Chapter 6

FL: A Functional Language

Things used as language are inexhaustible attractive.

— *The Philosopher, Ralph Waldo Emerson*

6.1 Decomposing Language Descriptions

The study of a programming language can often be simplified if it is decomposed into three parts:

1. A **kernel** language that forms the essential core of the language.
2. **Syntactic sugar** that extends the kernel with convenient constructs. Such constructs can be automatically translated into kernel constructs via a process known as **desugaring**.
3. A **standard library** of primitive constants and operators supplied with the language.

We shall refer to the combination of a kernel, syntactic sugar, and a standard library as a **full language** to distinguish it from its components.

Decomposing a programming language definition into parts relieves a common tension in the design and analysis of programming languages. From the standpoint of reasoning about a language, it is desirable for a language to have only a few simple parts. However, from the perspective of programming in a language, it is desirable to concisely and conveniently express common programming idioms. A language that is too pared down may be easy to reason about but horrendous to program in — try writing factorial in PostFix+{`dup`}. On the other hand, a language with many features may be convenient to program in but difficult to reason about — try proving some non-trivial properties about your next JAVA, C, ADA, or COMMON LISP program.

The technique of viewing a full language as mostly sugar coating around a kernel lets us have our cake and eat it too. When we want to reason about the language, we consider only the small kernel upon which everything else is built. But when we want to program in the language, we make heavy use of the syntactic sugar and standard library to express what we want in a readable fashion. Indeed, we can even add new syntactic sugar and new primitives without changing the properties of the kernel.

There are limitations to this approach. We'd like the kernel and full language to be close enough so that the desugaring is easy to understand. Otherwise we might have the situation where the kernel is a machine instruction set and the desugaring is a full-fledged compilation from high-level programs into object code. For this reason, we require that syntactic sugar be expressed via simple local transformations; no global program analysis is allowed.

We shall study this language decomposition technique in the context of a mini-language we call FL (for *F*unctional *L*anguage).[1] FL provides us with the opportunity to use the semantic tools developed in the previous chapters to analyze a programming language that is much closer to a "real" programming language than POSTFIX or EL. Along the way, we will introduce two approaches for modeling names in a programming language: substitution and environments.

FL is a language that examplifies what is traditionally known as the **functional programming paradigm**. As we shall see, functional programming languages are characterized by an emphasis on the manipulation of values that model mathematical functions. The name "functional language" is a little bit odd, since it suggest that languages not in this paradigm are somehow *dys*functional — a perception that many functional language aficionados actively promote! Perhaps **function-oriented languages** would be a more accurate term for this class of languages.

6.2 The Structure of FL

FL is a typical functional programming language for computing with numeric, boolean, symbolic, procedural, and compound data values. The computational model of FL is based on the functional programming paradigm exemplified by

[1] Our FL language is not to be confused with any other similarly-named language. In particular, our FL is *not* related to the FL functional programming language [BWW90, BWW$^+$89] based on Backus's FP [Bac78].

such languages as Erlang, FX, Haskell, ML, and Scheme. FL programs are free of side effects and make heavy use of first-class functional values (here called **procedures**[2]. Syntactically, FL bears a strong resemblance to Scheme, but semantically we shall see that it is closer to so-called **purely functional lazy languages** like Haskell and Miranda.

6.2.1 FLK: The Kernel of the FL Language

We begin by presenting the syntax and informal semantics of FLK, the FL kernel.

6.2.1.1 The Syntax of FLK

A well-formed FLK program is a member of the syntactic domain Program defined by the s-expression grammar in Figure 6.1. FLK programs have the form `(flk (`I_{formal}`*)` E_{body}`)`, where I_{formal}* are the **formal parameters** of the program and E_{body} is the **body expression** of the program. Intuitively, the formal parameters name program inputs and the body expression specifies the result value computed by the program for its inputs.

FLK expressions are s-expression syntax trees whose leaves are either literals or variable references. FLK literals include the unit literal, booleans, integers, and symbols. We adopt the Scheme convention of writing the boolean literals as `#t` (true) and `#f` (false). The unit literal (`#u`) is used in situations where the value of an expression is irrelevant, such as contexts in C and Java modeled by the `void` type. For symbolic (i.e., non-numeric) processing, FLK supports the Lisp-like notion of a **symbol**. Symbols are similar to strings in traditional languages, except that they a written with a different syntax (using the keyword `symbol` rather than double quotes) and they are atomic entities that cannot be decomposed into their component characters. For simplicity, FLK assumes a Lisp-like convention in which symbols are sequences of characters (1) that do not include whitespace, bracket characters ({, }, `(`, `)`, `[`, `]`), or quote characters (`"`, `‘`, `’`); (2) that do not begin with `#`; and (3) in which case is ignored. So the symbols `xcoord`, `xCoord`, and `XCOORD` are considered equivalent.

[2]We shall consistently use the term **procedure** to refer to entities in programming languages that denote mathematical functions, and **function** to refer to the mathematical notion of function. In some languages, these two terms are used to distinguish different kinds of programming language entities. For example, in Pascal, “function” refers to a subroutine that returns a result whereas “procedure” refers to a subroutine performs its work via side effect and returns no result. Much of the functional programming literature uses the term “function” to refer both to the programming language entity and the mathematical entity it denotes.

$P \in$ Program
$E \in$ Exp
$L \in$ Lit
K $\in$ Keyword $= \{$`call`, `if`, `pair`, `primop`, `proc`, `rec`, `symbol`, `error`$\}$
$Y \in$ Symlit $= \{$`x`, `lst`, `make-point`, `map_tree`, `4/3*pi*r^21`, $\ldots\}$
$I \in$ Identifier $=$ Symlit $-$ Keyword
$B \in$ Boollit $= \{$`#t`, `#f`$\}$
$N \in$ Intlit $= \{\ldots,$ `-2`, `-1`, `0`, `1`, `2`, $\ldots\}$
$O \in$ Primop $=$ *Defined by standard library (Section 6.2.3).*

P ::=	(`flk` (I_{formal}*) E_{body})	[Program]
E ::=	L	[Literal]
|	I	[Variable Reference]
|	(`primop` O_{name} E_{arg}*)	[Primitive Application]
|	(`proc` I_{formal} E_{body})	[Abstraction]
|	(`call` E_{rator} E_{rand})	[Application]
|	(`if` E_{test} E_{then} E_{else})	[Branch]
|	(`pair` E_{fst} E_{snd})	[Pairing]
|	(`rec` I_{var} E_{body})	[Recursion]
|	(`error` I_{msg})	[Errors]
L ::=	`#u`	[Unit Literal]
|	B	[Boolean Literal]
|	N	[Integer Literal]
|	(`symbol` I)	[Symbolic Literal]

Figure 6.1: An s-expression grammar for FLK

A key difference between FLK and PostFix/EL is that that FLK provides constructs (`flk`, `proc`, and `rec`) that introduce names for values. Syntactically, names are expressed via **identifiers**. The rules for what constitutes a well-formed identifier differs from language to language. In FLK we shall assume that any symbol that is not one of the **reserved keywords** of the language (`call`, `if`, `pair`, `primop`, `proc`, `rec`, `symbol`, and `error`) can be used as an identifier. This means that expressions like `x-y` and `4/3*pi*r^2` are treated as atomic identifiers in FLK. In many other languages, these would be infix specifications of trees of binary operator applications.

For compound expressions, FLK supports procedural abstractions (`proc`) and applications (`call`), primitive applications (`primop`), conditional branches (`if`), pair creation (`pair`), simple recursion (`rec`), and error signaling (`error`).

Although many of the syntactic conventions of FLK are borrowed from Lisp-like languages, especially Scheme, it's worth emphasizing that FLK differs from these languages in some fundamental ways. For example, in Scheme, abstractions may take any number of formal parameters, are introduced via the keyword `lambda`, and are invoked via an application syntax with no keyword. In contrast, FLK abstractions have exactly one formal parameter, are introduced via the keyword `proc`, and are applied via the keyword `call`.

6.2.1.2 An Informal Semantics for FLK

Intuitively, every FLK expression denotes a value that is tagged with its type in addition to whatever information distinguishes it from other values of the same type. The primitive values supported by FLK include the unit value, boolean truth values, integers, and textual symbols. The unit value is the unique value of a distinguished type that has a single element. In addition, FLK supports pairs and procedures. A pair is a compound value that allows any two values (which may themselves be pairs) to be glued together to form a single value. A procedure is a value that represents a mathematical function by specifying how to map an input value to an output value.

We will informally describe the semantics of FLK by considering some sample evaluations of FLK expressions. We use the notation $E \xrightarrow[FLK]{} V$ to indicate that the expression E evaluates to the value V. Here are some example values that indicate our conventions for writing FLK values:

unit	The unit value
false, *true*	The boolean values
17, *−3*	Integer values
'abstraction, *'captain*	Symbolic values
procedure	Procedural values
$\langle 17, true\rangle$, $\langle procedure, \langle 'abstraction, unit\rangle\rangle$	Pair values
error:not-an-integer	Errors
∞−*loop*	Non-termination (represents an infinite loop)

Additionally, the following abbreviation will be handy for representing lists of values that are encoded as a unit-terminated chain of pairs:

$$[V_1, V_2, \ldots, V_n] = \langle V_1, \langle V_2, \ldots\ \langle V_n, unit\rangle\ \ldots\rangle\rangle$$

For example, the notation $[17, true, \langle 'foo, procedure\rangle]$ is an abbreviation for a three-element list values $\langle 17, \langle true, \langle\langle 'foo, procedure\rangle, unit\rangle\rangle\rangle$.

Our value notation does not distinguish procedural values that denote different mathematical functions. For instance, a squaring procedure and a doubling procedure are both written *procedure*. This is because our operational semantics will not allow us to directly observe the function designated by a procedural value that is the outcome of a program. As explained in Section 3.4.4, intentionally blurring distinctions between certain values is sometimes necessary to enable program transformations. However, our notation for errors *does* distinguish errors with different messages.

The literal expressions designate constants in the language:

`#u` $\xrightarrow[FLK]{}$ *unit*
`#t` $\xrightarrow[FLK]{}$ *true*
`23` $\xrightarrow[FLK]{}$ *23*
`(symbol captain)` $\xrightarrow[FLK]{}$ *'captain*

The primitive application `(primop` O E_1 … E_n`)` denotes the result of applying the primitive operator named by O to the n values of the argument expressions E_i. The behavior of most of the primitive operators should be apparent from their names. E.g.,

`(primop not? #t)` $\xrightarrow[FLK]{}$ *false*
`(primop integer? 1)` $\xrightarrow[FLK]{}$ *true*
`(primop integer? #t)` $\xrightarrow[FLK]{}$ *false*
`(primop + 1 2)` $\xrightarrow[FLK]{}$ *3*
`(primop / 17 5)` $\xrightarrow[FLK]{}$ *3* {*integer division*}
`(primop rem 17 5)` $\xrightarrow[FLK]{}$ *2*
`(primop sym=? (symbol captain) (symbol abstraction))` $\xrightarrow[FLK]{}$ *false*
`(primop sym=? (symbol captain) (symbol Captain))` $\xrightarrow[FLK]{}$ *true*

The last example illustrates that FLK symbols are case-insensitive. The full list of primitive operations is specified by the FL standard library in Section 6.2.3.

The value of a primitive application is not defined when primitive functions are given the wrong number of arguments, when an argument has an unexpected type, or when integer division or remainder by 0 is performed. These situations are considered program errors:

(primop + 1) $\xrightarrow[FLK]{}$ *error:too-few-args*
(primop not? 1) $\xrightarrow[FLK]{}$ *error:not-a-bool*
(primop + #t 1) $\xrightarrow[FLK]{}$ *error:not-an-integer*
(primop / 1 0) $\xrightarrow[FLK]{}$ *error:divide-by-zero*

The abstraction (proc I E) specifies a procedural value that represents a mathematical function. The application (call E_1 E_2) stands for the result of applying the procedure denoted by E_1 to the operand value denoted by E_2. It is an error to use any value other than a procedure as an operator. Multiple-argument procedures can be simulated by currying (see Section A.2.5.1).

(proc x (primop * x x)) $\xrightarrow[FLK]{}$ *procedure*
(call (proc x (primop * x x)) 5) $\xrightarrow[FLK]{}$ *25*
(call (call (proc a (proc b (primop - b a))) 2) 3) $\xrightarrow[FLK]{}$ *1*
(call 3 5) $\xrightarrow[FLK]{}$ *error:non-procedural-rator*
(call not? #t) $\xrightarrow[FLK]{}$ *error:unbound-variable*
{not? *is a primop, not a variable name*}
(call (proc x (call x x)) (proc x (call x x))) $\xrightarrow[FLK]{}$ ∞*-loop*

As in Haskell, FLK's procedures are **non-strict**. This means that a call to a procedure may return a value even if one of its arguments denotes an error or a non-terminating computation. Intuitively, non-strictness indicates that an expression will never be evaluated if the rest of the computation does not require its value. For example:

```
(call (proc x 3) (primop / 1 0)) -FLK-> 3
(call (proc x (primop + x 3))
      (primop / 1 0)) -FLK-> error:divide-by-zero
(call (proc x 3)
      (call (proc x (call x x))
            (proc x (call x x)))) -FLK-> 3
(call (proc x (primop + x 3) )
      (call (proc x (call x x))
            (proc x (call x x)))) -FLK-> ∞-loop
```

Unlike FLK, most real-world languages (including C, Java, Pascal, Scheme, and ML) have **strict** procedures. In these langauges, operands of procedure applications are always evaluated, even if they are never referenced by the procedure body.

The branch expression (`if` E_{test} E_{then} E_{else}) requires the value of E_{test} to be a boolean, and evaluates one of E_{then} or E_{else} depending on whether the test is true or false:

(if (primop > 8 7) (primop + 2 3) (primop * 2 3)) $\xrightarrow[FLK]{}$ *5*
(if (primop < 8 7) (primop + 2 3) (primop * 2 3)) $\xrightarrow[FLK]{}$ *6*
(if (primop - 8 7) (primop + 2 3) (primop * 2 3))
$\xrightarrow[FLK]{}$ *error:non-bool-in-if-test*

The pairing expression (`pair` E_{fst} E_{snd}) is the means of gluing two values together into a single value of the pair type. The values of the two components can be extracted via the primitive operators `fst` and `snd`. A chain pairs linked by their second components and terminated by the unit value is a standard way of encoding a list:

(pair 1 (pair 2 (pair 3 #u))) $\xrightarrow[FLK]{}$ [*1*, *2*, *3*]

Like procedure calls, pairing in FLK is non-strict. The result of `pair` is always a well-defined pair even if one (or both) of its argument expressions is not an FLK value. The unspecified nature of a contained value can only be detected when it is extracted from the pair.

(pair (primop not? #f) (primop / 1 0))
$\xrightarrow[FLK]{}$ ⟨*true*, *error:divide-by-zero*⟩
(primop fst (pair (primop not? #f) (primop / 1 0))) $\xrightarrow[FLK]{}$ *true*
(primop snd (pair (primop not? #f) (primop / 1 0)))
$\xrightarrow[FLK]{}$ *error:divide-by-zero*

As we shall see in Section 10.1.3, non-strict data structures are an important mechanism for supporting modularity in programs.

We choose to make `pair` a special form rather than a primitive like `not?` or `+` to emphasize the fact that pairing is non-strict. If we made `pair` a primitive operator, we would still have to treat it specially when we describe the semantics of the `primop` form because all the other primitives are strict. Treating `pair` as a special form provides a cleaner description of the semantics. This is a purely stylistic decision; it is also possible to treat `pair` as a binary primitive operator (see Exercise 6.20).

The recursion expression (`rec` I E) allows the expression of recursion equations over one variable. The value of the `rec` expression is the value of its body, where the value of I within E is the value of the entire `rec` expression. That is, the value returned by a recursion is the solution to the equation $I = E$. `rec` is used to specify recursive procedures and data structures. For example:

```
(rec fact (proc n
            (if (primop = n 0)
                1
                (primop * n (call fact (primop - n 1))))))
```
$\xrightarrow[FLK]{}$ *procedure* {*A factorial procedure.*}

```
(rec ones (pair 1 ones))
```
$\xrightarrow[FLK]{}$ $[1,\ 1,\ 1,\ \ldots]$ {*An infinite sequence of 1s.*}

FLK programs are parameterized expressions. We use the notation $P \xrightarrow[FLK]{[V_1,\ldots,V_n]} V_{result}$ to indicate that running the FLK program P on argument values V_1, …, V_n yields the result value V_{result}. For example:

```
(flk (x) (* x x))
```
$\xrightarrow[FLK]{[5]}$ *25*

```
(flk (a b) (/ (+ a b) 2))
```
$\xrightarrow[FLK]{[2,8]}$ *5*

```
(flk (a b) (/ (+ a b) 2))
```
$\xrightarrow[FLK]{[2,8,11]}$ *error:wrong-number-of-args*

```
(flk (x nums)
  (call (rec scale
          (proc ys
            (if (primop unit? ys)                    {Is ys the empty list?}
                ys                                   {If so, return it;}
                (pair                                {otherwise, prepend the}
                  (primop * x (primop fst ys))       {scaled first number}
                  (call scale                        {to the result of scaling}
                        (primop snd xs))))))         {the rest of the numbers.}
        nums))
```
$\xrightarrow[FLK]{[4,[1,2,3]]}$ $[4, 8, 12]$

The penultimate example illustrates that it is an error if the number of arguments supplied to the program differs from the number of formal parameters declared. The final example illustrates that FLK program arguments may include values other than integers, such as lists of integers in this case.

