` as an argument, and returns a producer that yields the integers from `1` up to and including `n`.

```
(define up-to
  (lambda (n)
    (producer emit
      (letrec ((loop (lambda(i)
                       (if (> i n)
                           #f
                           (begin (emit i)
                                  (loop (+ i 1)))))))
        (loop 1)))))
```

The `sum` procedure adds all the numbers yielded by a given producer:

```
(define sum
  (lambda (prod)
    (let ((ans (cell 0)))
      (begin
        (consume prod n (cell-set! ans (+ n (cell-ref ans))))
        (cell-ref ans)))))
```

For example, `sum` can be used to add up the values produced by the producer `(up-to 5)`:

`(sum (up-to 5))` $\xrightarrow[eval]{}$ *33*

Note that when a producer does not yield additional values and returns a normal value v, execution of the invoking `consume` form is terminated and v is returned.

`(consume (up-to 5) i 7)` $\xrightarrow[eval]{}$ *false*

Assume in the following questions that FL! is desugared in the usual way to FLK!.

a. Alyssa wants to update the standard semantics of FLK! in order to specify the formal semantics of the newly introduced constructs. Alyssa starts by creating a new domain for producers, and adds it to the value domain:

$$v \in \mathit{Value} = \mathit{Unit} + \mathit{Bool} + \ldots + \mathit{Procedure} + \mathit{Producer}$$
$$q \in \mathit{Producer} = \mathit{Procedure} \rightarrow \mathit{Expcont} \rightarrow \mathit{Cmdcont}$$

Alyssa's valuation clause for the `consume` construct is as follows:

$$\begin{array}{l}\mathcal{E}[\![\texttt{(consume } E_{producer}\ I_{current}\ E_{body}\texttt{)}]\!] = \\ \quad \lambda e k_{normal} \,.\, (\mathcal{E}[\![E_{producer}]\!]\ \ e \\ \qquad\qquad (\mathit{test\text{-}producer} \\ \qquad\qquad (\lambda q \,.\, (q\ \ (\lambda v_{yielded} k_{after-yield} \,. \\ \qquad\qquad\qquad\qquad (\mathcal{E}[\![E_{body}]\!]\ \ [I_{current} : v_{yielded}]e \\ \qquad\qquad\qquad\qquad\quad (\lambda v \,.\ (k_{after-yield}\ \ (\mathit{Unit} \mapsto \mathit{Value}\ \ \mathit{unit}))))) \\ \qquad\qquad\qquad\qquad k_{normal}))))\end{array}$$

$$\begin{array}{l}\mathit{test\text{-}producer} : (\mathit{Producer} \rightarrow \mathit{Cmdcont}) \rightarrow \mathit{Expcont} = \\ \quad \lambda f \,.\, (\lambda v \,.\, \textbf{matching}\ v \\ \qquad\qquad \triangleright\, (\mathit{Producer} \mapsto \mathit{Value}\ \ q)\ [\![\ \ (f\ \ q) \\ \qquad\qquad \triangleright\, \textbf{else}\ \mathit{error\text{-}cont} \\ \qquad\qquad \textbf{endmatching}\ )\end{array}$$

Write the evaluation clause for the `producer` construct.

b. Alyssa also decides to specify the behavior of `producer` and `consume` in terms of their operational semantics. She starts with the SOS semantics for FLK! from Section 8.2.3.

State the modifications to the SOS semantics of FLK! that are necessary to handle `producer` and `consumer`, including any relevant rules.

c. Ben Bitdiddle discovers how to desugar Alyssa's constructs into normal FL! constructs. Ben's desugaring for `producer` is

$$\mathcal{D}\texttt{(producer } I_{yield}\ \ E_{body}\texttt{)} = \texttt{(lambda (}I_{yield}\texttt{)}\ \ E_{body}\texttt{)}$$

Write a corresponding desugaring for `consume`. ◁

▷ **Exercise 9.6** Sam Antics is aggressively using the standard semantics to define the meaning of some really non-standard FL! constructs. Most recently, he extended FL! with some special constructs for POP (i.e., "Politically Oriented Programming"). He extended the FLK! grammar as follows:

$$\begin{array}{rl}E ::= & \ldots \texttt{ existing FLK! constructs } \ldots \\ & \mid \texttt{(elect } E_{pres}\ \ E_{vp}\texttt{)} \\ & \mid \texttt{(reelect)} \\ & \mid \texttt{(impeach)}\end{array}$$

Here's the informal semantics of the newly introduced constructs:

- `(elect` E_{pres} E_{vp}`)` evaluates to the value of E_{pres} unless `(impeach)` is evaluated in E_{pres}, in which case it evaluates to the value of E_{vp}. It is possible to have nested `elect` constructs.
- If `(reelect)` is evaluated inside the E_{pres} part of a `(elect` E_{pres} E_{vp}`)` construct, it goes back to the beginning of the `elect` construct. Otherwise, it signals an error.
- If `(impeach)` is evaluated within the E_{pres} part of a `(elect` E_{pres} E_{vp}`)` construct, it causes the `elect` expression to evaluate to E_{vp}. Otherwise, it signals an error.

Here's a small example that Sam plans to use in his advertising campaign for the FL! *2000 Presidential Edition (TM)*:

```
(let ((scandals (cell 0)))
   (elect (if (< (cell-ref scandals) 5)
              (begin (cell-set! scandals
                                (+ (cell-ref scandals) 1))
                     (reelect))
              (impeach))
          (* (cell-ref scandals) 2)))
```

$\xrightarrow[eval]{}$ *10*

You are hired by Sam Antix to modify the standard denotational semantics of FLK! in order to define the formal semantics of the newly introduced constructs. Sam has already added the following semantic domains:

$r \in \mathit{Prescont} = \mathit{Cmdcont}$
$i \in \mathit{Vpcont} = \mathit{Cmdcont}$

He also changed the signature of the meaning function:

$\mathcal{E} : \mathrm{Exp} \to \mathit{Environment} \to \mathit{Prescont} \to \mathit{Vpcont} \to \mathit{Expcont} \to \mathit{Cmdcont}$

a. Give the definition of the top level meaning function $\mathcal{TL}[\![E]\!]$.

b. Give the meaning function clauses for $\mathcal{E}[\![$`(elect` E_{pres} E_{vp}`)`$]\!]$, `(reelect)`, and `(impeach)`.

c. Use the meaning functions you defined to compute $\mathcal{TL}[\![$`(elect (reelect 1))`$]\!]$.

◁

▷ **Exercise 9.7** This problem requires you to modify the standard denotational semantics for FLK!.

