Chapter 2

Slivers

This chapter introduces a class of programming idioms that I call *slivers*. Slivers capture the generating, mapping, filtering, and accumulating idioms commonly found in loops and recursive procedures. Many programs can be visualized as *sliver diagrams* in which information flows through a static network of slivers. This type of program organization is certainly not new, and in the next chapter we will see how it is supported in various popular programming paradigms. What *is* new is a dynamic framework for combining the operational aspects of slivers in a reasonable way. I will only hint at that framework in this chapter, but will develop it in detail in Chapters 4 and 5.

I motivate slivers in the context of two extended examples:

1. Simple manipulations of a linearly-structured database. This example introduces many of the issues relevant to modularity in the signal processing style of programming.

2. Alpha renaming of lambda calculus terms. This tree-structured example illustrates important issues and patterns of computation that do not arise in linear examples.

For the purpose of presentation, both examples have intentionally been kept simple. However, even though the examples are somewhat contrived, the issues they raise are very real.

2.1 Linear Example: Database Manipulation

2.1.1 Overview

In this example, we consider simple programs manipulating an employee database. Suppose that the interface to the database is the following set of procedures:[1]

- `(first-record` *database-descriptor*`)` returns the first record of the specified database.
- `(next-record` *record*`)` returns the database record following the given one.
- `(end-of-database?` *record*`)` tests for a distinguished database termination record.
- `(record-get` *record field*`)` retrieves the contents of the specified field from the given record.

The first three procedures effectively make the database accessible as a linked list of records. The `record-get` procedure is the means of extracting information from an individual record. For the purposes of this example, we will assume that the records in every database are sorted alphabetically by surname.

The procedural interface hides many details about how the database is implemented. For all we know, the database records might be stored in a tree-like fashion; different fields might be stored in distinct tables; or parts of the database may be stored remotely, perhaps even distributed across several physical servers and sites. Even a bizarre scenario in which every call to `next-record` initiates a request for a data entry operator to type in the next employee record on the fly is consistent with this interface! These details do not affect the *values* returned by the procedures, although they may show through in other ways (e.g., `next-record` may take a long time if records are stored remotely).

Below, we investigate two procedures describing computations on an employee database:

- `(mean-age` *database-descriptor*`)` returns the average age of employees in the specified database.

[1] The procedure specifications have parentheses because they, like all programming examples in this dissertation, are written in Scheme, a dialect of Lisp ([CR$^+$91], [ASS85]). I use Scheme because its support for first-class procedures, side effects, and tail recursion permits concise expression of a wide range of programming styles. However, the computational issues I am investigating are independent of the particular language in which the examples are phrased.

- (fat-cats *threshold database-descriptor*) returns a list (sorted alphabetically) of the names of all employees in the specified database whose salaries are greater than the given threshold.

These procedures will serve as a basis for comparing several styles of programming. We will explore issues of modularity by considering the ease with which these procedures can be modified and combined.

2.1.2 Monolithic Style: Functional Paradigm

In the *monolithic style*, the example programs are implemented as single recursive procedures that collect information during a traversal of the database. Here is a monolithic implementation of mean-age written in the functional programming paradigm:[2]

```
(define (mean-age_fun database)
  (define (loop record age-total count)
    (if (end-of-database? record)
        (/ age-total count)
        (loop (next-record record)
              (+ age-total (record-get record 'age))
              (+ 1 count))))
  (loop (first-record database) 0 0))
```

The internal loop procedure performs an iterative traversal of the database while maintaining three state variables: record points to the current record, age-total names the running sum of employee ages, and count names the number of records examined.

The monolithic functional version of fat-cats has a similar structure:

```
(define (fat-cats_fun threshold database)
  (define (gather record)
    (if (end-of-database? record)
        '()
        (if (> (record-get record 'salary)
               threshold)
            (cons (record-get record 'name)
                  (gather (next-record record)))
            (gather (next-record record)))))
  (gather (first-record database)))
```

[2]Identifiers naming procedures (such as mean-age) will often be subscripted (in this case, with $_{fun}$) to distinguish different implementations of the same procedure. The subscript is *not* part of the identifier; it is merely a convenient way to refer to a particular definition.

The internal `gather` procedure traverses the entire database and collects into a list the names of the employees satisfying the salary predicate. The list collection strategy used here, in which an employee name is prepended to the list resulting from a recursive call to `gather`, preserves the relative ordering of the selected employees. Because the databases are ordered alphabetically, so is the resulting list.

2.1.3 Monolithic Style: Imperative Paradigm

Both of the above procedures are written in the functional paradigm, which does not permit assignment. For comparison, Figure 2.1 presents monolithic versions of the two procedures written in the imperative paradigm. The imperative programs are rather similar to their functional counterparts. The main difference is that immutable formal parameters in the functional versions become mutable state variables in the imperative version.

2.1.4 Computational Shapes

Following [ASS85], I carefully distinguish *procedures* from the *computations* that they specify.[3] A procedure is just a specification for a computational process, whereas a computation is the process that dynamically unfolds when the procedure is called.

Despite the fact that the functional and imperative versions of `mean-age` are written in different styles, they specify computations with similar operational characteristics. In both cases, even though the internal `loop` procedure is syntactically recursive, the tail-recursive property of Scheme ([CR$^+$91], [Ste77]) guarantees that `loop` executes iteratively, just as if it had been written as a `do`, `for`, or `while` loop in other languages. And both `loop` procedures iterate over the same three state variables, though in one case they are explicit arguments and in the other case they are implicit. The operational behavior of $\texttt{fat-cats}_{fun}$ and $\texttt{fat-cats}_{imp}$ is likewise very similar.

On the other hand, even though $\texttt{mean-age}_{fun}$ and $\texttt{fat-cats}_{fun}$ are both written in the functional style, they specify computations that differ in fundamental ways. In addition to the obvious fact that these programs perform different calculations, the `gather` procedure

[3] I use the term "computation" in place of the term "process" used by [ASS85]. I make this change to distinguish the standard usage of "process" in the concurrency community from the notion for computational unfoldment presented in [ASS85].

```
(define (mean-age_imp database)
  (let ((record (first-record database))
        (age-total 0)
        (count 0))
    (define (loop)
      (if (end-of-database? record)
          (/ age-total count)
          (begin
            (set! age-total (+ age-total (record-get record 'age)))
            (set! count (+ count 1))
            (set! record (next-record record))
            (loop))))
    (loop)))

(define (fat-cats_imp threshold database)
  (let ((record (first-record database)))
    (define (gather)
      (if (end-of-database? record)
          '()
          (if (> (record-get record 'salary)
                 threshold)
              (let ((name (record-get record 'name)))
                (begin
                  (set! record (next-record record))
                  (cons name (gather))))
              (begin
                (set! record (next-record record))
                (gather)))))
    (gather)))
```

Figure 2.1: Imperative versions of the database procedures.

does not have the purely iterative character of `loop`. When the salary predicate is satisfied, the implementation must "remember" to perform a pending `cons` operation upon returning from the recursive call to `gather`. Conceptually, such calls to `gather` push information on an implicit control stack. Calls made to `gather` when the salary predicate is not satisfied push no information on the stack (they act like `goto`s). The amount of control space required to execute `fat-cats` is thus proportional to the number of names satisfying the salary predicate.

We'd like some method of characterizing these sorts of operational similarities and differences between computations. One way of doing this is to adopt a standard evaluation model that captures operational features. For example, [ASS85] uses an expresssion-rewriting model to analyze procedures in terms of traces of their evaluation. Here is a summary of such a trace of $\texttt{mean-age}_{fun}$ on a sample database (where the details of database and record representations have been surpressed):

```
(mean-age_fun  database)
(loop  record1    0 0)
(loop  record2   43 1)
(loop  record3  103 2)
(loop  record4  125 3)
(loop  record5  174 4)
(loop  record6  202 5)
(loop  record7  233 6)
...
```

On every line of the trace, the state of the computation is entirely captured in the values of the three arguments to `loop`. Thus, $\texttt{mean-age}_{fun}$ exhibits the constant space behavior of an iterative computation.