In general, the values considered to be valid program arguments will be a proper subset of the values manipulated by a language. In languages such as C and JAVA, program arguments must be strings, and these can be parsed into other kinds of values (such as integers, floating-point numbers, arrays of numbers, etc.) where necessary. Program arguments are typically limited to literal data with simple textual representations, which excludes procedural values as program arguments. In the case of FLK, we shall assume that program arguments may be any of the literal values (unit, booleans, integers, symbols) and binary trees (i.e., trees with `pair` nodes) whose leaves are such literals. Since s-expressions can be represented as such trees, this will allow us to write FLK programs that manipulate representations of programming language ASTs.

This concludes our informal description of the semantics of FLK. While FLK has considerably more expressive punch than PostFix or FL, expressing even simple programs within FLK is rather cumbersome. In the next section, we will see how to extend FLK to another language, FL, that maintains simplicity in the semantics but yields a language in which it is practical to write (and read!) non-trivial functional programs.

6.2.2 FL Syntactic Sugar

6.2.2.1 Syntactic Sugar Forms

P	$\in$	Program		
D	$\in$	Def		
SX	$\in$	SExp		
E	$\in$	Exp		
I	$\in$	Identifier		
B	$\in$	Boollit	=	{`#t`, `#f`}
N	$\in$	Intlit	=	{. . . , `-2`, `-1`, `0`, `1`, `2`, . . .}

P ::=	`(fl (`I_{formal}`*) `E_{body}` `$D_{definitions}$`*)`	[Program]
D ::=	`(define `I_{name}` `E_{value}`)`	[Definition]
E ::=	. . .	[FLK constructs]
	| `(lambda (`I_{formal}`*) `E_{body}`)`	[Multi-Abstraction]
	| `(`E_{rator}` `E_{rand}`*)`	[Multi-Application]
	| `(list `$E_{element}$`*)`	[List]
	| `(quote `SX`)`	[S-Expression]
	| `'`SX	[S-Expression Shorthand]
	| `(cond (`E_{test}` `E_{action}`)* (else `$E_{default}$`))`	[N-Way Branch]
	| `(scand `$E_{conjunct}$`*)`	[Short-Circuit And]
	| `(scor `$E_{disjunct}$`*)`	[Short-Circuit Or]
	| `(let ((`I_{var}` `E_{defn}`)*) `E_{body}`)`	[Local Binding]
	| `(letrec ((`I_{var}` `E_{defn}`)*) `E_{body}`)`	[Recursive Binding]
SX ::=	I	[Symbol]
	| `#u`	[Unit]
	| B	[Boolean]
	| N	[Integer]
	| `(`SX_{elt}`*)`	[List]

Figure 6.2: Grammar for FL syntactic sugar.

The syntax of FL's syntactic abbreviations are specified in the grammar presented in Figure 6.2. In the definition of E, the ellipses ... stand for all the expression productions in the FLK grammar. The new expression forms in Figure 6.2 can be used anywhere the nonterminal E appears in the kernel FLK constructs as well as in the new syntactic abstractions. We first explain informally the meaning of each abbreviation before showing how to desugar them into FLK. Many of these syntactic abbreviations are inspired by constructs in LISP dialects, but we shall see that some of them have somewhat different meanings in FL than in LISP.

FL's `lambda` construct can bind any number (possibly zero) of identifiers within a procedure body. In the tagless multi-application form, a procedure can be applied to any number (possibly zero) of arguments. Because multi-applications are the only tagless form, the lack of an explicit tag is not ambiguous. Because applications tend to be the most common kind of compound expression, eliminating the explicit tag for this case makes expressions more concise. The multi-abstraction and multi-application forms are inspired by SCHEME syntax. Unlike SCHEME, FL supports implicit currying with these constructs. For example, suppose that E_{abs3} is the three-parameter multi-abstraction

```
(lambda (a b c) (primop * a (primop + b c))).
```

Then (E_{abs3} `2 3 4`) denotes *14*, (E_{abs3} `2 3`) denotes the same procedure as `(lambda (c) (primop * 2 (primop + 3 c)))`, and (E_{abs3} `2`) denotes the same procedure as `(lambda (b c) (primop * 2 (primop + b c)))`.

The `list` construct is a shorthand for creating lists by a sequence of nested pairings. (`list` E_1 ... E_n) constructs a unit-terminated, chain of n pairs linked by their second components where the value of E_i is the value of the first element of the ith pair in the chain. For example,

```
(list (primop + 1 2) (primop = 3 4) (pair 4 5))
```

is equivalent to

```
(pair (primop + 1 2)
      (pair (primop = 3 4)
            (pair (pair 4 5)
                  #u))).
```

The `quote` construct facilitates the construction of **s-expressions** , which are recursively defined to be literals (unit, numeric, boolean, and symbolic) and lists of s-expressions. Quoted s-expressions are a very concise way to specify tree-structured data. The `quote` form can be viewed as a means of a constructing a tree from a printed representation of the tree. For example, the s-expression

```
(quote (1 (#t three) (four 5 six)))
```

is a shorthand for

```
(list 1
      (list #t (symbol three))
      (list (symbol four) 5 (symbol six))).
```

To make the abbreviation even more concise, we adopt the LISP convention that `'`*SX* is a shorthand for `(quote` *SX*`)`, so the above example can also be written `'(1 (#t three) (four 5 six))`. The ability to express s-expressions so concisely with the quotation forms makes them very handy for specifying programs that manipulate program phrases from languages with s-expression syntax. For example, the POSTFIX program `(postfix 1 (2 mul) exec)` can be represented as the FL s-expression form `'(postfix 1 (2 mul) exec)`.

The `cond` construct is an n-way conditional branch that stands for a nested sequence of `if` expressions. For example,

```
(cond ((primop > temp 80) (symbol hot))
      ((primop < temp 50) (symbol cold))
      (else (symbol mild)))
```

is equivalent to

```
(if (primop > temp 80)
    (symbol hot)
    (if (primop < temp 50)
        (symbol cold)
        (symbol mild))).
```

The `scand` and `scor` expressions provide for so-called **short-circuit** evaluation of boolean conjunctions and disjunctions, respectively. If a false value is encountered in the left-to-right evaluation of the conjuncts of a `scand` form, then the result of the form is the false value, regardless of whether subsequent conjuncts contain errors or infinite loops. So

```
(scand (primop = 1 2) (primop / 3 0))
```

evaluates to *false* but

```
(scand (primop / 3 0) (primop = 1 2))
```

signals a divide-by-zero error. Similarly, if a true value is encountered in the left-to-right evaluation of the disjuncts of a `scor` form, then the result of the form is the true value, regardless of whether the subsequent disjuncts contain errors or infinite loops.

The `(let ((`I_1 E_1`) ... (`I_n E_n`))` E_0`)` expression evaluates E_0 in a context where the names $I_1 \ldots I_n$ are bound to the values of the expressions $E_1 \ldots E_n$. For example,

```
(let ((a (primop * 4 5))
      (b (primop + 3 4)))
  (/ (primop + a b) (primop - a b)))
```

evaluates to *2*.

The `(letrec ((`I_1 E_1`) ... (`I_n E_n`))` E_{body}`)` expression is similar to the `let` expression except that the names $I_1 \ldots I_n$ are visible inside of the expressions $E_1 \ldots E_n$. The `letrec` expression is similar to the `rec` expression, except that it can be thought of as solving a group of mutually recursive equations. For example,

```
(letrec ((even? (lambda (x)
                  (if (primop = x 0)
                      #t
                      (odd? (primop - x 1)))))
         (odd? (lambda (y)
                 (if (primop = y 0)
                     #f
                     (even? (primop - y 1))))))
  (list (even? 0) (odd? 1) (odd? 2) (even? 3)))
```

evaluates to $[\mathit{true}, \mathit{true}, \mathit{false}, \mathit{false}]$.

The top-level program construct `(program (`$I_{formals}{}^*$`)` E_{body} $D_{definitions}{}^*$`)` evaluates the body expression E_{body} in a context where

- the formal program parameters $I_{formals}{}^*$ are bound to the program arguments;
- the definition names $D_{definitions}{}^*$ in are bound to the values of the corresponding definition expressions;
- and each member of a set of **standard identifiers** (names in the standard library) is bound to the value specified by the library.

The advantage of a standard library is that many primitive constants and procedures can be factored out of the syntax of the language. Of course, it is still necessary to specify the components of the library somewhere in a language description. Typically the library is specified by listing all elements in the library along with a description of the semantics of each one. We will do this for the FL library in Section 6.2.3.

Definitions make it convenient to name top-level values (typically procedures) that are used within E_{body}. The value expressions of the definitions are evaluated in a mutually recursive context: the expression in a definition may refer to any name introduced by the definitions.

Consider the following sample FL program:

```
(fl (ns) (pair (map even? ns) (map odd? ns))
  (define even? (lambda (x)
                  (if (= x 0)
                      #t
                      (odd? (- x 1)))))
  (define odd? (lambda (y)
                 (if (= y 0)
                     #f
                     (even? (- y 1)))))
  (define map (lambda (f xs)
                (if (unit? xs)
                    xs
                    (pair (f (fst xs))
                          (map f (snd xs)))))))
```

The body expression `(pair (map even? ns) (map odd? ns))` refers to the procedures `even?`, `odd?`, and `map` defined via definitions within P. As above, `even?` and `odd?` are mutually recursive. The fact that standard identifiers are bound to appropriate procedures in the program body and definitions means that `=`, `-`, `unit?`, `fst`, and `snd` can all be used without the `primop` tag.

6.2.2.2 Desugaring

The transformation that desugars FL into FLK is presented in Figures 6.3 and 6.4. The transformation is specified by two desugaring functions:

1. $\mathcal{D}_{\text{exp}}$ maps an FL expression to a FLK expression.

2. $\mathcal{D}_{\text{prog}}$ maps FL programs to FLK programs.

As these desugaring functions walk down FL program and expression ASTs, they perform local transformations that replace the syntactic sugar constructs of FL by FLK constructs. Some clauses of the functions require the introduction of an identifier. In these cases, we want to ensure that the name does not conflict with any identifiers used by the programmer (or other identifiers introduced by the rules themselves). An implementation of the desugaring rules will include a way to generate such new names. We refer to these variables as **fresh.** (See page 237 for further discussion of fresh variables.)

In Figure 6.3, the top clauses descend the syntactic constructs of FL that are inherited from FLK, recursively applying $\mathcal{D}_{\text{exp}}$ to all subexpressions. This will

$\mathcal{D}_{\text{exp}} : \text{Exp}_{FL} \rightarrow \text{Exp}_{FLK}$

$\mathcal{D}_{\text{exp}}[\![L]\!] = L$

$\mathcal{D}_{\text{exp}}[\![I]\!] = I$

$\mathcal{D}_{\text{exp}}[\![\texttt{(primop } O\ E_1 \ldots E_n\texttt{)}]\!] = \texttt{(primop } O\ \mathcal{D}_{\text{exp}}[\![E_1]\!] \ldots \mathcal{D}_{\text{exp}}[\![E_n]\!]\texttt{)}$

$\mathcal{D}_{\text{exp}}[\![\texttt{(call } E_1\ E_2\texttt{)}]\!] = \texttt{(call } \mathcal{D}_{\text{exp}}[\![E_1]\!]\ \mathcal{D}_{\text{exp}}[\![E_2]\!]\texttt{)}$

$\mathcal{D}_{\text{exp}}[\![\texttt{(if } E_{test}\ E_{then}\ E_{else}\texttt{)}]\!] = \texttt{(if } \mathcal{D}_{\text{exp}}[\![E_{test}]\!]\ \mathcal{D}_{\text{exp}}[\![E_{then}]\!]\ \mathcal{D}_{\text{exp}}[\![E_{else}]\!]\texttt{)}$

$\mathcal{D}_{\text{exp}}[\![\texttt{(pair } E_{fst}\ E_{snd}\texttt{)}]\!] = \texttt{(pair } \mathcal{D}_{\text{exp}}[\![E_{fst}]\!]\ \mathcal{D}_{\text{exp}}[\![E_{snd}]\!]\texttt{)}$

$\mathcal{D}_{\text{exp}}[\![\texttt{(rec } I_{var}\ E_{body}\texttt{)}]\!] = \texttt{(rec } I_{var}\ \mathcal{D}_{\text{exp}}[\![E_{body}]\!]\texttt{)}$

$\mathcal{D}_{\text{exp}}[\![\texttt{(error } I_{msg}\texttt{)}]\!] = \texttt{(error } I_{msg}\texttt{)}$

$\mathcal{D}_{\text{exp}}[\![\texttt{(lambda () } E\texttt{)}]\!] = \texttt{(proc } I_{fresh}\ \mathcal{D}_{\text{exp}}[\![E]\!]\texttt{)}$, where I_{fresh} is fresh

$\mathcal{D}_{\text{exp}}[\![\texttt{(lambda (}I\texttt{) } E\texttt{)}]\!] = \texttt{(proc } I\ \mathcal{D}_{\text{exp}}[\![E]\!]\texttt{)}$

$\mathcal{D}_{\text{exp}}[\![\texttt{(lambda (}I_1\ I_{rest}{}^{+}\texttt{) } E\texttt{)}]\!] = \texttt{(proc } I_1\ \mathcal{D}_{\text{exp}}[\![\texttt{(lambda (}I_{rest}{}^{+}\texttt{) } E\texttt{)}]\!]\texttt{)}$

$\mathcal{D}_{\text{exp}}[\![\texttt{(}E\texttt{)}]\!] = \texttt{(call } \mathcal{D}_{\text{exp}}[\![E]\!]\ \texttt{\#u)}$

$\mathcal{D}_{\text{exp}}[\![\texttt{(}E_1\ E_2\texttt{)}]\!] = \texttt{(call } \mathcal{D}_{\text{exp}}[\![E_1]\!]\ \mathcal{D}_{\text{exp}}[\![E_2]\!]\texttt{)}$

$\mathcal{D}_{\text{exp}}[\![\texttt{(}E_1\ E_2\ E_{rest}{}^{+}\texttt{)}]\!] = \mathcal{D}_{\text{exp}}[\![\texttt{((call } E_1\ E_2\texttt{) } E_{rest}{}^{+}\texttt{)}]\!]$

$\mathcal{D}_{\text{exp}}[\![\texttt{(list)}]\!] = \texttt{\#u}$

$\mathcal{D}_{\text{exp}}[\![\texttt{(list } E_1\ E_{rest}{}^{*}\texttt{)}]\!] = \texttt{(pair } \mathcal{D}_{\text{exp}}[\![E_1]\!]\ \mathcal{D}_{\text{exp}}[\![\texttt{(list } E_{rest}{}^{*}\texttt{)}]\!]\texttt{)}$

$\mathcal{D}_{\text{exp}}[\![\texttt{(quote \#u)}]\!] = \texttt{\#u}$

$\mathcal{D}_{\text{exp}}[\![\texttt{(quote } B\texttt{)}]\!] = B$

$\mathcal{D}_{\text{exp}}[\![\texttt{(quote } N\texttt{)}]\!] = N$

$\mathcal{D}_{\text{exp}}[\![\texttt{(quote } I\texttt{)}]\!] = \texttt{(symbol } I\texttt{)}$

$\mathcal{D}_{\text{exp}}[\![\texttt{(quote (}SX_1 \ldots SX_n\texttt{))}]\!] = \mathcal{D}_{\text{exp}}[\![\texttt{(list (quote } SX_1\texttt{)} \ldots \texttt{(quote } SX_n\texttt{))}]\!]$

$\mathcal{D}_{\text{exp}}[\![\texttt{(cond (else } E_{default}\texttt{))}]\!] = \mathcal{D}_{\text{exp}}[\![E_{default}]\!]$

$\mathcal{D}_{\text{exp}}[\![\texttt{(cond (}E_{test1}\ E_{action1}\texttt{) (}E_{testi}\ E_{actioni}\texttt{)}^{*}\ \texttt{(else } E_{default}\texttt{))}]\!]$
$\quad = \texttt{(if } \mathcal{D}_{\text{exp}}[\![E_{test1}]\!]$
$\qquad\quad \mathcal{D}_{\text{exp}}[\![E_{action1}]\!]$
$\qquad\quad \mathcal{D}_{\text{exp}}[\![\texttt{(cond (}E_{testi}\ E_{actioni}\texttt{)}^{*}\ \texttt{(else } E_{default}\texttt{))}]\!]\texttt{)}$

$\mathcal{D}_{\text{exp}}[\![\texttt{(scand } E_{conjunct}{}^{*}\texttt{)}]\!]$ and $\mathcal{D}_{\text{exp}}[\![\texttt{(scor } E_{disjunct}{}^{*}\texttt{)}]\!]$ ***Left as exercises.***