Sam Antics is working on a new language with hot new features that will appeal to government customers. He was going to base his language on Caffeine from Moon

Microsystems, but negotiations broke down. He has therefore decided to extend FLK! and has hired you, a top FLK! consultant, to assist with modifying the language to support these new features. The new language is called FLK#, part of Sam Antics' new .GOV platform. The big feature of FLK# is user tracking and quotas in the store. An important customer observed that government users tended to use the store carelessly, resulting in expensive memory upgrades. To improve the situation, the FLK# store will maintain a per-user quota. The Standard Semantics of FLK! are changed as follows:

$$\begin{aligned} w \in \mathit{UserID} &= \mathit{Int} \\ q \in \mathit{Quota} &= \mathit{UserID} \to \mathit{Int} \\ \mathit{gamma} \in \mathit{Cmdcont} &= \mathit{UserID} \to \mathit{Quota} \to \mathit{Store} \to \mathit{Answer} \end{aligned}$$

UserID is just an integer. 0 is reserved for the case when no one is logged in. *Quota* is a function that when given a *UserID* returns the number of cells remaining in the user's quota. The quota starts at 100 cells. *Cmdcont*, the command continuation, takes the currently logged in user ID, the current quota, and the current store to yield an answer. Plus, FLK# adds the following commands:

E ::= `...` [Classic FLK! expressions]
| `(login!` w`)` [Log in user w]
| `(logout!)` [Log out current user]

`(login!`w`)` — logs in the user associated with the identifier; returns the identifier (returns an error if a user is already logged in or if the UserID is 0)
`(logout!)` — logs the current user out; returns the last user's identifier (returns an error if there is no user logged in)
`(check-quota)` — returns the amount of quota remaining

The definition of $\mathcal{E}[\![$`(check-quota)`$]\!]$ is:

$$\begin{aligned} &\mathcal{E}[\![\texttt{(check-quota)}]\!] = \\ &\quad \lambda ekwq\,.\ \textbf{if}\ w = 0 \\ &\qquad\quad \textbf{then}\ (\mathit{error\text{-}cont}\ \texttt{no-user-logged-in}\ \ w\ \ q) \\ &\qquad\quad \textbf{else}\ (\mathrm{k}\ (\mathit{Int} \mapsto \mathit{Value}\ (q\ \ w))\ w\ \ q)\ \ \textbf{fi} \end{aligned}$$

a. Write the meaning function clause for $\mathcal{E}[\![$`(login!` E`)`$]\!]$.

b. Write the meaning function clause for $\mathcal{E}[\![$`(logout!)`$]\!]$.

c. Give the definition of $\mathcal{E}[\![$`(cell` E`)`$]\!]$. Remember you cannot create a cell unless you are logged in.

d. Naturally, Sam Antics wants to embed some "trap doors" into the .GOV platform to enable him to "learn more about his customers." One of these trap doors is the undocumented `(raise-quota! n)` command, which adds `n` cells to the quota of the current user and returns `0`. Give the definition of $\mathcal{E}[\![$`(raise-quota!` E`)`$]\!]$. ◁

9.4 Non-local Exits

A denotational semantics equipped with continuations is especially useful for modelling advanced control features of programming languages. One such feature is a **non-local exit**, a mechanism that aborts a pending computation by forcing control to jump to a specified place in the program.

To study non-local exits, we extend FLK! with two new constructs:

E ::= ... | `(label` I_{ctrl_pt} E_{body}`)` | `(jump` E_{ctrl_pt} E_{body}`)`

The informal semantics of these constructs is as follows:

- `(label` I_{ctrl_pt} E_{body}`)` evaluates E_{body} in an environment where the name I_{ctrl_pt} is bound to the **control point** that receives the value of the `label` expression.
- `(jump` E_{ctrl_pt} E_{body}`)` returns the value of E_{body} to the control point that is the value of E_{ctrl_pt}. If E_{ctrl_pt} does not evaluate to a control point, `jump` generates an error.

```
(+ 1 (label exit (* 2 (- 3 (/ 4 1))))) ⟶FL!⟶ -1

(+ 1 (label exit (* 2 (- 3 (/ 4 (jump exit 5)))))) ⟶FL!⟶ 6

(+ 1 (label exit
       (* 2 (- 3 (/ 4 (jump exit (+ 5 (jump exit 6)))))))) ⟶FL!⟶ 7

(+ 1 (label exit1
       (* 2 (label exit2 (- 3 (/ 4 (+ (jump exit2 5)
                                      (jump exit1 6)))))))) ⟶FL!⟶ 11
```

Figure 9.18: Some examples using `label` and `jump`.

Figure 9.18 shows some simple examples using `label` and `jump`. The first example illustrates that the value of `(label` I E`)` is the value of E if E performs no `jump`s. In the second example, `(jump exit 5)` aborts the pending `(* 2 (- 3 (/ 4 □)))` computation and returns `5` as the value of the `label` expression. The third example demonstrates that a pending `jump` can itself be aborted by a `jump` within one of its subexpressions. In the final example, the left-to-right evaluation of `call` subexpressions causes `5` to be returned as the value of `(label exit2 ...)`. If `call` subexpressions were evaluated in right-to-left order, the result of the final example would be `7`.

In practice, non-local exits are a convenient means of communicating information between two points of a program separated by pending operations without performing any of the pending operations. For instance, here is yet another version of a recursive procedure for computing the product of a list of numbers (see Section 9.2.2):

```
(define (prod-list a-list)
  (label return
    (letrec ((prod (lambda (lst)
                     (cond ((null? lst) 1)
                           ((= 0 (car lst)) (jump return 0))
                           (else (* (car lst)
                                    (prod (cdr lst))))))))
      (prod a-list))))
```

Upon encountering a `0` in the list, the internal `prod` procedure uses `jump` to immediately return `0` as the result of a call to `prod-list`. Any pending multiplications generated by recursive calls to `prod` are flushed.

The semantics for `label` and `jump` appear in Figure 9.19. Control points are modelled as expression continuations that are treated as first-class values. The valuation clauses for `label` and `jump` are presented in two styles: the traditional style of standard semantics, and a style based on the computation abstraction. `label` redefines its continuation k as a control point value and evaluates E_{body} in the environment e extended with a binding between I_{ctrl_pt} and the control point value. `jump` ignores its default continuation and instead evaluates E_{body} with the continuation determined by E_{ctrl_pt}.

Note that `label` refers to its continuation twice: it both names it and uses it as the continuation of E_{body}. (This is easier to see in the standard style than in the computation style.) This means that a value can be returned from a `label` expression in two ways: (1) by normal evaluation of E_{body} (without any `jump`s) and (2) by using `jump` with a control point that is extracted from the environment. In contrast, `jump` does not refer to its continuation at all. This means that a `jump` expression can never return! So it is meaningless to ask what the value of a `jump` expression is. Similarly, expressions containing `jump` expressions may also have no value. This is the first time we have seen expressions without values in a dialect of FL.

