Chapter 7

Naming

A good name is rather to be chosen than great riches

— *Proverbs 22:1*

Naming is a central issue in programming language design. The fact that programming languages use names to refer to various objects and processes is at the heart of what makes them languages.

At the very least, a programming language must have a primitive set of names (literals and standard identifiers) and a means of combining the names into compound names (expressions). In a purely functional programming language, every expression is a name for the value it computes. In FL, for instance, `9`, `(+ 4 5)`, and `((lambda (a) (* a a)) (+ 1 2))` are just three different names for the number nine. In non-functional languages, there are more complex relationships between names and values that we shall explore later.

Expressions built merely out of primitives and a means of combination quickly become complex and cumbersome. Any practical language must also provide a means of abstraction for abbreviating a long name with a shorter one. Programming languages typically use symbolic identifiers as abbreviations and have binding constructs that specify the association between the abbreviation and the entity for which it stands. FL has the binding constructs `lambda`, `let`, `letrec`, and `define`; these are built on top of FLK's binding constructs: `proc` and `rec`. Using such constructs, it is possible to remove duplications to obtain more concise, readable, and efficient expressions. For example, naming allows us to transform the procedure:

```
(lambda (a b c)
  (list (+ (- 0 b)
           (sqrt (- (* b b)
                    (* 4 (* a c)))))
        (- (- 0 b)
           (sqrt (- (* b b)
                    (* 4 (* a c)))))
```

into the equivalent procedure:

```
(lambda (a b c)
  (let ((discriminant (sqrt (- (* b b)
                               (* 4 (* a c)))))
        (-b (- 0 b)))
      (list (+ -b discriminant)
            (- -b discriminant))))
```

Naming seems like such a simple idea that it's hard to imagine the subtleties hidden therein. A sampling of naming facilities in modern programming languages reveals a surprising number of ways to think about names. Some of the dimensions along which these facilities vary are:

- *Denotable Values*: What entities in a language can be named by global variables? By local variables? By formal parameters of procedures? By field names of a record?

- *Parameter Passing Mechanisms*: What is the relationship between the actual arguments provided to a procedure call and the values named by the formal parameters of the procedure?

- *Scoping*: How are new variables declared? Over what part of the program text and its associated computation does a declaration extend? How are references to a variable matched up with the associated declaration?

- *Name Control*: What mechanisms exist for structuring names to minimize name clashes in large programs?

- *Multiple Namespaces*: Can an identifier refer to more than one variable within a single expression?

- *Name Capture*: Does the language exhibit any name capture problems like those that cropped up with naïve substitution in FLK?

- *Side Effects*: Can the value associated with a name change over time?

The goal of this chapter is to explore many of the above dimensions. Our discussion of FL already introduced some of the basic concepts and terminology of naming, e.g., scope, free and bound variables, name capture, substitution, and environments. Here we give a fuller account of the issues involved in naming. Along the way, we shall pay particular attention to the effects that choices in naming design have on the expressive power of a language.

Certain naming issues (e.g., side effects, many parameter passing mechanisms) are intertwined with other aspects of dynamic semantics that we will cover later: state, control, data, nondeterminism, and concurrency. We defer these topics until the necessary concepts have been introduced.

7.1 Parameter Passing

Procedure application is the inverse operation to procedural abstraction. An abstraction packages formal parameters together with a body expression that refers to them, while application unpackages the body and evaluates it in a context where the formal parameters are associated with the arguments to the call. There are numerous methods for associating the formal parameter names with the arguments. These methods are called **parameter passing mechanisms**. Here we shall focus on two such mechanisms:

- In the **call-by-name (CBN)** mechanism, a formal parameter names the computation designated by an unevaluated argument expression. This corresponds to the non-strict argument evaluation strategy exhibited by FL in the previous chapter. CBN evolved out of the lambda calculus, and variants of CBN have found their way into ALGOL 60 and various functional programming languages (such as HASKELL and MIRANDA).

- In the **call-by-value (CBV)** mechanism, a formal parameter names the value of an evaluated argument expression. This corresponds to the strict argument evaluation strategy used by most modern languages (e.g., C, PASCAL, SCHEME, ML, SMALLTALK, POSTSCRIPT, etc.).

Later we shall explore additional parameter passing mechanisms (e.g., call-by-denotation on page 275 and call-by-reference in Chapter 8).

7.1.1 Call-by-Name and Call-by-Value: The Operational View

Figure 7.1 summarizes the difference between CBN and CBV in an operational framework. Both mechanisms share the following progress rule for the operator of a `call` expression, which is not shown in the figure:

$$\texttt{(call (proc } I\ E_1\texttt{) } E_2\texttt{)} \Rightarrow [E_2/I]E_1 \qquad [\textit{cbn-call}]$$

Call-By-Name

$$\frac{E_2 \Rightarrow E_2{}'}{\texttt{(call } V_1\ E_2\texttt{)} \Rightarrow \texttt{(call } V_1\ E_2{}'\texttt{)}} \qquad [\textit{cbv-rand-progress}]$$

$$\texttt{(call (proc } I\ E_1\texttt{) } V\texttt{)} \Rightarrow [V/I]E_1 \qquad [\textit{cbv-call}]$$

Call-By-Value

Figure 7.1: Essential operational semantics of CBN and CBV parameter passing.

$$\frac{E_1 \Rightarrow E_1{}'}{\texttt{(call } E_1\ E_2\texttt{)} \Rightarrow \texttt{(call } E_1{}'\ E_2\texttt{)}} \qquad [\textit{rator-progress}]$$

Under CBN, the entire operand expression (not just its value) is substituted for the formal parameter of the abstraction. Figure 7.2 illustrates how this substitution works in some particular examples. Notice that the number of times the operand expression is evaluated under CBN depends on how many times the formal parameter is used within the body. If the formal is never used, the operand is never evaluated.

```
(call (proc x (primop * x x)) (primop + 2 3))
⇒ (primop * (primop + 2 3) (primop + 2 3))
⇒ (primop * 5 (primop + 2 3))
⇒ (primop * 5 5)
⇒ 25

(call (proc x 3) (primop / 1 0)) ⇒ 3

(call (proc x 3) (call (proc a (call a a))
                       (proc a (call a a)))) ⇒ 3
```

Figure 7.2: Under CBN, the entire operand expression is substituted for a formal parameter.

In the CBV strategy, the operand of an application is first completely evaluated, and then the resulting value is substituted for the formal parameter within the body of the abstraction. The [*cbv-call*] rule is only applicable when the operand of the application is a value in the syntactic domain ValueExp of value expressions. The [*cbv-rand-progress*] rule permits evaluation of the operand. Together, these two rules force complete evaluation of the operand position before substitution.

Figure 7.3 shows some examples of CBV evaluation. The first example shows that in CBV, the operand expression is evaluated exactly once, regardless of how many times the formal is needed within the evaluation of the abstraction body. The other examples illustrate that CBV can yield errors or nontermination in cases where CBN would return a value.

```
(call (proc x (primop * x x)) (primop + 2 3))
    ⇒ (call (proc x (primop * x x)) 5)
    ⇒ (primop * 5 5)
    ⇒ 25

    (call (proc x 3) (primop / 1 0)) {This stuck expression models
                                      an error.}
    (call (proc x 3) (call (proc a (call a a))
                           (proc a (call a a))))
    ⇒ (call (proc x 3) (call (proc a (call a a))
                               (proc a (call a a))))
    ⇒ ...                        {An infinite loop.}
```

Figure 7.3: Under CBV, argument expressions are evaluated before being substituted for formal parameters.

Each of the two mechanisms has benefits and drawbacks. The above examples showed that CBN can evaluate an operand expression multiple times when the formal parameter is referenced more than once. This is less efficient than CBV, which is guaranteed to evaluate the operand exactly once. On the other hand, the CBV strategy of evaluating the operand exactly once may cause the computation to hang even when the operand value is not required by the procedure body. This is the situation where the CBN strategy of evaluating the operand at every parameter reference pays off; since there are no references, the operand is *never* evaluated. As Joseph Stoy has noted, CBN means evaluating the operand as many times as necessary, but sometimes this means no times at all! In this case CBN is "infinitely" more efficient than CBV, because it produces an answer when CBV does not.

The CBN mechanism has its roots in the lambda calculus, which is essentially a pared down version of FL that supports only applications, abstractions, and variable references. CBN corresponds to a leftmost, outermost reduction strategy for the lambda calculus called **normal order reduction**. An important feature of normal order reduction in the lambda calculus is that it is guaranteed to find a **normal form** (i.e., value) for an expression if one exists. Furthermore, if any other reduction strategy finds a normal form, it must find the same one as the normal order strategy (modulo alpha equivalence).

The reduction strategy that corresponds to CBV, i.e., arguments must be reduced to normal form before substitution for formals, is called **applicative order reduction**.

We believe that a similar statement holds for FL:

> FLK **CBN/CBV Conjecture:** If an FLK expression $E \overset{*}{\Rightarrow}$CBV V, then $E \overset{*}{\Rightarrow}$CBN V', where V and V' are equivalent in some appropriate sense.

The fuzziness of "some appropriate sense" is due to the fact that ValueExp is different for the two mechanisms. In CBN, ValueExp includes `pair`s with arbitrary expression as parts, while in CBV, it includes only `pair`s with component values. As of this writing, we are still fleshing out a formal proof of this conjecture. But the intuition is clear: CBN terminates more often than CBV, and if they both terminate, they must terminate with "equivalent" values.

From the theoretical perspective, CBN clearly seems superior to CBV. Then why do so many languages use CBV and hardly any use CBN? As hinted above, a pragmatic reason is that CBN implies certain implementation overheads. Perhaps an even more important reason is that CBN and side-effects do not mix. As we shall see in the next chapter, imperative programs using CBN are notoriously hard to reason about. But here we shall focus only on the issue of overheads.

As a non-trivial example, let's compare the CBN and CBV mechanisms on the following call to an iterative factorial procedure written in FL:[1]

```
((rec fact-iter (lambda (n ans)
                  (if (= n 0)
                      ans
                      (fact-iter (- n 1) (* n ans)))))
 3 1)
```

Transition sequences for the two parameter passing mechanisms are shown in Figures 7.4 and 7.5. In the transition sequences, the abbreviation *FACT-ITER* stands for the expression

[1]We use FL rather than FLK for this example because it would be too cumbersome to express in FLK. We assume in the example an SOS in which FL expressions are appropriate configurations. For instance, in this SOS, multi-argument applications are performed in a single rewrite step.

```
(rec fact-iter (lambda (n ans)
                 (if (= n 0)
                     ans
                     (fact-iter (- n 1) (* n ans)))))
```

while the abbreviation *UNWOUND-FACT-ITER* stands for the expression

```
(lambda (n ans)
  (if (= n 0)
      ans
      (FACT-ITER (- n 1) (* n ans))))
```

(*FACT-ITER* 3 1)
$\overset{*}{\Rightarrow}$ (*UNWOUND-FACT-ITER* 3 1)
$\overset{*}{\Rightarrow}$ (if (= 3 0) 1 (*FACT-ITER* (- 3 1) (* 3 1)))
$\overset{*}{\Rightarrow}$ (if #f 1 (*FACT-ITER* (- 3 1) (* 3 1)))
$\overset{*}{\Rightarrow}$ (*FACT-ITER* (- 3 1) (* 3 1)))
$\overset{*}{\Rightarrow}$ (*UNWOUND-FACT-ITER* (- 3 1) (* 3 1)))
$\overset{*}{\Rightarrow}$ (*UNWOUND-FACT-ITER* 2 (* 3 1)))
$\overset{*}{\Rightarrow}$ (*UNWOUND-FACT-ITER* 2 3)
$\overset{*}{\Rightarrow}$ (if (= 2 0) 3 (*FACT-ITER* (- 2 1) (* 2 3)))
$\overset{*}{\Rightarrow}$ (if #f 3 (*FACT-ITER* (- 2 1) (* 2 3)))
$\overset{*}{\Rightarrow}$ (*FACT-ITER* (- 2 1) (* 2 3))
$\overset{*}{\Rightarrow}$ (*UNWOUND-FACT-ITER* (- 2 1) (* 2 3))
$\overset{*}{\Rightarrow}$ (*UNWOUND-FACT-ITER* 1 (* 2 3))
$\overset{*}{\Rightarrow}$ (*UNWOUND-FACT-ITER* 1 6)
$\overset{*}{\Rightarrow}$ (if (= 1 0) 6 (*FACT-ITER* (- 1 1) (* 1 6)))
$\overset{*}{\Rightarrow}$ (if #f 6 (*FACT-ITER* (- 1 1) (* 1 6)))
$\overset{*}{\Rightarrow}$ (*FACT-ITER* (- 1 1) (* 1 6)))
$\overset{*}{\Rightarrow}$ (*UNWOUND-FACT-ITER* (- 1 1) (* 1 6)))
$\overset{*}{\Rightarrow}$ (*UNWOUND-FACT-ITER* 0 (* 1 6))
$\overset{*}{\Rightarrow}$ (*UNWOUND-FACT-ITER* 0 6)
$\overset{*}{\Rightarrow}$ (if (= 0 0) 6 (*FACT-ITER* (- 0 1) (* 0 6)))
$\overset{*}{\Rightarrow}$ 6

Figure 7.4: CBV transition path computing the iterative factorial of 3.