$\mathcal{D}_{\text{exp}}[\![\texttt{(let ((}I_1\ E_1\texttt{)} \ldots \texttt{(}I_n\ E_n\texttt{)) } E_0\texttt{)}]\!]$
$\quad = \mathcal{D}_{\text{exp}}[\![\texttt{((lambda (}I_1 \ldots I_n\texttt{) } E_0\texttt{) } E_1 \ldots E_n\texttt{)}]\!]$

$\mathcal{D}_{\text{exp}}[\![\texttt{(letrec ((}I_1\ E_1\texttt{)} \ldots \texttt{(}I_n\ E_n\texttt{)) } E_0\texttt{)}]\!]$
$\quad = \mathcal{D}_{\text{exp}}[\![\texttt{(call (rec } I_{churchList}$
$\qquad\qquad\qquad \texttt{(proc } I_{selector}$
$\qquad\qquad\qquad\ \texttt{(}I_{selector}\ \texttt{(}I_{churchList}\ \texttt{(lambda (}I_1 \ldots I_n\texttt{) } E_1\texttt{))}$
$\qquad\qquad\qquad\qquad \vdots$
$\qquad\qquad\qquad \texttt{(}I_{churchList}\ \texttt{(lambda (}I_1 \ldots I_n\texttt{) } E_n\texttt{)))))}$
$\qquad\qquad\quad \texttt{(lambda (}I_1 \ldots I_n\texttt{) } E_0\texttt{))}]\!]$

$\quad$ where $I_{churchList} \neq I_{selector}$ are fresh and $\notin \bigcup_{i=0}^{n}$ ***FreeIds***$[\![E_i]\!]$

Figure 6.3: Desugaring FL expressions into FLK expressions.

```
$\mathcal{D}_{\text{prog}}$ : Program$_{FL}$ $\rightarrow$ Program$_{FLK}$
$\mathcal{D}_{\text{prog}}$ $[\![$(fl ($I_{formal}$*) $E_{body}$ (define $I_1$ $E_1$) ... (define $I_n$ $E_n$))$]\!]$
  = (flk ($I_{formal}$*)
       $\mathcal{D}_{\text{exp}}[\![$(let ((unit? (lambda (x) (primop unit? x)))
                    (boolean? (lambda (x) (primop boolean? x)))
                      ⋮
                    (+ (lambda (x y) (primop + x y)))
                      ⋮
                    (unit #u)
                    (true #t)
                    (false #f)
                    (cons (lambda (x y) (pair x y)))
                    (car (lambda (p) (primop fst p)))
                    (cdr (lambda (p) (primop snd p)))
                    (null? (lambda (x) (primop unit? x)))
                    (null (lambda () #u))
                    (nil #u)
                    (equal? ...) {Definition of this predicate left as an exercise.})
                    )
                (letrec (($I_1$ $E_1$)
                           ⋮
                         ($I_n$ $E_n$))
                  $E_{body}$))$]\!]$
```

Figure 6.4: Desugaring FL programs into FLK programs.

expand any syntactic sugar constructs appearing in the subexpressions. Note that $\mathcal{D}_{\text{exp}}$ acts as the identity function when applied to an FLK expression.

The rules for desugaring multi-abstraction into `proc` and multi-applications into `call` are based on the same currying trick that we use extensively in the metalanguage. (See Exercise 6.15 for an alternative approach to desugaring these constructs.) The recursive `list` desugaring creates a unit-terminated chain of pairs. The recursive `quote` desugaring descends an s-expression tree and builds up a corresponding tree of pairs with constants as leaves. The `cond` construct desugars into a nested sequence of `if`s. The `scand` and `scor` desugarings are left as exercises.

A `let` desugars into an application of an abstraction. This underscores the fact that abstractions are a fundamental means of naming in FL. Note that $E_1 \ldots E_n$ are outside the scope of $I_1 \ldots I_n$ and therefore cannot refer to the variables named by these identifiers.

However, in a `letrec`, the $E_1 \ldots E_n$ are *inside* the scope of $I_1 \ldots I_n$ and should refer to the variables named by these identifiers. Achieving this effect is challenging. We will present the desugaring in two stages. Suppose that `nth` is a standard identifier bound to a procedure that takes a list and an integer n and returns the nth element of the list (where elements are numbered from 1 up). Then an almost-correct desugaring for `letrec` is:

```
𝒟exp⟦(letrec ((I_1 E_1) ... (I_n E_n)) E_0)⟧
=
𝒟exp⟦(let ((I_outer (rec I_inner
                           (let ((I_1 (nth I_inner 1))
                                      ⋮
                                 (I_n (nth I_inner n)))
                             (list E_1 ... E_n)))))
        (let ((I_1 (nth I_outer 1))
                    ⋮
              (I_n (nth I_outer n)))
          E_0))⟧
   where I_outer ≠ I_inner are fresh identifiers
   and I_outer, I_inner ∉ ⋃_{i=0}^{n} FreeIds⟦E_i⟧
```

FreeIds is defined in Section 6.3.1 and in Figure 6.10. Assume for the moment that I_{outer} and I_{inner} are brand new names that don't conflict with any other names.

The basic idea of the desugaring is this: since `rec` can only find a single fixed point, design that fixed point to be a list of the n fixed points we really want. Inside the `rec`, the value of formal I_{inner} (which is a list of length n)

is destructured into its n elements, and the `list` expression is evaluated in a context where I_i is bound to the ith element of the list. Since `let` and `list` are both non-strict in FL, the solution to the `rec` is nontrivial. The name I_{outer} is bound to the solution of the `rec` and this list of length n is similarly destructured so that the body expression E_0 can be evaluated in a context where each I_i is bound to its individual solution.

The above result is an adequate desugaring, but it is complicated, and its use of the standard identifier `nth` is not only unaesthetic but also can lead to bugs due to name capture. For this reason, we will present an alternative desugaring that is more elegant. This desugaring is based on the same idea but represents lists as procedures. In this representation, which we shall call a **Church list**, an n-element list is a unary procedure whose single argument is an n-argument selector procedure that is applied to the n elements of the list. If $I_{churchList}$ is bound to an n-element Church list, then the application

($I_{churchList}$ `(lambda (`I_1 ... I_n`)` I_i`))`

extracts the ith element of the list. More generally, the application

($I_{churchList}$ `(lambda (`I_1 ... I_n`)` E`))`

returns the value of E in a context where each I_i is bound to the ith element of the list. Church lists give rise to the desugaring for recursive bindings shown in Figure 6.3.

$\mathcal{D}_{\text{prog}}$ is defined by a single clause, which transforms the definitions and body of a program using the `let` and `letrec` constructs. The desugaring makes standard identifiers available to the definitions and the body of the program by using an outer `let` to binding them to functions that perform the corresponding primitive applications via `primop`. Since multi-argument `lambda`s are used in the bindings, functions associated with the binary function names are appropriately curried. For example, in an FL program, `(+ 1)` stands for the incrementing function. The standard identifier `cons` makes FLK's `pair` construct available as a curried FL procedure, and the traditional LISP names `car`, `cdr`, `null?`, and `nil` are provided as synonyms for `fst`, `snd`, `unit?`, and `#u`. There are a few other handy synonyms as well. The mutually recursivene nature of the definitions is implemented by desugaring them into a `letrec`.

Note that $\mathcal{D}_{\text{exp}}$ is applied once again to the result of $\mathcal{D}_{\text{exp}}$ on `letrec` and $\mathcal{D}_{\text{prog}}$ on `program`.

▷ **Exercise 6.1** Provide the missing desugarings for FL's `scand` and `scor` constructs (see Figure 6.3). ◁

▷ **Exercise 6.2** The desugaring for `letrec` in Figure 6.4 requires a pair of fresh identifiers. There is another desugaring for `letrec` that requires no fresh identifiers whatsoever. This desugaring has a recursive structure not exhibited by the other versions. Below is a skeleton of the desugaring.

$$\mathcal{D}_{\text{exp}}[\![\texttt{(letrec ((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ \ E_0\texttt{)}]\!]$$
$$=$$
$$\mathcal{D}_{\text{exp}}[\![\texttt{(let ((}I_1\ \texttt{(rec}\ I_1\ \square_1\texttt{))}\ \ldots\ \texttt{(}I_n\ \texttt{(rec}\ I_n\ \square_n\texttt{)))}\ \ E_0\texttt{)}]\!]$$

where the boxes $\square_i$ are to be filled in appropriately.

a. Give the general form for expressions that fill the boxes $\square_i$ in such a way that the above skeleton defines a correct desugaring for `letrec`.

b. Using your approach, how many `rec`s will appear in a desugaring of a `letrec` with 5 bindings?

c. Give a closed form solution for the number of `rec`s that will appear in a desugaring of a `letrec` with n bindings.

d. Comment on the practicality of this `letrec` desugaring. ◁

▷ **Exercise 6.3** Two constructs are said to be **idempotent** (roughly, "of equal power") if each can be expressed as a desugaring into the other. For example, multi-argument procedures and single-argument procedures are idempotent: multi-argument abstractions and calls can be desugared into single-argument ones via currying; and single-argument abstractions and calls are a special subcase of the multi-argument ones. On the other hand, pairs and procedures are *not* idempotent; although Church's techniques give a desugaring of pairs into procedures, procedure abstractions and calls cannot be desugared into pairs.

We have considered a version of FLK where `rec` is the kernel recursion construct and FL's `letrec` is desugared into `rec`. Show that `rec` and `letrec` are idempotent by providing a desugaring of `rec` into `letrec`. ◁

▷ **Exercise 6.4** Many LISP dialects support an alternative version of `define` for constructing new functions. The syntax is of the form

(`define` (*function-name arg-1* … *arg-n*) *function-body*)

For example, the squaring function can be defined as:

```
(define (square x) (* x x))
```

Extend the desugaring for FL to handle this syntax. Hint: It is easier to add another processing step for definitions rather than modifying the desugaring of `program` expressions. ◁

▷ **Exercise 6.5** It is often useful for the value of a `let`-bound variable to depend on the value of a previous `let`-bound variable. In the current version of FL, achieving this behavior requires nested `let` expressions. For example:

```
(let ((r (+ 1 2)))
  (let ((square-r (* r r)))
    (let ((circum (* 2 (* pi square-r))))
      ... code using r, square-r, and circum ...
      )))
```

Many Lisp dialects support a `let*` construct that looks just like `let` except that its variables are guaranteed to be bound to their associated values in the order that they appear in the list of bindings. A val expression in `let*` can refer to the result of a previous binding within the same `let*`. Using `let*`, the above example could be rendered:

```
(let* ((r (+ 1 2))
       (square-r (* r r))
       (circum (* 2 (* pi square-r))))
  ... code using r, square-r, and circum ...
  )
```

Write an appropriate desugaring for `let*`. ◁

▷ **Exercise 6.6** It is common to create locally recursive procedures and then call them immediately to start a process. For example, iterative factorial can be expressed in FL as:

```
(define fact
  (lambda (n)
    (letrec ((iter (lambda (num ans)
                     (if (= num 0)
                         ans
                         (iter (- num 1) (* num ans))))))
      (iter n 1)))))
```

Some versions of Lisp have a "named `let`" or "`let` loop" construct that makes this pattern easier to express. The construct is of the form

$$\texttt{(let } I_{name} \texttt{ ((}I_{var}\ E_{val}\texttt{)*) } E_{body}\texttt{)}$$

It looks like a `let` expression except that it has an additional identifier I_{name}. The n variables I_{var} are first bound to the values E_{val} and then the E_{body} is evaluated in a context where these bindings are in effect *and* the name I_{name} refers to a procedure of the n variables I_{var} that computes E_{body}. Using named let, the iterative factorial construct can be expressed more succinctly as:

```
(define fact
  (lambda (n)
    (let iter ((num n) (ans 1))
      (if (= num 0)
          ans
          (iter (- num 1) (* num ans))))))
```

Extend the desugaring for `let` to handle named `let`. ◁

▷ **Exercise 6.7** In FL, definitions are only allowed within the `program` construct at "top-level"; yet a local form of definition within `lambda` and `let` expressions would often be useful. Generalize the idea of definitions by modifying FL to support local definitions. Design a syntax for your change, and show how to express it in terms of a desugaring. ◁

▷ **Exercise 6.8** Ben Bitdiddle is upset by the desugaring for nullary (i.e., zero-argument) abstractions and applications. He argues (correctly) that, according to the desugarings, the FL expression `((lambda (x) x))` will return `#u`. He believes that evaluating this expression should give an error.

One way to fix this problem is to package up multiple arguments into some sort of data structure. See Exercise 6.15 for an example of this approach. Here we will consider other approaches for handling nullary abstractions and applications.

a. Bud Lojack suggests desugaring `(lambda ()` E`)` into E and `(`E`)` into E. Give examples of FL expressions that have a questionable behavior under this desugaring.

b. Paula Morwicz suggests a desugaring in which

$$\begin{aligned}
\mathcal{D}_{\text{exp}}[\![\texttt{(}E\texttt{)}]\!] &= \texttt{(call (call } \mathcal{D}_{\text{exp}}[\![E]\!] \texttt{ \#t) \#u)}\\
\mathcal{D}_{\text{exp}}[\![\texttt{(}E_1\ \ E_2\texttt{)}]\!] &= \texttt{(call (call } \mathcal{D}_{\text{exp}}[\![E_1]\!] \texttt{ \#f) } \mathcal{D}_{\text{exp}}[\![E_2]\!]\texttt{)}\\
\mathcal{D}_{\text{exp}}[\![\texttt{(}E_1\ \ E_2\ \ E_{rest}{}^{+}\texttt{)}]\!] &= \texttt{((call (call } \mathcal{D}_{\text{exp}}[\![E_1]\!] \texttt{ \#f) } \mathcal{D}_{\text{exp}}[\![E_2]\!]\texttt{)}\\
&\qquad\qquad \mathcal{D}_{\text{exp}}[\![E_{rest}{}^{+}]\!]\texttt{)}
\end{aligned}$$

i. Give the corresponding desugarings for multi-abstractions.

ii. What value does `((lambda (x) x))` have under this desugaring?

c. Ben reasons that the fundamental problem exhibited by the nullary desugarings is that there is no way to call a procedure without passing it an argument. He decides to extend FLK with the following kernel forms for parameterless procedures:

`(freeze` E`)`: Return a "frozen" value that suspends the evaluation of E.

`(thaw` E`)`: Unsuspends the expression frozen within a frozen value. Gives an error if called on any value other than one created by `freeze`.

Show how `freeze` and `thaw` can be used to fix Ben's problem.

d. Sam Antix doesn't like the fact that multi-abstractions and multi-applications both have three desugaring clauses. Figuring that only two clauses should suffice in each case, he develops the following desugaring rules based on Ben's `freeze` and `thaw` commands:

$$\mathcal{D}_{\text{exp}}[\![\texttt{(lambda () } E\texttt{)}]\!] = \mathcal{D}_{\text{exp}}[\![\texttt{(freeze } E\texttt{)}]\!]$$
$$\mathcal{D}_{\text{exp}}[\![\texttt{(lambda (}I_1\ I_{rest}{}^*\texttt{) } E\texttt{)}]\!]$$
$$= \texttt{(proc } I_1\ \mathcal{D}_{\text{exp}}[\![\texttt{(lambda (}I_{rest}{}^*\texttt{) } E\texttt{)}]\!]\texttt{)}$$
$$\mathcal{D}_{\text{exp}}[\![\texttt{(}E\texttt{)}]\!] = \mathcal{D}_{\text{exp}}[\![\texttt{(thaw } E\texttt{)}]\!]$$
$$\mathcal{D}_{\text{exp}}[\![\texttt{(}E_1\ E_{rest}{}^*\texttt{)}]\!] = \mathcal{D}_{\text{exp}}[\![\texttt{((call } E_1\ E_2\texttt{) } \mathcal{D}_{\text{exp}}[\![E_{rest}{}^*]\!]\texttt{)}]\!]$$

Discuss the strengths and weaknesses of Sam's desugaring. ◁

▷ **Exercise 6.9†** Show that a desugaring process based on the rules in Figures 6.3 and 6.4 is guaranteed to terminate. ◁

6.2.3 The FL Standard Library

The FL standard library is shown in Figure 6.5. All of FLK's primitives (those names that can be used in `primop`) are included as curried procedures. Note that FL only supports integers and not floating point numbers, so arithmetic operations like `+`, `*`, `<=`, etc. only work on integers. It would be straightforward to extend FL to support floating point numbers, and in some code examples it will be convenient to assume that FL does support floating point numbers. In such examples, we will use arithmetic operation names prefixed with `f` to indicate floating point operations: e.g., `f+`, `f*`, and `f<=`.

The standard library also includes a number of other standard identifiers that are convenient, such as constants (`unit`, `true`, `false`, and `nil`), Scheme-style operations on lists (`cons`, `car`, `cdr`, `null?`), a generic binary equality tester (`equal?`) that tests for equality between any two FL values that are not procedures.

6.2.4 Examples

Although FL is a toy language, it packs a fair bit of expressive punch. In this section, we illustrate the expressive power of FL in the context of a few examples.