In contrast, here is a trace for `fat-cats` on a sample database:

```
(fat-cats_fun 250000  database)
(gather  record1)
(gather  record2)
(cons "Raws P. Arrow" (gather  record3))
(cons "Raws P. Arrow" (gather  record4))
(cons "Raws P. Arrow" (gather  record5))
(cons "Raws P. Arrow" (cons "Gill Bates" (gather  record6)))
(cons "Raws P. Arrow" (cons "Gill Bates" (gather  record7)))
...
```

Some calls to `gather` simply rewrite to another call of `gather` on the next record. However, when the salary predicate is satisfied, a call to `gather` rewrites to a `cons` application whose second argument is a call to `gather`. The rightward-growing "bulge" of the trace expressions is due to the pending calls to `cons`, which textually encode the implicit control stack required by the computation.

These examples show how the expression-rewriting model differentiates procedures according to the pattern by which their computations evolve. In keeping with [ASS85], I will refer to these patterns of computational evolution as *shapes of computation*. The notion of shape is intended to capture operational features of a computation, such as time and space complexity and the relative ordering of various events. The work described in this report was motivated by the desire to formalize the notion of computational shape and to incorporate shape-based ideas into a programming language.

Certain patterns are so common that programmers give them names. For example, `mean-age` is a *linear iteration* in three state variables while `fat-cats` is a *linear recursion* in one argument. Both shapes are said to be linear because because each line of the trace contains only a single recursive call. Later we will study more general *tree shapes* in which several recursive calls are potentially active.

2.1.5 Monolithic Programs Lack Modularity

`Mean-age` and `fat-cats` exhibit several common idioms for linear computations. Despite the differences in their shape and what they compute, both procedures generate all the records of the database in succession, and accumulate information from each record. `Mean-age` uses two instances of a running sum idiom, in which a numeric variable initialized to 0 is used to accumulate a sum. `Fat-cats` uses a filtering idiom to eliminate unwanted records, a mapping idiom to find the name in each record, and a list accumulation idiom to collect information in a list. These kinds of idioms (generate, map, filter, accumulate) arise repeatedly in manipulation of linear data. For example, Waters found that 90% of the code in the Fortran Scientific Subroutine Package could be expressed wholly in terms of these idioms [Wat79].

A major problem with the above procedures is that the idioms are not localized in the program text. For example, the database enumeration idiom is spread out across each procedure body in the calls to `first-record`, `next-record`, and `end-of-database?`. Similarly, the running sum idiom conceptually consists of the declaration, initialization, and update of a variable that maintains the sum. However, in `mean-age`$_{fun}$, these three parts of the running sum idiom are textually separated from each other. The situation is marginally better in `mean-age`$_{imp}$, where the declaration and initialization occur together.

The problem with non-localized idioms is that they are hard to read, write, and modify. Idiom recognition is hampered when the programmer has to hunt for fragments of an idiom in different parts of the code. First, it is necessary to make sure that all the right pieces are present, and then it is necessary to become convinced that the rest of the code doesn't prevent the idiom from working as expected. Reasoning is similarly complicated when writing or modifying code; the programmer has to mentally juggle pieces from various idioms and guarantee that they don't adversely interact. The non-locality forces the programmer to wade through details unrelated to the idiom.

With non-localized idioms, the author of a programs may very well be guided by mental notions of such idioms, and an attuned reader of programs can recognize the idioms. But the idioms are only *implicit*. A fundamental principle of modularity in programming is that idioms should be made *explicit*. When idioms are captured and named, programs can be expressed by explicitly composing program fragments embodying the idioms. The above procedures are said to be *monolithic* because they are expressed as densely interwoven idiom fragments rather than as compositions of reusable idiom components.

Programs written in the monolithic style are difficult to combine and modify. For example, suppose we want to compute the average age of employees earning more than a given salary. Intuitively, we would like to connect `fat-cats` and `mean-age` in series. But the result of `fat-cats` is a list of names, and `mean-age` expects a database. Even if we reach inside the procedures, we do not find a common interface through which they can be connected. Instead, the monolithic style forces us to construct an entirely new procedure from scratch:[4]

[4] Here, and throughout the rest of this section, we will only consider procedures written in the functional style. However, all the analyses and conclusions hold for the imperative style as well.

```
(define (fat-cat-age_fun threshold database)
  (define (loop record age-total count)
    (if (end-of-database? record)
        (/ age-total count)
        (if (> (record-get record 'salary)
               threshold)
            (loop (next-record record)
                  (+ age-total (record-get record 'age))
                  (+ 1 count))
            (loop (next-record record)
                  age-total
                  count))))
  (loop (first-record database) 0 0))
```

Almost all of the expression fragments used in the resulting procedure are taken from the two original functional procedures. But we cannot point to single entities that represent the salary filtering, the running total for ages, or the running count of filtered employees. As before, these idioms are smeared throughout the (new) `loop` procedure. The work that went into implementing these idioms in the original procedures must be repeated because the monolithic style does not create reusable language artifacts embodying the idioms.

An interesting feature of `fat-cat-age`$_{fun}$ is that it is iterative, even though it is partially derived the non-iterative `fat-cats`$_{fun}$ procedure. Examination of that procedure reveals that the stacking behavior is due only to the list accumulation idiom. Since `fat-cat-age`$_{fun}$ does not use that idiom, it does not require a procedure call stack. This example suggests that we should be able to associate shapes of computation with individual idioms and then determine the shape generated by a program from the shapes of the idioms out of which it is constructed. Later, we will explore this idea in depth.

As another example, consider a procedure that returns both the fat cats and the average employee age for a given database. Here we want to connect `mean-age` and `fat-cats` in parallel. In the simplest approach, we can just encapsulate the two existing computations within one procedure:

```
(define (fat-cats&mean-age_nice threshold database)
  (list (fat-cats threshold database)
        (mean-age database)))
```

This extremely simple means of combination is a key advantage of the black-box modularity offered by procedures. This procedure works regardless of the styles in which `fat-cats` and `mean-age` happen to have been written.

Unfortunately, while this procedure computes the desired *values*, it doesn't necessarily compute them in the desired *way*. In particular, the calls to `fat-cats` and `mean-age` will each traverse the entire database independently. This means that `next-record` will be called twice for every record in the database — once by `fat-cats` and once by `mean-age`. The overhead of calling `next-record` twice for every record may be deemed unacceptable, especially in the case where calls to `next-record` are particularly expensive (e.g., when the database is stored remotely).

An alternative is to merge the computations of `fat-cats` and `mean-age` such that the database is traversed only once. Here is the merged version for the functional implementations:[5]

```
(define (fat-cats&mean-agefun threshold database)
  (define (both record age-total count)
    (if (end-of-database? record)
        (list '() (/ age-total count))
        (if (> (record-get record 'salary)
               threshold)
            (mlet (((rest-names avg)
                    (both (next-record record)
                          (+ age-total (record-get record 'age))
                          (+ 1 count))))
              (list (cons (record-get record 'name) rest-names) avg))
            (both (next-record record)
                  (+ age-total (record-get record 'age))
                  (+ 1 count)))))
  (both (first-record database) 0 0))
```

Here, the internal `both` procedure performs a single traversal of the database, during which it accumulates both a list of names and an average age. It returns the two results as a two-element list. Since this procedure does use the linearly-recursive list accumulation idiom, its shape is a linear recursion rather than a linear iteration.

Although `fat-cats&mean-age`$_{fun}$ is "better" than `fat-cats&mean-age`$_{nice}$ in terms of execution time, it is a whole lot worse in terms of understandability and modifiability. The idioms here are exactly the same as those used in `fat-cat-age`, but their interleaving makes

[5] I assume that Scheme has been extended with a pattern matching version of `let` called `mlet`. A `mlet` expression has the same form as a `let` expression, except that the name position of a binding may contain a *name pattern* rather than just a single name. A name pattern is any list structure whose atomic elements are names. The name pattern is matched against the value of the binding expression, and the resulting name/value associations are accessible in the body of the `mlet`. It is an error if a name pattern does not match the binding value.

It is possible to dispense with `mlet` by instead using explicit selectors or Scheme's multiple value return mechanism. However, I find both of these alternatives too unwieldy for expository purposes.

them even harder to recognize. Additionally, the program is complicated by the details of managing two return values rather than one. For example, the fact that the calculation of the average age is still effectively iterative is obscured by the inherently recursive nature of the name collection.

As a final example, we consider a modification to the original `fat-cats` procedure. By a clever trick based on mutable pairs, it is possible to transform the linear recursive version into a linear iteration. The original version generates a recursive computation because no pair is allocated for `cons` until both of its subexpressions are fully evaluated. This means that the pairs composing the spine of the returned list are actually allocated in order from back to front, requiring the implementation to maintain a stack of pending `cons`es.