As indicated by the figures, CBN can be much less efficient than CBV. There are two important sources of overhead:

1. CBN often requires more *time*[2] than CBV in the case where an argument

[2]Here we assume that the time taken by an evaluation is related to the number of evaluation steps in the operational semantics. This is often a reasonable assumption.

```
(FACT-ITER 3 1)
⇒* (UNWOUND-FACT-ITER 3 1)
⇒* (if (= 3 0) 1 (FACT-ITER (- 3 1) (* 3 1)))
⇒* (if #f 1 (FACT-ITER (- 3 1) (* 3 1)))
⇒* (FACT-ITER (- 3 1) (* 3 1)))
⇒* (UNWOUND-FACT-ITER (- 3 1) (* 3 1)))
⇒* (if (= (- 3 1) 0) (* 3 1) (FACT-ITER (- (- 3 1) 1) (* (- 3 1) (* 3 1))))
⇒* (if (= 2 0) (* 3 1) (FACT-ITER (- (- 3 1) 1) (* (- 3 1) (* 3 1))))
⇒* (if #f (* 3 1) (FACT-ITER (- (- 3 1) 1) (* (- 3 1) (* 3 1))))
⇒* (FACT-ITER (- (- 3 1) 1) (* (- 3 1) (* 3 1)))
⇒* (UNWOUND-FACT-ITER (- (- 3 1) 1) (* (- 3 1) (* 3 1)))
⇒* (if (= (- (- 3 1) 1) 0)
       (* (- 3 1) (* 3 1))
       (FACT-ITER (- (- (- 3 1) 1) 1) (* (- (- 3 1) 1) (* (- 3 1) (* 3 1)))))
⇒* (if (= (- 2 1) 0)
       (* (- 3 1) (* 3 1))
       (FACT-ITER (- (- (- 3 1) 1) 1) (* (- (- 3 1) 1) (* (- 3 1) (* 3 1)))))
⇒* (if (= 1 0)
       (* (- 3 1) (* 3 1))
       (FACT-ITER (- (- (- 3 1) 1) 1) (* (- (- 3 1) 1) (* (- 3 1) (* 3 1)))))
⇒* (if #f
       (* (- 3 1) (* 3 1))
       (FACT-ITER (- (- (- 3 1) 1) 1) (* (- (- 3 1) 1) (* (- 3 1) (* 3 1)))))
⇒* (FACT-ITER (- (- (- 3 1) 1) 1) (* (- (- 3 1) 1) (* (- 3 1) (* 3 1))))
⇒* (UNWOUND-FACT-ITER (- (- (- 3 1) 1) 1) (* (- (- 3 1) 1) (* (- 3 1) (* 3 1))))
⇒* (if (= (- (- (- 3 1) 1) 1) 0)
       (* (- (- 3 1) 1) (* (- 3 1) (* 3 1)))
       (FACT-ITER (- (- (- (- 3 1) 1) 1) 1) (* (- (- (- 3 1) 1) 1) (* (- (- 3 1) 1) (* (- 3 1) (* 3 1))))))
⇒* (if (= (- (- 2 1) 1) 0)
       (* (- (- 3 1) 1) (* (- 3 1) (* 3 1)))
       (FACT-ITER (- (- (- (- 3 1) 1) 1) 1) (* (- (- (- 3 1) 1) 1) (* (- (- 3 1) 1) (* (- 3 1) (* 3 1))))))
⇒* (if (= (- 1 1) 0)
       (* (- (- 3 1) 1) (* (- 3 1) (* 3 1)))
       (FACT-ITER (- (- (- (- 3 1) 1) 1) 1)
                  (* (- (- (- 3 1) 1) 1) (* (- (- 3 1) 1) (* (- 3 1) (* 3 1))))))
⇒* (if (= 0 0)
       (* (- (- 3 1) 1) (* (- 3 1) (* 3 1)))
       (FACT-ITER (- (- (- (- 3 1) 1) 1) 1) (* (- (- (- 3 1) 1) 1) (* (- (- 3 1) 1) (* (- 3 1) (* 3 1))))))
⇒* (if #t
       (* (- (- 3 1) 1) (* (- 3 1) (* 3 1)))
       (FACT-ITER (- (- (- (- 3 1) 1) 1) 1) (* (- (- (- 3 1) 1) 1) (* (- (- 3 1) 1) (* (- 3 1) (* 3 1))))))
⇒* (* (- (- 3 1) 1) (* (- 3 1) (* 3 1)))
⇒* (* (- 2 1) (* (- 3 1) (* 3 1)))
⇒* (* 1 (* (- 3 1) (* 3 1)))
⇒* (* 1 (* 2 (* 3 1)))
⇒* (* 1 (* 2 3))
⇒* (* 1 6)
⇒* 6
```

Figure 7.5: CBN transition path computing the factorial of 3.

is used more than once in the body of an abstraction, because then the same argument expression must be evaluated multiple times. For example, in Figure 7.5, the value of `(- 3 1)` is calculated five times, compared to only once in Figure 7.4.

2. CBN often requires more *space* than CBV because expressions whose values are not currently needed may grow as their evaluations are deferred until later. For example, the expression in the `ans` operand position of a call to *FACT-ITER* grows by one multiplication with every recursive call.

In practice, there are techniques for ameliorating both of these sources of overhead. The time inefficiency is typically finessed by **memoization**, a technique that evaluates an operand and caches its value the first time it is referenced. Further references simply return the cached value rather than evaluating the operand again. We shall bump into it again when we study **lazy evaluation** in Chapter 8.

The space overhead is perhaps more insidious. It can be improved by graph-based expression representations that share substructure, but this trick does not stop the space consumed by operands for parameters like `ans` from growing in size with every call. This problem, known as a **dragging tail**, is disturbing because it destroys desirable space management properties for FL. The CBV version of factorial requires only a constant amount of space to keep track of control and state variables.[3] In contrast, the CBN version requires space for the unevaluated state variables that grows linearly with the input to factorial. When executing these kinds of programs on a machine with finite storage resources, a CBN strategy is more likely to run out of space than a CBV strategy. A technique called **strictness analysis** can improve CBN by modifying it to use CBV for operand evaluation when it is possible to prove that the operand will be required at least once.

The various tricks for improving CBN make it much more palatable, but the techniques still require overheads that some implementors find unacceptable. For example, memoization implies that a flag must be tested at every variable reference. Since variable references are rather common, the extra check is often considered prohibitive without special hardware support.

Parameter passing mechanisms are used to describe not only procedure calls, but also other constructs that bind names to values. For example, FL's binding constructs (`lambda`, `let`, `letrec`, and `define`) have an implicit method for associating names with values because they desugar into FLK's abstractions and

[3]For simplicity, we ignore the fact that the larger numbers required for larger inputs to factorial actually require more space.

applications. We shall assume (unless stated otherwise) that all binding constructs inherit their parameter passing mechanism from procedure calls. Thus, `(let ((a (/ 1 0))) 3)` evaluates to `3` in a CBN language, but generates an error in a CBV language.

▷ **Exercise 7.1** In a CBV language, it is often useful to delay the evaluation of an argument until a later time. This behavior can be specified with the pair of constructs `(lazy` E`)` and `(touch` E`)`. Informally, `(lazy` E`)` wraps E up without evaluating it, while `(touch` E`)` unwraps E until it is no longer embedded in a `lazy`. On a non-`lazy` value, `touch` acts as an identity. For example, in CBV FL:

(touch (+ 1 2)) $\xrightarrow[FL]{}$ 3

(touch (lazy (+ 1 2))) $\xrightarrow[FL]{}$ 3

(touch (lazy (lazy (+ 1 2)))) $\xrightarrow[FL]{}$ 3

```
(let ((a (lazy (+ 1 2)))
      (b (lazy (/ 1 0))))
  (touch a))
```
$\xrightarrow[FL]{}$ 3

```
(let ((a (lazy (+ 1 2)))
      (b (lazy (/ 1 0))))
  (touch b))
```
$\xrightarrow[FL]{}$ *error:divide-by-zero*

a. Extend the operational semantics of FLK to handle `lazy` and `touch`. Recall that an SOS has five parts; make whatever changes are necessary to each of the parts.

b. Can `lazy` and `touch` be implemented by syntactic sugar? If so, give the desugarings; if not, explain why.

c. Show how to translate CBN FL into a CBV version of FL that is equipped with `lazy` and `touch`. ◁

▷ **Exercise 7.2** The desugaring rule for `letrec` specified in Section 6.2 was designed for a CBN language. Does it still work in a CBV language? Explain, using concrete example(s) to justify your answer.

◁

▷ **Exercise 7.3** Show by example that an FL expression that diverges under CBN need not diverge under CBV. How does this fact relate to the CBN/CBV conjecture made above? ◁

7.1.2 Call-by-Name and Call-by-Value: The Denotational View

The denotational descriptions of CBN and CBV FL semantics have an interesting and important difference. This difference is found in the kinds of things that a formal parameter can name. In CBN, a formal parameter can name a value, an error, or a non-terminating computation. In short, a formal parameter in CBN can name the unrestricted meaning of an FL expression. This makes sense, as in CBN we conceptually pass an unevaluated expression as an actual parameter, and we only evaluate an actual parameter when absolutely necessary. In CBN, a formal parameter can name the arbitrary result of an FL expression. A CBV FL semantics is substantially more restrictive than a CBN semantics. In CBV, we can only pass values (e.g., integers, booleans, pairs, etc.) as actual parameters, and thus formal parameters can only name values. As we shall describe in detail below, the difference in the kinds of entities that can be named by formal parameters is reflected in the definition of the *Denotable* domain.

Rather than giving the clauses of the valuation function $\mathcal{E}$ in full for each of the parameter passing mechanisms, we shall only describe clauses for those constructs that are pertinent to parameter passing: variable references, `proc`, and `call`. (The `rec` and `pair` clauses are also relevant, but we defer discussion of them until later.) The valuation clause for `proc` is the same for both mechanisms:

$$\mathcal{E}[\![(\texttt{proc}\ I\ E)]\!] = \lambda e\,.\ (\mathit{val\text{-}to\text{-}comp}\ (\mathit{Procedure} \mapsto \mathit{Value}\ (\lambda\delta\,.\ (\mathcal{E}[\![E]\!]\ [I:\delta]e))))$$

Figure 7.6 gives the denotational semantics for CBN versus CBV. The essential difference is that CBN environments name elements of *Computation* while CBV environments can only name elements of *Value*. In FL,

$$\mathit{Computation} = \mathit{Expressible} = (\mathit{Value} + \mathit{Error})_\bot$$

so that CBN environments can name all expressible values (including error and divergence), while CBV environments can only name "regular" values.

The CBN domain equation

$$\delta \in \mathit{Denotable} = \mathit{Computation}$$

indicates that a formal parameter can denote any computation, which in the case of FL includes error and divergence. A CBN procedure can return an element of *Value* even when it is passed one of these irregular values. Thus, call-by-name procedures are not strict. Here's an example of CBN (where e is an arbitrary environment):

$$\delta \in \mathit{Denotable} = \mathit{Computation}$$

$$\mathcal{E}[\![I]\!] = \lambda e \,.\, (\mathit{with\text{-}denotable}\ (\mathit{lookup}\ e\ I)\ (\lambda\delta \,.\, \delta))$$

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

Call-By-Name

$$\delta \in \mathit{Denotable} = \mathit{Value}$$

$$\mathcal{E}[\![I]\!] = \lambda e \,.\, (\mathit{with\text{-}denotable}\ (\mathit{lookup}\ e\ I)\ (\lambda\delta \,.\, (\mathit{val\text{-}to\text{-}comp}\ \delta)))$$

$$\begin{aligned}\mathcal{E}[\![\texttt{(call}\ E_1\ E_2\texttt{)}]\!] = \lambda e \,.\, (\mathit{with\text{-}procedure}\ &(\mathcal{E}[\![E_1]\!]\ e)\\ &(\lambda p \,.\, (\mathit{with\text{-}value}\ (\mathcal{E}[\![E_2]\!]\ e)\ p)))\end{aligned}$$

Call-By-Value

Figure 7.6: Essential denotational semantics of CBN and CBV parameter passing. For FLK, *Computation* = *Expressible*, but the CBN semantics will still be valid later when *Computation* is updated to reflect extensions to FLK. Likewise, the CBV semantics will still be valid when the *Value* domain is extended.

$$
\begin{array}{l}
(\mathcal{E}[\![\texttt{(call (proc x 3) (primop / 1 0))}]\!]\ e) \\
= (\textit{with-procedure}\ \ (\mathcal{E}[\![\texttt{(proc x 3)}]\!]\ e)\ \ (\lambda p\,.\,(p\ \ (\mathcal{E}[\![\texttt{(primop / 1 0)}]\!]\ e)))) \\
= (\textit{with-procedure}\ (\textit{val-to-comp} \\
\qquad\qquad\qquad\qquad (\textit{Procedure}\mapsto\textit{Value} \\
\qquad\qquad\qquad\qquad\qquad\qquad (\lambda\delta\,.\,(\mathcal{E}[\![3]\!]\ [\texttt{x}:\delta]e)))) \\
\qquad (\lambda p\,.\,(p\ \ \textit{error}))) \\
= ((\lambda\delta\,.\,(\mathcal{E}[\![3]\!]\ [\texttt{x}:\delta]e))\ \ \textit{error}) \\
= (\mathcal{E}[\![3]\!]\ [\texttt{x}:\textit{error}]e) \\
= (\textit{val-to-comp}\ (\textit{Int}\mapsto\textit{Value}\ 3))
\end{array}
$$

The CBV domain equation

$$\delta \quad\in\quad \textit{Denotable} \quad=\quad \textit{Value}$$

indicates that identifiers may be bound to values such as integers, booleans, symbols, procedures, and pairs, but may *not* be bound to objects denoting an error or nontermination. The treatment of error and bottom is the only semantic difference between CBV and CBN.

The CBV clause for `call` uses *with-value* to guarantee that only elements of the domain *Value* are passed to p. This accounts for the strict nature of CBV evaluation. As an illustration of CBV, let's once again consider the meaning of the FLK expression `(call (proc x 3) (primop / 1 0))`:

$$
\begin{array}{l}
(\mathcal{E}[\![\texttt{(call (proc x 3) (primop / 1 0))}]\!]\ e) \\
= (\textit{with-procedure}\ (\mathcal{E}[\![\texttt{(proc x 3)}]\!]\ e) \\
\qquad (\lambda p\,.\,(\textit{with-value}\ \ (\mathcal{E}[\![\texttt{(primop / 1 0)}]\!]\ e)\ \ p))\ ) \\
= (\textit{with-procedure}\ \ (\mathcal{E}[\![\texttt{(proc x 3)}]\!]\ e)\ \ (\lambda p\,.\,(\textit{with-value}\ \textit{error}\ p))) \\
= (\textit{with-procedure}\ \ (\mathcal{E}[\![\texttt{(proc x 3)}]\!]\ e)\ \ (\lambda p\,.\,\textit{error})) \\
= (\textit{with-procedure}\ (\textit{val-to-comp} \\
\qquad\qquad\qquad\qquad (\textit{Procedure}\mapsto\textit{Value} \\
\qquad\qquad\qquad\qquad\qquad\qquad (\lambda\delta\,.\,(\mathcal{E}[\![3]\!]\ [\texttt{x}:\delta]e))\ )) \\
\qquad (\lambda p\,.\,\textit{error})\ ) \\
= \textit{error}
\end{array}
$$

7.1.3 Discussion

7.1.3.1 Extensions

Numerous additional features may be layered on top of the above mechanisms to yield further variations in parameter passing for functional languages. For example, it is possible to pass parameters by keyword, to specify optional arguments, or to describe formal parameters that are pattern-matched against arguments that are compound data structures. While these are important ways of capturing common patterns of usage, they are orthogonal to and less fundamental than the CBN vs. CBV distinction. The introduction of side-effects, on the other hand, will lead to fundamental variations of the above mechanisms. We will explore these in Chapter 8.

It is possible to include more than one parameter passing mechanism within a single language. This possibility is explored in Exercise 7.7.