6.2.4.1 List Utilities

As a simple example of FL procedures, consider the list procedures in Figure 6.6. The `list?` procedure takes a value and determines if it is a list – i.e., a sequence of pairs terminated with the unit value. The `length` procedure returns the length

Primitives (can be used in `primop`):

`unit?`	Unary type predicate for the unit value.
`boolean?`	Unary type predicate for booleans.
`integer?`	Unary type predicate for integers.
`symbol?`	Unary type predicate for symbols.
`procedure?`	Unary type predicate for procedures (i.e., a functional value).
`pair?`	Unary type predicate for pairs.
`not?`	Unary boolean negation.
`and?`	Binary boolean conjunction (not short-circuit).
`or?`	Binary boolean disjunction (not short-circuit).
`bool=?`	Binary boolean equality predicate.
`+`	Binary integer addition.
`-`	Binary integer subtraction.
`*`	Binary integer multiplication.
`/`	Binary integer division.
`%`	Binary integer remainder.
`=`	Binary integer equality predicate.
`!=`	Binary integer inequality predicate.
`<`	Binary integer less-than predicate.
`<=`	Binary integer less-than-or-equal-to predicate.
`>`	Binary integer greater-than predicate.
`>=`	Binary integer greater-than-or-equal-to predicate.
`sym=?`	Binary symbol equality.
`fst`	Unary selector of the first element of a given pair.
`snd`	Unary selector of the second element of a given pair.

Other Standard Identifiers:

`unit`	The unit value.
`true`	Boolean truth.
`false`	Boolean falsity.
`cons`	Binary list constructor.
`car`	Unary list selector — head of list.
`cdr`	Unary list selector — tail of list.
`nil`	The empty list (synonym for the unit value).
`null`	Unary empty list constructor.
`null?`	Unary empty list predicate.
`equal?`	Generic binary equality test

Figure 6.5: FL Standard Library.

of a list. The `member?` procedure determines if a value is an element of a list. The `merge` procedure takes a less-than-or-equal-to predicate `leq` and two lists `xs` and `ys` that are assumed to be sorted according to this predicate and returns the sorted list containing all the elements of both lists (including duplicates, if any). The `alts` procedure returns a pair of (1) all the odd-indexed[3] elements and (2) all the even-indexed elements of a given list, preserving the relative order of elements in each sublist. The `merge-sort` procedure takes an ordering predicate and a list of elements and returns a list of the same elements ordered according to the ordering predicate.

Here are some sample uses of these procedures:

`(list? 17)` $\xrightarrow[FL]{}$ *false*
`(list? (list 7 2 5))` $\xrightarrow[FL]{}$ *true*
`(list? (pair 3 (pair 4 5)))` $\xrightarrow[FL]{}$ *false*

`(length (list))` $\xrightarrow[FL]{}$ *0*
`(length (list 7 2 5))` $\xrightarrow[FL]{}$ *3*

`(member? 2 (list 7 2 5))` $\xrightarrow[FL]{}$ *true*
`(member? 17 (list 7 2 5))` $\xrightarrow[FL]{}$ *false*
`(member? '* '(+ - * /))` $\xrightarrow[FL]{}$ *true*

`(merge < (null) (list 3 4 6))` $\xrightarrow[FL]{}$ $[3, 4, 6]$
`(merge < (list 1 6 8) (list 3 4 6))` $\xrightarrow[FL]{}$ $[1, 3, 4, 6, 6, 8]$

`(alts (null))` $\xrightarrow[FL]{}$ $\langle [], [] \rangle$
`(alts (list 7))` $\xrightarrow[FL]{}$ $\langle [7], [] \rangle$
`(alts (list 7 2))` $\xrightarrow[FL]{}$ $\langle [7], [2] \rangle$
`(alts (list 7 2 4 5 1 4 3))` $\xrightarrow[FL]{}$ $\langle [7, 4, 1, 3], [2, 5, 4] \rangle$

`(merge-sort <= (list 7 2 4 1 5 4 3))` $\xrightarrow[FL]{}$ $[1, 2, 3, 4, 4, 5, 7]$
`(merge-sort >= (list 7 2 4 1 5 4 3))` $\xrightarrow[FL]{}$ $[7, 5, 4, 4, 3, 2, 1]$
```
(merge-sort (lambda (a b) (<= (% a 4) (% b 4)))
            (list 7 2 4 1 5 4 3))
```
$\xrightarrow[FL]{}$ $[4, 4, 1, 5, 7, 2, 3]$

[3]Assume that list elements are indexed starting with 1.

6.2.4.2 An ELM Interpreter

As a more interesting example of an FL program, in Figure 6.7 we use FL to write an interpreter for the ELM subset of the EL language (Exercise 3.10). Recall that ELM is EL without conditional and boolean expressions. The `elm-eval` procedure evaluates an ELM expression relative to a list of numbers, `args`, which are the the program inputs. ELM expressions are represented as FL s-expressions. `elm-eval` is written as a dispatch on the type of expression, which is determined by the syntax predicates `lit?`, `arg?`, and `arithop?`. The selectors `lit-num`, `arg-index`, `arithop-op`, `arithop-rand1`, `arithop-rand2` extract components of syntax nodes. The `arg-index` procedure returns the `index`th element of the given list `nums` (where indices are assumed to start at 1). The `primop->proc` procedure converts a symbol (such as `'+`) to a binary FL procedure (such as the addition procedure `+`).

Here are some examples of the `elm-eval` procedure in action:

`(elm-eval '(* (arg 1) (arg 1)) '(5))` $\xrightarrow[FL]{}$ *25*
`(elm-eval '(/ (+ (arg 1) (arg 2)) 2) '(6 8))` $\xrightarrow[FL]{}$ *7*
`(elm-eval '(+ (arg 1) (arg 2)) '(3))` $\xrightarrow[FL]{}$ *error:arg-index-out-of-bounds*

6.2.4.3 A Pattern Matcher

Programs that match a pattern against a tree structure are so useful that they should be part of every programmer's bag of tricks. Figures 6.8 and 6.9 present a simple pattern matching program written in FL.

The pattern matcher manipulates trees represented as s-expressions. Patterns are trees whose leaves are either constants (unit, booleans, integers, or symbols) or pattern variables. We represent the pattern variable named *I* by the s-expression `(?` *I*`)`. Because of this convention, the symbol `?` is considered special and should never be used as one of the symbol constants in the pattern or the structure being matched. Examples of legal patterns include:

```
(? pat)
(The (? adjective) programmer (? adverb) hacked (? noun))
((? a) is equal to (? a))
(((? a) (? b)) is the reflection of ((? b) (? a)))
```

A pattern *p* matches an s-expression *s* if there is some set of bindings between

```
(define list?
  (lambda (val)
    (scor (null? val)
          (scand (pair? val) (list? (snd val))))))

(define length
  (lambda (lst)
    (if (null? lst)
        0
        (+ 1 (length (cdr lst))))))

(define member?
  (lambda (elt lst)
    (scand (not (null? lst))
           (scor (equal? elt (car lst))
                 (member? elt (cdr lst))))))

(define merge
  (lambda (leq xs ys)
    (cond ((null? xs) ys)
          ((null? ys) xs)
          ((leq (car xs) (car ys))
           (cons (car xs) (merge leq (cdr xs) ys)))
          (else
           (cons (car ys) (merge leq xs (cdr ys)))))))

(define alts
  (lambda (ws)
    (if (null ws)
        (pair (null) (null))
        (let ((alts-rest (alts (cdr ws))))
          (pair (cons (car ws) (snd alts-rest))
                (fst alts-rest))))))

(define merge-sort
  (lambda (leq zs)
    (if (scor (null? zs) (null? (cdr zs)))
        zs
        (let ((split (alts zs)))
          (merge (fst split) (snd split))))))
```

Figure 6.6: Some list procedures written in FL.

```
(fl (pgm args)
  (cond ((not? (elm-program pgm)) (error ill-formed-program))
        ((not? (list? args)) (error ill-formed-argument-list))
        ((not? (= (elm-nargs pgm) (length args)))
         (error wrong-number-of-args))
        (else  (elm-eval (elm-body pgm) args)))

  (define elm-eval
    (lambda (exp args)
      (cond ((lit? exp) (lit-num exp))
            ((arg? exp) (get-arg (arg-index exp) args))
            ((arithop? exp)
            ((primop->proc (arithop-op exp))
             (elm-eval (arithop-rand1 exp) args)
             (elm-eval (arithop-rand2 exp) args)))
            (else (error illegal-expression)))))

  (define get-arg
    (lambda (index nums)
      (cond ((scor (<= index 0) (null? nums))
             (error arg-index-out-of-bounds))
            ((= index 1) (car nums))
            (else (get-arg (- index 1) (cdr nums))))))

  (define primop->proc
    (lambda (sym)
      (cond ((sym=? sym '+) +) ((sym=? sym '-) -)
            ((sym=? sym '*) *) ((sym=? sym '/) /)
            (else (error illegal-op)))))

  ;; Abstract syntax
  (define elm-program?
    (lambda (sexp)
      (scand (list? sexp) (= (length sexp) 3) (sym=? (car exp) 'elm))))
  (define elm-program-nargs (lambda (sexp) (car (cdr sexp))))
  (define elm-program-body (lambda (sexp) (car (cdr (cdr sexp)))))
  (define lit? integer?)
  (define lit-num (lambda (lit) lit))
  (define arg?
    (lambda (exp)
      (scand (list? exp) (= (length exp) 2) (sym=? (car exp) 'arg))))
  (define arg-index (lambda (exp) (car (cdr exp))))
  (define arithop?
    (lambda (exp)
      (scand (list? exp) (= (length exp) 3) (member? (car exp) '(+ - * /)))))
  (define arithop-op (lambda (exp) (car exp)))
  (define arithop-rand1 (lambda (exp) (car (cdr exp))))
  (define arithop-rand2 (lambda (exp) (car (cdr (cdr exp)))))

  ;; List utilities
  (define list? (lambda (sexp) ...))
  (define length (lambda (xs) ...))
  (define member? (lambda (x xs) ...))
```

Figure 6.7: An interpreter for ELM, a subset of EL.

pattern variables and s-expressions such that instantiating the variables with their bindings in p yields s. Constraints on the form of the pattern can be specified by using the same pattern variable in more than one place.

For example, consider the pattern `((? a) is equal to (? a))`. It matches the following two s-expressions:

```
(1 is equal to 1)   and
((Ben Bitdiddle) is equal to (Ben Bitdiddle))
```

but not

```
(1 is equal to 2)   or
(Ben Bitdiddle is equal to Ben Bitdiddle)
```

In the final example, `Ben Bitdiddle` is two s-expressions and cannot be matched by a single pattern.

The entry point of the pattern matcher is the `match-sexp` procedure, which takes a pattern and an s-expression as arguments. If the pattern does not match the s-expression, `match-sexp` returns the symbol `*failed*`. Otherwise, `match-sexp` returns a dictionary structure that contains pattern variable bindings that make the match successful. `match-sexp` just passes responsibility to `match-with-dict`, which does the real work.

In addition to a pattern and an s-expression, `match-with-dict` takes a dictionary. It matches the pattern to the s-expression in the context of the dictionary. That is, any match of a variable in the pattern must be consistent with the binding that is already in the dictionary. In high-level terms, `match-with-dict` performs a left-to-right depth-first walk in lock-step over both the pattern tree and s-expression tree. A dictionary representing the bindings of variables seen so far flows along this depth-first path. Along the path, the matching process checks whether:

- an internal node of the pattern tree has the same number of subtrees as the corresponding internal node of the s-expression.
- a constant leaf in the pattern is matched by exactly the same constant leaf in the corresponding position of the pattern.
- a variable leaf in the pattern is matched by an s-expression that is consistent with the bindings represented by the current dictionary.

A successful check allows the dictionary to flow to the next part of the path, possibly extended with a new binding. After an unsuccessful check, the dictionary is replaced by a failure symbol that propagates through the rest of the path.

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

(define match-with-dict
  (lambda (pat sexp dict)
    (cond ((failed? dict) dict)    ; Propagate failures.
          ((null? pat)
           (if (null? sexp)
               dict                ; PAT and SEXP both ended.
               (fail)))            ; PAT ended but SEXP didn't.
          ((null? sexp) (fail))    ; SEXP ended but PAT didn't.
          ((pattern-constant? pat)
           (if (sexp=? pat sexp) dict (fail)))
          ((pattern-variable? pat)
           (dict-bind (pattern-variable-name pat) sexp dict))
          (else (match-with-dict (cdr pat)
                                 (cdr sexp)
                                 (match-with-dict (car pat)
                                                  (car sexp)
                                                  dict))))))

(define pattern-variable?
  (lambda (pat) (if (pair? pat)
                    (sexp=? (car pat) '?)
                    #f)))

(define pattern-variable-name (lambda (sexp) (car (cdr sexp))))

(define pattern-constant?
  (lambda (p) (or (symbol? p) (integer? p) (boolean? p) (unit? p))))

(define fail (lambda () '*failed*))
(define failed? (lambda (dict) (sexp=? dict '*failed*)))
```

Figure 6.8: A pattern matcher in FL, part 1.

There are many possible representations for dictionaries. We represent a dictionary as a list of bindings, where each binding is a pair of a pattern variable identifier and the associated s-expression.

Examples of using `match-sexp`:

```
(match-sexp '(a short sentence) '(a short sentence))
```
$\xrightarrow[FL]{}$ [] {*Match succeeds with the empty dictionary.*}

```
(match-sexp '(a short sentence) '(a longer sentence))
```
$\xrightarrow[FL]{}$ $'{*}failed{*}$ {*Match failed.*}

```
(match-sexp '((? article) (? adjective) sentence)
            '(a longer sentence))
```
$\xrightarrow[FL]{}$ $[\langle 'article, 'a\rangle, \langle 'adjective, 'longer\rangle]$

```
;; Can make use of FL's currying
(define m1 (match-sexp '((a (b (? c))) (((? c) b) a))))
```

`(m1 '((a (b (c (d)))) (((c (d)) b) a)))` $\xrightarrow[FL]{}$ $[\langle 'c, ['c, ['d]]\rangle]$

`(m1 '((a (b (c (d)))) ((((d) c) b) a)))` $\xrightarrow[FL]{}$ $'{*}failed{*}$

6.3 Variables and Substitution

Intuitively, the meaning of an FLK abstraction (`proc` I E) shouldn't depend on the particular name chosen for I, which is known as its **formal parameter**. Just as we expect the meaning of an integral to be independent of the choice of the variable of integration (so that $\int_a^b f(x)dx = \int_a^b f(y)dy$), we expect the meaning of an FLK abstraction to be invariant under a change to the name of its variable. Thus, the identity abstraction `(proc a a)` should also be expressible as `(proc x x)` or `(proc square square)`. Furthermore, the variable references named by `a`, `x`, and `square` are logically distinct from any variable references coincidentally sharing the same name in other expressions.

This section formalizes this intuition about variables in FLK expressions.

6.3.1 Terminology

First, it's important to tease apart several related but distinct concepts in our terminology concerning names. We reserve the word **variable** for the logical entity that is introduced by an abstraction and is referenced by a variable reference. The word **identifier** designates the name that stands for a given variable within an expression. The identity abstraction discussed above has a

```
;;; Dictionaries
(define dict-bind
  (lambda (sym1 sexp dict)
    (let ((value (dict-lookup sym1 dict)))
      (cond ((unbound? value)
             (dict-adjoin-binding (binding-make sym1 sexp) dict))
            ((sexp=? value sexp) dict)
            (else (fail))))))

(define dict-lookup
  (lambda (name dict)
    (cond ((dict-empty? dict) (unbound))
          ((sym=? name (binding-name (dict-first-binding dict)))
           (binding-value (dict-first-binding dict)))
          (else (dict-lookup name (dict-rest-bindings dict))))))

(define dict-empty (lambda () (list)))
(define dict-empty? null?)
(define dict-adjoin-binding cons)
(define dict-first-binding car)
(define dict-rest-bindings cdr)
(define unbound (lambda () '*unbound*))
(define unbound? (lambda (sym) (sexp=? sym '*unbound*)))

;;; Bindings
(define binding-make cons)
(define binding-name car)
(define binding-value cdr)

;; Utilities
(define sexp=?
  (lambda (obj1 obj2)
    (cond ((unit? obj1) (unit? obj2))
          ((and (boolean? obj1) (boolean? obj2)) (boolean=? obj1 obj2))
          ((and (integer? obj1) (integer? obj2)) (= obj1 obj2))
          ((and (symbol? obj1) (symbol? obj2)) (sym=? obj1 obj2))
          ((and (pair? obj1) (pair? obj2))
           (and (sexp=? (car obj1) (car obj2))
                (sexp=? (cdr obj1) (cdr obj2))))
          (else #f))))
```

Figure 6.9: A pattern matcher in FL, part 2.

single variable, and the identifier that names it is arbitrary. In the expression `(proc x (call x (proc x x)))` there are two logically distinct variables, but they happen to be named by the same identifier.

Sometimes it is useful to distinguish different **occurrences** of an identifier or subexpression within an expression. In the expression