The trick is to allocate the pairs from front to back by using side effects. Here is a procedure that embodies this trick:[6]

```
(define (fat-cats-iter_mixed threshold database)
  (let ((ans-pair (cons 'ignore 'ignore)))
    (define (gather! record prev-pair)
      (if (end-of-database? record)
          (begin
            (set-cdr! prev-pair '())
            (cdr ans-pair))
          (if (> (record-get record 'salary)
                 threshold)
              (let ((next-pair (cons (record-get record 'name)
                                     'ignore)))
                (begin
                  (set-cdr! prev-pair next-pair)
                  (gather! (next-record record) next-pair)))
              (gather! (next-record record) prev-pair))))
    (gather! (first-record database) ans-pair)))
```

The iterative `gather!` procedure takes the previously allocated pair in addition to the current record. Every time a record satisfies the condition, it bashes the `cdr` of the previous pair to point to a newly allocated pair, and then passes the new pair to the recursive call. The initial pair passed to `gather!` is a dummy whose `cdr` will ultimately be the final answer list; `gather!` returns this answer list when the end of the database is reached.

This trick, which I will call *cdr-bashing list collection*, is clearly handy in many list processing programs. However, the monolithic approach does not allow us to package

[6]The resulting procedure is annotated with a $_{mixed}$ subscript because it combines aspects of both the functional and imperative styles.

this trick into a component that can be used elsewhere. In fact, the convoluted structure of `fat-cats-iter`$_{mixed}$ makes it difficult to even notice that it has been obtained from `fat-cats` by applying a trick!

2.2 Tree Example: Alpha Renaming

2.2.1 Overview

All the programs considered thus far generate linear computations. In this section, we will study a program that generates a tree-structured computation. Tree-shaped computations are important because, in the absence of control features like non-local exits and continuations, the procedure calls dynamically executed in any program can naturally be arranged in a tree. Linear computations are trivially a subset of tree computations. Moreover, we will see that tree-shaped computations support a richer set of evolution patterns than do linear computations.

The program we consider is *alpha renaming*, an operation on abstract syntax trees that is common in interpreters and compilers. Alpha renaming is one of the simplest practical programs that involves combining different shapes of tree walks in a nontrivial way.

Alpha renaming is a mechanism for consistently renaming the variables in a program so that each variable has a unique name.[7] This transformation is especially useful in lexically scoped languages, which permit the same identifier to name different logical variables within a single program. Many other expression transformations (e.g., substitution) are easier to perform on an expression that has been alpha renamed.

For simplicity, I will initially present alpha renaming in the context of an extremely simple language, the lambda calculus. The syntax of lambda calculus terms E is given by the following grammar:

[7] Technically, alpha renaming is any renaming transformation that maintains the same "connectivity" of variable declarations and references within a program. It does not necessarily imply making all variable names unique. However, in practice, the term is typically used to indicate that the resulting variable names are all distinct.

$$
\begin{array}{rcll}
E & ::= & I_{\text{use}} & \text{[Variable reference]} \\
 & | & \texttt{(lambda } I_{\text{def}} \; E_{\text{body}}\texttt{)} & \text{[Abstraction]} \\
 & | & \texttt{(call } E_{\text{rator}} \; E_{\text{rand}}\texttt{)} & \text{[Application]} \\
I & ::= & \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \ldots \mid \texttt{aa} \mid \texttt{ab} \mid \ldots &
\end{array}
$$

For our purposes, it is acceptable to view the lambda calculus as a restricted Lisp dialect where all procedures take exactly one argument and applications are tagged with an explicit `call` keyword. The only detail that matters for the present example is that the identifier I_{def} introduced by a `lambda` term declares a variable that can be referenced anywhere within E_{body} (as long as there is no intervening declaration of another variable with the same name).

The lambda calculus is about as spartan as a programming language can be. Later, we will consider extending it with extra features that make it more palatable to program in. We will see that a good test of the modularity of an alpha-renaming program is how little it needs to be changed in order to accomodate such features.

As an example of alpha renaming, consider the lambda calculus term:

```
(lambda x
  (lambda y
    (call (lambda x x)
          (call (lambda y x)
                (call (lambda x y)
                      z)))))
```

In this term, there are three logically distinct variables named `x`, two named `y`, and one named `z`. Variable references that occur within the scope of a declaration are said to be *bound*; those, like `z`, that are not in the scope of a declaration are said to be *free*.

The following is an alpha renamed instance of the above term:

```
(lambda x_1
  (lambda y_2
    (call (lambda x_3 x_3)
          (call (lambda y_4 x_1)
                (call (lambda x_5 y_2)
                      z)))))
```

Here, all declared variables and bound references have been consistently renamed by extending the original name with a unique number. (The free variable reference `z` cannot been renamed.) Any other method for consistent renaming that guarantees distinct names for distinct logical variables would also be acceptable.

The standard technique for alpha renaming can be visualized as the superposition of three tree-walking computations, as sketched in Figure 2.2:

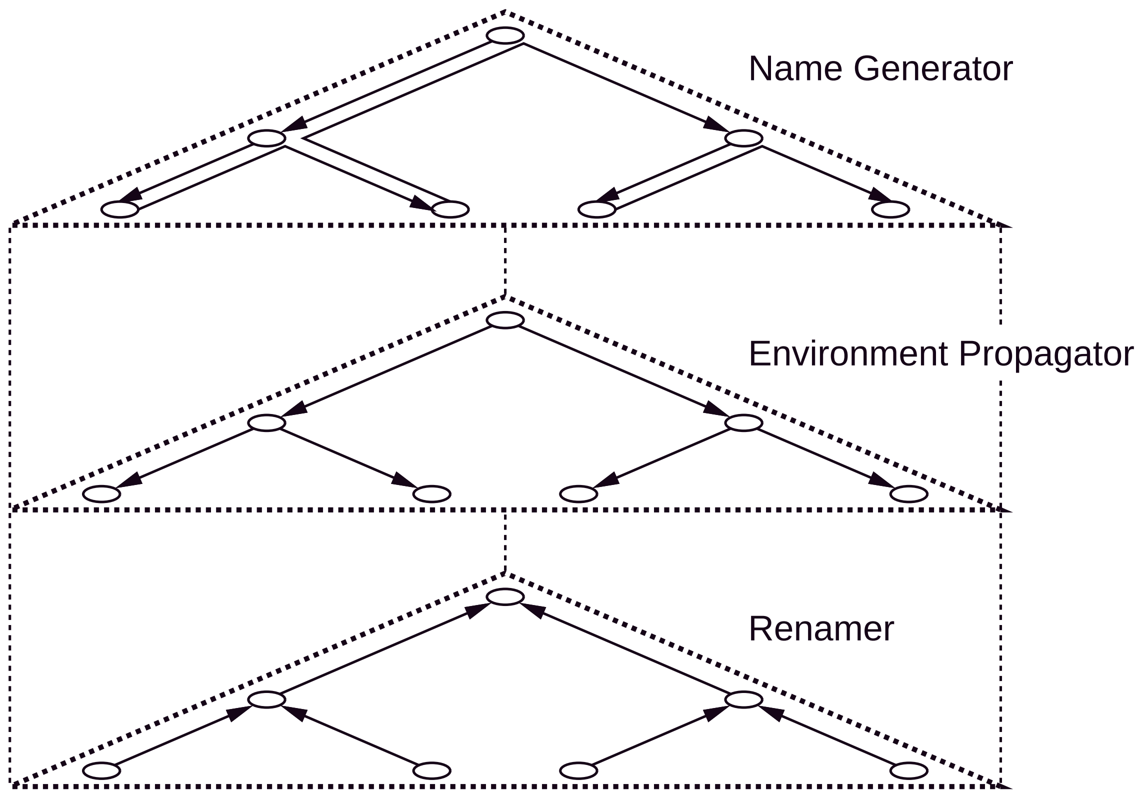

Figure 2.2: Sketch suggesting the different shapes the computations involved in alpha-renaming.

1. A *name generator* that creates a fresh name for every declared variable. To guarantee distinctness among the fresh names, the generator must conceptually carry with it some history of the names generated so far. Since the history at any point in time depends on the history at the previous point in time, the name generation computation effectively traces out a single-threaded dependence path through the syntax tree specified by the term. Whether the generator walks the term tree left-to-right, right-to-left, or by some other route is immaterial as long as the path is a single thread.