Like all other values in FL!, control point values are first-class: they can be named, passed as arguments, returned as results, and stored in data structures (pairs, cells). An interesting consequence of this fact is that it is possible to return to the same control point more than once. Consider the following FL! expression:

Abstract Syntax:

$$
\begin{array}{lll}
E ::= \ldots & & \text{[As before]} \\
\quad | \ \texttt{(label}\ I_{name}\ \ E_{body}\texttt{)} & & \text{[Label]} \\
\quad | \ \texttt{(jump}\ E_{control-point}\ \ E_{val}\texttt{)} & & \text{[Jump]}
\end{array}
$$

Semantic Domains:

$$
\begin{array}{l}
\quad \mathit{ControlPoint} = \mathit{Expcont} \\
v \in \mathit{Value} = \ldots + \mathit{ControlPoint}
\end{array}
$$

$$\mathit{test\text{-}control\text{-}point} : (\mathit{ControlPoint} \to \mathit{Cmdcont}) \to \mathit{Expcont}$$

Valuation functions (standard version):

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(label}\ I_{ctrl\_pt}\ \ E_{body}\texttt{)}]\!] \\
\quad = \lambda ek\ .\ (\mathcal{E}[\![E_{body}]\!]\ [I_{ctrl\_pt} : (\mathit{ControlPoint} \mapsto \mathit{Value}\ \ k)]e\ \ k)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(jump}\ E_{ctrl\_pt}\ \ E_{val}\texttt{)}]\!] \\
\quad = \lambda ek_{ignore}\ .\ (\mathcal{E}[\![E_{ctrl\_pt}]\!]\ \ e\ \ (\mathit{test\text{-}control\text{-}point}\ (\lambda k_{ctrl\_pt}\ .\ (\mathcal{E}[\![E_{val}]\!]\ \ e\ \ k_{ctrl\_pt}))))
\end{array}
$$

Valuation functions (computation version):

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(label}\ I_{ctrl\_pt}\ \ E_{body}\texttt{)}]\!] \\
\quad = \lambda e\ .\ (\mathit{capturing\text{-}cont}\ (\lambda k\ .\ (\mathcal{E}[\![E_{body}]\!]\ [I_{ctrl\_pt} : (\mathit{ControlPoint} \mapsto \mathit{Value}\ \ k)]e)))
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(jump}\ E_{ctrl\_pt}\ \ E_{val}\texttt{)}]\!] \\
\quad = \lambda e\ .\ (\mathit{with\text{-}control\text{-}point} \\
\qquad\quad (\mathcal{E}[\![E_{ctrl\_pt}]\!]\ \ e)\ (\lambda k_{ctrl\_pt}\ .\ (\mathit{install\text{-}cont}\ \ k_{ctrl\_pt}\ \ (\mathcal{E}[\![E_{val}]\!]\ \ e))))
\end{array}
$$

Figure 9.19: The semantics of `label` and `jump` in FLK!.

```
(let ((c (cell 'later)))
  (let ((n (label bind-n
             (begin (cell-set! c bind-n)
                    1))))
    (if (> n 17)
        n
        (jump (cell-ref c) (* 2 n))))) ⟶_{FL!} 32
```

Here, `bind-n` names the control point that: (1) accepts a value, (2) binds the value to `n`, and (3) evaluates the `if` expression. This control point is stashed away in the cell `c` for later use, and then a 1 is returned to the normal flow of control. Since this value for `n` is less than 17, the `jump` is performed, which returns the value of 2 to the same `bind-n` control point. This causes `n` to be rebound to 2 and the `if` expression to be evaluated a second time. Continuing in this manner, the expression behaves like a loop that successively binds `n` to the values 1, 2, 4, 8, 16, and 32. The final result is 32 because that is the first power of two that is greater than 17.

A similar trick can be used to phrase an imperative version of an iterative factorial procedure in terms of `label` and `jump`:

```
(define factorial
  (lambda (n)
    (let ((loop (cell 'later))
          (num (cell n))
          (ans (cell 1)))
      (begin
        (label top (cell-set! loop (lambda ()
                                     (jump top 'ignore))))
        (if (= (cell-ref num) 0)
            (cell-ref ans)
            (begin
              (cell-set! ans (* (cell-ref num) (cell-ref ans)))
              (cell-set! num (- (cell-ref num) 1))
              ((cell-ref loop)))))))))
```

It turns out that mutation is not necessary for exhibiting this sort of looping behavior via `label` and `jump` (see Exercise 9.9).

The above examples of first-class continuations (i.e., control points) are rather contrived. However, in languages that support them (such as SCHEME and some dialects of ML), first-class continuations provide a powerful mechanism by which programmers can implement advanced control features. For instance, coroutines, backtracking, and multi-threading can all be implemented in terms of first-class continuations. But any control abstraction mechanism

this powerful can easily lead to programs that are virtually impossible to understand. After all, it turns the notion of `goto`-less programming on its head by making `goto` labels first-class values! Thus, great restraint should be exercised when using first-class continuations.

In SCHEME, first-class continuations are made accessible by the standard procedure `call-with-current-continuation`, which we will abbreviate as `cwcc` (another common abbreviation is `call/cc`.) This procedure can be written in terms of `label` and `jump` as follows:

```
(define (cwcc proc)
  (label here
    (proc (lambda (val) (jump here val)))))
```

The `proc` argument is a unary procedure that is applied to an **escape procedure** that, when called, will return a result from the call to `cwcc`. Here is a version of `prod-list` written in terms of `cwcc`.

```
(define (prod-list a-list)
  (cwcc
    (lambda (return)
      (letrec ((prod (lambda (lst)
                       (cond ((null? lst) 1)
                             ((= 0 (car lst)) (return 0))
                             (else (* (car lst)
                                      (prod (cdr lst))))))))
        (prod a-list)))))
```

The advantage of `cwcc` as an interface to capturing continuations is that it does not require extending a language with any new special forms. The binding performed by `label` is instead handled by the usual binding mechanism (`lambda`), and a `jump` is encoded as a procedure call.

Some languages put restrictions on capturable continuations that make them easier to reason about and to implement. For example, the DYLAN language provides a `(bind-exit (`*I*`)` *E*`)` form that is similar to `(cwcc (lambda (`*I*`)` *E*`))` except that the lifetime of the escape procedure is limited by the lifetime of the `bind-exit` form. The `catch` and `throw` constructs of COMMON LISP are similar to `label` and `jump` except that `throw` jumps to a named control point declared by a dynamically enclosing `catch`. Dynamically declared control points are a good mechanism for exception handling, which is our next topic of study.