```
(call (proc x x) (proc x x))
```

there are four occurrences of the identifier `x` and two occurrences of the subexpression `(proc x x)`. In order to refer to a particular occurrence, we can imagine that each distinct identifier or expression has been numbered from left to right starting with 1. Thus, we could view the above application as

$$\texttt{(call (proc x}^{1}\ \texttt{x}^{2}\texttt{)}^{1}\ \texttt{(proc x}^{3}\ \texttt{x}^{4}\texttt{)}^{2}\texttt{)}\ ,$$

where the superscripts distinguish the occurrences of an identifier or subexpression. When we say "the ith occurrence of `x`" we mean $\texttt{x}^{i}$.

We shall say that the formal parameter I appearing in an FLK abstraction (`proc` I E) is a **binding occurrence** of I and that the abstraction **binds** I. An occurrence of an identifier in an FLK expression I is **bound** if it is a binding occurrence or it occurs in the body of some abstraction that binds I; otherwise, that occurrence of the identifier is said to be **free**. For example, in `(proc a (proc b (call a c)))`, the single occurrence of `b` and both occurrences of `a` are bound, while the single occurrence of `c` is free. The freeness or boundness of an identifier occurrence depends on the context in which the identifier is viewed. Thus, in the previous example, the second occurrence of `a` is free in `(call a c)` and in `(proc b (call a c))` but not in `(proc a (proc b (call a c)))`. It is possible in one expression to have some occurrences of an identifier that are bound and other occurrences of the same identifier that are free. In `(call (proc a a) a)` the first and second occurrences of `a` are bound, while the third occurrence is free.

An identifier (as opposed to an *occurrence* of an identifier) is said to be a **free identifier** (likewise, **bound**) in an expression if at least one of its occurrences is free (likewise, bound) in the expression. For instance, in the expression

```
(call b (proc a (proc b (call a c)))),
```

`a` and `b` are bound identifiers and `b` and `c` are free identifiers. Similarly, a variable is said to be **free** (likewise, **bound**) in an expression if the identifier that names it is free (likewise, bound). Note that an identifier may be both bound and free in an expression, but a variable can only be one or the other. An expression is **closed** if it contains no free identifiers (or, equivalently, no free variables). Expressions with free variables often arise when considering subexpressions of a given expression. For instance, in the subexpression `(proc b (call b a))` of the closed expression `(proc a (proc b (call b a)))`, the identifier `a` names a free variable.

Using definition by structural induction, it is straightforward to define functions *FreeIds* and *BoundIds* that map FLK expressions to sets of their free and bound identifiers, respectively. These functions are presented in Figure 6.10. Both functions have signature

$$\text{Exp} \to \mathcal{P}(\text{Identifier})$$

where $\mathcal{P}(\text{Identifier})$ is the power set (set of all subsets) of Identifier. For example,

$$\begin{aligned} \mathit{FreeIds}[\![\texttt{(call b (proc a (proc b (call a c))))}]\!] &= \{\texttt{b}, \texttt{c}\} \\ \mathit{BoundIds}[\![\texttt{(call b (proc a (proc b (call a c))))}]\!] &= \{\texttt{a}, \texttt{b}\} \end{aligned}$$

One subtle note deserves mention. An I that appears within double brackets on the left hand side of the definitions stands for a variable reference that is an element of the syntactic domain Exp. On the other hand, an unbracketed I on the right hand side of the definitions stands for an element of the syntactic domain Identifier.

▷ **Exercise 6.10** For each of the following FLK expressions:

- Indicate for every occurrence of an identifier whether it is bound or free.
- Determine the free identifiers and bound identifiers of the expression.

a. `(proc x (call x y))`

b. `(call (proc z (proc x (call (call x y) z))) z)`

c. `(call z (proc y (call (proc z (call x y)) z)))`

d. `(proc x (call (call (proc y (call (proc z (call x r)) y)) y) z))` ◁

6.3.2 General Properties of Variables

Throughout mathematical and computational notation, variables serve as syntactic placeholders that range over some set of semantic entities. Variables are manipulated in two different kinds of expressions:

1. A **variable declaration** introduces a new placeholder into an expression.
2. A **variable reference** uses a placeholder within an expression.

$$\begin{array}{rcl}
\mathit{FreeIds} & : & \text{Exp} \rightarrow \mathcal{P}(\text{Identifier}) \\
\mathit{FreeIds}[\![L]\!] & = & \{\} \\
\mathit{FreeIds}[\![I]\!] & = & \{I\} \\
\mathit{FreeIds}[\![\texttt{(primop}\ O\ E_1\ \ldots\ E_n\texttt{)}]\!] & = & \bigcup_{i=1}^{n} \mathit{FreeIds}[\![E_i]\!] \\
\mathit{FreeIds}[\![\texttt{(proc}\ I\ E\texttt{)}]\!] & = & \mathit{FreeIds}[\![E]\!] - \{I\} \\
\mathit{FreeIds}[\![\texttt{(call}\ E_1\ E_2\texttt{)}]\!] & = & \mathit{FreeIds}[\![E_1]\!] \cup \mathit{FreeIds}[\![E_2]\!] \\
\mathit{FreeIds}[\![\texttt{(if}\ E_1\ E_2\ E_3\texttt{)}]\!] & = & \mathit{FreeIds}[\![E_1]\!] \cup \mathit{FreeIds}[\![E_2]\!] \\
 & & \cup \mathit{FreeIds}[\![E_3]\!] \\
\mathit{FreeIds}[\![\texttt{(pair}\ E_1\ E_2\texttt{)}]\!] & = & \mathit{FreeIds}[\![E_1]\!] \cup \mathit{FreeIds}[\![E_2]\!] \\
\mathit{FreeIds}[\![\texttt{(rec}\ I\ E\texttt{)}]\!] & = & \mathit{FreeIds}[\![E]\!] - \{I\}
\end{array}$$

$$\begin{array}{rcl}
\mathit{BoundIds} & : & \text{Exp} \rightarrow \mathcal{P}(\text{Identifier}) \\
\mathit{BoundIds}[\![L]\!] & = & \{\} \\
\mathit{BoundIds}[\![I]\!] & = & \{\} \\
\mathit{BoundIds}[\![\texttt{(primop}\ O\ E_1\ \ldots\ E_n\texttt{)}]\!] & = & \bigcup_{i=1}^{n} \mathit{BoundIds}[\![E_i]\!] \\
\mathit{BoundIds}[\![\texttt{(proc}\ I\ E\texttt{)}]\!] & = & \{I\} \cup \mathit{BoundIds}[\![E]\!] \\
\mathit{BoundIds}[\![\texttt{(call}\ E_1\ E_2\texttt{)}]\!] & = & \mathit{BoundIds}[\![E_1]\!] \cup \mathit{BoundIds}[\![E_2]\!] \\
\mathit{BoundIds}[\![\texttt{(if}\ E_1\ E_2\ E_3\texttt{)}]\!] & = & \mathit{BoundIds}[\![E_1]\!] \cup \mathit{BoundIds}[\![E_2]\!] \\
 & & \cup \mathit{BoundIds}[\![E_3]\!] \\
\mathit{BoundIds}[\![\texttt{(pair}\ E_1\ E_2\texttt{)}]\!] & = & \mathit{BoundIds}[\![E_1]\!] \cup \mathit{BoundIds}[\![E_2]\!] \\
\mathit{BoundIds}[\![\texttt{(rec}\ I\ E\texttt{)}]\!] & = & \{I\} \cup \mathit{BoundIds}[\![E]\!]
\end{array}$$

Figure 6.10: Definition of the free and bound identifiers of a FLK expression.

The region of an expression in which a particular variable may be referenced is called the **scope** of that variable.

In standard notations, variables are typically represented by identifiers, and declarations and references are distinguished in the format of expressions. For example, compare how variables are declared and referenced in notations for FLK, integration, summation, union, and logical quantification (in each case, the declaring occurrence of the variable x has been boxed):

$$\texttt{(proc } \boxed{\texttt{x}} \texttt{ x)} \qquad \int_a^b x\, d\boxed{x} \qquad \sum_{\boxed{x}=1}^{n} x^2 \qquad \bigcup_{\boxed{x}\in A} x \qquad \forall \boxed{x}. f(x) = g(x)$$

Notations in which variables are represented by identifiers share the following properties:

1. Modulo certain restrictions to be discussed shortly, it is possible to consistently rename a variable within its scope without changing the meaning of the entire expression. Thus, in each of the above notations, the x can be changed to y without changing the meaning:

$$\texttt{(proc y y)} \qquad \int_a^b y\, dy \qquad \sum_{y=1}^{n} y^2 \qquad \bigcup_{y\in A} y \qquad \forall y. f(y) = g(y)$$

2. Within the scope S of a variable I, the declaration of a new variable with the same name I creates a new scope S' in which the outer variable cannot be referenced. The region S' is called a **hole in the scope** of S. For example, any reference to x within the empty box ($\Box$) in the following examples would refer to the variable declared by the inner x, not the outer x.

$$\texttt{(proc x (call x (proc x }\Box\texttt{)))} \qquad \int_a^b x \cdot \left(\int_c^x \Box\, dx\right) dx \qquad \prod_{x=1}^{n} \left(\sum_{x=1}^{x} \Box\right)$$

$$\bigcup_{x\in A} \langle x, \bigcap_{x\in B} \Box \rangle \qquad \forall x. \left((f(x) = g(x)) \wedge \exists x. \Box\right)$$

6.3.3 Abstract Syntax DAGs and Stoy Diagrams

The chief structural feature of variables is that they permit sharing in an expression: the same variable introduced by a declaration can be used by many variable reference occurrences. We have said before that syntactic expressions can be viewed as abstract syntax trees, but since trees allow no sharing of substructure, they are inadequate for illustrating the sharing nature of variables.

We need the more general directed acyclic graph (DAG) to faithfully show the structure of an expression with variables.

As an example, consider the following FLK expression:[4]

```
(call (proc a (call a a)) (proc a (proc b a)))
```

In this expression, there are two distinct variables named `a`, and the variable named by `b` is declared without being referenced. Figure 6.11 shows an abstract syntax DAG corresponding to this expression. In the DAG, the three distinct variables in the expression are represented by distinct nodes labeled `variable`.

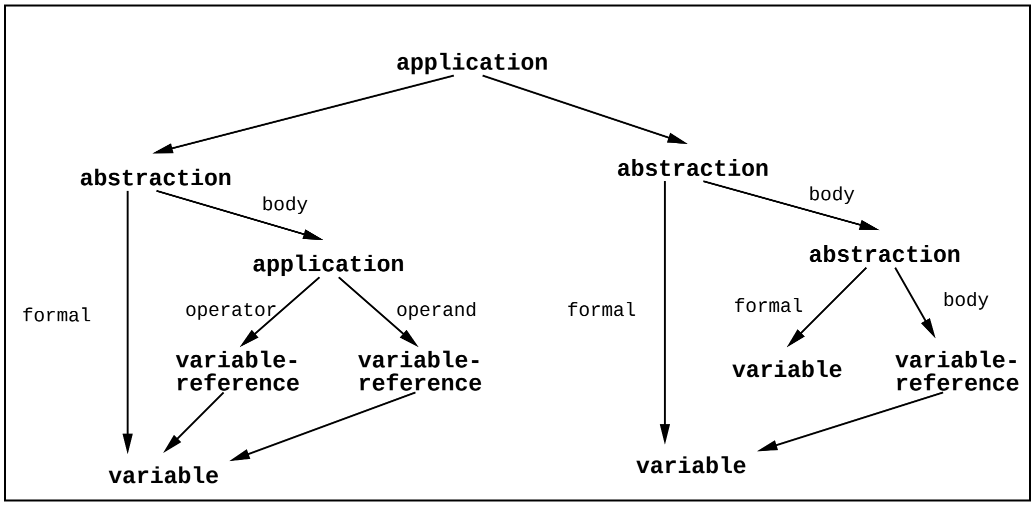


Figure 6.11: Abstract syntax DAG for `(call (proc a (call a a)) (proc a (proc b a)))`

Since sharing is explicit in the structure of the DAG, no identifiers are necessary in the DAG representation of the expression. The key reason variables are traditionally represented with identifiers is that they allow DAGs to be encoded within linear and tree-based notational frameworks. Unfortunately, encodings of DAGs based on identifiers complicate reasoning about expressions because of incidental properties of the identifiers. For example, the notion of a "hole in the scope" introduced earlier is not inherent in the nature of variables, but is a side effect of the fact that when variables are represented by identifiers, a nested pair of variables can accidentally share the same name. We'll see below that identifiers are the major sore spot when defining notions of renaming and substitution on FLK expressions.

[4]In the following discussion, we shall focus only on FLK expressions, but the same techniques could be applied to any notation using variables.

Every closed expression can always be represented by a DAG with no identifiers. However, expressions containing free variables pose a problem because they contain references to a variable without also containing its declaration. Since expressions with free variables are common, we'd like to handle them within the DAG framework. The DAG representation must include the names of any free identifiers because the names of free identifiers actually matter (for example, the expression `(proc b (call b a))` does not have the same meaning as `(proc b (call b c))` in every context). Figure 6.12 shows the DAG representation of `(proc b (call b a))`. The free variable is declared by a special free variable node annotated with the name of the variable.

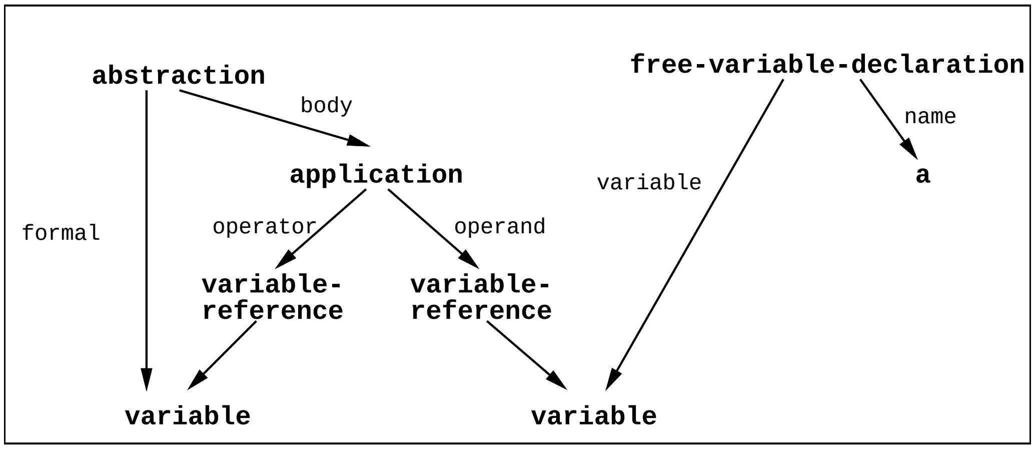
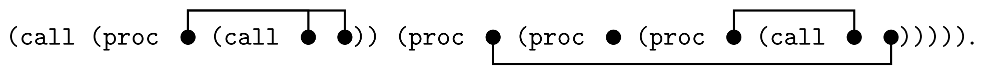


Figure 6.12: Abstract syntax DAG for `(proc b (call b a))`

Abstract syntax DAGs take up a lot of real estate on the printed page, so we shall use a more compact notation due to Joseph Stoy [Sto85]. Stoy's notation is a kind of wiring diagram for expressions in which the position corresponding to a variable reference is connected by a wire to the position corresponding to the variable declaration. For example, a **Stoy diagram** for the expression

```
(call (proc a (call a a)) (proc a (proc b (proc c (call c a)))))
```

is



```
(call (proc ● (call ● ●)) (proc ● (proc ● (proc ● (call ● ●))))).
```

We extend Stoy's notation to handle free variables by simply leaving every free variable reference where it occurs in the expression. Thus, the Stoy diagram for `(proc b (call a (call b a)))` is:

```
(proc ● (call a (call ● a)))
```

Observe that all identifiers sharing the same name in a Stoy diagram must name the same free variable.

6.3.4 Alpha-Equivalence

Since we really care about the implied DAG structure of an expression and not the vagaries of particular choices of identifiers for variable names, it is natural to equate FLK expressions that share the same DAG representation. We shall use the notation

$$E_1 =_\alpha E_2$$

(pronounced "E_1 is alpha-equivalent to E_2") to mean that E_1 and E_2 designate the same abstract syntax DAG. Thus,

$$\begin{array}{rcl} \texttt{(proc a (proc b (call b a)))} & =_\alpha & \texttt{(proc b (proc a (call a b)))} \\ & =_\alpha & \texttt{(proc one} \\ & & \quad\texttt{(proc two (call two one)))} \end{array}$$

and

$$\texttt{(proc b (call b a))} =_\alpha \texttt{(proc c (call c a))}$$

but

$$\texttt{(proc a (proc b (call b a)))} \neq_\alpha \texttt{(proc a (proc a (call a a)))}$$

and

$$\texttt{(proc b (call b a))} \neq_\alpha \texttt{(proc b (call b c))}$$

Since alpha-equivalence is an equivalence relation, it partitions FLK expressions into equivalence classes that share the same DAG. We shall generally assume throughout the rest of our discussion on FLK that each FLK expression serves as a representative of its equivalence class and that syntactic manipulations on expressions are functions on these equivalence classes rather than on individual expressions. For example, *FreeIds* is a well-defined function not only on FLK expressions but also on alpha-equivalence classes of FLK expressions because

$$E_1 =_\alpha E_2$$

implies

$$FreeIds[\![E_1]\!] = FreeIds[\![E_2]\!].$$

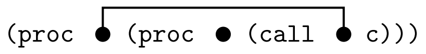
On the other hand, *BoundIds* is not a meaningful function on alpha-equivalence classes because it depends on syntactic details of an expression that are not represented in its DAG structure. Thus `(proc a a)` $=_\alpha$`(proc b b)`, but

$$BoundIds[\![\texttt{(proc a a)}]\!] = \{\texttt{a}\} \neq_\alpha \{\texttt{b}\} = BoundIds[\![\texttt{(proc b b)}]\!]\ .$$

6.3.5 Renaming and Variable Capture

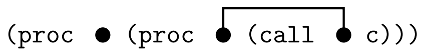
Equipped with a deeper understanding of the structure of variables, we're ready to consider the subtleties of renaming a variable introduced by an abstraction. A correct variable renaming is one that preserves the alpha-equivalence class of the expression — i.e., does not alter its abstract syntax DAG or Stoy diagram. The naïve approach of consistently renaming the declaration occurrence of the variable and all its references is not always appropriate because of a situation known as **variable capture**. There are two kinds of variable capture, both of which will be illustrated in the following example.

Consider the expression `(proc a (proc b (call a c)))`, whose Stoy diagram is shown below:



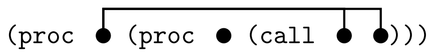
Suppose we want to rename the variable named `a` in this expression. For almost all possible identifiers, a simple consistent renaming will do. For example, renaming `a` to `x` produces the expression `(proc x (proc b (call x c)))` which has the same Stoy diagram as the original.

Suppose, however, that we choose the identifier `b` as the new name for `a`. Then the naïve renaming method yields `(proc b (proc b (call b c)))`, whose Stoy diagram,



is *not* the same as that for the original expression. The inner binding occurrence of `b` has created a hole in the scope of the outer binding occurrence of `b` in which the outer `b` cannot be seen. Because an inner abstraction just happens to bind the new name, all references to the new name within the body of the inner abstraction are accidentally captured by that abstraction. We shall refer to this situation as **internal variable capture**.

A slightly different problem is encountered if we choose `c` as the new name for `a`. In that case, naïve renaming yields `(proc c (proc b (call c c)))`, whose Stoy diagram is



The free identifier `c` has accidentally been captured by the declaration occurrence of the new name. Here the declaration of the new name has captured a free identifier in the body of the renamed abstraction; above, the internal abstraction captured a reference to the renamed variable. Since the captured variable is declared external to the renamed abstraction, we shall refer to this second situation as **external variable capture**.

Internal and external variable capture are not unique to FLK. They can occur in any naming system in which logically distinct variables can accidentally be identified. As we shall see later, variable capture commonly rears its ugly head in languages supporting dynamic scoping or macro expansion.

We would like it to be the case that such coincidental choices of identifiers in renamings do not destroy the structural integrity of an FLK expression. One way of doing this is to guarantee that each new variable name introduced by a renaming appears nowhere else in the FLK expression. However, this approach is overly restrictive and gives little insight into the true nature of the problem. Below, we shall precisely define a general syntactic renaming operator that avoids both forms of variable capture.

6.3.6 Substitution

Variable renaming is a special case of a more general syntactic operation on FLK expressions called **substitution**. It is often desirable to substitute a given expression for all free variable references of the variable named by a given identifier within another expression. For example, we might want to replace each free `a` within

```
(call a (proc b (call (proc a (call a b)) a)))
```

by the application `(call c d)` to yield

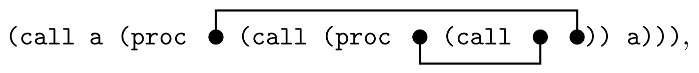
```
(call (call c d) (proc b (call (proc a (call a b)) (call c d))))
```

We use the notation $[E/I]$ to denote a function that maps a given expression into another expression in which E has been substituted for all free variable references named by I. Thus, $[E_1/I]E_2$ denotes the result of substituting E_1 for the free occurrences of I in E_2. Using this notation, the above example can be expressed as:

```
[(call c d)/a](call a (proc b (call (proc a (call a b)) a)))
= (call (call c d) (proc b (call (proc a (call a b)) (call c d))))
```

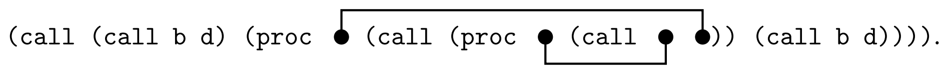
A correct substitution is one which preserves the logical structure both of the expression being substituted (E_1) and the expression substituted into (E_2) — except, of course, for the free variable being substituted for. Although substitution might seem like a straightforward idea, it is plagued with variable capture subtleties similar to those that lurk in renaming. In fact, several well-known logicians gave incorrect definitions for substitution before a correct one was found.

As an example of a problematic situation, suppose that `(call b d)` rather than `(call c d)` were being substituted for `a` in the above example. Since the expression being substituted into has the Stoy diagram



```
(call a (proc ● (call (proc ● (call ● ●)) a))),
```

[`(call b d)`/`a`]`(call a (proc b (call (proc a (call a b)) a)))` should have the Stoy diagram

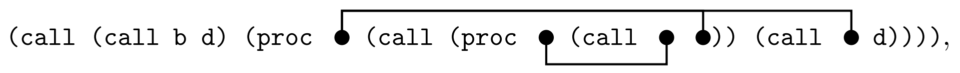


```
(call (call b d) (proc ● (call (proc ● (call ● ●)) (call b d)))).
```

However, a naïve syntactic approach to substitution would yield the expression

```
(call (call b d) (proc b (call (proc a (call a b)) (call b d)))),
```

whose Stoy diagram,



```
(call (call b d) (proc ● (call (proc ● (call ● ●)) (call ● d)))),
```

shows that variable capture violates the integrity of the free variable `b` within the second occurrence of `(call b d)`.

Figure 6.13 presents a method of substitution that avoids variable capture. Substitution is defined by structural induction on the expression substituted into. However, there is sometimes more than one clause per expression type because some expression types have subcases that depend on interactions between the variable I_{subst} being replaced and variables within the expression substituted into. For example, $[E/I_{subst}]I_{exp}$ is E if I_{subst} and I_{exp} are syntactically identical, but is the original expression I_{exp} if I_{subst} and I_{exp} are not the same. These different subcases are expressed in Figure 6.13 by implicit pattern matching or explicit restrictions.

As seen in Figure 6.13, most of the rules straightforwardly distribute the substitution over the subexpressions of an expression. The tricky case is substituting into a variable declaration construct (`proc` or `rec`). For example, consider the case for `proc`:

$$[E_{new}/I_{subst}]\texttt{(proc } I_{bound}\ E_{body}\texttt{)},$$

In the case where I_{subst} and I_{bound} are the same, no substitutions can be permitted inside the abstraction because I_{bound} declares a variable that is distinct from the one named by I_{subst}. Without this restriction, we could derive results like

$$[\texttt{b}/\texttt{a}]\texttt{(proc a (call a b))} = \texttt{(proc b (call b b))}$$

in which external variable capture invalidates the purported substitution.

When I_{subst} and I_{bound} are distinct, the crucial situation to handle is where I_{subst} appears free in E_{body} (so a substitution will definitely take place) *and*

$$
\begin{array}{rcl}
[E_{new}/I_{sub}]L & = & L \\
[E_{new}/I_{sub}]I_{sub} & = & E_{new} \\
[E_{new}/I_{sub}]I_{expr} & = & I_{expr},\ \text{where } I_{sub} \neq I_{expr} \\
[E_{new}/I_{sub}]\texttt{(primop}\ O & = & \texttt{(primop}\ O\ [E_{new}/I_{sub}]E_1\ \ldots\ [E_{new}/I_{sub}]E_n\texttt{)} \\
E_1\ \ldots\ E_n\texttt{)} & & \\
[E_{new}/I_{sub}]\texttt{(proc}\ I_{sub}\ E_{body}\texttt{)} & = & \texttt{(proc}\ I_{sub}\ E_{body}\texttt{)} \\
[E_{new}/I_{sub}]\texttt{(proc}\ I\ E_{body}\texttt{)} & = & \texttt{(proc}\ I_{fresh}\ [E_{new}/I_{sub}]([I_{fresh}/I]E_{body})\texttt{)}\ , \\
 & & \quad \text{where } I_{sub} \neq I \text{ and} \\
 & & \quad I_{fresh} \notin \{I_{sub}\} \cup \mathit{FreeIds}[\![E_{new}]\!] \\
 & & \qquad\qquad \cup \mathit{FreeIds}[\![E_{body}]\!] \\
[E_{new}/I_{sub}]\texttt{(call}\ E_{rator}\ E_{rand}\texttt{)} & = & \texttt{(call}\ [E_{new}/I_{sub}]E_{rator}\ [E_{new}/I_{sub}]E_{rand}\texttt{)} \\
[E_{new}/I_{sub}]\texttt{(if}\ E_1\ E_2\ E_3\texttt{)} & = & \texttt{(if}\ [E_{new}/I_{sub}]E_1 \\
 & & \qquad [E_{new}/I_{sub}]E_2 \\
 & & \qquad [E_{new}/I_{sub}]E_3\texttt{)} \\
[E_{new}/I_{sub}]\texttt{(pair}\ E_1\ E_2\texttt{)} & = & \texttt{(pair}\ [E_{new}/I_{sub}]E_1\ [E_{new}/I_{sub}]E_2\texttt{)} \\
[E_{new}/I_{sub}]\texttt{(rec}\ I_{sub}\ E_{body}\texttt{)} & = & \texttt{(rec}\ I_{sub}\ E_{body}\texttt{)} \\
[E_{new}/I_{sub}]\texttt{(rec}\ I\ E_{body}\texttt{)} & = & \texttt{(rec}\ I_{fresh}\ [E_{new}/I_{sub}]([I_{fresh}/I]E_{body})\texttt{)}\ , \\
 & & \quad \text{where } I_{sub} \neq I \text{ and} \\
 & & \quad I_{fresh} \notin \{I_{sub}\} \cup \mathit{FreeIds}[\![E_{new}]\!] \\
 & & \qquad\qquad \cup \mathit{FreeIds}[\![E_{body}]\!]
\end{array}
$$

Figure 6.13: The definition of substitution.

E_{new} contains a free reference to I_{bound}. This reference will be captured by the bound variable of the abstraction unless we're careful. A simple example of this situation is:

[b/a](proc b (call b a)).

Here, the substituted expression b contains (in fact, is) a free reference to a variable whose name happens to be the same as the name of the variable bound by the abstraction. A naïve substitution would yield (proc b (call b b)), in which the outer variable named b has been accidentally captured by the inner variable of the same name. To prevent this internal variable capture, it is necessary to first consistently rename the bound variable of the abstraction with an identifier that is not the same as I_{subst} and is free neither in E_{new} nor in E_{body}. After this renaming, substitution can be performed on E_{body} without threat of variable capture. In our example, the bound variable b can be renamed to c, say, yielding the alpha-equivalent abstraction (proc c (call c a)). Then substitution can be performed on the body to yield the correct expression

(proc c [b/a](call c a)) = (proc c (call c b)).

In the case where $I_{subst} \neq I_{bound}$, it is always correct to perform the described renaming of the bound variable of the abstraction, but it is not always necessary. If I_{subst} is not free in E_{body}, renaming is not required because no substitution will be performed inside the abstraction anyway. And if I_{bound} doesn't appear in E_{new}, no internal variable capture can arise, and it is safe to directly substitute into the body of the abstraction without a renaming step.

In the rule for substituting into an abstraction, it is necessary to choose an identifier that is not the same as I_{subst} and is free neither in E_{new} nor in E_{body}. The notion of choosing an identifier that satisfies certain properties often arises when manipulating syntactic expressions in which variables are represented by identifiers. Such an identifier is said to be **fresh**. When describing a syntactic manipulation, it is always necessary to specify any constraints involved in choosing the fresh identifiers.

Keep in mind that all the complexity for renaming and substitution arises from dealing with linear (in this case, textual) representations for declaration/reference relationships that are not linear or even tree-like. If FLK expressions were represented instead as DAGs or Stoy diagrams, renaming would be unnecessary and substitution would be straightforward.

▷ **Exercise 6.11** Use the definition of substitution in Figure 6.13 to determine the results of the following substitutions. Assume that fresh identifiers are taken from the list v_1, v_2, v_3, ..., and that the first identifier from the list that satisfies the given constraint is chosen as the fresh identifier.

a. [(call (call b c) d)/a](proc a (proc b (call (call c b) a)))

b. [(call (call b c) d)/b](proc a (proc b (call (call c b) a)))

c. [`(call (call b c) d)`/c]`(proc a (proc b (call (call c b) a)))`

d. [`(call (call b c) d)`/d]`(proc a (proc b (call (call c b) a)))`

e. [`(call (call b c) d)`/b]`(proc a (proc b (call c a)))` ◁

▷ **Exercise 6.12** Consider the case for substituting into `proc` abstractions,

$$[E_{new}/I_{subst}](\texttt{proc}\ I_{bound}\ E_{body}),$$

where $I_{subst} \neq I_{bound}$. Here I_{bound} is consistently renamed to be a variable I_{fresh} that is not free in either E_{new} or E_{body} and is not equal to I_{subst}.

a. Provide an example of an incorrect substitution that would be permitted if the restriction $I_{fresh} \notin \mathit{FreeIds}[\![E_{new}]\!]$ were lifted.

b. Provide an example of an incorrect substitution that would be permitted if the restriction $I_{fresh} \notin \mathit{FreeIds}[\![E_{body}]\!]$ were lifted.

c. Provide an example of an incorrect substitution that would be permitted if the restriction $I_{fresh} \neq I_{subst}$ were lifted.

d. Would it be possible to consistently rename the free variables of E_{new} (within both E_{new} and E_{body}) instead of renaming I_{bound}? Explain your answer, using examples where appropriate. ◁

▷ **Exercise 6.13** Assuming that I_1 and I_2 are distinct, and that $I_2 \notin \mathit{FreeIds}[\![E_1]\!]$, prove the following useful equivalence:

$$[E_1/I_1]([E_2/I_2]E_3) = [([E_1/I_1]E_2)/I_2]([E_1/I_1]E_3)$$

(Hint: Do the proof by induction on the height of E_3.) ◁

▷ **Exercise 6.14** The notion of **simultaneous substitution** is an extension to the substitution function we have seen. A simultaneous substitution, $[E_1 \ldots E_n/I_1 \ldots I_n]$, is a function of a single expression that performs the substitutions $[E_1/I_1] \ldots [E_n/I_n]$ in parallel on that expression. It differs from a sequence of substitutions in that an I_i appearing in one of the E_j is never substituted for. For example, simultaneous substitution of I_2 for I_1 and I_1 for I_2 in the expression $(\texttt{call}\ I_1\ I_2)$ swaps the two identifiers:

$$[I_2, I_1/I_1, I_2](\texttt{call}\ I_1\ I_2) = (\texttt{call}\ I_2\ I_1)$$

whereas neither ordering of two single substitutions has this behavior:

$$[I_2/I_1]([I_1/I_2](\texttt{call}\ I_1\ I_2)) = (\texttt{call}\ I_2\ I_2)$$

$$[I_1/I_2]([I_2/I_1](\texttt{call}\ I_1\ I_2)) = (\texttt{call}\ I_1\ I_1)$$

Write a formal definition of simultaneous substitution for FLK. ◁

▷ **Exercise 6.15** Suppose that FL is extended with the following constructs for manipulating tuples of elements:

`(tuple` E*`)`:	Non-strict constructor of a tuple with any number of elements.
`(tuple-ref` E i`)`:	Suppose i is a positive integer i and E is a tuple t. Return the ith element of t (assume 1-based indexing).
`(tuple?` E`)`:	Predicate determining if E is a tuple.
`(tuple-length` E`)`:	Returns the number of elements in the tuple.

Tuples provide an alternate way to desugar multi-abstractions and multi-applications. Multi-applications can package arguments into a tuple that is unpackaged by a multi-abstraction.

a. Provide tuple-based desugarings for multi-abstractions and multi-applications. You may find substitution helpful. Explain any design choices that you make.

b. Discuss the advantages and disadvantages of the tuple-based desugaring versus the desugaring based on currying. ◁

6.4 An Operational Semantics for FLK

6.4.1 An SOS for FLK

Figure 6.14 presents an SOS for FLK. In addition to the semantic domains of FLK, the SOS uses the following domains:

- The ValueExp domain is a subset of Exp$_{FLK}$ consisting of expressions that model the values manipulated by FLK programs. The notations {`(symbol` I`)`}, {`(proc` I E`)`}, {`(pair` E_1 E_2`)`}, and {`(error` I_{msg}`)`} indicate the set of all expressions that match the given pattern. The value expressions include all the literals, as well as abstractions (representing procedural values), pairings (representing pair values), and error expressions.

- Each input to an FLK program is an s-expression value from the SExpVal domain. This is a subset of ValueExp that excludes all `proc` and `error` forms.

- The Answer domain models final answers in the execution of FLK programs. It is similar to ValueExp except that it replaces all `proc` expressions by the procedure value token `procval` and replaces all `pair` expressions

Domains

$$\begin{array}{rcllll}
V & \in & \text{ValueExp} & = & \{\texttt{\#u}\} \cup \text{Boollit} \cup \text{Intlit} \cup \{\texttt{(symbol } I\texttt{)}\} \\
 & & & & \cup\{\texttt{(proc } I\ E\texttt{)}\} \cup\{\texttt{(pair } E_1\ E_2\texttt{)}\} \cup\{\texttt{(error } I_{msg}\texttt{)}\} \\
SV & \in & \text{SExpVal} & = & \{\texttt{\#u}\} \cup \text{Boollit} \cup \text{Intlit} \cup \{\texttt{(symbol } I\texttt{)}\} \\
 & & & & \{\texttt{(pair } SV_1\ SV_2\texttt{)}\} \\
I & \in & \text{Inputs} & = & \text{SExpVal*} \\
A & \in & \text{Answer} & = & \{\texttt{\#u}\} \cup \text{Boollit} \cup \text{Intlit} \cup \{\texttt{(symbol } I\texttt{)}\} \\
 & & & & \cup\{\texttt{procval}\} \cup\{\texttt{pairval}\} \cup\{\texttt{(error } I_{msg}\texttt{)}\}
\end{array}$$

SOS

The FLK SOS has the form $FLKSOS = \langle \text{Exp}_{FLK}, \Rightarrow, \text{ValueExp}, IF, OF\rangle$, where:

$\Rightarrow$ is a deterministic transition relation defined in Figures 6.15 and 6.16.

$IF : \text{Program}_{FLK} \times \text{Inputs} \to \text{Exp}_{FLK}$
$= \lambda\langle \texttt{(flk (}I_1 \ \ldots\ I_n\texttt{)}\ E_{body}\texttt{)}, [SV_1, \ldots, SV_k]\rangle\ .$
if n = k **then** $([SV_i/A_i]_{i=1}^{n})E_{body}$
else `(error wrong-number-of-args)` **fi**

$OF : \text{ValueExp} \to \text{Answer}$
$= \lambda V\ .$ **matching** V
$\triangleright$ `(proc` $I\ E$`)` $\|$ `procval`
$\triangleright$ `(pair` $E_1\ E_2$`)` $\|$ `pairval`
$\triangleright$ **else** V **endmatching**

Figure 6.14: An SOS for FLK.

by the pair value token `pairval`. These tokens distinguish the types of procedure and pair values, but the structure of these values is not observable.

The configuration space for the FLK SOS consists of FLK expressions. The input function *IF* maps an FLK program and a sequence of s-expression argument values to an initial configuration by substituting the arguments for the formal parameter names in the body of the program. The final configurations of the SOS are modeled by the ValueExp domain. The output function *OF* erases the details of all procedure and pair values.

The SOS rewrite relation $\Rightarrow$ is defined by the rewrite rules in Figures 6.15 and 6.16. Applications are handled by the [*call-apply*] and [*call-operator*] rules. The [*call-apply*] rule makes use of the FLK substitution operator to evaluate the application of an abstraction. The [*call-operator*] progress rule permits rewrites on the operator. No rewrites are performed on the operand so these rules are non-strict, like `if` and unlike `primop`.

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

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

$$\texttt{(if \#t}\ E_1\ E_2\texttt{)} \Rightarrow E_1 \qquad [\textit{if-true}]$$

$$\texttt{(if \#f}\ E_1\ E_2\texttt{)} \Rightarrow E_2 \qquad [\textit{if-false}]$$

$$\frac{E_1 \Rightarrow E_1{}'}{\texttt{(if}\ E_1\ E_2\ E_3\texttt{)} \Rightarrow \texttt{(if}\ E_1{}'\ E_2\ E_3\texttt{)}} \qquad [\textit{if-test}]$$

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

$$\frac{E \Rightarrow E'}{\texttt{(primop}\ O\ E\texttt{)} \Rightarrow \texttt{(primop}\ O\ E'\texttt{)}} \qquad [\textit{unary-arg}]$$

$$\frac{E_1 \Rightarrow E_1{}'}{\texttt{(primop}\ O\ E_1\ E_2\texttt{)} \Rightarrow \texttt{(primop}\ O\ E_1{}'\ E_2\texttt{)}} \qquad [\textit{binary-arg-1}]$$

$$\frac{E_2 \Rightarrow E_2{}'}{\texttt{(primop}\ O\ V\ E_2\texttt{)} \Rightarrow \texttt{(primop}\ O\ V\ E_2{}'\texttt{)}} \qquad [\textit{binary-arg-2}]$$

Figure 6.15: FLK rewrite rules, part 1.

Strict languages would include progress rules for operands of procedure calls (as FLK does for primitives), and these rules would reflect constraints on evaluation order. However, even strict languages have non-strict conditionals to avoid errors (e.g., division by zero) and infinite loops (the base case of a recursion, such as the factorial of 0).

The semantics of recursion is especially simple in the SOS framework. It is obtained by simply "unwinding" the recursion equation one level. Programmers often follow the same approach when trying to hand-simulate the behavior of recursive procedures.

The three progress rules [*unary-arg*], [*binary-arg-1*], and [*binary-arg-2*] suffice for forcing the evaluation of arguments in a primitive application. The metavariable V in rule [*binary-arg-2*] is used to express a constraint that the first operand must be a value; thus the first argument must be fully evaluated before the second argument is evaluated. These three rules are actually instantiations of a single general rule to evaluate any number of arguments in left-to-right order:

$$\frac{E_i \Rightarrow E_i{}'}{\begin{array}{l}\texttt{(primop } O\ V_1\ \ldots V_{i-1}\ E_i\ \ldots E_n\texttt{)}\\ \Rightarrow\texttt{(primop } O\ V_1\ \ldots V_{i-1}\ E_i{}'\ \ldots E_n\texttt{)}\end{array}} \qquad [\textit{prim-arg}]$$

where n can be any nonnegative integer (including 0) and i ranges between 0 and n. The notation is intended to indicate that the first $i-1$ arguments have all been fully evaluated, and the ith expression is in the process of evaluation.

A sampling of the remaining primitive operator rules are given in Figure 6.16. These rules define the behavior of each primitive operator. The *calculate* function used in the [+] rule serves the same purpose as it did in the PostFix SOS.

Like the PostFix SOS, the FLK SOS models most errors with stuck states. If the final configuration happens to be an `error` form, then this will be returned as the outcome of the program. But if a configuration is stuck because it contains a problematic subexpression such as `(primop + 1 #t)` or an `error` form, the outcome of the program will be `stuck`. See Exercise 6.21 for an alternative approach to handle errors in FLK.

6.4.2 Example

Figure 6.17 illustrates a sample proof-structured evaluation of the expression