2. An *environment propagator* that transmits the fresh name for each declared variable down to all references of that variable. Since both subterms of an application see the same set of old-name/new-name bindings, the environment computation conceptually fans out into independent subcomputations at each `call` node. This downward fan shape contrasts with the single-threaded nature of the name generator.

3. A *renamer* that makes a copy of the term tree in which variable declarations and references have been replaced by the appropriate fresh name passed down by the environment propagator. The renaming computation conceptually starts at the leaves of the term tree and builds a new term on its way up the tree.

Although the above algorithm is standard, the form of the description — a decomposition into superposed computations — is not. One of the main themes of this dissertation is that conceptualizing programs in terms of interacting processes is essential to good modularity regardless of whether or not the programs are actually run on multi-processor machines. Each of the computations described above is a natural modular unit. Such computations are valuable building blocks for other programs; for example, environment propagating computations crop up all the time in interpreters and static analyzers. And such computations have the interchangeability usually associated with modules; for instance, in some contexts it might be worthwhile to replace a left-to-right name generator with a right-to-left one.

From this perspective, it seems reasonable to judge alpha-renaming programs by how well they reflect the modular structure described above. Unfortunately, the typical monolithic implementations exhibit little of this structure.

2.2.2 Monolithic Style: Functional Approach

Figure 2.3 presents an alpha renaming procedure written in the functional paradigm. The internal `walk` procedure takes three arguments: the term to be renamed; an environment that associates original variable names with their new names; and a positive integer representing a counter that is incremented every time a new variable declaration is encountered. The integer is used as an argument to `fresh-name`, which is responsible for generating unique names.[8] `Walk` returns two results: the alpha-renamed term, and the current value of the counter. Appropriate procedures for environment manipulation, fresh name generation, and syntactic abstraction (e.g., `lambda?`, `make-call`) are straightforward and have been omitted.

The $\texttt{alpha-rename}_{fun}$ procedure exhibits *none* of the modular structure suggested by Figure 2.2. Name generation, environment propagation, and construction of the renamed term are all inextricably intertwined in the single `walk` procedure. There is no handle for reusing one of the computations in a different program, or for easily modifying one of these computatations independently of the others. For example, the name generator hidden in the above program performs a left-to-right walk over the tree. What changes would be required to yield a right-to-left walk?[9] Answering this question requires consideration of the whole program rather than just a name generation piece. Furthermore, when making the changes, we must be careful to maintain the integrity of the other conceptual computations.

Finally, consider how $\texttt{alpha-rename}_{fun}$ would have to be modified in response to extending the base language with the conditional term (`if` E_{test} E_{then} E_{else}). The dispatch within `walk` would need to be extended with a clause like the following:

```
((if? exp)
 (mlet (((new-test num1) (walk (test exp) env num)))
   (mlet (((new-then num2) (walk (then exp) env num1)))
     (mlet (((new-else num3) (walk (else exp) env num2)))
       (list (make-if new-test new-then new-else) num3)))))
```

This is an awfully complicated mess for something as simple as a conditional! The problem is that the bookkeeping details associated with returning both the renamed term *and* the updated counter obscure the following essential facts:

- The counter flows in a left-to-right fashion through the three subterms of the `if`.
- The environment flows unchanged into the three subterms of the `if`.

[8] For $\texttt{alpha-rename}_{fun}$ to be correct, it is necessary to assume that `fresh-name` returns a name disjoint from the set of free variables of the term.

[9] In the alpha renaming program, the order of traversal doesn't affect the correctness of the result. However, it's easy to imagine similarly structured programs in which some traversal orders are more desirable than others.

```
(define (alpha-rename_fun exp)

  (define (walk exp env num)
    (cond
     ((variable? exp)
      (list (env-lookup exp env) num))
     ((lambda? exp)
      (let ((old-formal (formal exp)))
        (let ((new-formal (fresh-name old-formal num)))
          (mlet (((new-body num1) (walk (body exp)
                                        (env-extend old-formal
                                                    new-formal
                                                    env)
                                        (+ num 1))))
            (list (make-lambda new-formal new-body) num1)))))
     ((call? exp)
      (mlet (((new-rator num1) (walk (rator exp) env num)))
        (mlet (((new-rand num2) (walk (rand exp) env num1)))
          (list (make-call new-rator new-rand) num2))))
     ))

  (mlet (((new-exp final-num) (walk exp env-standard 0)))
    new-exp))
```

Figure 2.3: Monolithic version of an alpha renamer written in the monolithic style.

Since these are the default flows associated with the name generation and environment computations, we would prefer a modularization in which we didn't have to specify them explicitly for the `if` case at all! After all, one of the measures of good modularity is the localizability of changes — internal modifications to one module shouldn't require changing connected modules.

2.2.3 Monolithic Style: Imperative Approach

Judicious use of side effects can improve the modularity of the alpha renaming program. Consider the `alpha-rename`$_{imp}$ procedure in Figure 2.4. The key difference between this

```
(define (alpha-rename_imp exp)

  (define gensym!
    (let ((num 0))
      (lambda (name)
        (let ((new-name (fresh-name name num)))
          (begin (set! num (+ num 1))
                 new-name)))))

  (define (walk exp env)
    (cond
     ((variable? exp) (env-lookup exp env))
     ((lambda? exp)
      (let ((old-formal (formal exp)))
        (let ((new-formal (gensym! old-formal)))
          (make-lambda new-formal
                       (walk (body exp)
                             (env-extend (formal exp)
                                         new-formal
                                         env))))))
     ((call? exp)
      (make-call (walk (rator exp) env)
                 (walk (rand exp) env)))
     ))

  (walk exp env-standard))
```

Figure 2.4: Monolithic version of an alpha-renamer written in the monolithic style.

procedure and the previous one is that most of the name generation computation has been captured in the `gensym!` procedure. The `gensym!` procedure owns a local state variable (`num`) that maintains the counter which was spread throughout the entire `walk` procedure within alpha-rename$_{fun}$. Because the single threading of the name generation computation is now being managed implicitly by side-effects rather than explicitly by data flow, the interface to the `walk` procedure is much simpler: it takes one less argument and returns one less result than in the functional version. This makes the resulting code easier to read and extend. For example, an `if` term can be handled by extending `walk` with the following clause, which is far simpler than the solution for the functional version:

```
((if? exp)
 (make-if (walk (test exp) env)
          (walk (then exp) env)
          (walk (else exp) env)))
```

This example underscores the importance of state and side effects as a technique for modularizing programs. This point is certainly not new and is elegantly argued elsewhere ([ASS85], [Bar92]). However, it is worth emphasizing that side effects are an important tool that cannot be neglected in our goal of modularizing programs.

In spite of the improvements, alpha-rename$_{imp}$ fails to achieve modularity in some important ways:

- The notion of a downward-flowing environment computation is still intertwined with the tree-construction computation in the single `walk` procedure. This means that the environment computation is not an entity that can be reused elsewhere. Furthermore, this organization forces the `if` handler to explicitly indicate that the environment is passed unchanged to each recursive call to `walk` on the subterms of the `if`. In an ideal scenario, this default flow shouldn't have to be explicit.

- More subtly, while *most* of the name generation computation has been localized to `gensym`, not *all* of it has. The order in which name generation visits the nodes of the term tree is inherited from the order in which the implementation language visits the arguments to a procedure call. In Scheme, where the order of argument evaluation is unspecified, no particular order can be relied upon. If for some reason we want to guarantee that the name generator visits subterms in left-to-right order, it is necessary to rewrite the `call` handler as:

```
((call? exp)
 (let ((new-rator (walk (rator exp) env)))
   (let ((new-rand (walk (rand exp) env)))
     (make-call new-rator new-rand))))
```

Here, Scheme's (strict) `let` construct forces the operator walk to be performed before the operand walk. A similar approach would have to be used for for cases like `if`.

Again, the order of the name generator doesn't matter in this particular problem, but traversal order does matter in other problems with a similar structure. In general, we would like a mechanism by which we can easily choose from options like left-to-right, right-to-left, and don't-care. A modularity technique providing finer control over the ordering of side effects (when it matters) is preferable to one that offers no control.

We have given numerous examples demonstrating that the monolithic approach fails to capture an important class of programming idioms in an effective way. It is obviously desirable to consider program organizations in which such idioms can be captured as modular units.