▷ **Exercise 9.8** What are the values of the following expressions? (Assume `prod-list` is defined as above.)

a. `(prod-list '(2 3 4))`

b. `(prod-list '(2 0 yow!))`

c.
```
(prod-list '(yow! 0 2))
```

d.
```
(let ((twice (lambda (f x) (f (f x)))))
  (let ((f (label bind-f (lambda (new-f)
                           (jump bind-f new-f)))))
    ((f twice) (+ 1) 0)))
```

e.
```
(jump (label a a) (label b b))
```
◁

▷ **Exercise 9.9** It is possible to implement loops with `label` and `jump` without using mutation. As an example, here is a template for an iterative factorial procedure in FL + {`label`, `jump`} (recall that FL does not support mutation):

```
(define (factorial n)
  (let ((triple E_triple))
    (let ((loop (first triple))
          (num  (second triple))
          (acc  (third triple)))
      (if (= num 0)
          acc
          (loop (list loop (- num 1) (* acc num)))))))
```

(Assume that `first`, `second`, and `third` are the appropriate list accessing procedures.) Using `label` and `jump`, write an expression E_{triple} such that `factorial` behaves as advertised. ◁

▷ **Exercise 9.10** Chris Krenshall[4] is dissatisfied with FLK!+{`label`,`jump`}. He's never sure where his thread of control will end up! Therefore, Chris would like you to give him some control over his control points. Chris wants to have *applets* — syntactically distinguished regions of code across which control points cannot be used. Here are the proposed FLK! extensions:

E ::= ... | `(applet` I E`)` | `(label` I E`)` | `(jump` E_1 E_2`)`

Informally, the `label` and `jump` constructs work as described above: `label` establishes first class control points and `jump` transfers control to them. However, there is one important difference, related to the `applet` construct: it is only legal to jump to control points created by the current applet, which is determined by the identifier of the nearest *lexically enclosing* `applet`.

For example, the following program is legal and evaluates to 0.

[4]Recall the C. Krenshall Program for eliminating concurrency from government programming contracts.

```
(applet hot
  (let ((p (applet cool
             (proc x
               (label cool-return
                 (+ (if (= x 1)
                        (jump cool-return 0)
                        x)
                    7))))))
    (p 1))) ⟶eval 0
```

On the other hand, the following program should signal an error:

```
(applet hot
  (let ((p (proc x
             (label hot-return
               (applet cool
                 (+ (if (= x 1)
                        (jump hot-return 0)
                        x)
                    7))))))
    (p 1))) ⟶eval error
```

In this problem you will modify the standard semantics for FLK! to specify the semantics of the `applet`, `label`, and `jump` constructs.

a. Suppose we define an *ControlPoint* domain and modify the *Value* domain accordingly:

$$q \in \mathit{ControlPoint} = \mathit{Applet} \times \mathit{Expcont}$$
$$a \in \mathit{Applet} = \text{Identifier}$$
$$v \in \mathit{Value} = \mathit{ControlPoint} + \ldots$$

Modify the signature of $\mathcal{E}$ as necessary to support applets.

b. Give a new definition for *top-level*. In your semantics, use the special applet identifer *global-applet* $\in$ *Applet* no applet has been defined for a `label` or `jump`.

c. Give the meaning function clause for `(applet` I E`)`.

d. Give the meaning function clause for `(label` I E`)`.

e. Give the meaning function clause for `(jump` E_1 E_2`)`. ◁

▷ **Exercise 9.11** This exercise explores the semantics of `cwcc` in more detail:

a. We have shown that `cwcc` can be written in terms of `label` and `jump`. Show how `label` and `jump` can be desugared in a language that provides `cwcc`.

b. Write a standard style valuation clause for the `cwcc` primitive.

c. Write a computation style valuation clause for the `cwcc` primitive. ◁

.

9.5 Exception Handling

A common reason to alter the usual flow of control in a program is to respond to exceptional conditions. For example, upon encountering a divide-by-zero error, the caller of the division procedure may want the computation to proceed with a large number rather than terminate with an error. Dynamically responding to exceptional conditions is known as **exception handling**.

One strategy for exception handling is for every procedure to return values that are tagged with a **return code** that indicates whether the procedure is returning normally or in some exceptional way. The caller can then test for the return code and handle the situation accordingly. Although popular, the return code technique is unsatisfactory in many ways. For one, it effectively requires every call to a procedure to explicitly test for all return codes the procedure could potentially generate. By treating normal and exceptional returns in the same fashion, return codes fail to capture the notion that exceptions are generally perceived as rare events compared to normal returns. In addition, return codes provide a very limited way in which to respond to exceptional conditions. All responsibility for dealing with the condition resides in the caller; in particular, the point at which the condition was generated has been lost.

An alternate way to view exceptional conditions is that procedures can **raise** (or **signal**) an **exception** as an alternative to returning a value. The immediate caller may then **handle** the exception, or it might decline to handle the exception and instead allow other callers in the current call chain to handle the exception. There are two basic strategies for handling the exception:

1. In **termination semantics**, the handler receives control from the signaler of the exception and keeps it. This is the approach taken by ML's `raise` and `handle`, Common Lisp's `throw` and `catch`, and CLU's `signal` and `except when`.

2. In **resumption semantics**, the handler receives control from the signaler of the exception but later passes control back to the computation that raised the exception. Operating system traps usually follow this model.

Some languages (such as CLU) require the caller to explicitly resignal user exceptions to propagate them up the call chain. In other languages, unhandled exceptions propagate up the call chain until an appropriate handler is found. In these languages, programs are implicitly wrapped in a default handler that handles otherwise uncaught exceptions.

As a concrete example of exception handling, we extend FLK! to accommodate a rudimentary resumption-style exception facility:

E ::= ... | `(raise` I_{except} E_{val}`)` | `(trap` I_{except} $E_{handler}$ E_{body}`)`

The informal semantics of these constructs is as follows:

- `(raise` I_{except} E_{val}`)` raises an exception named I_{except} with argument E_{val}.

- `(trap` I_{except} $E_{handler}$ E_{body}`)` evaluates E_{body} in such a way that if a `raise` of I_{except} is encountered during the evaluation of E_{body}, the value of the `raise` form is the result of applying the **handler procedure** computed by $E_{handler}$ to the argument supplied by the `raise`. If there is more than one handler with the same name, the one associated with the nearest dynamically enclosing `trap` is used. The value of the `trap` form is the value returned by E_{body}. If $E_{handler}$ does not designate a procedure, `trap` generates an error.