```
(call (call (proc f (call f (primop + 4 1)))
            (proc a (proc b (primop - b a))))
      3)
```

- `not`: $\texttt{(primop not? \#f)} \Rightarrow \texttt{\#t}$ [*not-1*]

  $\texttt{(primop not? \#t)} \Rightarrow \texttt{\#f}$ [*not-2*]

- `left` and `right`: $\texttt{(primop left (pair } E_1\ E_2\texttt{))} \Rightarrow E_1$ [*left*]

  $\texttt{(primop right (pair } E_1\ E_2\texttt{))} \Rightarrow E_2$ [*right*]

- `integer?` (other predicates are defined similarly):

  $\texttt{(primop integer? } N\texttt{)} \Rightarrow \texttt{\#t}$ [*integer?-integer*]

  $\texttt{(primop integer? \#u)} \Rightarrow \texttt{\#f}$ [*integer?-unit*]

  $\texttt{(primop integer? } B\texttt{)} \Rightarrow \texttt{\#f}$ [*integer?-boolean*]

  $\texttt{(primop integer? (symbol } I\texttt{))} \Rightarrow \texttt{\#f}$ [*integer?-symbol*]

  $\texttt{(primop integer? (proc } I\ E\texttt{))} \Rightarrow \texttt{\#f}$ [*integer?-abstraction*]

  $\texttt{(primop integer? (pair } E_1\ E_2\texttt{))} \Rightarrow \texttt{\#f}$ [*integer?-pair*]

- `and?` (`or?` is defined similarly):

  $\texttt{(primop and? \#t \#t)} \Rightarrow \texttt{\#t}$ [*and-true-true*]

  $\texttt{(primop and? \#t \#f)} \Rightarrow \texttt{\#f}$ [*and-true-false*]

  $\texttt{(primop and? \#f \#t)} \Rightarrow \texttt{\#f}$ [*and-false-true*]

  $\texttt{(primop and? \#f \#f)} \Rightarrow \texttt{\#f}$ [*and-false-false*]

- `+` (other binary operators are similar, except for `/` and `rem`):

  $\texttt{(primop + } N_1\ N_2\texttt{)} \Rightarrow (\textit{calculate}\ \texttt{+}\ N_2\ N_1)$ [*+*]

- `/` (`rem` is similar):

  $\texttt{(primop / } N_1\ N_2\texttt{)} \Rightarrow (\textit{calculate}\ \texttt{/}\ N_2\ N_1)$, where $N_2 \neq \texttt{0}$ [*/*]

Figure 6.16: FLK rewrite rules, part 2.

based on the above rewriting rules. Each rewriting step is annotated with a justification that explains how the step follows from previous steps and a rewrite rule.

A more condensed form of the evaluation in Figure 6.17 treats as a single rewrite any axiom rewrite in conjunction with any number of rewrites implied by progress rules. This gives rise to a linear sequence of rewrites, where the rewrite arrow can be subscripted with the name of the axiom applied. The example from the figure then becomes:

```
(call (call (proc f (call f (primop + 4 1)))
            (proc a (proc b (primop - b a))))
      3)
⇒[call-apply] (call (call (proc a (proc b (primop - b a)))
                          (primop + 4 1))
                    3)
⇒[call-apply] (call (proc b (primop - b (primop + 4 1)))
                    3)
⇒[call-apply] (primop - 3 (primop + 4 1))
⇒[+] (primop - 3 5)
⇒[−] -2
```

▷ **Exercise 6.16** Use the rewrite rules to show the evaluation of the following expressions:

a. `(primop left (pair 1 (primop not? 3)))`

b.
```
(primop left (primop right (primop right
                                      (rec p (pair 1 (pair 2 p))))))
```

The first expression illustrates the non-strictness of `pair` while the second illustrates the unwinding nature of `rec`. ◁

▷ **Exercise 6.17** Since FLK is non-strict, it is not necessary for `if` to be a distinguished construct. Instead, `if` could be a unary primitive operator that returns a (curried) binary function. That is, instead of being written `(if` E_1 E_2 E_3`)`, conditionals could be expressed as

`(call (call (primop if` E_1`)` E_2`)` E_3`)`

Give the rewrite rules for `if` as a unary primitive operator. ◁

▷ **Exercise 6.18** Functional computation in a dynamically typed language can be viewed as a bureaucracy where envelopes (values containing a type and other information) are shuffled around by the interpreting agent that performs the computation.[5] In many steps of the computation, envelopes are simply moved around without being

[5]Phil Agre introduced us to this point of view.

```
(call (proc f (call f (primop + 4 1)))
      (proc a (proc b (primop - b a))))
⇒ (call (proc a (proc b (primop - b a)))                    1: call-apply
        (primop + 4 1))

(call (call (proc f (call f (primop + 4 1)))
            (proc a (proc b (primop - b a))))
      3)
⇒ (call (call (proc a (proc b (primop - b a)))        2: 1 & call-operator
              (primop + 4 1))
          3)

(call (proc a (proc b (primop - b a)))
      (primop + 4 1))
⇒ (proc b (primop - b (primop + 4 1)))                      3: call-apply

(call (call (proc a (proc b (primop - b a)))
            (primop + 4 1))
      3)
⇒ (call (proc b (primop - b (primop + 4 1)))          4: 3 & call-operator
          3)

(call (proc b (primop - b (primop + 4 1)))
      3)
⇒ (primop - 3 (primop + 4 1))                               5: call-apply

(primop + 4 1) ⇒ 5                                                 6: +

(primop - 3 (primop + 4 1)) ⇒ (primop - 3 5)            7: binary-arg-2

(primop - 3 5) ⇒ -2                                                8: +

(call (call (proc f (call f (primop + 4 1)))
            (proc a (proc b (primop - b a))))
      3)
*
⇒ -2                                                 9: 2 & 4 & 5 & 7 & 8
```

Figure 6.17: Example evaluation of an FLK expression.

opened. In the formation of a non-strict pair, for instance, two envelopes are simply stuffed into a larger envelope without ever having their contents examined. During other stages — a primitive addition, for instance — the contents (type and content information) of envelopes must definitely be examined.

With this perspective in mind, for each FLK expression describe when the contents of envelopes must be examined. In other words, which contexts demand the value of an expression? ◁

▷ **Exercise 6.19** Suppose we want to extend FL with a `least` construct. Given a numeric predicate, `least` returns the least non-negative integer that satisfies the predicate. For example,

`(least (proc (x) (= x (* x x))))` $\xrightarrow[FL]{}$ *0*
`(least (proc (a) (> (* a a) 10)))` $\xrightarrow[FL]{}$ *4*
`(least (proc (x) (< x 0)))` $\xrightarrow[FL]{}$ *∞-loop* {*Looks, but no solution*}
`(least (proc (x) x))` $\xrightarrow[FL]{}$ *error:Non-bool-in-if-test*

a. Must the argument to `least` always be an abstraction? If so, explain why; if not, give a counterexample.

b. One way to add `least` is to extend the syntax of FLK to include `(least` *E*`)` as a new expression type. Extend the operational semantics of FLK to handle the `least` expression. Keep in mind that a SOS has five parts; make the appropriate modifications to each of the parts.

   Hint: In addition to adding `(least` *E*`)` to the configuration space, it is also desirable to add a configuration of the form `(*least*` *E N*`)`. Configurations like `*least*` that are not valid as expressions in the language are often useful for representing intermediate states of computations.

c. Alternately, `least` could be written as a user-defined procedure that is standardly available in the body of a `program`. Show how to implement `least` with this approach. ◁

▷ **Exercise 6.20** In FLK, `pair` is a primitive construct built into the syntax. In a non-strict language, though, there is no need for `pair` to be primitive.

a. One option is to include `pair` as a primitive primop operator. Implement this change by modifying the operational semantics of FLK.

b. Is it possible to define `pair` as a user-defined procedure? How would you implement `left`, `right`, and `pair?` ? ◁

▷ **Exercise 6.21** Like the PostFix SOS, the FL SOS uses stuck states to model errors. For example, all of the following stuck states correspond to error situations:

```
a                               ; Unbound variable
(primop / 1 0)                  ; Division by 0
(primop + 1 #t)                 ; Inappropriate argument type
(primop + 1 2 3)                ; Inappropriate number of arguments
(call 1 2)                      ; Attempt to apply a non-procedure
(if (symbol nonbool) 2 3)       ; Non-boolean test in an IF.
```

Rather than using stuck states to model errors, we can use the fact that ValueExp includes the form `(error` I_{msg}`)` to explicit represent and propagate errors. For this approach, the rewrite rules need to (1) convert stuck expressions to an appropriate `error` form and (2) propagate `error` forms so that they eventually become final configurations. For example, we could have the rule

$$(\texttt{call}\ N\ E) \Rightarrow (\texttt{error non-procedural-rator}) \qquad [\textit{integer-operator-error}]$$

to express the fact that it is an error to use an integer in the operator position of an application.

Make all necessary modifications and additions to the FLK rewrite rules in order to handle the explicit introduction and propagation of `error` forms. Make sure that errors propagate appropriately; e.g.,