2.3 Slivers Capture Programming Idioms

2.3.1 Two Approaches to Decomposing Computations

We have seen that procedures like `mean-age`, `fat-cats`, and `alpha-rename` are conceptually composed out of many idiomatic units. But what is the nature of these units, and how are they combined to yield a program?

Viewing computations graphically gives some insight into these questions. Figure 2.5 shows a *computation diagram* for `fat-cats`. A computation diagram is a kind of circuit diagram for computations. In the diagram, every procedure application is represented as a labelled rectangular *device*, except for calls to `fat-cats` and `gather`, which have been expanded in terms of their definitions.[10] (Procedure calls represented by devices will be called *unexpanded*, while those replaced by the structure of their bodies will be called *expanded*.) Triangles pointing into a device are *input ports* that represent the procedure's arguments,

[10] In the diagram, `salary` and `name` are abbreviations for calls to `record-get`.

while triangles pointing out of a device are *output ports* for the procedure's results. Devices are connected by *wires* that specify the data dependences between procedures. Values can be thought of as flowing along the wires in the direction of the arrows.

This picture is similar to graphical depictions of dataflow programs [Den75, DK82] except that here the devices can compute only once, and the wires can transmit only a single value token before they are "used up". A computation diagram can be viewed as a dataflow diagram unwound so that every spatial entity is also associated with a unique point of time during the execution of the program. We will have much more to say about the static and dynamic properties of such computation diagrams in Chapter 8. For now, we will focus on ways in which the diagram can be decomposed into modular units.

The dashed boxes in Figure 2.6 indicate a *layer decomposition* of the computation diagram for `fat-cats`. Each box, called a *layer*, represents an expanded procedure call: the topmost box represents a call to `fat-cats`, while the other boxes represent calls to `gather`. Each layer contains devices for the unexpanded subcalls in the procedure's body, and sits on top of the layers for the expanded subcalls in the procedure's body. Layers are glued together by wires that pass arguments down to sublayers and receive results up from sublayers. Some of the arguments are explicit in the code (e.g., the results of the `next-record` devices correspond to the `record` parameter of `gather`) while others are implicit (the `threshold` "bus" down the middle of the diagram indicates that lexical scoping effectively makes `threshold` an implicit argument of `gather`).

Figure 2.7 shows a so-called *sliver decomposition* of the same diagram. Here the dashed boxes, called *slivers*, parse the diagram into vertical components rather than horizontal ones. The slivers are glued together by *cables*, collections of wires that pass information in and out of the sides of the slivers.

The rules for what comprises a legal sliver are very loose. The only real requirement is that if a sliver contains one instance of a device corresponding to a particular call in the procedure underlying the diagram, then it must contain all such instances. But beyond that, there are just some general guidelines for choosing slivers. The slivers are typically chosen to encapsulate repeated units of related functionality while minimizing the structure of the cables between them. All instances of the three devices for database manipulation

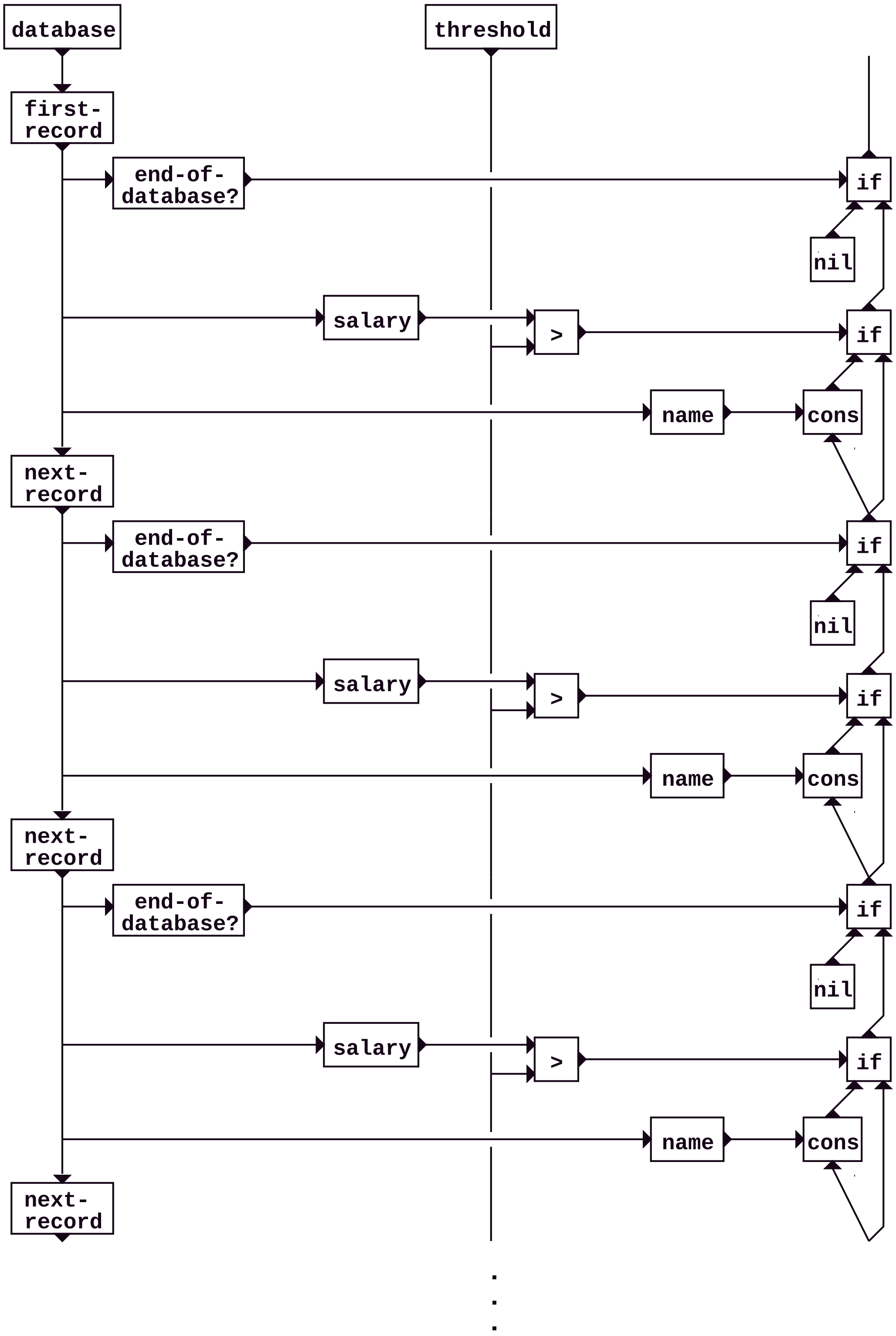

Figure 2.5: Computation diagram for `fat-cats`.