7.1.3.2 Non-Strict vs. Strict Pairs

Data constructors (such as `pair`) are typically non-strict in a CBN language but strict in a CBV language. Figure 7.7 summarizes the operational and denotational differences between non-strict and strict pairs. In both perspectives, the difference boils down to whether the components of pair values must themselves be values. The figure omits the semantics of the `left` and `right` primitives, which do not differ between the non-strict and strict versions.

7.1.3.3 Denotable vs. Passable Values vs. Component Values

We have assumed in our discussion that every entity that is nameable may be passed as an argument or bundled into a pair. While this tends to be true in (and is a major source of power for) functional languages, it is not true in general. For example, while procedures are almost universally nameable, there are many languages (e.g., FORTRAN, BASIC, PASCAL) in which procedures cannot be passed as arguments, or can only be passed in a limited way. Similarly, many languages do not permit data structure components to be procedures. In order to give an accurate denotational description of such languages, it is necessary to distinguish the class of nameable entities from those which may be passed as arguments and those which can be components of data structures. We could therefore introduce new domains, *Passable* and *Component*, that describe these classes of values.

7.1.3.4 Semantic Derivation of Thunking

The denotational descriptions of parameter passing emphasize that CBN and CBV differ only in their treatment of error and nontermination. They also give another perspective on simulating CBN in a CBV language. From a denotational view, the essence of such a simulation in CBV FL is to find a way of naming *error* and $\perp_{Computation}$. It is not possible to name these directly in a CBV language, but it is always possible to name them indirectly, via procedures. For every computation c, we can construct a procedural value that returns c when called:

$$(\mathit{val\text{-}to\text{-}comp}\ (\mathit{Procedure} \mapsto \mathit{Value}\ \ (\lambda\delta\ .\ c)))$$

Operational:

$$V \quad \in \quad \text{ValueExp} \quad = \quad \ldots \cup \{\texttt{(pair } E_1 \; E_2\texttt{)}\}$$

Non-Strict Pairs

$$V \quad \in \quad \text{ValueExp} \quad = \quad \ldots \cup \{\texttt{(pair } V_1 \; V_2\texttt{)}\}$$

$$\frac{E_1 \Rightarrow E_1{}'}{\texttt{(pair } E_1 \; E_2\texttt{)} \Rightarrow \texttt{(pair } E_1{}' \; E_2\texttt{)}} \qquad [\textit{pair-left-progress}]$$

$$\frac{E_2 \Rightarrow E_2{}'}{\texttt{(pair } V_1 \; E_2\texttt{)} \Rightarrow \texttt{(pair } V_1 \; E_2{}'\texttt{)}} \qquad [\textit{pair-right-progress}]$$

Strict Pairs

Denotational:

$$a \quad \in \quad \mathit{Pair} \quad = \quad \mathit{Computation} \times \mathit{Computation}$$

$$\mathcal{E}[\![\texttt{(pair } E_1 \; E_2\texttt{)}]\!] = \lambda e \,.\, (\textit{val-to-comp} \; (\mathit{Pair} \mapsto \mathit{Value} \; \langle (\mathcal{E}[\![E_1]\!] \; e), \; (\mathcal{E}[\![E_2]\!] \; e)\rangle))$$

Non-Strict Pairs

$$a \quad \in \quad \mathit{Pair} \quad = \quad \mathit{Value} \times \mathit{Value}$$

$$\begin{aligned}
&\mathcal{E}[\![\texttt{(pair } E_1 \; E_2\texttt{)}]\!] = \\
&\quad \lambda e \,.\, (\textit{with-value} \; (\mathcal{E}[\![E_1]\!] \; e) \\
&\qquad\quad (\lambda v_1 \,.\, (\textit{with-value} \; (\mathcal{E}[\![E_2]\!] \; e) \\
&\qquad\qquad\quad (\lambda v_2 \,.\, (\textit{val-to-comp} \; (\mathit{Pair} \mapsto \mathit{Value} \; \langle v_1, v_2\rangle)))
\end{aligned}$$

Strict Pairs

Figure 7.7: Operational and denotational views of non-strict and strict pairs.

This means that we can effectively put any computation into an environment by transforming it into the above form before it is bound to a name and then perform the inverse transformation when the name is looked up. Since the parameter δ of the above form is ignored, the transform and its inverse are equivalent to, respectively, creating a procedure of no arguments (a so-called **thunk**) and calling that procedure on no arguments.

7.1.3.5 CBV Versions of `rec`

In an operational semantics, `rec` is handled by the same rule regardless of whether the language is CBN or CBV:

$$(\texttt{rec}\ I\ E) \Rightarrow [(\texttt{rec}\ I\ E)/I]E \qquad [rec]$$

Unforutnately, things are not so simple in a denotational semantics. In CBN, where *Denotable* = *Computation*, the valuation clause for a CBN version of `rec` is very pretty:

$$\mathcal{E}[\![(\texttt{rec}\ I\ E)]\!] = \lambda e\,.\ \mathbf{fix}_{Computation}(\lambda c\,.\,(\mathcal{E}[\![E]\!]\ [I : c]e))$$

The fixed point defined by this clause is well-defined as long as *Computation* is a pointed CPO. For this reason, we will always guarantee that *Computation* is a pointed domain.

However, developing a valuation clause for `rec` in a CBV language is rather tricky. In CBV, the corresponding version of the CBN clause is:

$$\mathcal{E}[\![(\texttt{rec}\ I\ E)]\!] = \lambda e\,.\ \mathbf{fix}_{Value}(\lambda v\,.\,(\mathcal{E}[\![E]\!]\ [I : v]e))$$

But since *Denotable* = *Value* is *not* a pointed domain, the fixed point is not well defined, and the clause is nonsensical.

There are several ways of circumventing this impasse. Here we present two approaches:

1. In `(rec` I E`)`, we can limit E to a subset of expressions that are syntactically guaranteed to be procedures. In CBV,

   $$Proc = Value \to Computation$$

   is pointed (because *Computation* is always pointed), so it is always possible to fix over *Procedure*. That is, suppose we modify the syntax of FLK as follows:

   E ::= ...
   | `(rec` I_{var} A_{body}`)` [Recursion]

   A ::= `(proc` I_{formal} E_{body}`)` [Abstraction]

If we suppose that $\mathcal{A}$ is a valuation function of signature

$$\text{Abstraction} \to \mathit{Environment} \to \mathit{Procedure}$$

then `rec` is definable as:

```
$\mathcal{E}[\![$(rec $I$ $A$)$]\!]$ =
  $\lambda e$ . ($\mathit{Value} \mapsto \mathit{Expressible}$
            ($\mathit{Procedure} \mapsto \mathit{Value}$
              ($\mathbf{fix}_{\mathit{Procedure}}$
                ($\lambda p$ . (($\mathcal{A}$ $[\![A]\!]$)($\mathit{extend}$ $e$
                                                $I$
                                                ($\mathit{Value} \mapsto \mathit{Denotable}$
                                                  ($\mathit{Procedure} \mapsto \mathit{Value}$ $p$) )))))))
```

Unfortunately, the syntactic restriction results in a restriction of the expressive power of `rec`. It is no longer possible to specify recursions over pairs or over procedures whose describing expression is not a manifest `proc`. The following examples, though contrived, are indicative of useful patterns that are disallowed by an approach that requires the body of a `rec` to be a manifest abstraction in a CBV language:

```
(rec ones (pair 1 (lambda () ones)))

(rec fact
  (let ((fact-of-0 1))
    (lambda (f)
      (lambda (n)
        (if (= n 0)
            fact-of-0
            (* n (fact (- n 1))))))))
```

2. An alternative is to make use of $\perp_{Binding}$ to compute fixed points. Here is a version that works in the case of *Computation* = *Expressible*:

```
$\mathcal{E}[\![$(rec $I$ $E$)$]\!]$ = $\lambda e$ . ($\mathbf{fix}_{\mathit{Computation}}$
                        ($\lambda c$ . ($\mathcal{E}[\![E]\!]$ $[I :: (\mathit{extract\text{-}value}\ c)]e$))
```

```
$\mathit{extract\text{-}value} : \mathit{Computation} \to \mathit{Binding}$
= $\lambda c$ . matching $c$
        $\triangleright$ ($\mathit{Value} \mapsto \mathit{Expressible}$ $v$) $[\!]$ ($\mathit{Denotable} \mapsto \mathit{Binding}$ $v$)
        $\triangleright$ ($\mathit{Error} \mapsto \mathit{Expressible}$ error) $[\!]$ $\perp_{\mathit{Binding}}$
        endmatching
```

extract-value coerces a computation into a binding. The resulting binding is either an element of *Denotable* = *Value*, or it is $\perp_{Binding}$. (Recall that

matching is strict, so that *extract-value* maps $\bot_{Computation}$ to $\bot_{Binding}$.) By effectively naming a bottom element in the environment, this trick gives a starting point for the fixed-point iteration. It would also be possible to add a bottom element directly to the *Denotable* domain, but that would not faithfully model the intuition behind CBV. The $\bot_{Binding}$ element helps to clarify the difference between using bottom to solve a recursion equation expressed by a `rec` and allowing bottom to be passed as an argument to a procedure.

It is worth noting that the *extract-value* function works only for *Computation* = *Expressible* and needs to be tweaked if the domain of computations changes.

7.1.3.6 The Perils of Reading Denotational Descriptions Operationally

The valuation clause for `call` in CBN illustrates the potential dangers of giving operational interpretations to denotational definitions. If $(p\ \ (\mathcal{E}[\![E_2]\!]\ e))$ were read as if it were, for example, a LISP or SCHEME program fragment, it would say something like: "First evaluate the expression E_2 in the environment e and then call p on the resulting value." Unfortunately, this reading introduces inappropriate notions of evaluation order and time (based on when the argument is evaluated) that are inherited from the CBV nature of LISP or SCHEME. Such a reading can cause confusion in the cases where E_2 denotes error or bottom; the reader might (incorrectly) think that, as in LISP, errors or bottom in an argument would propagate to errors or bottom for the call.

Rather than reading $\mathcal{E}$ as "evaluate" (in the operational sense), it is safer to read it as "the meaning of." Thus $(p\ \ (\mathcal{E}[\![E_2]\!]\ e))$ means "Apply p to the meaning of E_2 in the environment e." Additionally, it is important to remember that the application of a total function (as opposed to the application of a procedure in a programming language) is well-defined as long as the arguments are in the appropriate domains. In particular, the result is independent of any evaluation strategy that might be associated with the metalanguage expressions used to represent the application. For example, the term

$$(\lambda y\,.\,3)\,((\lambda x\,.\,xx)\,(\lambda x\,.\,xx))$$

denotes 3 even though a CBV-like strategy for equation rewriting would not find this value.

In the case of $(p\ \ (\mathcal{E}[\![E_2]\!]\ e))$, if the meaning of E_2 is error or bottom, these are simply handed to p, which is constructed by the clause for `proc`. The `proc` clause shows that any such argument is simply associated with an identifier in the environment, while the variable reference clause indicates that this denotable value is retrieved upon lookup without any ado.

It is natural to wonder at this point how error or nontermination in an argument can *ever* lead to error or nontermination for an application in a CBN language. That is, if an argument is simply inserted into the environment upon call and retrieved upon lookup, who ever actually *examines* the value denoted by the argument? There are several spots in the denotational definition where information about the values is required. For example, in the clause for $[\![$`(call` E_1 E_2`)`$]\!]$ the value of E_1 is required to be the denotation of a procedure; the semantics must check the value to ensure this is the case (such checks are hidden in the abstraction *with-procedure*). Similarly, an `if` clause must not only check that the test expression denotes a boolean, but also uses the boolean value in order to determine which arm is denoted by the entire conditional construct. Handling primitive operators in FLK is perhaps the most common case where details of the values must be examined.

These facts imply that in any implementation of CBN FLK, the argument expression E_2 *cannot* be evaluated before the procedure p is invoked. For if E_2 initiated a nonterminating computation, p would never be invoked. The moral of this discussion is that operational conclusions of this sort aren't always obvious from a denotational definition. Indeed, factoring out operational concerns is a source of power for denotational semantics. It is not necessary to worry about details like the practical implications of binding errors or nontermination in an environment. Instead, the denotational approach helps us to focus on high-level descriptions like: "The essential difference between CBN and CBV is that the former allows errors and nontermination to be named whereas the latter does not."

7.1.3.7 Call-by-Denotation

Sometimes a denotational semantics suggests alternative perspectives. A case in point is **call-by-denotation (CBD)**, a parameter passing mechanism that is obtained by tweaking call-by-name semantics in a straightforward way (see Figure 7.8). Whereas call-by-name determines the meaning of an argument expression relative to the environment available at the point of call, call-by-denotation instead determines the meaning of an argument expression relative to the environment where the formal parameter is referenced.

In this case, the domain equation

$$\delta \quad \in \quad Denotable \quad = \quad Environment \rightarrow Computation$$

indicates that the nameable entities in the language are functions that map

$$\delta \in \mathit{Denotable} = \mathit{Environment} \to \mathit{Computation}$$

$$\mathcal{E}[\![I]\!] = \lambda e\,.\,(\textit{with-denotable}\ (\textit{lookup}\ e\ I)\ (\lambda\delta\,.\,(\delta\ e)))$$

$$\mathcal{E}[\![\texttt{(call}\ E_1\ E_2\texttt{)}]\!] = \lambda e\,.\,(\textit{with-procedure}\ (\mathcal{E}[\![E_1]\!]\ e)\ (\lambda p\,.\,(p\ \mathcal{E}[\![E_2]\!])))$$

Call-By-Denotation

Figure 7.8: Essential semantics of call-by-denotation.

environments to computations. The `call` clause simply embeds the actual parameter in a function of this type. Variable references are handled by applying the function associated with the variable to the environment in effect where the variable is referenced.