```
(primop + 1 (primop / 1 0))
```

should rewrite to an error because it has a subexpression that rewrites to an error. ◁

▷ **Exercise 6.22** After carefully studying the SOS for FLK, Paula Morwicz proclaims that it is safe to use a naive substitution strategy (i.e., one that does not rename bound variables) in the [*call-apply*] and [*rec*] rules *as long as* the original expression being evaluated does not contain any unbound variables (i.e., free identifiers).

a. Show that Paula is right. That is, show that the name capture problems addressed by the definition of substitution in Figure 6.13 cannot occur during the evaluation of an FLK expression that has no unbound variables.

b. Give an example of an FLK expression containing an unbound variable that evaluates to the wrong answer if the the naive substitution strategy is used.

c. Suppose that every FLK expression were alpha-renamed so that all variables had distinct identifiers and no bound variable used the same identifier as any unbound variable. Under these conditions, is it always safe to use the naive substitution strategy? If so, explain; if not, give a counter-example. ◁

▷ **Exercise 6.23** After reading up on the the lambda calculus, Sam Antix decides to experiment with some new rewrite rules for the FL SOS.

a. The first rule he tries is the so-called **eta rule**:

$$\begin{array}{l}(\texttt{proc}\ I\ (E\ I)) \Rightarrow E, \\ \text{where}\ \ I \notin \textit{FreeIds}[\![E]\!]\end{array} \qquad [\textit{eta}]$$

Although this rule is reasonable in the lambda calculus, it greatly changes the semantics of FLK. Demonstrate this fact by showing a FLK expression that can evaluate to two different values along two different transition paths.

b. The eta rule can be made safe by restricting the form of E. Describe such a restriction, and explain why the rule is safe.

c. After getting rid of the [*eta*] rule, Sam experiments with a rule that allows rewrites within the body of an abstraction:

$$\frac{E \Rightarrow E'}{\texttt{(proc } I\ E\texttt{)} \Rightarrow \texttt{(proc } I\ E'\texttt{)}} \qquad [\textit{proc-body}]$$

How does the addition of this rule change the semantics of FLK? For example, does it make it possible for an expression to rewrite to two different values via two different transition paths? Does it enable new kinds of transition paths? ◁

6.5 A Denotational Definition for FLK

In this section, we develop a denotational semantics for FLK. A complete denotational semantics for FLK appears in Figures 6.18–6.22. The semantic algebras for this semantics appear in Figure 6.18, and Figures 6.19 and 6.20 define auxiliary functions and values. These definitions provide the landscape that serves as the backdrop for our future denotational definitions, as well as for the valuation functions in Figures 6.21 and 6.22.

It is always best to begin a study of a denotational semantics with a careful look at the semantic algebras. Here is what we can see by looking at the FLK semantic algebras in Figure 6.18.

The values that can be expressed by an FLK expression are modeled by the *Expressible* domain, which is a lifted sum of *Value* and *Error*. Errors, like symbols, are modeled as identifiers.[6] *Value* contains unit, boolean, integer, and symbol values, as well as pair and procedure values, which are defined recursively in terms of *Expressible*. The bottom element of the *Expressible* domain represents a non-terminating computation in FLK.

Whereas the SOS for FLK used substitution to model naming, the denotational semantics uses **environments** as a kind of virtual substitution. When a value is bound to an identifier, that binding is stored in the environment used to evaluate expressions within the scope of that binding. Identifiers that represent

[6]We are being a little loose here. Program identifiers often exclude language key words, like `let`. Such restrictions should not be applied to program data or errors.

c	$\in$	*Computation*	$=$	*Expressible*
δ	$\in$	*Denotable*	$=$	*Computation*
p	$\in$	*Procedure*	$=$	*Denotable* $\rightarrow$ *Computation*
β	$\in$	*Binding*	$=$	$(\mathit{Denotable} + \mathit{Unbound})_{\perp}$
e	$\in$	*Environment*	$=$	Identifier $\rightarrow$ *Binding*
		Unbound	$=$	$\{\mathit{unbound}\}$
x	$\in$	*Expressible*	$=$	$(\mathit{Value} + \mathit{Error})_{\perp}$
v	$\in$	*Value*	$=$	*Unit* + *Bool* + *Int* + *Sym* + *Pair* + *Procedure*
		Unit	$=$	$\{\mathit{unit}\}$
i	$\in$	*Int*	$=$	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$
b	$\in$	*Bool*	$=$	$\{\mathit{true}, \mathit{false}\}$
y	$\in$	*Sym*	$=$	Identifier
a	$\in$	*Pair*	$=$	*Computation* $\times$ *Computation*
		Error	$=$	Identifier

Figure 6.18: The semantic algebras for FLK.

variable references are looked up in the current environment. Environments map identifiers to bindings, where *Binding* is a lifted sum of denotable values and the trivial domain *Unbound*. The trivial element acts as an "unbound marker" that indicates that an identifier is not bound in an environment.

The environment functions (Figure 6.19) have been updated to be consistent with the *Binding* domain. In particular, there is now a distinction between *extend*, which associates a name with a denotable in an environment, and *bind*, which associates a name with a binding in an environment. The figure introduces shorthand notation for these functions that will be used in future valuation clauses.

There is no a priori reason why the class of entities that can be named in an environment has to be the same as that denoted by arbitrary expressions. For this reason, there is a separate semantic domain, *Denotable*, for the set of values that can be associated with names in environments. There are many possible relationships between *Denotable* and *Expressible*:

- *Denotable* may be the same as *Expressible*. This is the case in FL.

- *Denotable* may be a superset of *Expressible* — some entities may be named but not computed. For example, languages in which procedures are not first-class typically have ways to name procedures (usually via a declaration) even though procedures cannot be values of expressions.

- *Denotable* may be a subset of *Expressible* — some entities may be computed, but not named. For example, in certain languages variables cannot

Environment operations:

$empty\text{-}env : Environment$
$\quad = \lambda I \,.\, (\mathit{Unbound} \mapsto \mathit{Binding}\ \text{unbound})$

$lookup : Environment \to \text{Identifier} \to Binding$
$\quad = \lambda e I \,.\, (e\ I)$

$bind : Environment \to \text{Identifier} \to Binding \to Environment$
$\quad = \lambda e I_1 \beta \,.\, \lambda I_2 \,.\, \textbf{if}\ (same\text{-}identifier?\ I_1\ \ I_2)\ \textbf{then}\ \beta\ \textbf{else}\ (lookup\ \ e\ \ I_2)\ \textbf{fi}$
$(bind\ \ e\ \ I\ \ \beta)$ will be abbreviated $[I :: \beta]e$; this notation associates to the right:
$\quad [I_2 :: \beta_2][I_1 :: \beta_1]e\ = [I_2 :: \beta_2]([I_1 :: \beta_1]e)$

$extend : Environment \to \text{Identifier} \to Denotable \to Environment$
$\quad = \lambda e I \delta \,.\, (bind\ \ e\ \ I\ \ (\mathit{Denotable} \mapsto \mathit{Binding}\ \delta))$
$(extend\ \ e\ \ I\ \ \delta)$ will be abbreviated $[I : \delta]e$; this notation associates to the right:
$\quad [I_2 : \delta_2][I_1 : \delta_1]e\ = [I_2 : \delta_2]([I_1 : \delta_1]e)$

Figure 6.19: Auxiliary functions and values for FLK, Part II.

name values that represent errors and nontermination. We shall study this example in detail when we discuss call-by-value semantics in Chapter 7.

- The relationship between *Denotable* and *Expressible* may be more complex. Consider a language in which procedures are denotable but not expressible, and errors are expressible but not denotable. (FORTRAN is in this category.)

Thus, the definitions of *Denotable* and *Expressible* in the denotational semantics of a given language contain some important information about high-level features of the language. The availability of this kind of information is the reason why, when reading a denotational semantics, it is advisable to first carefully study domain equations and function signatures before delving into the details of the valuation functions.

The meaning of an expression with respect to an environment depends on the formulation of the meaning function used. To provide a level of abstraction, we will define a new domain called *Computation*. The *Computation* domain names the domain of meanings that we can get from evaluating an expression in an environment:

$\mathcal{E} : \text{Exp} \to Environment \to Computation$

The *Computation* domain, with the helper functions in Figure 6.20 (described more below), allows us to factor out some complex details and have compact clauses in our valuation functions. In FLK, the benefit is largely that we can factor out much of the error checking. When we extend FL, e.g., in order to add state in Chapter 8, *Computation* will become more complex, but it will allow the valuation functions to remain relatively simple.

The *Computation*, *Denotable*, and *Value* domains all serve as knobs that can be tweaked to specify different languages. The *Procedure* domain's argument value must be denotable (otherwise the argument could not be named by a formal parameter).

We assume that the *Computation* domain comes equipped with a set of helper functions shown in Figure 6.20. *val-to-comp* treats a value as computation, while *err-to-comp* treats an error as one. *with-value* is a generalized version of the various functions we have already seen with this name. It unpackages a computation into a value (if possible) and applies to this value a function that returns another computation. In the case where the computation cannot be coerced to a value, it is passed along unchanged. The other *with-* functions (which can be written in terms of *with-value*), are similar, except that they may also generate new error computations rather than just passing along old ones.

The valuation functions of Figures 6.21 and 6.22 are relatively compact, thanks in large part to the *Computation* abstraction and the associated helper functions. However, semantics written in this style can take some time to get used to. It is helpful to keep in mind the signatures of all functions, as well as the purposes of the various auxiliary functions. To see how much more complicated the valuation clauses would be, compare the one-line `if` clause of Figure 6.21 with:

```
$\mathcal{E}[\![$(if $E_1$ $E_2$ $E_3$)$]\!]$ =
  $\lambda e$ . **matching** ($\mathcal{E}[\![E_1]\!]\ e$)
        $\triangleright$ ($Value \mapsto Computation\ v$) $[\!]$ **matching** $v$
                                          $\triangleright$ ($Bool \mapsto Value\ b$) $[\!]$
                                            **if** $b$ **then** ($\mathcal{E}[\![E_2]\!]\ e$) **else** ($\mathcal{E}[\![E_3]\!]\ e$) **fi**
                                          $\triangleright$ **else** (*err-to-comp* non-bool-in-if-test)
                                          **endmatching**
        $\triangleright$ **else** ($\mathcal{E}[\![E_1]\!]\ e$)
        **endmatching**
```

This sort of error checking would be repeated throughout the valuation clauses.

▷ **Exercise 6.24** Recall that the integer division and remainder operators (`/` and `rem`) are different than other binary operators because they are ill-defined when the second argument is *0*. Write the valuation clause for $\mathcal{P}[\![/]\!]$. ◁

Usual operations on *Bool*: *not*, *and*, *or*
Usual operations on *Int*: +, -, `*`, /, ...
Equality operation on Identifier: *same-identifier?*

$$\begin{array}{l}
\mathit{val\text{-}to\text{-}comp} : \mathit{Value} \to \mathit{Computation} \;= \text{Value} \mapsto \text{Computation}\\
\mathit{err\text{-}to\text{-}comp} : \mathit{Error} \to \mathit{Computation} \;= \text{Error} \mapsto \text{Computation}\\
\mathit{den\text{-}to\text{-}comp} : \mathit{Denotable} \to \mathit{Computation} \;= \lambda\delta\,.\,\delta\\
\mathit{error\text{-}comp} : \mathit{Computation} \;= (\text{err-to-comp}\ \texttt{error})
\end{array}$$

```
$with\text{-}value : Computation \to (Value \to Computation) \to Computation$
$= \lambda c f\,.$ matching $c$
          $\triangleright (Value \mapsto Computation\ \ v)\ \|\ (f\ \ v)$
          $\triangleright$ else $c$
          endmatching
```

```
$with\text{-}values : Computation^* \to (Value^* \to Computation) \to Computation$
$= \lambda c^* f\,.$ matching $c^*$
          $\triangleright [\,]_{Computation}\ \|\ (f\ [\,]_{Value})$
          $\triangleright c_{fst}\ .\ c_{rest}{}^*\ \|\ (with\text{-}value\ c_{fst}$
                      $(\lambda v_{fst}\,.\ (with\text{-}values\ c_{rest}{}^*\ (\lambda v_{rest}{}^*\,.\ (f\ (v_{fst}\ .\ v_{rest}{}^*))))))$
          endmatching
```

```
$with\text{-}boolean\text{-}val : Value \to (Bool \to Computation) \to Computation$
$= \lambda v\,.$ matching $v$
          $\triangleright (Bool \mapsto Value\ \ b)\ \|\ (f\ \ b)$
          $\triangleright$ else $(err\text{-}to\text{-}comp$ not-a-boolean$)$
          endmatching
```

Similar for *with-unit-val*, *with-integer-val*, *with-symbol-val*, *with-pair-val*.

$$\begin{array}{l}
\mathit{with\text{-}boolean\text{-}comp} : \mathit{Computation} \to (\mathit{Bool} \to \mathit{Computation}) \to \mathit{Computation}\\
= \lambda c f\,.\ (\mathit{with\text{-}value}\ \ c\ \ (\lambda v\,.\ (\mathit{with\text{-}boolean\text{-}val}\ \ v\ \ f)))
\end{array}$$

Similar for *with-procedure-comp*.

```
$with\text{-}denotable : Binding \to (Denotable \to Computation) \to Computation$
  $= \lambda \beta f\,.$ matching $\beta$
            $\triangleright (Denotable \mapsto Binding\ \ \delta)\ \|\ (f\ \ \delta)$
            $\triangleright (Unbound \mapsto Binding\ \ Unbound)\ \|\ (err\text{-}to\text{-}comp$ unbound-var$)$
            endmatching
```

Figure 6.20: Auxiliary functions and values for FLK, Part I.

$\mathcal{E}$: Exp $\rightarrow$ *Environment* $\rightarrow$ *Computation*
$\mathcal{E}*$: Exp* $\rightarrow$ *Environment* $\rightarrow$ *Computation**
$\mathcal{L}$: Lit $\rightarrow$ *Value*
$\mathcal{P}$: Primop $\rightarrow$ *Value** $\rightarrow$ *Computation*
$\mathcal{B}$: Boollit $\rightarrow$ *Bool*
$\mathcal{N}$: Intlit $\rightarrow$ *Int*

$\mathcal{E}[\![L]\!] = \lambda e \,.\, (\textit{val-to-comp}\ \mathcal{L}[\![L]\!])$

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

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

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

$\mathcal{E}[\![\texttt{(if}\ E_1\ \ E_2\ \ E_3\texttt{)}]\!] =$
$\quad \lambda e \,.\, (\textit{with-boolean-comp}\ \ (\mathcal{E}[\![E_1]\!]\ \ e)\ \ (\lambda b \,.\, \textbf{if}\ b\ \textbf{then}\ (\mathcal{E}[\![E_2]\!]\ \ e)\ \textbf{else}\ (\mathcal{E}[\![E_3]\!]\ \ e)\ \textbf{fi}))$

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

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

$\mathcal{E}[\![\texttt{(primop}\ O\ \ E^*\texttt{)}]\!] = \lambda e \,.\ \ (\textit{with-values}\ \ (\mathcal{E}^*[\![E^*]\!]\ \ e)\ \ (\lambda v^* \,.\ (\mathcal{P}[\![O]\!]\ \ v^*)))$

$\mathcal{E}[\![\texttt{(error}\ I\texttt{)}]\!] = \lambda e \,.\ \ (\textit{err-to-comp}\ I)$

$\mathcal{E}^*[\![]\!] = \lambda e \,.\ \ [\,]_{\textit{Computation}}$

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

$\mathcal{L}[\![\texttt{\#u}]\!] = (\textit{Unit} \mapsto \textit{Value}\ \ \mathrm{unit})$
$\mathcal{L}[\![B]\!] = (\textit{Bool} \mapsto \textit{Value}\ \ \mathcal{B}[\![B]\!])$
$\mathcal{L}[\![N]\!] = (\textit{Int} \mapsto \textit{Value}\ \ \mathcal{N}[\![N]\!])$
$\mathcal{L}[\![\texttt{(symbol}\ I\texttt{)}]\!] = (\textit{Sym} \mapsto \textit{Value}\ \ I)$

$\mathcal{B}$ and $\mathcal{N}$ defined as usual.

Figure 6.21: Valuation functions for FLK, Part I

```
$\mathcal{P}[\![$not?$]\!] = \lambda v^* .$ matching $v^*$
                    $\triangleright [v]_{Value}$ ‖ $(\textit{with-boolean-val}$
                                 $v$
                                 $\lambda b . \ (\textit{val-to-comp} \ (\textit{Bool} \mapsto \textit{Value} \ (\textit{not} \ b))))$
                    $\triangleright$ else $(\textit{err-to-comp}$ not?-wrong-number-of-args$)$
                    endmatching

$\mathcal{P}[\![$left$]\!] = \lambda v^* .$ matching $v^*$
                    $\triangleright [v]_{Value}$ ‖ $(\textit{with-pair-val} \ v \ \lambda c_l c_r . c_l)$
                    $\triangleright$ else $(\textit{err-to-comp}$ left-wrong-number-of-args$)$
                    endmatching
Similarly for right

$\mathcal{P}[\![$integer?$]\!] =$
  $\lambda v^* .$ matching $v^*$
        $\triangleright [v]_{Value}$ ‖ matching $v$
                     $\triangleright (\textit{Int} \mapsto \textit{Value} \ i)$ ‖ $(\textit{val-to-comp} \ (\textit{Bool} \mapsto \textit{Value} \ \textit{true}))$
                     $\triangleright$ else $(\textit{val-to-comp} \ (\textit{Bool} \mapsto \textit{Value} \ \textit{false}))$
                     endmatching
        $\triangleright$ else $(\textit{err-to-comp}$ integer?-wrong-number-of-args$)$
        endmatching

Similarly for other predicates

$\mathcal{P}[\![+]\!] = \lambda v^* .$ matching $v^*$
                $\triangleright [v_1 \ v_2]_{Value}$ ‖ $(\textit{with-integer} \ v_1$
                                  $(\lambda i_1 . \ (\textit{with-integer} \ v_2$
                                          $(\lambda i_2 . \ (\textit{val-to-comp}$
                                                  $(\textit{Int} \mapsto \textit{Value} \ (+ \ i_1 \ i_2)))))))$
                $\triangleright$ else $(\textit{err-to-comp}$ +-wrong-number-of-args$)$
                endmatching
```

Similarly for other binary operators, except / and rem, which give an error on a second argument of 0.

Figure 6.22: Valuation functions for FLK, Part II

▷ **Exercise 6.25** In FLK, `error` expressions take a manifest constant as the name of the error. There are other possible error strategies. One is to have only a single error value, which might simplify the semantics while making errors less helpful in practice. Another approach is to allow the argument of `error` to be a computed value. If we alter the syntax of FLK to support the form (`error` E), then

a. Write the evaluation clause for (`error` E).

b. What is the meaning of an `error` expression whose argument results in an error? ◁

▷ **Exercise 6.26** Construct an operational semantics for FLK that uses explicit environments rather than substitutions. [Hint: it is a good idea to introduce a closure object that pairs a lambda expression with the environment it is evaluated in.] ◁

▷ **Exercise 6.27** Write a denotational semantics for FL that does not depend on its desugaring into FLK. That is, the valuation clauses should directly handle features such as `define`, `let`, `letrec`, and procedures with multiple arguments. ◁