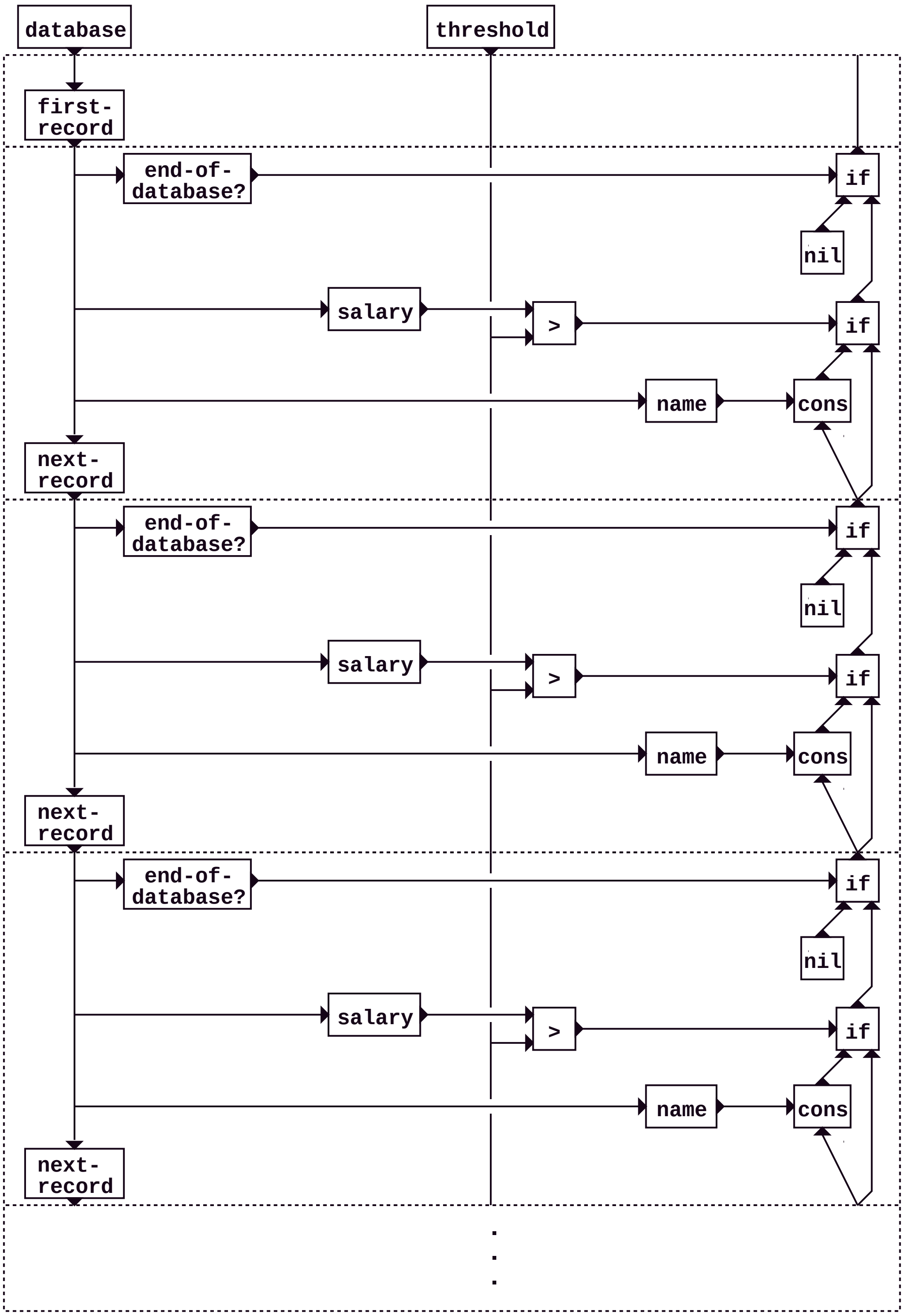

Figure 2.6: Layer decomposition of `fat-cats`.

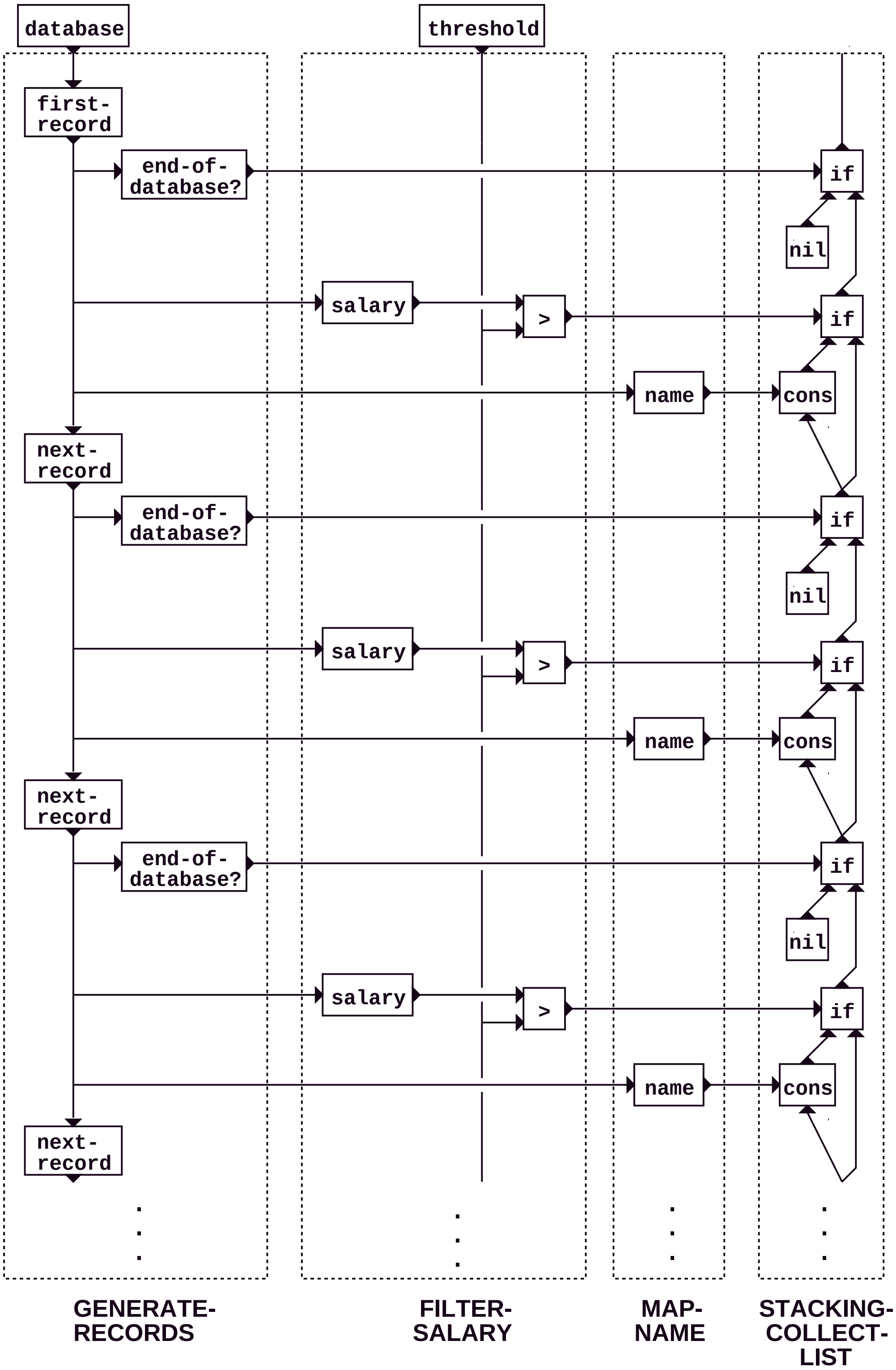

Figure 2.7: Sliver decomposition of `fat-cats`.

are bundled into the sliver labelled `GENERATE-RECORDS`, while the devices responsible for list collection are packaged up into `STACKING-COLLECT-LIST`. (The term "stacking" is intended to indicate that the sliver builds the resulting list from bottom up.) It would be possible to decompose these slivers up even further, say by factoring the `end-of-database?` devices or the `nil` devices into their own separate slivers, but this did not seem particularly desirable in this case. The devices within a sliver often form a single connected component of the computation diagram, but slivers like `FILTER-SALARY` and `MAP-NAME` consist of repeated connected components, all of which are mutually disjoint.

The most remarkable feature of the sliver decomposition in Figure 2.7 is that the individual slivers correspond closely to the kinds of idioms mentioned in the discussion of `mean-age` and `fat-cats`. The `GENERATE-RECORDS` sliver takes a database and spits out the individual records of the database in succession, along with a boolean *termination flag* indicating whether the database termination record has been found. Conceptually, this component occurs in both `mean-age` and `fat-cats`. The `STACKING-COLLECT-LIST` sliver accumulates elements into a list from the bottom up; elements are conditionally included in the resulting list depending on the value of an associated boolean *presence flag*. This can be made into a general linear recursive accumulator by abstracting the `cons` and `nil` to be any binary operator and base value. `FILTER-SALARY` effectively filters the records based on salary by providing the presence flag to the list collector, while `MAP-NAME` simply extracts the names from each record.

A host of other idioms can be depicted as slivers. For example, running sum and cdr-bashing list collection are two linear iterative accumulators that can be encapsulated into slivers. And the three tree-traversal idioms for `alpha-rename` sketched in Figure 2.2 can also be fleshed out into tree-shaped slivers.

What feature of sliver decompositions allows it to capture so many idioms? Whereas a layer decomposition focuses on the recursive pattern of a computation, a sliver decomposition focuses on recursionless operators. In effect, the recursion has been distributed over the slivers rather than being the main organizational principle for the program. The sliver-based organization directs attention away from the structural details of recursion and towards a more functional view of how program units fit together.