CBD is not very useful but does model some of the name capture problems associated with macro expansion. As a somewhat bizarre example of CBD, consider the meaning of the FL expression

```
((let ((x 3))
   (lambda (y) y))
 x)
```

in an environment e_0 in which the identifier `x` is not bound. In both call-by-name and call-by-value, the meaning of this expression is an error because the value of (the outer) `x` is required but nowhere defined. In call-by-denotation, however, the body of the identity procedure — i.e., the variable `y` — will eventually be evaluated in an environment e_1 where `x` is bound to

$$(\mathit{Denotable} \mapsto \mathit{Binding}\ (\lambda e\,.\,(\mathit{Value} \mapsto \mathit{Expressible}\ (\mathit{Int} \mapsto \mathit{Value}\ 3))))$$

and `y` is bound to

$$(\mathit{Denotable} \mapsto \mathit{Binding}\ (\lambda e\,.\,(\textit{with-denotable}\ (\textit{lookup}\ e\ \texttt{x})\ (\lambda\delta\,.\,(\delta\ e)))))$$

(We leave the details of how this point is reached as an exercise.) At this point, the denotation of `y` will be applied to e_1, with the following result:

$$\begin{aligned}
&((\lambda e\,.\,(\textit{with-denotable}\ (\textit{lookup}\ e\ \texttt{x})\ (\lambda\delta\,.\,(\delta\ e))))\ e_1)\\
&= (\textit{with-denotable}\ (\textit{lookup}\ e_1\ \texttt{x})\ (\lambda\delta\,.\,(\delta\ e_1)))\\
&= (\textit{with-denotable}\ (\mathit{Denotable} \mapsto \mathit{Binding}\\
&\qquad\qquad (\lambda e\,.\,(\mathit{Value} \mapsto \mathit{Expressible}\ (\mathit{Int} \mapsto \mathit{Value}\ 3))))\\
&\quad (\lambda\delta\,.\,(\delta\ e_1)))\\
&= ((\lambda\delta\,.\,(\delta\ e_1))\ (\lambda e\,.\,(\mathit{Value} \mapsto \mathit{Expressible}\ (\mathit{Int} \mapsto \mathit{Value}\ 3))))\\
&= (\mathit{Value} \mapsto \mathit{Expressible}\ (\mathit{Int} \mapsto \mathit{Value}\ 3))
\end{aligned}$$

Thus, in a CBD semantics, the meaning of this expression is the number 3!

The weird behavior of call-by-denotation in this example is due to a kind of name capture. The evaluation of the outer `x` yields not what we would normally think of as a value but an environment accessor that is eventually applied to an environment with a binding for the inner `x`. Had the inner `x` been named something other than `x` or `y`, no capture would have occurred, and the expression would have denoted an error, as expected. But `x` is not the only outer name which would cause trouble; if we replace the outer `x` by a reference to `y`, the expression diverges! (Check it and see.) Because the semantics of call-by-denotation are so convoluted, it is hardly surprising that this mechanism is not used in any real programming language. (However, call-by-denotation does exhibit some of the behavior of macro languages.)

Then what is our purpose in introducing so contrived a mechanism? First, we wanted to emphasize that in a denotational semantics, names can be bound to entities much more complex than simple values or expressible values. We shall see many examples of this in the future. Second, we wanted to emphasize that just because a mechanism has an elegant denotational description doesn't necessarily mean that it is of any use in real programming languages. Although in some cases denotational descriptions *do* suggest powerful language constructs, other extensions suggested by denotational semantics (like call-by-denotation) turn out to be downright duds.

▷ **Exercise 7.4** For each of the following FL expressions, use the denotational semantics to give the meaning of the expression in CBN, CBV, and CBD. (Recall that `let` and `lambda` inherit their semantics from `proc`.)

a.
```
(let ((x 3)
      (y (/ 1 0)))
  x)
```

b.
```
(let ((x 7))
  (let ((f (lambda (y) (+ x y))))
    (let ((x 10))
      (f x))))
```

c.
```
(let ((x 3))
  ((let ((y 19))
     (lambda (z) y))
   x))
```

d.
```
(let ((x 23))
  (let ((x x))
    x))
```
◁

▷ **Exercise 7.5**

a. Write a single FL expression that has a different meaning in each of the three parameter passing mechanisms. Give the meaning of your expression in each mechanism.

b. Bud Lojack hopes to solve part a. above with an expression that evaluates to one of the symbols `call-by-name`, `call-by-value`, or `call-by-denotation` depending on which parameter passing mechanism is being used. Kindly explain why the expression that Bud desires does not exist. ◁

▷ **Exercise 7.6**

a. Can a CBN FL interpreter be written in CBV FL?

b. Can a CBV FL interpreter be written in CBN FL?

Justify your answers. ◁

▷ **Exercise 7.7** In this exercise we explore combining call-by-name and call-by-value in a single language.

Imagine a language NAVALNAVAL (NAme/VAlue Language) that is just like CBN FLK except that (`proc` *I E*) has been replaced by the two constructs (`vproc` *I E*) and (`nproc` *I E*). Both of these constructs act like `proc` in that they create single-argument procedures. The only difference between them is that procedures created by `nproc` pass parameters using CBN, while those created by `vproc` use CBV.

For example:

`(call (nproc x 3) (primop / 1 0))` $\xrightarrow[FL]{}$ *3*
`(call (vproc x 3) (primop / 1 0))` $\xrightarrow[FL]{}$ *error:divide-by-zero*

a. Provide a denotational semantics for NAVAL (only give those domain equations and valuation clause that differ from those for CBN FLK). Hint: In a simple approach to this problem, by-name and by-value procedures can both be elements of a single *Procedure* domain. What should *Denotable* be?

b. Just as it was convenient to extend FLK with the notion of multiple argument procedures, it would be nice to extend NAVAL with a similar notion. Some method must be chosen for specifying which parameters are by name and which are by value. For example, parameters might default to the by-value mechanism, but could be declared by-name with the token `name`, as illustrated below:

```
(define unless
  (lambda (test (name default) (name exception))
    (if test exception default)))
```

(unless (= 1 2) (+ 3 4) (/ 5 0)) $\xrightarrow[eval]{}$ *[int 7]*

Give the rules for desugaring such a multiple-argument `lambda` construct into NAVAL's one-argument `nproc` and `vproc`.

c. We have seen how CBN can be simulated in a CBV language using thunks. Formalize this transformation by showing how to translate NAVAL into CBV FLK. ◁

▷ **Exercise 7.8** Write a program that translates CBN FL into CBV FL. (Note that a very similar program could be used to translate CBN FL into Scheme.) ◁

▷ **Exercise 7.9** Show that when *Computation* = *Expressible*, the CBV valuation clause for `rec` denotes the same function as the CBN valuation clause for `rec`. ◁

▷ **Exercise 7.10** Use both the operational and denotational `rec` semantics to compute the values of the following expressions in both CBN and CBV versions of FL:

a. `(rec a 4)`

b. `(rec a (1 0))`

c. `(rec a a)`

d. `(rec a b)`

e. `(rec a (if #t 3 a))`

f. `(rec a (lambda (x) a))`

g. `(rec a (pair 1 a))`

h. `(rec a (pair (lambda (x) a)))`

i. `(let ((b 3)) (rec a b))` ◁

7.2 Name Control

The phrase "too much of a good thing" evokes images such as a child getting a stomach ache after eating too much candy or the crew of the starship Enterprise being swamped by the cute but prolific tribbles. The recent explosion in information technology has added a new twist to this phrase. Information consumers, such as television viewers, magazine subscribers, and readers of electronic mail, now have rapid access to incredible stores of information. But these changes in the information landscape have brought new problems, perhaps the most daunting of which is information overload: there is simply too much information to weed through, to absorb, to remember.

The area of naming in programming languages harbors its own version of the information overload problem. While names are an indispensable means of abstraction, the abundance of names in even modestly sized programs can lead to a host of complications.

From a cognitive point of view, more names can mean more learning, remembering, and model building for programmers. One of the simplest naming strategies, a single global namespace (in which distinct variables are named by distinct identifiers), is also one of the most nightmarish for program writers and readers alike. This approach foists a tremendous amount of mental bookkeeping on the programmer:

- Nonlocality of naming structure impairs readability because there are no constraints on which names the user needs to search or remember in order to understand a given fragment of code. At all times, the reader must potentially be aware of the entire namespace. For this reason, a global namespace is not scalable in a cognitive sense; large programs are much harder to comprehend than shorter ones.

- The reader has to infer structural groupings intended by the writer but not expressed due to the flatness of the namespace.

- Every time a new name is needed, the writer must find one that does not clash with any names already in use.

In order to reduce such unreasonable cognitive demands, programming languages typically provide mechanisms for reusing names and structuring the scope of names. Even when these are not supported by the naming system, programmers often develop naming conventions to simulate such mechanisms.

From an engineering point of view, more names can mean more complex interactions between program parts. One of the chief methods of controlling the complexity of large programs is to break them up into smaller units having well-defined **interfaces** that separate the use of a unit from its implementation. An interface specifies:

- the names defined external to the unit that are to be **imported** for use within the implementation of the unit; and

- the names defined internal to the unit that are to be **exported** for use outside of the unit.

It is desirable to make such interfaces narrow — i.e., importing and exporting few names — to limit dependencies among program parts. Wide interfaces give rise to spaghetti-like dependencies among program units that are difficult for a programmer to keep track of. And the complexities are only exacerbated in the more common situation where a large program is developed in collaboration with others. In this case, wider interfaces imply increased communication and coordination between members of a programming team.

From this engineering perspective, a programming language should provide mechanisms that facilitate the construction of narrow interfaces. The simple approach of a single global namespace strikes out again because it allows every name to be used everywhere throughout a program. A crucial ingredient for narrow interfaces is some means of **name hiding**, whereby names purely local to the implementation of a unit are effectively hidden from the rest of a program.

In this section, we shall investigate techniques for **name control** that address the cognitive and engineering problems outlined above. Unlike our discussion of names up to this point, these issues are largely orthogonal to the choice of denotable values. Rather, they specify the relationship between patterns of name usage and the logical structure of variables in the program.

7.2.1 Hierarchical Scoping: Static and Dynamic

Recall the following terms from our study of variables in FLK:

- A **variable** is an entity that names a value.
- An **identifier** is a name for a variable. Distinct variables may be named by the same identifier.
- A **variable declaration** is a construct that introduces a variable.
- A **variable reference** is a construct that stands for the value of a variable.
- The **scope** of a variable declaration is the portion of the program text over which the declared variable may be referenced.

For example, in FL, variables are declared in `lambda`, `letrec`, and `let` expressions. All variable references are written as unadorned identifiers.

For a given language, it may or may not be possible to determine the scope of a given declaration without running the program. If the scope of a declaration can always be determined from the abstract syntax tree of a program, the scope of the declaration is said to be **static** or **lexical**. In this case, the variable declaration associated with any variable reference is apparent from the lexical structure of the program. If the scope of the declaration depends on details of the run-time behavior of the program, the declaration is said to have **dynamic scope**. A language in which all declarations have static (dynamic) scope is said to be a **statically (dynamically) scoped** language.

Figure 7.9 summarizes the difference between static and dynamic scoping. We explain these in turn.

$$p \in \mathit{Procedure} = \mathit{Denotable} \to \mathit{Computation}$$

$$\mathcal{E}[\![\texttt{(proc}\ I\ E\texttt{)}]\!] = \lambda e_{proc} .\ (\mathit{val\text{-}to\text{-}comp}\ (\mathit{Procedure} \mapsto \mathit{Value}\ (\lambda\delta\ .\ (\mathcal{E}[\![E]\!]\ [I:\delta]e_{proc}))))$$

$$\mathcal{E}[\![\texttt{(call}\ E_1\ E_2\texttt{)}]\!] = \lambda e_{call} .\ (\mathit{with\text{-}procedure}\ (\mathcal{E}[\![E_1]\!]\ e_{call})\ (\lambda p\ .\ (p\ (\mathcal{E}[\![E_2]\!]\ e_{call}))))$$

Statically (Lexically) Scoped Procedures

$$p \in \mathit{Procedure} = \mathit{Denotable} \to \mathit{Environment} \to \mathit{Computation}$$

$$\mathcal{E}[\![\texttt{(proc}\ I\ E\texttt{)}]\!] = \lambda e_{proc} .\ (\mathit{val\text{-}to\text{-}comp}\ (\mathit{Procedure} \mapsto \mathit{Value}\ (\lambda\delta\, e_{call}\ .\ (\mathcal{E}[\![E]\!]\ [I:\delta]e_{call}))))$$

$$\mathcal{E}[\![\texttt{(call}\ E_1\ E_2\texttt{)}]\!] = \lambda e_{call} .\ (\mathit{with\text{-}procedure}\ (\mathcal{E}[\![E_1]\!]\ e_{call})\ (\lambda p\ .\ (p\ (\mathcal{E}[\![E_2]\!]\ e_{call})\ e_{call})))$$

Dynamically Scoped Procedures

Figure 7.9: The essential semantics of statically and dynamically scoped CBN procedures (CBV is analogous). In a statically (lexically) scoped procedure, free identifiers appearing in a `proc` body are resolved relative to e_{proc}, the environment determined by the text enclosing the `proc` expression. In a dynamically scoped procedure, free identifiers appearing in a `proc` body are resolved relative to e_{call}, the environment determined by the dynamic chain of procedure calls in which the procedure is being called.

7.2.1.1 Static Scope

All of the languages we have studied so far (other than call-by-denotation FL) have been statically scoped. In fact, all of these languages support a particular discipline of static scoping known as **block structure**. In a block structured language, declarations can be nested arbitrarily, and every variable reference refers to the variable introduced by the nearest lexically enclosing variable declaration of that identifier. The nearest lexically enclosing declaration is found by starting at the identifier and walking up the abstract syntax tree until a declaration introducing the identifier name is found.

As an example, consider the CBV FL expression

```
(let ((x 20))
  ((let ((increment-by-x (lambda (y) (+ x y)))
         (double (lambda (x) (* x 2))))
     (letrec ((x (cons 1 x)))
       (lambda (z)
         (cons (double (increment-by-x (double z)))
               x))))
   (- x 15)))
```

In this expression there are three distinct variables named `x` introduced by three declarations:

1. `(let ((x 20)) ...)` declares `x` and binds it to the number 20.
2. The `(lambda (x) (* x 2))` expression named by `double` declares `x` but does not bind it; `x` will be bound on application of the procedural value of this abstraction. (In fact, the binding of `x` may be different for every distinct application of this procedure.)
3. `(letrec ((x (cons 1 x))) ...)` declares `x` and binds it to an infinite list of 1s.

There are also five variable references involving `x`:

1. In `(+ x y)`, `x` is a reference to the `let`-bound variable named `x`, so its meaning in this context is 20. This means that `increment-by-x` is a procedure that always adds 20 to its argument, regardless of the binding for `x` in whatever environment it happens to be applied.
2. In `(* x 2)`, `x` is a reference to the `lambda`-bound variable named `x`, whose meaning will be determined at application time.
3. In `(cons 1 x)`, `x` is a reference to the `letrec`-bound variable named `x`, so its meaning is an infinite list of 1s.

4. In `(cons (double (increment-by-x (double z))) x)`, `x` refers to the variable introduced by the first lexically enclosing declaration of `x`, which in this case is the `letrec`-bound variable. So here `x` is an infinite list of 1s as well.

5. In `(- x 15)`, `x` is a reference to the variable introduced by the first lexically enclosing declaration of `x`, which in this case is the `let`-bound variable. So here `x` means the number 20.