As an example of exception handling, consider an FL! `add` procedure that normally returns the sum of its two arguments, but raises a `non-integer` exception if one of its arguments is not an integer:

```
(define add
  (lambda (x y)
    (let ((check-integer
           (lambda (a)
             (if (integer? a) a (raise non-integer a)))))
      (+ (check-integer x) (check-integer y)))))
```

(Even better, we could change the semantics of the `+` primitive to raise exceptions rather than generate errors.) Now suppose we use `add` within a procedure that sums the elements of a list:

```
(define sum-list
  (lambda (lst)
    (if (null? lst)
        0
        (add (car lst) (sum-list (cdr lst))))))

(sum-list '(1 2 3)) ⟶ 6
```

If we call `sum-list` on a list containing non-integer elements, we can use `trap` to specify how these elements should be handled. For example, here is a handler that treats false as 0, true as 1, and all symbols as 10; elements that are not booleans or symbols abort with an error:

```
(define simple-handler
  (lambda (x)
    (cond ((boolean? x) (if x 1 0))
          ((symbol? x) 10)
          (else (error not-boolean-or-symbol)))))

(trap non-integer simple-handler
  (sum-list '(5 yes #t)))
```
$\xrightarrow[eval]{}$ *16*

```
(trap non-integer simple-handler
  (sum-list '(5 (yes no) #t)))
```
$\xrightarrow[eval]{}$ $error : not - boolean - or - symbol$

We will assume that exceptions not handled by any dynamically enclosing `trap`s are converted into errors by a default top-level exception handler; e.g.:

```
(sum-list '(5 #t yes))
```
$\xrightarrow[eval]{}$ $error : non - integer$

While the informal semantics for `raise` and `trap` given above may seem like an adequate specification, it harbors some ambiguities. For example, what should be the result of the following program?

```
(trap a (lambda (x) (+ 4000 x))
  (trap b (lambda (x) (+ 300 (raise a (+ x 4))))
    (trap a (lambda (x) (+ 20 x))
      (+ 1 (raise b 2)))))
```

The `raise` of `b` invokes a handler that raises the exception `a`. But which of the two `a` handlers should be used?

Once again, formal semantics comes to the rescue. In fact, because complex control constructs can easily befuddle our intuitions, we look more than ever to the guidance of formal semantics. Standard semantics is an excellent tool for precisely wiring down the meaning of complex control constructs like `raise` and `trap`.

Our approach is to treat `trap` as a binding construct that associates an exception name with an exception handler in a dynamic environment. An exception handler is just a procedure. `raise` looks up the handler associated with the given exception name in the current dynamic environment and applies the resulting procedure to the argument of `raise`.

To express these extensions formally, we will modify the standard semantics of FLK!. Figures 9.20 and 9.21 summarize the changes needed to accommodate `raise` and `trap`. Exception handlers are represented as procedure values that are named in a special environment, *Handler-Env*. Augmenting computations with this handler environment treats them as dynamic (as opposed to lexical) environments. That is, the domain *Procedure*, which is *Denotable* $\rightarrow$

$c \in \mathit{Computation} = \mathit{Handler\text{-}Env} \to \mathit{Expcont} \to \mathit{Cmdcont}$
$w \in \mathit{Handler\text{-}Env} = \text{Identifier} \to \mathit{Procedure}$
$p \in \mathit{Procedure} = \mathit{Denotable} \to \mathit{Computation}$; *As usual*

New auxiliaries for handler environments:

$\mathit{extend\text{-}handlers} : \mathit{Handler} - \mathit{Env} \to \text{Identifier} \to \mathit{Procedure} \to \mathit{Handler} - \mathit{Env}$
$= \lambda w I_1\, p\,.\ \lambda I_2\,.\ \textbf{if}\ (\mathit{same\text{-}identifier?}\ I_1\ I_2)\ \textbf{then}\ p\ \textbf{else}\ (w\ I_2)\ \textbf{fi}$

$\mathit{get\text{-}handler} : \mathit{Handler} - \mathit{Env} \to \text{Identifier} \to \mathit{Procedure}\ = \lambda w I\,.\,(w\ I)$

$\mathit{default\text{-}handlers} : \mathit{Handler} - \mathit{Env}\ = \lambda I\,.\ \lambda p\,.\ (\mathit{err\text{-}to\text{-}comp}\ I)$

New computation auxiliaries:

$\mathit{extending\text{-}handlers} : \text{Identifier} \to \mathit{Procedure} \to \mathit{Computation} \to \mathit{Computation}$
$= \lambda I p c\,.\ \lambda w\,.\ (c\ \ (\mathit{extend\text{-}handlers}\ w\ I\ p))$

$\mathit{getting\text{-}handler} : \text{Identifier} \to (\mathit{Procedure} \to \mathit{Computation}) \to \mathit{Computation}$
$= \lambda I f\ .\ \lambda w\,.\ (f\ \ (\mathit{get\text{-}handler}\ w\ I)\ \ w)$

Modifications to other computation auxiliaries:

$\mathit{val\text{-}to\text{-}comp} : \mathit{Value} \to \mathit{Computation}\ = \lambda v\,.\ \lambda w k\,.\ (k\ v)$

$\mathit{err\text{-}to\text{-}comp} : \mathit{Error} \to \mathit{Computation}\ = \lambda I\,.\ \lambda w k s\,.\ (\mathit{Error} \mapsto \mathit{Expressible}\ I)$

$\mathit{with\text{-}value} : \mathit{Computation} \to (\mathit{Value} \to \mathit{Computation}) \to \mathit{Computation}$
$= \lambda c f\ .\ \lambda w k\,.\ (c\ \ w\ \ (\lambda v\,.\ (f\ \ v\ \ w\ \ k)))$

Other computation auxiliaries similarly pass around handler environments.

Figure 9.20: Semantics of `raise` and `trap`, Part I.