The notion of distributing loops over program components to enhance modularity is an old one. It has its origins in Lisp's higher-order list operators and APL's array operators, and is now used extensively as a technique in data parallel languages, functional and mostly functional languages, concurrent languages, and stream-based languages. We will study many of these techniques in detail in Chapter 3.

The new idea in slivers is to provide decompositions for a more general class of recursive computations than those handled by existing techniques. We shall see that most of the techniques alluded to above are limited to expressing linear iterative computations. Even the techniques that handle more general computations exhibit other limitations that constrain the class of decomposable computations. The goal of slivers is to express general tree-structured computations as compositions of mix-and-match parts. The strategy is to gain insight into the nature of these parts by studying sliver decompositions of monolithic computations that contain them.

Before we go on, it is worth pointing out that the sliver decomposition of Figure 2.7 is not an ideal modular decomposition. The problem is that the list collection sliver "knows" about the filtering sliver in the way that it handles the presence flag. In a sliver decomposition for a program that lists all employees, the list collection sliver would not handle any presence flags at all; for a similar program with two filtering predicates, the list collection sliver might need to handle two presence flags per record. It is clearly unreasonable to require different list collection slivers for each of these situations. In some sense, the instances of `if` that manipulate the presence flag really belong in the filtering sliver. But this change would greatly complicate the wiring between the filtering and accumulation slivers; it would add a cable loop between the slivers, and the clear upward flow of information in `STACKING-COLLECT-LIST` would disappear. In Chapter 5, we will investigate various methods for enhancing the modularity of computations that involve filtering.

2.3.2 Procedural Slivers

Just as it is important to distinguish procedures from the computations they specify, it is necessary to distinguish *procedural slivers* from *computational slivers*. A computational sliver is what we have simply been calling a sliver up to this point: a repeated pattern of devices in a computation diagram that embodies a programming idiom. A procedural sliver is a specification of this repeated pattern. Intuitively, it is that fragment of a procedure that encapsulates the idiom.

How can such a fragment be specified? Investigating answers to this question is one of the main goals of this report. For the moment, the following approach may be helpful for envisioning procedural slivers. Imagine starting with the text of a procedure and erasing all the program structure that is not relevant for generating a given computational sliver. Then what's left over is a crude kind of procedural sliver.

Figure 2.8 shows the descriptions that result from using this approach for three of the computational slivers of `fat-cats`. The resulting procedural slivers don't make much sense individually, and certainly aren't executable. However, a sensible program like `fat-cats` can be constructed by textually overlaying its component procedural slivers. In computation space, the textual overlaying of procedural slivers corresponds to combining computational slivers in a side-by-side manner to yield the computation diagram for the entire procedure. The challenge is to develop a programming model in which the specification and combination of procedural slivers is no harder than procedure specification and combination.

For the remainder of this report, we will loosely use the term *sliver* for both computational slivers and procedural slivers when the meaning is clear from context or the distinction doesn't matter. The modifiers "computational" and "procedural" will only be sprinkled in when we wish to emphasize the distinction.

2.3.3 Sliver Diagrams

Sliver decompositions can be abstracted into *sliver diagrams* that summarize the sliver and cable interconnections while suppressing many details. Figure 2.9 shows one possible diagram for the `fat-cats` sliver decomposition. Each sliver is represented as a box, while cables are represented as thick, directed conduits between slivers. Individual devices and wires are represented as before.

The sliver diagram in Figure 2.9 is a very literal interpretation the sliver decomposition in Figure 2.7 in the sense that the slivers and cables are assumed to have exactly the structure shown in the decomposition. For example, in Figure 2.9, the input cable to the

```
;;; Fragment responsible for GENERATE-RECORDS
;;;
(define (fat-cats           database)
  (define (gather record)
        (end-of-database? record)





            (gather (next-record record)))
  (gather (first-record database)))

;;; Fragment responsible for FILTER-SALARY.
;;; (The fragment for MAP-NAME is similar)
;;;
(define (fat-cats threshold         )
  (define (gather      )


            (> (record-get record 'salary)
               threshold)



            (gather                     ))
  (gather                        ))

;;; Fragment responsible for STACKING-COLLECT-LIST
;;;
(define (fat-cats                  )
  (define (gather      )
    (if
        '()
        (if

            (cons
                  (gather                      )
            (gather                     )))))
  (gather                        )
```

Figure 2.8: Procedural fragments approximating the procedural slivers that generate the computational slivers of `fat-cats`. These are obtained by "whiting out" structure that does not appear in a given computation sliver. The original program can be obtained by textually overlaying all of its component fragments.

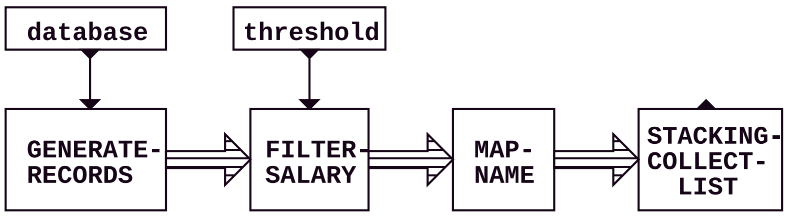

Figure 2.9: A sliver diagram summarizing the sliver decomposition for `fat-cats`.

`MAP-NAME` sliver is assumed to carry three types of wires:

1. *Termination wires* that transmit a boolean indicating whether the end of the database has been reached.

2. *Presence wires* that transmit a boolean indicating whether the filter has passed the associated record.

3. *Element wires* that transmit the current record when the corresponding terminations and presence wires both transmit a true value.

The output cable of the `MAP-NAME` sliver also carries the same three types of wires. `MAP-NAME` passes the termination and presence wires unchanged, but transmits a string rather than a record on the element wire.

However, not every wire that happens to pass across a vertical dotted box in a sliver decomposition need be considered part of that sliver. Figure 2.10 depicts a sliver diagram for a different perspective on the sliver decomposition of `fat-cats`. Here, the cable produced by the record generator fans out and feeds both `FILTER-SALARY` and `MAP-NAME`, while the list collector now takes two input cables:

1. A cable of presence wires from the salary comparison.

2. A cable of element wires from the name mapper.

(Presumably one or both of these cables also carries the termination wires.) It is easy to imagine yet other diagrams in which each type of wire is carried by a distinct cable. Of course, the interfaces to the abstracted slivers depends on the chosen interpretation. The challenge is to design the diagrams to maximize the reusability of the abstracted slivers. We will discuss this in detail later; for now, we will choose Figure 2.9 as the "standard" sliver diagram for `fat-cats`.

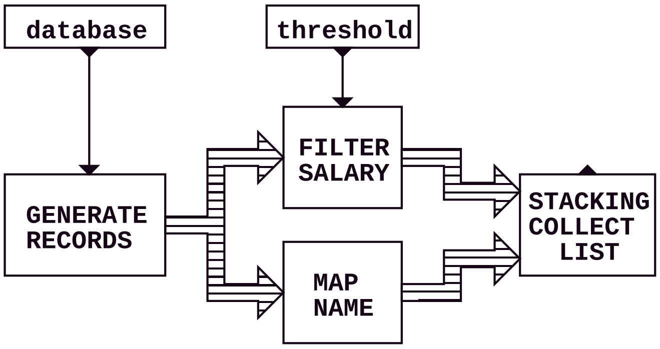

Figure 2.10: An alternate sliver diagram for `fat-cats`.

Sliver diagrams underscore the advantages of capturing idioms in modular units. They provide a convenient framework in which to understand and compare programs. For example, the block diagram in Figure 2.11 elucidates the structure of `mean-age`. The fact that there are two running sums is obvious from the structure of the diagram. The `MAP-ONE` sliver maps every input record into the constant `1`; summing these `1`s up gives the employee count. From the sliver diagrams, it is easy to see that `mean-age` uses the same database enumeration strategy as `fat-cats`.

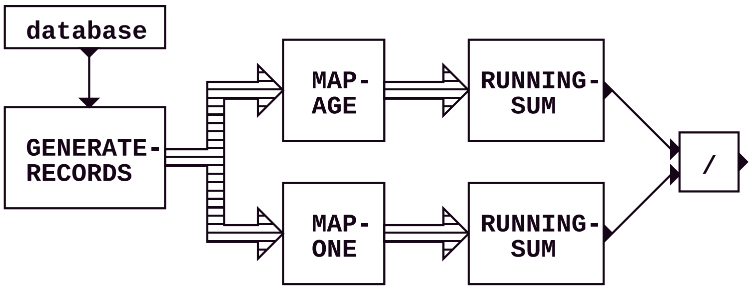

Figure 2.11: A sliver diagram for `mean-age`.