Putting together all of the above information, the value of the example expression is an infinite list whose first element is 60, and the rest of whose elements are all 1.

Variables in a block structured language have a structure reminiscent of variables in the lambda calculus.

As we noted before for FLK, when the scope of a declaration contains another declaration of the same name, the inner declaration carves out a **hole in the scope** of the outer one. The Stoy diagrams we used to represent the structure of lambda terms could easily be adapted to show declaration/reference relationships in any block structured language.

The essence of block structure is in the way environments are handled by abstractions. Figure 7.9 shows the domains and valuation functions that are crucial for block structure in CBN FLK. The clause for `proc` dictates that the body of the abstraction will be evaluated with respect to the environment in effect *when the procedure was created*. In particular, the environment in which the procedure is called can have no effect on the meaning of names within the abstraction body. This is clear from the domain definition for *Procedure*, which simply maps denotable values to computations and ignores whatever the current environment might be. Though the details of expressible and denotable values might differ under other parameter passing mechanisms, the handling of environments will have this form in any block structured language.

7.2.1.2 Dynamic Scope

SNOBOL4, APL, most early LISP dialects, and many macro languages are dynamically scoped. In each of these languages, a free variable in a procedure (or macro) body gets its meaning from the environment at the point where the procedure is called rather than the environment at the point where the procedure is created. Thus, in these languages, it is not possible to determine a unique declaration corresponding to a given free variable reference; the effective declaration depends on where the procedure is called. It is therefore generally impossible to determine the scope of a declaration simply by considering the abstract syntax tree of the program. Instead, the scope of a variable declaration depends on the run-time tree of procedure calls.

Figure 7.9 also shows essential semantics of dynamic scoping for a CBN language. The *Procedure* domain has been modified to indicate that procedures take an extra argument: the dynamic environment (i.e., the call-time environment). In the valuation clause for `proc`, the body of the abstraction is evaluated in the dynamic environment rather than the lexical one. The clause for `call` has been modified to pass the current environment to the procedure being called.

As an example of static vs. dynamic scoping, consider the following expression in CBV FL:

```
(let ((a 1))
  (let ((f (lambda (x) (primop + x a))))
    (let ((a 20))
      (f 300))))
```

Informally, we can reason as follows. The procedure named `f` refers to a free variable `a`. Under static scoping, this variable is bound to the value of `a` where the procedure is defined (i.e., `1`). Thus, the binding between `a` and `20` is irrelevant, and the result of the call `(f 300)` is `301`. On the other hand, under dynamic scoping, the free variable gets its value from whatever binding of `a` is dynamically apparent. In the call `(f 300)`, the binding between `a` and `20` shadows the binding between `a` and `1`, so the value of the call is `320`.

We can use the denotational definitions of scoping to formally analyze this example. The example FL expression desugars into the following FLK program:

```
(call (proc a                              ; E_proc:a1
        (call (proc f                      ; E_proc:f
                (call (proc a (call f 300)) ; E_proc:a20
                      20))
              (proc x (primop + x a))))    ; E_proc:x
      1)
```

The four `proc` expressions have been commented with names that will be used to abbreviate them. Figure 7.10 and 7.11 highlight the key steps for using the denotational definitions to derive the value of the expression under static scoping and dynamic scoping.

A more graphical perspective of these derivations appears in Figure 7.12. Each derivation is summarized by an **environment diagram** that shows key expressions along with the environments they are evaluated in. An environment is represented by a chain of bindings that go up the page; this helps to clarify the relationship between the different environments. The static scoping example is depicted in Figure 7.12(a). The arrow from within the procedural value to the

STATIC SCOPING

$\mathcal{E}[\![\texttt{(call } E_{proc:a1}\ \texttt{1)}]\!]\ e_0$

$(\textit{with-procedure}\ (\mathcal{E}[\![E_{proc:a1}]\!]\ e_0)\ (\lambda p\ .\ (p\ (\textit{Int} \mapsto \textit{Value}\ 1))))$

$\mathcal{E}[\![\texttt{(call } E_{proc:f}\ E_{proc:x}\texttt{)}]\!]\ e_1$
where $e_1 = [\text{a: } (\textit{Int} \mapsto \textit{Value}\ 1)]\ e_0$

$(\textit{with-procedure}\ (\mathcal{E}[\![E_{proc:f}]\!]\ e_1)$
$(\lambda p\ .\ (p\ (\textit{Procedure} \mapsto \textit{Value}\ (\lambda\delta\ .\ (\mathcal{E}[\![\texttt{(primop + x a)}]\!]\ [\texttt{x} : \delta]e_1))))))$

$\mathcal{E}[\![\texttt{(call } E_{proc:a20}\ \texttt{20)}]\!]\ e_2$
where $e_2 = [\texttt{f}\text{: } (\textit{Procedure} \mapsto \textit{Value}$
$(\lambda\delta\ .\ (\mathcal{E}[\![\texttt{(primop + x a)}]\!]\ [\texttt{x} : \delta]e_1)))]e_1$

$(\textit{with-procedure}\ (\mathcal{E}[\![E_{proc:a20}]\!]\ e_2)\ (\lambda p\ .\ (p\ (\textit{Int} \mapsto \textit{Value}\ 20))))$

$\mathcal{E}[\![\texttt{(call f 300)}]\!]\ e_3$
where $e_3 = [\texttt{a} : (\textit{Int} \mapsto \textit{Value}\ 20)]e_2$

$(\textit{with-procedure}\ (\mathcal{E}[\![\texttt{f}]\!]\ e_3)\ (\lambda p\ .\ (p\ (\textit{Int} \mapsto \textit{Value}\ 300))))$

$((\lambda\delta\ .\ (\mathcal{E}[\![\texttt{(primop + x a)}]\!]\ [\texttt{x} : \delta]e_1))\ (\textit{Int} \mapsto \textit{Value}\ 300))$

$\mathcal{E}[\![\texttt{(primop + x a)}]\!]\ [\texttt{x} : (\textit{Int} \mapsto \textit{Value}\ 300)]e_1$

$\mathcal{E}[\![\texttt{(primop + x a)}]\!]\ [\texttt{x} : (\textit{Int} \mapsto \textit{Value}\ 300)][\texttt{a} : (\textit{Int} \mapsto \textit{Value}\ 1)]e_0$

$= (\textit{Int} \mapsto \textit{Value}\ 301)$

Figure 7.10: $\texttt{(call } E_{proc:a1}\ \texttt{1)}$ evaluation using static scoping

DYNAMIC SCOPING

$\mathcal{E}[\![$`(call` $E_{proc:a1}$ `1)`$]\!]\ e_0$

$(\textit{with-procedure}\ (\mathcal{E}[\![E_{proc:a1}]\!]\ e_0)\ (\lambda p\ .\ (p\ (\textit{Int} \mapsto \textit{Value}\ 1)\ e_0)))$

$\mathcal{E}[\![$`(call` $E_{proc:f}\ E_{proc:x}$`)`$]\!]\ e_1$
where $e_1 = [\mathtt{a} : (\textit{Int} \mapsto \textit{Value}\ 1)]e_0$

$(\textit{with-procedure}\ (\mathcal{E}[\![E_{proc:f}]\!]\ e_1)$
$(\lambda p\ .\ (p\ (\textit{Procedure} \mapsto \textit{Value}\ (\lambda \delta\, e'\ .\ (\mathcal{E}[\![\mathtt{(primop\ +\ x\ a)}]\!]\ [\mathtt{x} : \delta]e')))$
$e_1))$

$\mathcal{E}[\![$`(call` $E_{proc:a20}$ `20)`$]\!]\ e_2$
where $e_2 = [\mathtt{f}\text{: } (\textit{Procedure} \mapsto \textit{Value}$
$(\lambda \delta\, e'\ .\ (\mathcal{E}[\![\mathtt{(primop\ +\ x\ a)}]\!]\ [\mathtt{x} : \delta]e')))]e_1$

$(\textit{with-procedure}\ (\mathcal{E}[\![E_{proc:a20}]\!]\ e_2)\ (\lambda p\ .\ (p\ (\textit{Int} \mapsto \textit{Value}\ 20)\ e_2)))$

$\mathcal{E}[\![$`(call f 300)`$]\!]\ e_3$
where $e_3 = [\mathtt{a} : (\textit{Int} \mapsto \textit{Value}\ 20)]e_2$

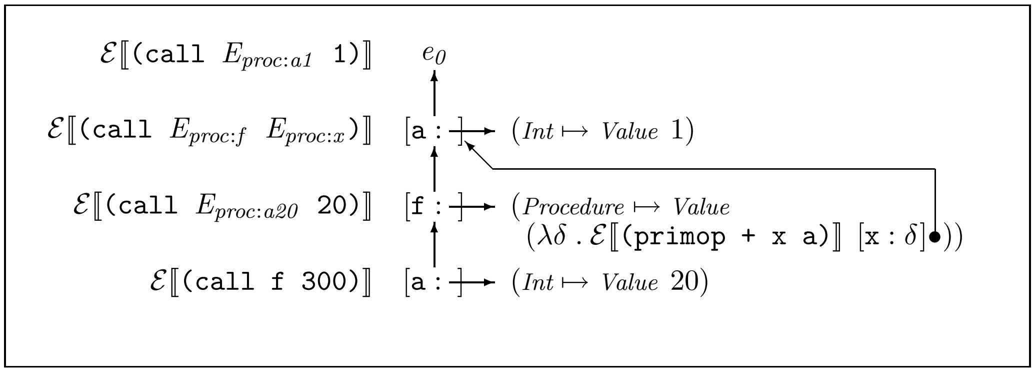
$(\textit{with-procedure}\ (\mathcal{E}[\![\mathtt{f}]\!]\ e_3)\ (\lambda p\ .\ (p\ (\textit{Int} \mapsto \textit{Value}\ 300)\ e_3)))$

$((\lambda \delta\, e'\ .\ (\mathcal{E}[\![\mathtt{(primop\ +\ x\ a)}]\!]\ [\mathtt{x} : \delta]e'))\ (\textit{Int} \mapsto \textit{Value}\ 300)\ e_3)$

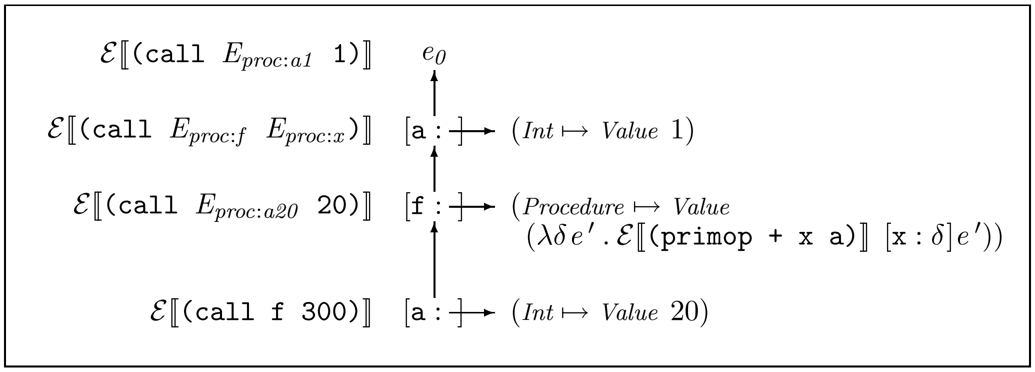
$\mathcal{E}[\![\mathtt{(primop\ +\ x\ a)}]\!]\ [\text{x: } (\textit{Int} \mapsto \textit{Value}\ 300)][\text{a: } (\textit{Int} \mapsto \textit{Value}\ 20)]\ e_2$

$= (\textit{Int} \mapsto \textit{Value}\ 320)$

Figure 7.11: `(call` $E_{proc:a1}$ `1)` evaluation using dynamic scoping


$\mathcal{E}[\![$(call $E_{proc:a1}$ 1)$]\!]$
e_0
$\mathcal{E}[\![$(call $E_{proc:f}$ $E_{proc:x}$)$]\!]$
[a :]
($Int \mapsto Value$ 1)
$\mathcal{E}[\![$(call $E_{proc:a20}$ 20)$]\!]$
[f :]
($Procedure \mapsto Value$
($\lambda\delta$. $\mathcal{E}[\![$(primop + x a)$]\!]$ [x : δ]●))
$\mathcal{E}[\![$(call f 300)$]\!]$
[a :]
($Int \mapsto Value$ 20)

(a) Environment diagram for the sample expression under static scoping.


$\mathcal{E}[\![$(call $E_{proc:a1}$ 1)$]\!]$
e_0
$\mathcal{E}[\![$(call $E_{proc:f}$ $E_{proc:x}$)$]\!]$
[a :]
($Int \mapsto Value$ 1)
$\mathcal{E}[\![$(call $E_{proc:a20}$ 20)$]\!]$
[f :]
($Procedure \mapsto Value$
($\lambda\delta e'$. $\mathcal{E}[\![$(primop + x a)$]\!]$ [x : δ]e'))
$\mathcal{E}[\![$(call f 300)$]\!]$
[a :]
($Int \mapsto Value$ 20)

(b) Environment diagram for the sample expression under dynamic scoping.

Figure 7.12: Environment diagrams illustrating the difference between statically and dynamically scoped procedures.

environment starting with [a : 1] emphasizes that a statically scoped procedure "remembers" the environment in which it was created. This **lexical environment** is determined by the text lexically surrounding the proc expression that gave rise to the procedure value.

The dynamic scoping example is depicted in Figure 7.12(b). Here, there is no arrow emanating from the procedural value because the environment e' in which the body is evaluated will be the **dynamic environment** in effect when the procedure is called. The dynamic environment is determined by the bindings in the current branch of the tree of procedure calls made during the execution of the program. In this example, it is constructed by the procedure calls associated with the three nested let expressions.

Although the environment chains happen to be the same for these two examples, lexically scoped languages tend to give rise to shallow, bushy environment diagrams, while dynamically scoped languages tend to give rise to deep thin ones (see Exercise 7.13).

Dynamic scoping seems rather odd. Is it useful? Yes! Dynamic scope is convenient for specifying the values of implicit parameters that are cumbersome to list explicitly as formal parameters to procedures. For example, consider the derivative procedure:

```
(define derivative
  (lambda (f x)
    (/ (- (f (+ x epsilon))
          (f x))
       epsilon)))
```

Note that epsilon appears as a free variable in derivative. With dynamic scoping, it is possible to dynamically specify the value of epsilon via any binding construct. For example, the expression

```
(let ((epsilon 0.001))
  (derivative (lambda (x) (* x x)) 5.0))
```

would evaluate (derivative (lambda (x) (* x x)) 5.0) in a context where epsilon is bound to 0.001.