Valuation functions (standard version):

$$\mathcal{E}[\![(\texttt{trap}\ I_{except}\ E_{handler}\ E_{body})]\!]$$
$$= \lambda ewk\,.\ (\mathcal{E}[\![E_{handler}]\!]$$
$$e\ \ w\ \ (\textit{test-procedure}$$
$$(\lambda p\,.\ (\mathcal{E}[\![E_{body}]\!]\ e\ (\textit{extend-handlers}\ w\ I_{except}\ p)\ k))))$$

$$\mathcal{E}[\![(\texttt{raise}\ I_{except}\ E_{body})]\!]$$
$$= \lambda ewk\,.\ (\mathcal{E}[\![E_{body}]\!]\ e\ w\ (\lambda v\,.\ ((\textit{get-handler}\ w\ I_{except})\ v\ w\ k)))$$

Valuation functions (computation version):

$$\mathcal{E}[\![(\texttt{trap}\ I_{except}\ E_{handler}\ E_{body})]\!]$$
$$= \lambda e\,.\ (\textit{with-procedure}\ \ (\mathcal{E}[\![E_{handler}]\!]\ e)$$
$$(\lambda p\,.\ (\textit{extending-handlers}\ I_{except}\ p\ \ (\mathcal{E}[\![E_{body}]\!]\ e))))$$

$$\mathcal{E}[\![(\texttt{raise}\ I_{except}\ E_{body})]\!]$$
$$= \lambda e\,.\ (\textit{with-value}\ \ (\mathcal{E}[\![E_{body}]\!]\ e)\ \ (\lambda v\,.\ (\textit{getting-handler}\ I_{except}\ (\lambda p\,.\ (p\ v)))))$$

Figure 9.21: Semantics of `raise` and `trap`, Part II.

Computation, is equivalent to the following:

$$\textit{Procedure} = \textit{Value} \to \textit{Handler} - \textit{Env} \to \textit{Expcont} \to \textit{Store} \to \textit{Expressible}$$

The auxiliaries *extend-handlers*, *get-handler*, and *default-handlers* are versions of *extend*, *lookup*, and *empty-env* for *Handler-Env*. The auxiliaries *extending-handlers* and *getting-handler* capture manipulations of the *Handler-Env* component of the computation in an abstract way. If the computation abstractions are used, it is not necessary to modify any of the valuation clauses from the semantics of FLK!. However, valuation clauses written in the standard style would have to be modified to pass along an extra w argument.

The valuation clauses for `trap` and `raise` are presented in both the standard style and the computation style. `trap` simply extends the dynamic handler environment with a new binding and evaluates E_{body} with respect to this new environment. `raise` invokes the dynamically bound handler on the value of E_{body}. Note that *default-handlers* initially binds every exception name to a handler that converts an exception into an error. A handler procedure is called with the dynamic handler environment in effect at the point of the `raise`. This gives rise to the following behavior:

```
(trap a (lambda (x) (+ 4000 x))
  (trap b (lambda (x) (+ 300 (raise a (+ x 4))))
    (trap a (lambda (x) (+ 20 x))
      (+ 1 (raise b 2))))) ⟶FL! 327

(trap a (lambda (x) (* x 10))
  (+ 1 (raise a (+ 2 (raise a 4))))) ⟶FL! 421
```

Exception handling is an excellent example of the utility of dynamic scoping. Suppose `trap` were to bind I_{except} in a lexical environment rather than a dynamic one. Then `raise` could only be handled by lexically apparent handlers. It would be impossible to specify handlers for a procedure on a per call basis.

Termination semantics for exception handlers can be simulated by using `label` and `jump` in conjunction with `raise` and `trap`. For example, suppose we want a call to `sum-list` to abort its computation and return 0 if the list contains a non-integer. This can be expressed as follows:

```
(label exit
  (trap non-integer (lambda (x) (jump exit 0))
    (sum-list '(5 yes #t)))) ⟶FL! 0
```

Here the handler procedure forces the computation to abort to the `exit` point when the symbol `yes` is encountered.

An alternative to using `label` and `jump` in situations like these is to develop a new kind of handler clause:

(`handle` I_{except} $E_{handler}$ E_{body})

Like `trap`, `handle` dynamically binds I_{except} to the handler computed by $E_{handler}$. But unlike `trap` handlers, when a `handle` handler is invoked by `raise`, it uses the dynamic environment and continuation of the `handle` expression rather than the `raise` expression. For example:

```
(handle a (lambda (x) (+ 4000 x))
  (handle b (lambda (x) (+ 300 (raise a (+ x 4))))
    (handle a (lambda (x) (+ 20 x))
      (+ 1 (raise b 2))))) ⟶FL! 4006

(handle a (lambda (x) (* x 10))
  (+ 1 (raise a (+ 2 (raise a 4))))) ⟶FL! 40
```

We leave the semantics of `handle` as an exercise (`raise` need not be changed).

▷ **Exercise 9.12** Sam Antix decides to add the new `handle` exception handling primitive to FL! + {`raise`, `trap`}. He adds alters the grammar of FL! + {`raise`, `trap`}:

(`handle` I_{except} $E_{handler}$ E_{body})

As we described above, Sam's new expression is similar to

$$\texttt{(trap } I_{except} \ E_{handler} \ E_{body}\texttt{)}.$$

Both expressions evaluate $E_{handler}$ to a handler procedure and dynamically install the procedure as a handler for exception I_{except}. Then the body expression E_{body} is evaluated. If E_{body} returns normally, then the installed handler is removed, and the value returned is the value of E_{body}.

However, if the evaluation of E_{body} reaches an expression

$$\texttt{(raise } I_{except} \ E\texttt{)},$$

then E is evaluated and the handler procedure is applied to the resulting value. With `trap`, this application is evaluated at the point of the `raise` expression. But with `handle`, the application is evaluated at the point of the `handle` expression. In particular, both the dynamic environment and continuation are inherited from the `handle` expression, *not* the `raise` expression.

Here is another example besides the one given above:

```
(handle a (lambda (x) (* x 10))
  (+ 1 (raise a (+ 2 (raise a 4))))) ⟶eval 40
```

a. Extend the denotational semantics of call-by-value FLK! + {`raise`, `trap`} with a meaning function clause for `handle` (the meaning function clause for `raise` doesn't need to be changed).

b. Give a desugaring of `handle` into FL! + {`raise`, `trap`, `label`, `jump`}. ◁

▷ **Exercise 9.13** Ben Bitdiddle, whose company is fighting for survival in the competitive FL! market, has an idea for getting ahead of the competition: adding recursive exception handlers! He wants to extend FLK! as follows:

E ::= ... `existing` FLK! `constructs` ...
| `(handle` I_{except} $E_{handler}$ E_{body}`)`
| `(rec-handle` I_{except} $E_{handler}$ E_{body}`)`
| `(raise` I_{except} E_{val}`)`

The `handle` and `raise` constructs are unchanged from Exercise 9.12. Informally, the `rec-handle` construct has the following semantics: first, $E_{handler}$ is evaluated to a procedure p (it is an error if it evaluates to something that is not a procedure). Next, E_{body} is evaluated; if it raises the exception I_{except}, the procedure p handles it. So far, `rec-handle` is identical to `handle`. However, if the execution of p raises the exception I_{except}, this exception is handled by p. The exceptions have termination semantics.