However, with lexical scoping, the variable epsilon must be defined at top level, and, without using mutation, there is no way to temporarily change the value of epsilon while the program is running. If we really want to abstract over epsilon with lexical scoping, we must pass it to derivative as an explicit argument:

```
(define derivative
  (lamdba (f x epsilon)
    (/ (- (f (+ x epsilon))
          (f x))
       epsilon)))
```

But then any procedure that uses `derivative` and wants to abstract over `epsilon` must also include `epsilon` as a formal parameter. In the case of `derivative`, this is only a small inconvenience. But in a system with a large number of tweakable parameters, the desire for fine-grained specification of variables like `epsilon` can lead to an explosion in the number of formal parameters throughout a program.

As an example along these lines, consider the huge parameter space of a typical window system (colors, fonts, stippling patterns, line thicknesses, etc.). It is untenable to specify each of these as a formal parameter to every window routine. At the very least, all these parameters need to be bundled up into a data structure that represents the graphics state. But then we still want a means of executing window routines in a temporary graphics state in such a way that the old graphics state is restored when the routines are done. Dynamic scoping is one technique for achieving this effect; side-effects are another.

Another typical use of dynamic scope is to specify error handling routines that are in effect during the execution of an expression. We shall see an example of this in Chapter 9.

Although dynamic scoping is good for allowing the specification of implicit parameters, it is seriously at odds with modularity, especially in a language that has first-class procedure values. We explore this issue in the exercises.

▷ **Exercise 7.11**

a. Can a dynamically scoped FL interpreter be written in statically scoped FL?

b. Can a statically scoped FL interpreter be written in dynamically scoped FL? ◁

▷ **Exercise 7.12** Write a single FL expression that exhibits a different behavior in each of the four following scenarios:

a. statically scoped CBN FL

b. statically scoped CBV FL

c. dynamically scoped CBN FL

d. dynamically scoped CBV FL ◁

▷ **Exercise 7.13** This problem considers a dynamically scoped variant of FL called FLUID. The abstract syntax for FLUID is the same as that for FL except that the grammar for FLUID does not include any recursion constructs. That is, the FLUID kernel does not contain the `rec` construct `(rec` I E`)`; and FLUID does not contain the `letrec` construct `(letrec ((`I E`)*)` E`)`. The denotational semantics for FLUID is the same for that as FL except for the changes specified in Figure 7.9.

a. For each of the expressions below, show the result of evaluating the expression both in FL and in FLUID. Refer to the denotational semantics as necessary to reason about the evaluation process, but don't get lost in a symbol manipulation quagmire. You may find environment diagrams helpful for thinking about these problems.

   i.
   ```
   (let ((a 1))
     (let ((f (lambda (a) (primop + a 20))))
       (f a)))
   ```

   ii.
   ```
   (let ((a 1))
     (let ((f (lambda (a b) (primop + a b))))
       (f 20 300)))
   ```

   iii.
   ```
   (let ((a 1))
     (let ((a 20)
           (b 300))
       (primop + a b)))
   ```

   iv.
   ```
   (let ((a 1))
     (let ((f (lambda (b) (primop + a b))))
       (f (let ((a (f 20)))
            (f 300)))))
   ```

   v.
   ```
   (let ((a 1))
     (let ((f (lambda (b) (primop + a b))))
       (let ((g (lambda (a) (f a))))
         (g (g a)))))
   ```

b. In FLUID, the usual desugaring of multiple-argument abstractions into single argument abstractions no longer behaves as expected. Explain what goes wrong with the usual desugaring. (You do *not* need to describe how to fix the problem.)

c. In FLK, the factorial procedure is written as the expression:

```
(rec fact (proc n
            (if (primop = n 0)
                1
                (primop * n (fact (primop - n 1))))))
```

FLUID has no recursion constructs, but none are needed to write recursive definitions.

i. Briefly explain why the above claim is true.
ii. Show the definition for factorial procedure in FLUID.
iii. Explain why your FLUID definition for factorial wouldn't work in FL.

d. Consider the factorial procedure from part c. When using the denotational semantics to determine the meaning of `(call fact 3)` in environment e_0, the meaning of `(primop = n 0)` is determined in four distinct environments. For both CBV FL and for FLUID, draw an environment diagram that shows the relationship between these four environments. ◁

▷ **Exercise 7.14** Consider a version of FL called FLAT in which a procedure (a `lambda` or kernel `proc` expression) is not allowed to have free identifiers. Can the meaning of a FLAT expression differ under lexical and dynamic scope? If so, exhibit such an expression; if not, explain why. ◁

▷ **Exercise 7.15** Develop an operational semantics for CBV FL that uses explicit environments instead of substitution. ◁

▷ **Exercise 7.16** Develop an operational semantics for a dynamically scoped version of CBV FL. ◁

▷ **Exercise 7.17** The static scope expressed in Figure 7.9 is typical of block structured languages. However, other kinds of static scope are imaginable. For example, suppose that e_{global} is the top-level FL environment — the one that defines the meanings of all of the standard library names (e.g., `+`, `boolean?`, `cons`, etc.). Then **global scoping** is a static scoping mechanism in which free identifiers in a `proc` expression are resolved relative to e_{global} rather than the environment at the time of procedure creation or at the time of procedure call.

a. Write the valuation clauses for `proc` and `call` for a CBV variant of FL with global scoping.

b. Write a single FL expression whose value is a symbol (one of `global`, `block-structure`, or `dynamic`) indicating the scoping mechanism under which it is evaluated. ◁

▷ **Exercise 7.18** In this problem, we ask you to give a translation from dynamically scoped, call-by-value FLK to PostLisp, the language defined in Exercise 3.45. You are only required to translate a subset of FLK, defined by the following grammar:

E ::= U | I | `(proc` I E`)` | `(call` E_1 E_2`)` | `(primop /` E_1 E_2`)`

Your translation should map every expression E_{FLK} of the subset to a sequence $Q_{PostLisp}$ of PostLisp commands such that:

$$\begin{array}{rcl} E_{FLK} \xrightarrow[DCBV]{} U & \text{if and only if} & (Q_{PostLisp}) \xrightarrow[PostLisp]{} U, \\ E_{FLK} \xrightarrow[DCBV]{} \mathit{error} & \text{if and only if} & (Q_{PostLisp}) \xrightarrow[PostLisp]{} \mathit{error}, \text{ and} \\ E_{FLK} \xrightarrow[DCBV]{} \infty\!-\!\mathit{loop} & \text{if and only if} & (Q_{PostLisp}) \xrightarrow[PostLisp]{} \infty\!-\!\mathit{loop}, \end{array}$$

where $\xrightarrow[DCBV]{}$ means dynamically scoped, call-by-value FLK evaluation. ◁

▷ **Exercise 7.19** Alyssa P. Hacker is asked by Analog Equipment Corporation to change their version of FL to be dynamically scoped in response to customer demand. Alyssa is asked to do this over a weekend, but she does not panic. Instead, she realizes that by implementing just a few new primitives, the entire job can be accomplished with clever desugaring.

More specifically, Alyssa added the following three new primitives:

- `(%new)`: Creates a new, empty environment.
- `(%extend` ENV `(symbol` I`)` V`)`: Returns a new environment equal to ENV, except that I is bound to V.
- `(%lookup` ENV `(symbol` I`))`: Returns the value of identifier I in environment ENV. It is is a fatal error if I is not bound in ENV.

In Alyssa's desugaring, the `*dynenv*` variable is always bound to the current dynamic environment. For this problem, consider only single argument procedures and calls. Here is Alyssa's desugaring rule for `call`:

$\mathcal{D}[\![$`(call` E_1 E_2`)`$]\!]$ = `(call` $\mathcal{D}[\![E_1]\!]$ `*dynenv*` $\mathcal{D}[\![E_2]\!]$`)`

a. What is the desugaring rule for I (variable reference)?

b. What is the desugaring rule for `(lambda (`I`)` E`)`?

c. What is the desugaring rule for `(let ((`I_1 E_1`) ... (`I_n E_n`))` E_{body}`)`?

d. Do all identifiers have to be looked up in the dynamic environment? If not, state what optimizations of the desugarings for identifiers and lambda are possible, and when and how they could be accomplished.

e. Desugar the following expression in this dynamically scoped version of FL:

```
(lambda (g) (call g x))
```

◁

7.2.2 Multiple Namespaces

Sometimes a single environment is not sufficient to model the naming features of a programming language. Languages commonly support **multiple namespaces** — i.e., several different contexts in which names are associated with values of various sorts. For example, Figure 7.13 shows a piece of COMMON LISP code in which the name `x` is used to name five different entities at the same time: an exit point, a special (dynamic) variable, a lexical variable, a procedure, and a tagbody tag.

```
(block x                              ; x1, name of exit point
  (let ((x 2))                        ; x2, declared to be
    (declare (special x))             ;   a special variable
    (let ((x 3))                      ; x3, normal lexical variable
      (flet ((x (y)                   ; x4, names a procedure that
                (+ x y)))             ;   refers to x3 in its body
        (tagbody
           x                          ; x5, a tagbody tag
           (if (> x 6)                ; this x = x3 = 3
               (go x)                 ; go to x5 if we get here
               (return-from
                 x                    ; return from exit point x1
                 (locally (declare (special x))
                                      ; Make 2nd x special below
                          (x x)       ; Apply procedure x4 to
                  ))))))))            ;  special x2
; The value of this expression is 5.
```

Figure 7.13: COMMON LISP code that uses multiple namespaces

There are two typical situations in which multiple namespaces are useful:

1. The language provides multiple scoping mechanisms. In this case, different namespaces can be used for different scoping mechanims. COMMON LISP, for example, supports both lexical and dynamic scoping of variables; variables are ordinarily scoped lexically, but those marked as `special` are dynamically scoped.

2. Different namespaces are used to name different kinds of entities. For example, exit points, tag labels, and procedures are in non-overlapping namespaces in Common Lisp. Namespaces used this way are especially useful for modeling values that are not first-class.

Of course, any language with multiple namespaces must provide methods for both binding names and accessing names within each namespace. For example, consider the namespaces for exit points and tags in the Common Lisp example above. `block` introduces a name into the exit point namespace, and `return-from` accesses the exit point name, whereas `tagbody` introduces new tag names into the namespace of tags, and `go` can refer to these tags.

Multiple namespaces are modeled in denotational semantics by using multiple environments. For example, we could modify the semantics of FL to

- support both lexical and dynamic scoping;
- make procedures second-class objects.

These modifications are left as exercises.

▷ **Exercise 7.20** DYNALEX Understanding the virtues of both lexical and dynamic scoping, Sam Antix decides to design a language, DYNALEX, that supports both kinds of scoping mechanisms. The kernel of DYNALEX is statically scoped CBV FLK, extended with the following extra constructs to support dynamic scoping:

> `(dylambda (`I_{dyn}`*) `E_{body}`)` is like `lambda`, but binds the names I_{dyn}* in a dynamic environment rather than a static one.
>
> `(dyref `I`)` looks up I in the dynamic environment rather than the lexical one.

The full DYNALEX language includes the usual FL sugar as well as the following sugar for the `dylet` construct:

$$\mathcal{D}_{\text{exp}}[\![\texttt{(dylet ((}I_1\ E_1\texttt{)} \ldots \texttt{(}I_n\ E_n\texttt{))}\ E_{body}\texttt{)}]\!] = \texttt{((dylambda (}I_1 \ldots I_n\texttt{)}\ \mathcal{D}_{\text{exp}}[\![E_{body}]\!]\texttt{)}\ \mathcal{D}_{\text{exp}}[\![E_1]\!] \ldots \mathcal{D}_{\text{exp}}[\![E_n]\!]\texttt{)}$$

The following DYNALEX expression illustrates both dynamic and lexical scoping:

```
(let ((a 1) (b 20))
  (let ((f (lambda () (+ a (dyref b)))))
    (dylet ((a 300) (b 4000))
      (f)))) ──DYNALEX──> 4001
```

a. Sketch a denotational semantics for DYNALEX that includes the the signature of $\mathcal{E}$, and the valuation clauses for the following constructs: I, `proc`, `call`, `dyref`, and `dylambda`.

b. Explain why Sam chose to make the multi-argument `dylambda` abstraction a kernel form rather than treating `dylambda` as sugar for a single argument abstraction for dynamic variables.

c. Write a set of translation rules for translating the DYNALEX kernel into FLK. ◁

7.2.3 Non-hierarchical Scope

7.2.3.1 Philosophy

The binding constructs we have seen so far are all **hierarchical** in nature. Each construct establishes a parent-child relationship between an outer context in which the declaration is not visible and an inner (body) context in which the declaration is visible. In static scoping, the hierarchy is determined by the abstract syntax tree, while in dynamic scoping, the hierarchy is determined by the tree of procedure calls generated at run-time. In both these scoping mechanisms, there is no natural way to communicate a declaration *laterally* across the tree-structure imposed by the hierarchy.

For small programs, this is not ordinarily a problem, but when a large program is broken into independent pieces, or **modules**, the constraint of hierarchy can be a problem. Modules connect and communicate with each other via collections of bindings; a module provides services by exporting a set of bindings and makes use of other modules' services by importing bindings from those other modules. In a hierarchical language, the scope of a binding is a single region of a program, so all the clients of a module must reside in the region where the module's bindings are in scope.

The traditional solution to the problem of communicating modules is to use a global namespace. All exported bindings from all modules are defined in a single environment, so all exported bindings are available to all modules. This technique is certainly widespread, but it has some major drawbacks:

- In order to avoid accidental name collisions, every module must be aware of all definitions made by all other modules, even those definitions that are completely irrelevant.

- In practice, the dependencies among modules are often poorly documented, making intermodule dependencies difficult to track.

A way for languages to overcome the hierarchical scoping of binding constructs is to provide a value with named subparts. For this purpose, we will introduce a new module value that bundles up a set of bindings at one point in a program and can communicate them to a point that is related neither lexically nor dynamically to the declarations of those bindings. Typically, a module defines a set of named values, especially procedures, that provide a particular function. For example, a matrix module might provide a set of matrix manipulation procedures like `matrix-invert` and `gaussian-elimination`. The modules described here are similar to PASCAL records and C structures.

We will study modules here in the context of a record package for FL. (Chapter 15 will explore a more complete module system.) Figure 7.14 lists new kernel forms for records.

`(record (`I E`)*)`	Create a record.
`(select` I E`)`	Select field I from record E.
`(override` E_1 E_2`)`	Append the named components of two records, giving precedence to names in E_2.
`(conceal (`I`*)` E`)`	Return a new record without specified fields.

Figure 7.14: Kernel record constructs.

`record` builds a data structure of name/value bindings. Values can be extracted by name using the `select` construct. Two records can be combined into a new record with `override`, which gives precedence to the bindings in the second record argument. `conceal` returns a new record in which some bindings of the original record have been removed. For example:

```
(define m1 (record
             (a (+ 2 3))
             (square (lambda (x) (* x x)))))
```

`(select a m1)` $\xrightarrow[FL]{}$ `5`

`((select square m1) (select a m1))` $\xrightarrow[FL]{}$ `25`

`(select b m1)` $\xrightarrow[FL]{}$ *error:no-such-record-field*

`(define m2 (record (a 7) (b 11)))`

`(select a (override m1 m2))` $\xrightarrow[FL]{}$ `7`

`(select a (override m1 (conceal (a) m2)))` $\xrightarrow[FL]{}$ `5`

Notice that there is a design choice in the semantics of `conceal`: the language designer must choose whether or not it is an error when `conceal` attempts to hide a name that is not actually a field in the record.

Figure 7.15 shows some convenient sugar constructs for records. Like many desugarings, these capture handy idioms programmers would invent on their own. `recordrec` allows the fields of a record to be mutually recursive. This is useful when records are used to construct separate program modules that contain procedures. `with-fields` provides a lexical scope that binds the specified names exported by a record, saving the programmer the tiresome task of writing `select`

`(recordrec (`I E`)*)`	A record with mutually recursive bindings.
`(with-fields (`I*`)` E_1 E_2`)`	Bind the I* to the corresponding fields in the record value E_1 and then evaluate E_2.
`(restrict (`I*`)` E`)`	Return a new record with only the specified fields. The dual of `conceal`.
`(rename ((`I_{old} I_{new}`)*)` E`)`	Rename I_{old} fields to I_{new} fields in E.

Figure 7.15: Record sugar constructs.

everywhere (or introducing the `let`s manually). Note that `with-fields` requires the list of identifiers to be bound. This allows the bindings in this lexical scope to be apparent,[4] which is necessary in a block structured language, and it also allows the programmer to avoid introducing unnecessary names. `restrict` is the natural dual to `conceal`, useful when exporting comparatively few names. `rename` helps programmers avoid name conflicts.

The desugarings for these constructs appear in Figure 7.16. Just as with `conceal`, the language designer must choose whether names not defined by a record generate errors. In a CBV language, the desugaring rules will generate an error when the undefined name is selected.

```
D[[(recordrec (I_1 E_1) ... (I_n E_n))]] =
  (letrec ((I_1 D[[E_1]]) ... (I_n D[[E_n]]))
    (record (I_1 I_1) ... (I_n I_n)))

D[[(with-fields (I_1 ... I_n) E_1 E_2)]] =
  (let ((I_fresh D[[E_1]]))
    (let ((I_1 (select I_1 I_fresh)) ... (I_n (select I_n I_fresh)))
      D[[E_2]]))

D[[(restrict (I_1 ... I_n) E)]] =
  (let ((I_fresh D[[E]]))
    (record (I_1 (select I_1 I_fresh)) ... (I_n (select I_n I_fresh))))

D[[(rename ((I_1 I_1') ... (I_n I_n')) E)]] =
  (let ((I_fresh D[[E]]))
    (override (conceal (I_1 ... I_n) I_fresh)
              (record (I_1' (select I_1 I_fresh)) ... (I_n' (select I_n I_fresh)))))
```

Figure 7.16: Desugarings for the record syntactic sugar.

[4]We will see in Chapters ?? and 15 how a statically typed language could deduce this information.

7.2.3.2 Semantics

Figure 7.17 and 7.18 present a denotational semantics for records. Since both records and environments associate names and values, it is natural to model a record as an environment. You are encouraged to develop an operational semantics for the record constructs.

```
$v \in Value = \ldots + Record$
$r \in Record = Environment$

$with\text{-}record : Computation \to (Record \to Computation) \to Computation$
 Defined like with-boolean and with-procedure.

$extend\text{-}env^* : Environment \to \text{Identifier}^* \to Denotable^* \to Environment$
   $= \lambda e I^* \delta^* .$ matching $\langle I^*, \delta^* \rangle$
                 $\triangleright \langle [], [] \rangle \ \| \ e$
                 $\triangleright \langle I.I_{rest}{}^*, \delta.\delta_{rest}{}^* \rangle \ \| \ (extend\ (extend^*\ e\ I_{rest}{}^*\ \delta_{rest}{}^*)\ I\ \delta)$
                 $\triangleright$ else (err-to-comp no-such-record-field)
                 endmatching

$combine\text{-}env : Environment \to Environment \to Environment$
   $= \lambda e_1 e_2 .\ \lambda I .$ matching $(lookup\ e_1\ I)$
                      $(Denotable \mapsto Binding\ \delta) \ \| \ (lookup\ e_1\ I)$
                      $(Unbound \mapsto Binding\ \text{unbound}) \| \ (lookup\ e_2\ I)$
                      endmatching
```

Figure 7.17: Domains and auxiliary functions for a denotational semantics of the kernel record constructs in a CBV language.

7.2.3.3 Examples

Figure 7.19 presents two modules for arithmetic: one for integers and one for rational numbers.

The `gcd` routine is used by the `rats` module to remove common factors from the numerator and denominator. Note how `recordrec` (and not `record`) is used to create a recursive scope in which the `rat` constructor is visible to other procedures in the module. Here is a sample use of the two modules:

```
$\mathcal{E}[\![$(record ($I_1$ $E_1$) ... ($I_n$ $E_n$))$]\!]$ $e$ =
    ($with\text{-}values$ $\mathcal{E}^*[\![$[$E_1$ ... $E_n$]$]\!]$
      ($\lambda v^*$ . ($val\text{-}to\text{-}comp$ ($Record \mapsto Value$ (extend* [$I_1 \ldots I_n$] $v^*$ empty-env)))))

$\mathcal{E}[\![$(select $I$ $E$)$]\!]$$e$ =
    ($with\text{-}record$ ($\mathcal{E}[\![E]\!]$ $e$)
      ($\lambda r$ . ($with\text{-}denotable$ ($lookup$ ($Record \mapsto Environment$ $r$) $I$)
                    ($\lambda \delta$ . ($den\text{-}to\text{-}comp$ $\delta$))))

$\mathcal{E}[\![$(override $E_1$ $E_2$)$]\!]$$e$ =
    ($with\text{-}record$ ($\mathcal{E}[\![E_1]\!]$ $e$)
      ($\lambda r_1$ . ($with\text{-}record$ ($\mathcal{E}[\![E_2]\!]$ $e$)
                    ($\lambda r_2$ . ($val\text{-}to\text{-}comp$
                              ($Record \mapsto Value$
                                    (combine-env ($Record \mapsto Environment$ $r_2$)
                                                 ($Record \mapsto Environment$ $r_1$))))))))

$\mathcal{E}[\![$(conceal ($I_1$ ... $I_n$) $E$)$]\!]$$e$ =
    ($with\text{-}record$ ($\mathcal{E}[\![E]\!]$ $e$)
      ($\lambda r$ . ($val\text{-}to\text{-}comp$
                  ($Record \mapsto Value$
                        ($\lambda I$ . **if** $I \in [I_1 \ldots I_n]$
                             **then** ($Unbound \mapsto Binding$ unbound)
                             **else** ($lookup$ ($Record \mapsto Environment$ $r$) $I$)
                             **fi**)))))
```

Figure 7.18: Valuation functions for the kernel record constructs in a CBV language.

```
(define ints (record (zero 0) (add +) (sub -) (mul *) (div quotient)
                     (neg (lambda (x) (- 0 x)))
                     (recip (lambda (x) (quotient 1 x)))
                     (eq =) (lt <) (gt >)))

(define gcd (lambda (a b) (if (= b 0)
                              a
                              (gcd b (rem a b)))))
(define rats
  (recordrec
    (rat (lambda (numer denom)
           (let ((common (gcd numer denom)))
             (pair (/ numer common) (/ denom common)))))
    (numer car)
    (denom cdr)
    (zero (rat 0 1))
    (add (lambda (r1 r2)
           (rat (+ (* (numer r1) (denom r2)) (* (denom r1) (numer r2)))
                (* (denom r1) (denom r2)))))
    (sub (lambda (r1 r2) (add r1 (neg r2))))
    (mul (lambda (r1 r2)
           (rat (* (numer r1) (numer r2))
                (* (denom r1) (denom r2)))))
    (div (lambda (r1 r2) (mul r1 (recip r2))))
    (neg (lambda (r) (rat (- 0 (numer r)) (denom r))))
    (recip (lambda (r) (rat (denom r) (numer r))))
    (eq (lambda (r1 r2)
          (and (= (numer r1) (numer r2))
               (= (denom r1) (denom r2)))))
    (lt (lambda (r1 r2) (< (* (numer r1) (denom r2))
                           (* (denom r1) (number r2)))))
    (gt (lambda (r1 r2) (lt r2 r1)))))
```

Figure 7.19: Two modules for arithmetic.

```
(define sum-of-squares
  (lambda (mod)
    (with-fields (add mul) mod
      (lambda (a b)
        (add (mul a a) (mul b b))))))
```

`((sum-of-squares ints) 3 4)` $\xrightarrow[FL]{}$ *25*

```
(with-fields (rat) rats
  ((sum-of-squares rats) (rat 1 3) (rat 1 4)))
```
$\xrightarrow[FL]{}$ $\langle 25, 144\rangle$

As a meatier example of using these arithmetic modules, consider the matrix module generator in Figure 7.20. In this example, matrices are represented as lists of rows.[5] `make-matrix-module` takes a number n and an arithmetic module a and constructs a new module that implements $n \times n$ matrices whose components are manipulated by a. For example, if n is 3 and a is `rats`, then `make-matrix-module` returns a module of 3×3 matrices over the rational numbers. The input module a must supply a `zero` constant and binary `add` and `mul` procedures. The resulting module is a matrix module that also exports these names as matrix operations. This means it is possible to use $n \times n$ matrices as elements of another matrix. Figures 7.21–7.23 show some matrix examples that run on a CBV FL interpreter:

7.3 Object-Oriented Programming

Object-oriented programming has emerged as an extremely popular programming paradigm. Definitions of what constitutes object-oriented programming vary, but they typically involve state-based entities called **objects** that communicate via messages. The behavior of an object is defined by its **class**, which specifies the object's state variables and its responses to messages. Classes and objects can be organized into **inheritance hierarchies** that describe how the behavior of one object can be inherited from other objects or classes. Although we will not discuss issues of state until the next chapter, it is worthwhile to introduce object-oriented programming here because most of the issues involved in this paradigm are issues of naming, not issues of state.

We introduce a purely functional object-oriented kernel called HOOK (Humble Object-Oriented Kernel) and its associated full language, HOOPLA (Humble Object-Oriented Programming Language). Figure 7.24 presents an s-expression grammar for HOOK. Figure 7.25 gives the syntax of the syntactic sugar; the

[5]In practice, we would import the list routines from their own list module.

```
(define make-matrix-module
  (lambda (n element-module)
    (with-fields (elt-add elt-mul elt-zero)
      (rename ((add elt-add) (mul elt-mul) (zero elt-zero))
        element-module)
      (conceal (map map2 reduce make-list)
        (recordrec
          (zero (make-list n (make-list n elt-zero)))
          (add (lambda (m1 m2)
                 (map2 (lambda (row1 row2) (map2 elt-add row1 row2))
                       m1
                       m2)))
          (mul (lambda (m1 m2)
                 (map (lambda (row1)
                        (map (lambda (row2)
                               (reduce elt-add
                                       elt-zero
                                       (map2 elt-mul row1 row2)))
                             (transpose m2)))
                      m1)))
          (transpose (lambda (m) (if (null? (car m))
                                     nil
                                     (cons (map car m)
                                           (transpose (map cdr m))))))
          (map (lambda (f lst)
                 (if (null? lst)
                     nil
                     (cons (f (car lst)) (map f (cdr lst))))))
          (map2 (lambda (f lst1 lst2)
                  (if (or (null? lst1) (null? lst2))
                      nil
                      (cons (f (car lst1) (car lst2))
                            (map2 f (cdr lst1) (cdr lst2))))))
          (reduce (lambda (binop identity lst)
                    (if (null? lst)
                        identity
                        (binop (car lst)
                               (reduce binop identity (cdr lst))))))
          (make-list (lambda (n elt)
                       (if (= n 0)
                           nil
                           (cons elt (make-list (- n 1) elt))))))))))
```

Figure 7.20: A generator for NxN matrix modules.

```
(define 2x2-int-matrices (make-matrix-module 2 ints))

(define im1 '((1 2) (3 4)))
(define im2 '((2 3) (4 5)))

fl-CBV> ((select add 2x2-int-matrices) im1 im2)
(list (list 3 5) (list 7 9))

fl-CBV> ((select mul 2x2-int-matrices) im1 im2)
(list (list 10 13) (list 22 29))
```

Figure 7.21: 2×2 matrices of integers.

```
(define 2x2-rat-matrices (make-matrix-module 2 rats))

(define rm1 (with-fields (rat) rats
              (list (list (rat 1 4) (rat 2 4))
                    (list (rat 3 4) (rat 4 4)))))

(define rm2 (with-fields (rat) rats
             (list (list (rat 1 5) (rat 2 5))
                   (list (rat 3 5) (rat 4 5)))))

fl-CBV> ((select add 2x2-rat-matrices) rm1 rm2)
(list (list (pair 9 20) (pair 9 10))
      (list (pair 27 20) (pair 9 5)))

fl-CBV> ((select mul 2x2-rat-matrices) rm1 rm2)
(list (list (pair 7 20) (pair 1 2))
      (list (pair 3 4) (pair 11 10)))
```

Figure 7.22: 2×2 matrices of rationals.

```
(define 2x2-matrices-of-2x2-int-matrices
  (make-matrix-module 2 2x2-int-matrices))

(define im3 '((3 4) (5 6)))
(define im4 '((5 6) (7 8)))

(define imm1 (list (list im1 im2) (list im3 im4)))
(define imm2 (list (list im2 im3) (list im4 im1)))

fl-CBV> ((select add 2x2-matrices-of-2x2-int-matrices) imm1 imm2)
(list (list (list (list 3 5) (list 7 9))
            (list (list 5 7) (list 9 11)))
      (list (list (list 8 10) (list 12 14))
            (list (list 6 8) (list 10 12))))

fl-CBV>  ((select mul 2x2-matrices-of-2x2-int-matrices) imm1 imm2)
(list (list (list (list 41 49) (list 77 93))
            (list (list 24 32) (list 48 64)))
      (list (list (list 89 107) (list 125 151))
            (list (list 52 70) (list 76 102))))
```

Figure 7.23: 2×2 Matrices of matrices of integers.

desugarings themselves are defined in Figure 7.26.