Here's a short example that Ben has prepared for you:

```
(rec-handle I
  (lambda (x)
    (if (= x 0) 1 (raise I (- x 1))))
  (raise I 5)) ⟶eval 1
```

Give the meaning function clause for `(rec-handle` I_{except} $E_{handler}$ E_{body}`)` (the semantic domains and the meaning function clauses for `handle` and `raise` remain unchanged). ◁

▷ **Exercise 9.14** Alyssa P. Hacker really likes the exception system of a certain Internet applet language. She has decided to add that exception system to her favorite language, FL!. In particular, she wants to modify the grammar of FLK! as follows:

E ::= ... `existing` FLK! `constructs (except proc)` ...
| `(handle` I_{except} $E_{handler}$ E_{body}`)`
| `(raise` I_{except} E_{val}`)`
| `(finally` E_{body} $E_{finally}$`)`
| `(proc` I_{except} I E`)`
| `(try` E_{body} `((`I_{except} I $E_{handler}$`)*)` $E_{finally}$`)`

Note: To improve the readability of the examples, all the exception identifiers used in this exercise start with the character `%`.

The `handle` and `raise` constructs are old friends, but the others are new. Here are their informal semantics:

- `(handle` I_{except} $E_{handler}$ E_{body}`)` establishes an exception handler for exception I_{except}. First $E_{handler}$ is evaluated to a handler procedure. Then E_{body} is evaluated. If E_{body} raises I_{except}, the exception handler is called with the value of the exception, and the value returned by the handler is returned by the `handle` expression. (That is, `handle` provides *termination* semantics.)

- `(raise` I_{except} E_{val}`)` passes the value of E_{val} to the exception handler for I_{except}. A `raise` expression never returns a value, because the `handle` expression provides termination semantics.

- `(finally` E_{body} $E_{finally}$`)` ensures that $E_{finally}$ is evaluated. Specifically, it evaluates E_{body} and either (1) returns its value or (2) propagates the exception it raises. Whether E_{body} returns normally or via an exception, $E_{finally}$ is evaluated before `finally` returns. The value of $E_{finally}$ is discarded.

  For example, the following expression always closes the input file:

```
(let ((port (open-input-file "foo.txt")))
  (handle %invalid (lambda (value)
                     (begin
                       (display "invalid: ")
                       (display value)
                       (newline)
                       'invalid))
    (finally (let ((value (read port)))
               (if (valid? value)
                   value
                   (raise %invalid value)))
      (close-input-file port))))
```

If the file's contents are valid, the expression returns the file's contents. Otherwise, the exception `%invalid` is raised, the handler prints a message and then returns the symbol `invalid` instead of the file's contents. No matter what happens, `close-input-file` is called to clean up.

- (`proc` I_{except} I E) returns a procedure of one argument. However, the procedure is guaranteed to raise only exception I_{except} or `%illegal-exception`. If its body raises any other exceptions, they are converted to `%illegal-exception`.

  For example, the following procedure raises the `%f` exception if its argument is `#f`. Otherwise it automatically raises the `%illegal-exception` exception:

  ```
  (proc %f b (if b (raise %t b) (raise %f b)))
  ```

- (`try` E_{body} ((I_{except} I E)*) $E_{finally}$) corresponds to the `try-catch-finally` construction in a certain unmentionable language. It works like this:

  - E_{body} is evaluated and, if it doesn't raise any exceptions, its value is returned as the value of the `try` expression. However, if an exception is raised by E_{body}, the (I_{except} I E) clauses are consulted to handle the exception.
  - If an exception I_{except} is raised by E_{body}, the corresponding variable I is bound to the value of the exception and the corresponding expression E is evaluated. The value of the `try` expression becomes the value of E in this case.
  - Regardless of whether or not an exception is raised by E_{body}, the expression $E_{finally}$ is evaluated immediately before control leaves the `try` expression. Its value is discarded.

As usual, Alyssa has vanished, probably to another Internet startup. She's left you with a helper function, *insert-procedure*. Informally, *insert-procedure* takes a handler environment, an identifier predicate, and a procedure. It returns a new handler environment that *conditionally* inserts the procedure at the beginning of the handler chain: the procedure is executed for any exception that satisfies the corresponding identifier predicate and was not caught yet by a handler. For example,

(*insert-procedure* w $(\lambda I_{except}\,.\,true)$ p)returns a handler environment that is like w except that p will be called first on *every* exception. Here is the signature and the definition of *insert-procedure*:

$$\begin{aligned}
&\textit{insert-procedure}: \textit{Handler-Env} \to (\text{Identifier} \to \textit{Bool}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \to \textit{Procedure} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \to \textit{Handler-Env} \\
&= \lambda w f\, p\,.\,(\lambda I_{except}\,.\,\textbf{if}\,(f\ \ I_{except}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}\ \lambda\delta w' k\,.\ (p\ \ \delta\ \ w'\ \ (\lambda v\,.\,((w\ \ I_{except})\ \ \delta\ \ w'\ \ k))) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ (w\ \ I_{except}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{fi}
\end{aligned}$$

You may find *insert-procedure* useful in solving the following problems:

a. Give the meaning function clause for `finally`.

b. Give the meaning function clause for `proc`.

c. Give a desugaring for

$$\texttt{(try}\ \ E_{body}\ \ \texttt{((}I_{except\_1}\ \ I_1\ \ E_1\texttt{)}\ \ \ldots\ \ \texttt{(}I_{except\_n}\ \ I_n\ \ E_n\texttt{))}\ \ E_{finally}\texttt{)}$$

into `handle`, `raise`, `proc`, and `finally`. Assume that the handler expression E_i is permitted to raise only the exception it handles, I_{except_i}. If it raises any other exceptions, they are automatically converted to `illegal-exception`. ◁

▷ **Exercise 9.15** Sam Antix thinks that exception handlers should be able to choose dynamically between termination or resumption semantics. Sam likes the termination semantics of `handle` (Exercise 9.12), but occasionally he would prefer resumption semantics. He decides to extend FL!+{`raise`, `handle`} with a new construct `(resume` E`)`.

$$\begin{aligned}
E ::= &\ \ldots && \text{[As before]} \\
&\mid \texttt{(resume}\ E\texttt{)} && \text{[Resume at point of most recent raise]}
\end{aligned}$$

Informally, `(resume` E`)` will cause a handler to resume at the point of the `raise` rather than terminating at the point of the `handle`. `resume` first evaluates E using the current dynamic handler environment and then returns control to the point of the most recent `raise` with the value of E. Further, any program that does not use `resume` should behave just as it would in FL!+{`raise`, `handle`}.

Sam came up with some short examples demonstrating his new `(resume` E`)` construct:

```
(handle exn
     (lambda (x) (+ x 2))
   (+ 20 (raise exn 1))) ⟶eval 3

(handle exn
     (lambda (x) (resume (+ x 2)))
   (+ 20 (raise exn 1))) ⟶eval 23

(resume 7) ⟶eval error : no − raise

(let ((f (lambda (x) (resume (+ x 4)))))
   (handle exn
        (lambda (x) (f (+ x 2)))
     (+ 20 (raise exn 1)))) ⟶eval 27
```

When `resume` is invoked, any pending computation in the handler is discarded, including any other `resume`s:

```
(handle exn
     (lambda (x) (resume (+ 300 (resume (+ x 2)))))
   (+ 20 (raise exn 1))) ⟶eval 23

(handle exn1
     (lambda (x) (+ 50000 (resume (+ x 4))))
   (+ 4000 (handle exn2
              (lambda (x) (+ 300 (raise exn1 (+ x 2))))
             (+ 20 (raise exn2 1))))) ⟶eval 4307
(handle exn1
     (lambda (x) (+ 50000 (resume (+ x 4))))
   (+ 4000 (handle exn2
              (lambda (x)
                 (resume (+ 300 (raise exn1 (+ x 2)))))
             (handle exn1
                  (lambda (x) (+ 600000 x))
               (+ 20 (raise exn2 1)))))) ⟶eval 4327
```

Now handlers can choose between termination and resumption semantics:

```
(define example-fctn
   (lambda (argument)
      (handle exn
           (lambda (x) (if (< x 3)
                           (+ x 300)
                           (+ 500000 (resume (+ x 4000)))))
         (+ 20 (raise exn argument)))))
```

`(example-fctn 2)` $\xrightarrow[eval]{}$ *302*

`(example-fctn 4)` $\xrightarrow[eval]{}$ *4024*

a. Unfortunately, Sam has fallen ill. You must flesh out his design. You should extend the standard semantics for FL!+{`raise`, `handle`} as follows.

   i. Give the signature of $\mathcal{E}$ and the definition of the semantic domain *Procedure*.

   ii. Give the meaning function clauses for `raise`, `handle`, and `resume`.

b. The `trap` construct presented above has resumption semantics. It is possible to translate FL!+{`trap`, `raise`} into FL!+{`handle`, `raise`, `resume`}. We emphasize that this is a *translation* between two different languages and *not* a desugaring from a language to itself.

   The translation of most expressions merely requires translating their subexpressions. For example,

   $\mathcal{T}[\![$`(call` E_1 E_2`)`$]\!] =$`(call` $\mathcal{T}[\![E_1]\!]$ $\mathcal{T}[\![E_2]\!]$`)`

   Give the translation for `trap` and `raise`. ◁

▷ **Exercise 9.16** Consider a lexically-scoped, call-by-value variant of FLK, with the following twist: it has a `switch` construct that allows a program to both generate and handle exceptions. Here is the complete grammar:

E ::= L | I | `(if` E_1 E_2 E_3`)`
| `(proc` I E_B`)` | `(call` E_0 E_1`)`
| `(switch` E`)`

The idea behind `switch` is that every expression is implicitly provided with two continuations called A and B. All expressions pass their return value to the A continuation. The top level meaning function $\mathcal{TL}$ is modified to call $\mathcal{E}$ by passing the identity function as the A continuation (in order to get back the normal result), and an error handler as the B continuation. Thus, the A continuation usually corresponds to a normal return, while the B continuation usually corresponds to an exceptional return. We will call values that are passed to A continuations "A values" and values that are passed to B continuations "B values."

The `switch` form is used to swap the A and B continuations during the evaluation of its component expression. `switch` can be used to both generate and handle exceptions. Here are two example uses of `switch` (assuming the usual desugarings):

```
(define (my-func i j)
  (if (< 10 (+ i j))            ; make sure that  i+j > 10
      (g i j)                   ; compute the result
      (switch "size")))         ; else "size" error

(switch
  (let ((bval (switch (my-func 3 4))))
    (switch
      (if (string-equal? bval "size")
          1          ; return 1 for size error
          0))))      ; return 0 for other errors
```

In the last example, the `switch` around the application of `my-func` will cause B values of `my-func` to be bound to the variable `bval`. If `my-func` has an A value (a normal value) then the `switch` around `my-func` will cause execution to bounce out to the outermost `switch`, where the A value returned by `my-func` will be the A value of the entire expression. If a B value is returned by `my-func`, the entire expression will have an A value of either `1` or `0`, depending on the B value returned by `my-func`.

a. Construct the standard semantics for the language described above: Give the signature of $\mathcal{E}$ and the definition of the *Procedure* domain. Also give the meaning function clauses for `switch` and `call`. Either write out the other meaning function clauses, or describe how the corresponding clauses from the semantics for FLK! would be modified.

b. Using your semantics, prove that `(switch (switch` E`))` has the same meaning as E.

c. Suppose we have a (strict) `pair` construct to make a pair of two values, and the operations `left` and `right` to select the first and second values of a pair respectively. Then we might define an FLK!-like `raise` construct by the following desugaring:

$$\mathcal{D}(raise I_{except} E_{val}) = \texttt{(switch (pair (symbol } I_{except}\texttt{)} \ \mathcal{D}E_{val}\texttt{))}$$

Give a corresponding desugaring for `(handle` I_{except} $E_{handler}$ E_{body}`)`. Does your solution implement termination or resumption semantics? Explain. ◁

Reading

The notion of **continuation** was developed in the early 1970's by Christopher Strachey, Christopher Wadsworth, F. Lockwood Morris, and others. See [SW74]

which was more recently reprinted in [SW00] (that issue of *Higher-Order and Symbolic Computation* was dedicated to Strachey's work). Subsequently, continuations played an important role in actor languages [Hew77] and in SCHEME [SS76, Ste78].

For more information on control, continuations, and denotational semantics, see John Reynolds's history of continuations [Rey93], David Schmidt's textbook on denotational semantics [Sch86b], as well as Joseph Stoy's coverage of continuations in denotational semantics [Sto77, esp. pp. 251ff]. Stoy makes the argument that it is better for expressing the meaning of programs to embed continuations in the semantics rather than syntactically transforming programs into continuation passing style and using direct semantics.

For transforming programs into continuation-passing style (which we will explore further in Chapter 17), see [SF93, FSDF93, SF92].

Continuations can be used to understand other control structures, such as `shift` and `reset` [DF92] and the `amb` (for "ambiguous") operator [McC67, Cli82].

For more information on coroutines, see Melvin Conway coroutines [Con63]. See also C. A. R. Hoare's classic text on Communicating Sequential Processes, [Hoa85], as well as the OCCAM reference manual [occ95] and the description of JCSP in [Lea99].