Figure 7.27 contains some sample HOOPLA classes. For example, a point is created by sending `make` to the `point` class. The resulting point responds to the `x`, `y`, and `move` messages. The language does not support side-effects; `move` does not change the existing point but creates a new one. Every method has as its first formal parameter a **self variable** that names the object that originally received the message. This self variable is crucial for getting inheritance to work. The `turtle` class inherits behavior from both `points` and `directions`. Figure 7.28 shows some examples of interacting with a HOOPLA interpreter.

7.3.1 Semantics of HOOK

The inheritance structure in HOOK programs is reminiscent of records in FL. In fact, the similarity is so great that we will define the semantics of HOOK programs by compiling them into a version of the FL language extended with records. The key to this transformation is that objects are represented as records that bind message names to procedures that represent methods. A message send is then handled by simply looking up the method/procedure in the receiver record and applying it to the actual arguments.

$$\begin{array}{lcll}
E & \in & \text{Exp} & \\
M & \in & \text{Message} & = \text{Identifier} \\
I & \in & \text{Identifier} & \\
L & \in & \text{Lit} & = \text{Intlit} + \text{Boollit} + \ldots
\end{array}$$

$$\begin{array}{lll}
E ::= & L & \text{[Literal]} \\
 & \mid I & \text{[Identifier]} \\
 & \mid \texttt{(method } M_{message} \texttt{ (}I_{self}\ I_{formal}\texttt{*) } E_{body}\texttt{)} & \text{[Simple Object]} \\
 & \mid \texttt{(object-compose } E_{obj1}\ E_{obj2}\texttt{)} & \text{[Object Composition]} \\
 & \mid \texttt{(null-object)} & \text{[Null Object]} \\
 & \mid \texttt{(send } M_{message}\ E_{receiver}\ E_{arg}\texttt{*)} & \text{[Message Send]}
\end{array}$$

Figure 7.24: The abstract syntax for HOOK.

$$\begin{array}{lcl}
P & \in & \text{Program} \\
D & \in & \text{Def}
\end{array}$$

$$\begin{array}{lll}
P ::= & \texttt{(program } E_{body}\ D_{def}\texttt{*)} & \text{[Program]} \\
 & & \\
D ::= & \texttt{(define } I_{name}\ E_{value}\texttt{)} & \text{[Definition]} \\
 & & \\
E ::= & \ldots & \\
 & \mid \texttt{(object } E_{object}\texttt{*)} & \text{[Object]} \\
 & \mid \texttt{(class (}I_{init}\texttt{*) } E_{instance}\texttt{*)} & \text{[Class]} \\
 & \mid \texttt{(lambda (}I_{formal}\texttt{*) } E_{body}\texttt{)} & \text{[Abstraction]} \\
 & \mid \texttt{(}E_{rator}\ E_{rand}\texttt{*)} & \text{[Application]} \\
 & \mid \texttt{(let ((}I_{name}\ E_{value}\texttt{)*) } E_{body}\texttt{)} & \text{[Local-Binding]} \\
 & \mid \texttt{(if } E_{test}\ E_{consequent}\ E_{alternative}\texttt{)} & \text{[Branch]}
\end{array}$$

Figure 7.25: The syntax of the HOOPLA language.

$$\mathcal{D}[\![\texttt{(object)}]\!] = \texttt{(null-object)}$$
$$\mathcal{D}[\![\texttt{(object } E_{fst}\ E_{rest}\texttt{*)}]\!] = \texttt{(object-compose } \mathcal{D}[\![E_{fst}]\!]\ \mathcal{D}[\![\texttt{(object } E_{rest}\texttt{*)}]\!]\texttt{)}$$

$$\begin{aligned}&\mathcal{D}[\![\texttt{(class (}I_{init}\texttt{*) } E_{inst_1}\ \ldots\ E_{inst_n}\texttt{)}]\!] = \\ &\quad\texttt{(method make (}I_{ignore}\ I_{init}\texttt{*) } \mathcal{D}[\![\texttt{(object } E_{inst_1}\ \ldots\ E_{inst_n}\texttt{)}]\!]\texttt{)} \\ &\qquad\text{where } I_{ignore} \notin \textstyle\bigcup_{i=1}^{n} \mathit{FreeIds}[\![E_{inst_i}]\!]\end{aligned}$$

$$\begin{aligned}&\mathcal{D}[\![\texttt{(lambda (}I_{formal}\texttt{*) } E_{body}\texttt{)}]\!] = \texttt{(method call (}I_{ignore}\ I_{formal}\texttt{*) } \mathcal{D}[\![E_{body}]\!]\texttt{)} \\ &\quad\text{where } I_{ignore} \notin \mathit{FreeIds}[\![E_{body}]\!]\end{aligned}$$

$$\mathcal{D}[\![\texttt{(}E_{rator}\ E_{rand}\texttt{*)}]\!] = \texttt{(send call } \mathcal{D}[\![E_{rator}]\!]\ \mathcal{D}[\![E_{rand}\texttt{*}]\!]\texttt{)}$$

$$\mathcal{D}[\![\texttt{(let ((}I_{var}\ E_{val}\texttt{)*) } E_{body}\texttt{)}]\!] = \mathcal{D}[\![\texttt{((lambda (}I_{var}\texttt{*) } E_{body}\texttt{) } E_{val}\texttt{*)}]\!]$$

$$\begin{aligned}\mathcal{D}[\![\texttt{(if } E_{test}\ E_{con}\ E_{alt}\texttt{)}]\!] = \texttt{(send if-true } &\mathcal{D}[\![E_{test}]\!] \\ &\mathcal{D}[\![\texttt{(lambda () } E_{con}\texttt{)}]\!] \\ &\mathcal{D}[\![\texttt{(lambda () } E_{alt}\texttt{)}]\!]\texttt{)}\end{aligned}$$

Figure 7.26: Rules for desugaring HOOPLA into HOOK.

We formally define the transformation from HOOK code to CBV FL code in terms of the compilation function $\mathcal{T} : \mathrm{Exp}_{HOOK} \to \mathrm{Exp}_{FL}$. This function is defined in Figure 7.29. To be complete, we also would need functions that map HOOK programs to FL programs and HOOK definitions to FL definitions, but since these are straightforward, we will leave them out.

The core of the compilation is the handling of methods, objects, and message sends. A HOOK `method` construct is transformed into an FL `record` construct with a single binding of the message name to a procedure that does the work of the method. A HOOK `object-compose` construct compiles into an FL `override` construct; the semantics of `override` are such that methods from E_{obj1} will take precedence over methods from E_{obj2}. A HOOK message send compiles to a procedure application in FL; the procedure is found by looking up the message name in the record that represents the receiver.

The handling of literals (via $\mathcal{T}_{lit}$) is perhaps the trickiest part of the compilation. HOOK literals stand not for simple values but for full-fledged message-passing objects. A HOOK number object, for instance, must compile into an FL record that has methods for all the numeric operations. In addition, such a record must also be able to supply the unadorned version of the value it is holding onto; this is the purpose of the binding involving I_{int}. The identifier I_{int} must be the same for all literal objects. Note that operations returning the

```
(define point
  (class (init-x init-y)
    (method x (self) init-x)
    (method y (self) init-y)
    (method move (self dx dy)
      (object (send make point
                    (send + (send x self) dx)
                    (send + (send y self) dy))
              self)                        ; Allows mixins
      )))

(define direction
  (class (init-angle)
    (method angle (self) init-angle)
    (method turn (self delta)
      (object (send make direction
                    (send + (send angle self) delta))
              self))                       ; Allows mixins
    ))

(define turtle
  (class (x y angle)
    (method home (self)
      (object (send make turtle x y angle)
              self))                       ; Allows mixins
    (send make point x y)
    (send make direction angle)))

(define color
  (class (clr)
    (method color (self) clr)
    (method new-color (self new)
      (object (send make color new)
              self))))

(define colored-point
  (class (x y col)
    (send make point x y)
    (send make color col)))
```

Figure 7.27: Sample HOOPLA classes.

```
;;; Define a turtle T1
(define t1 (send make turtle 0 0 0))
(send x t1) ─HOOPLA→ 0    {This is the object 0}
(send y t1) ─HOOPLA→ 0
(send angle t1) ─HOOPLA→ 0

;;; Define T2 as a rotated and translated version of T1.
(define t2 (send move (send turn t1 45) 17 23))
(send x t2) ─HOOPLA→ 17
(send y t2) ─HOOPLA→ 23
(send angle t2) ─HOOPLA→ 45)

;;; Note that T1 is unchanged. E.g.:
(send x t1) ─HOOPLA→ 0

;;; Now define T3 as a version of T2 sent home.
(define t3 (send home t2))
(send x t3) ─HOOPLA→ 0
(send y t3) ─HOOPLA→ 0
(send angle t3) ─HOOPLA→ 0
```

Figure 7.28: Example interactions with a HOOPLA interpreter.

same kind of object being defined (e.g., `+`, `*`) return an extended version of `self` rather than just a fresh instance of the object. This means that the returned object retains all the behavior of the receiver that is not explicitly specified by the definition.

Booleans, symbols, and whatever other literals or standard identifiers we might support are handled like integers.

▷ **Exercise 7.21** What is the value of the following HOOPLA expression?

```
(let ((ob1 (object (method value (self) 1)))
      (ob2 (object (method value (self) 2)))
      (ob3 (object (method value (self) 3)
                   (method evaluate (self) (send value self)))))
  (send evaluate (object ob1 ob2 ob3)))
```
◁

▷ **Exercise 7.22** Following the example for integer literals, show how boolean literals in HOOK compile into FL. HOOK boolean objects handle the messages `not?`, `and?`, `or?`, `if-true`, and `if-false`. ◁

$\mathcal{T} : \text{Exp}_{HOOK} \rightarrow \text{Exp}_{FL}$
$\mathcal{T}_{lit} : \text{Lit}_{HOOK} \rightarrow \text{Exp}_{FL}$

$\mathcal{T}[\![I]\!] = I$

$\mathcal{T}[\![L]\!] = \mathcal{T}_{lit}[\![L]\!]$, where $\mathcal{T}_{lit}$ is described below.

$\mathcal{T}[\![$`(method` $M_{message}$ `(`I_{self} I_{formal}`*)` E_{body}`)`$]\!] =$
  `(record (`$M_{message}$ `(lambda (`I_{self} I_{formal}`*)` $\mathcal{T}[\![E_{body}]\!]$`)))`

$\mathcal{T}[\![$`(object-compose` E_{obj_1} E_{obj_2}`)`$]\!] =$`(override` $\mathcal{T}[\![E_{obj_2}]\!]$ $\mathcal{T}[\![E_{obj_1}]\!]$`)`

$\mathcal{T}[\![$`(null-object)`$]\!] =$`(record)`

$\mathcal{T}[\![$`(send` $M_{message}$ $E_{receiver}$ E_{arg_1} $\ldots E_{arg_n}$`)`$]\!] =$
  `(let ((`$I_{receiver}$ $\mathcal{T}[\![E_{receiver}]\!]$`))`
   `((select` $M_{message}$ $I_{receiver}$`)` $I_{receiver}$ $\mathcal{T}[\![E_{arg_1}]\!]$ $\ldots \mathcal{T}[\![E_{arg_n}]\!]$`))`
where $I_{receiver} \notin \bigcup_{i=1}^{n} \mathit{FreeIds}[\![E_{arg_i}]\!]$

$\mathcal{T}_{lit}[\![N]\!] =$
```
(letrec ((make-integer
           (lambda (n)
             (record
               (I_int n)
               (+ (lambda (self arg)
                    (override self
                      (make-integer
                        (primop + n (select I_int arg))))))
               (* (lambda (self arg)
                    (override self
                      (make-integer
                        (primop * n (select I_int arg))))))
                    ⋮
               ))))
   (make-integer N))
```
where I_{int} is the same for all integers but distinct from all other message names.

Similarly for other literals.

Figure 7.29: The rules for translating HOOK to FL.

▷ **Exercise 7.23** Anoop Hacker is confused about namespace issues in HOOPLA. In the syntax of the full language, there are several binding constructs: `class`, `lambda`, `let`, and `method`. The first three constructs all bind formal parameters; the last one binds a message name and a name for *self* in addition to the formal parameters of the method. You have volunteered to help Anoop answer the following questions. Carefully study the definitions of HOOPLA to HOOK desugaring and HOOK to FL translation to justify your answers. Give examples where appropriate.

a. How many distinct namespaces are there in HOOPLA?

b. Is it possible for a method formal parameter named `x` to be shadowed by a message named `x`?

c. Is it possible for a message named `x` to be shadowed by a method formal parameter named `x`?

d. Do the answers to parts b and c change if the `record` in the translation for `object-compose` becomes a `recordrec` instead? If so, how? ◁

▷ **Exercise 7.24** Does HOOPLA's `lambda` construct support currying? Explain. ◁

▷ **Exercise 7.25** Paula Morwicz doesn't like the fact that it's always necessary to explicitly name *self* within a HOOPLA method. She decides to implement a version of HOOPLA called SELFISH in which a reserved word `self` is implicitly bound within every method body. For example, in SELFISH the point class would be written as follows:

```
(define point
  (class (init-x init-y)
    (method x () init-x)
    (method y () init-y)
    (method move (dx dy)
      (object (send make point
                    (send + (send x self) dx)
                    (send + (send y self) dy))
              self)                        ; Allows mixins
      )))
```

In this example, the instances of `self` within the `move` method evaluate to the receiver of the `move` message. Because `self` is a reserved word in SELFISH, it is illegal to use it as a formal parameter to a method.

a. Describe what modifications would have to be made to the following in order to specify the semantics of SELFISH:

   i. The HOOK grammar.

   ii. The HOOPLA to HOOK desugarer.

   iii. The HOOK to FL translator.

b. Unfortunately, SELFISH doesn't always give the behavior Paula expects. For example, she makes a simple modification to the definition of the point class:

```
(define point
  (class (init-x init-y)
    (method x () init-x)
    (method y () init-y)
    (method move (dx dy)
      (let ((new-x (send + (send x self) dx))
            (new-y (send + (send y self) dy)))
        (object (method x () new-x)
                (method y () new-y)
                self)
        ))))
```

After this change, turtle objects (which are implemented in terms of points) no longer work as expected. Explain what has gone wrong.

c. Show how to get Paula's new point definition to work as expected. You can add new code, but not remove any. You may perform alpha-renaming where necessary.

d. Which do you think is better: the explicit `self` approach of HOOPLA or the implicit `self` approach of SELFISH? Explain your answer. ◁