Perhaps the biggest advantage of sliver diagrams is that they promote the notion that programs should be organized out of mix-and-match parts. For example, Figures 2.12

and 2.13 support the intuition that the `fat-cat-age` and `fat-cats&mean-age` procedures are essentially series and parallel combinations of `fat-cats` and `mean-age`.[11] Replacing the recursive `STACKING-COLLECT-LIST` sliver in Figure 2.9 with the iterative `CDR-BASHING-COLLECT-LIST` sliver (Figure 2.14) represents the constant-space computation generated by the `fat-cats-iter` procedure. It is apparent from the diagrams that the iterative version is obtained from the recursive one by a simple modification; moreover, that modification is encapsulated in a way such that it can easily be plugged in elsewhere. Components like `RUNNING-SUM`, `STACKING-COLLECT-LIST`, and `CDR-BASHING-COLLECT-LIST` are especially reusable because they do not depend on particular details of the database example.[12]

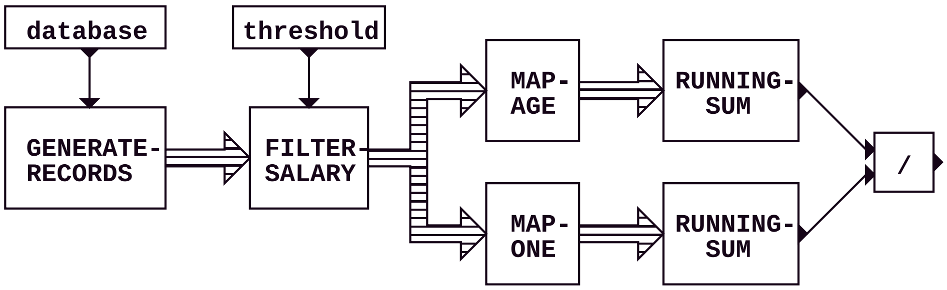

Figure 2.12: A sliver diagram for `fat-cat-age`.

As a final example, Figure 2.15 shows a sliver diagram for the tree-based `alpha-rename` computation. Here, each cable transmits tree-structured information between the slivers. The name generation, environment propagation, and renaming computations have each been encapsulated in their own sliver. There are some additional slivers, though. The `FILTER-FORMALS` sliver finds the names of the declared variables; this has been factored out of the name generation computation so that the names can be supplied to the environment computation as well. And the `TERM->TREE` and `TREE->TREE` slivers provide conversions between concrete terms and the "exploded" versions of their abstract syntax trees that are

[11] Figure 2.12 is somewhat misleading, since it uses versions of `MAP-AGE`, `MAP-ONE`, and `RUNNING-SUM` that must handle the presence flag produced by `FILTER-SALARY`. Yet, the corresponding slivers in Figure 2.11 do not manipulate a presence flag. This is another instance of the problems inherent in filtering that we will need to deal with later.

[12] Modulo the above-mentioned issues of filtering.

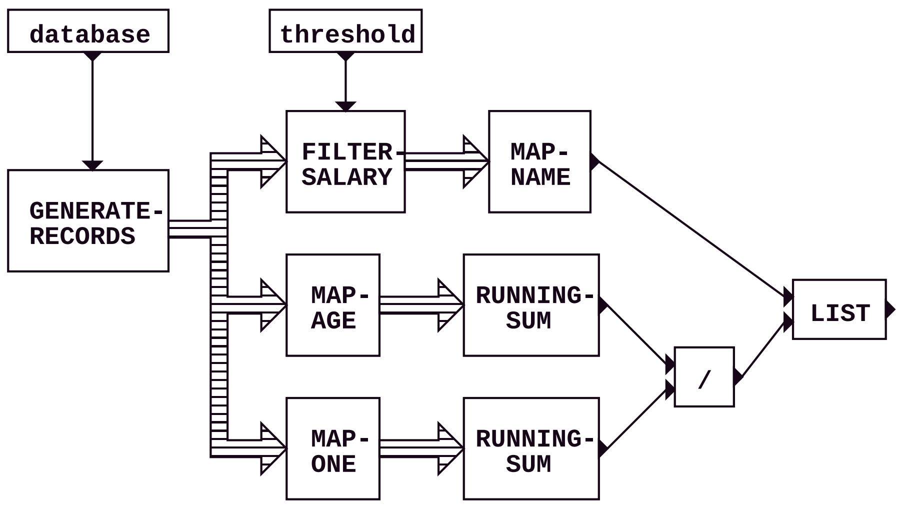

Figure 2.13: A sliver diagram for `fat-cats&mean-age`.

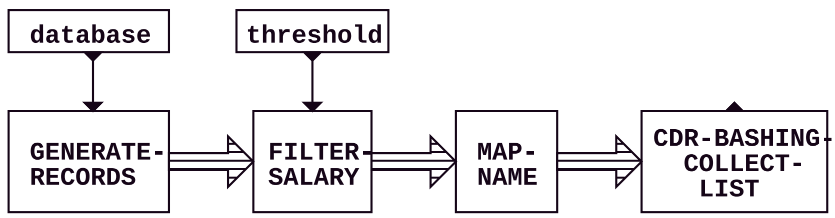

Figure 2.14: A sliver diagram for `fat-cats-iter`.

transmitted via the cables.[13]

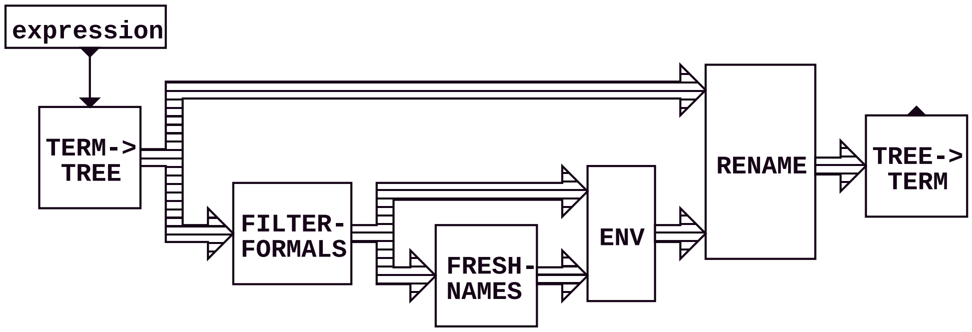

Figure 2.15: A sliver diagram for `alpha-rename`.

A key property of sliver diagrams is that they explicitly show only the flow of data, not of control. As noted above, an upshot of modularizing the kinds of idioms discussed above is that loops and recursions are distributed over individual devices rather than being the main organizing principle for programs. This approach directs attention away from the often complex details of control flow and towards a more functional view of how program units fit together. Reasoning about control is localized to the implementations of individual devices, where it is much more tractable.

2.3.4 Operational Interpretation of Sliver Diagrams

The previous examples demonstrate that it's easy to combine slivers in mix-and-match ways. But the operational behavior of the programs specified by the resulting sliver diagrams is rather ambiguous. Suppose we presented the sliver diagrams in Figures 2.9 – 2.15 to other programmers and described the high-level purpose of each box but did not explain how the diagrams were derived from monolithic programs. Then the programmers could easily construct many distinct but consistent stories about the meaning of these diagrams that were different from our intended meaning. Here are some alternate interpretations:

1. The boxes are procedures that take and return aggregate data structures.

[13] This is only true as long as there are no circularities in the data dependencies implied by the cables. While such "data loops" can be be useful programming techniques (e.g. [Bir84]), they can also complicate reasoning about programs by requiring programmers to think in terms of fixed points.

2. The boxes are demand-driven agents that request individual values from and return individual values to their neighbors.

3. The boxes are data-driven processes that concurrently consume and produce values sent over communicating channels.

These interpretations correspond to some of the classical techniques for achieving modularity by distributing loops across programming idioms.

But the existence of so many interpretations means that sliver diagrams fail to nail down important operational characteristics of computations. In the case where sliver diagrams are derived from monolithic procedures, we'd like the resulting diagrams to specify operational behavior similar to the original procedures. And when the diagrams aren't derived from a single procedure, but are pieced together from existing slivers, we want a theory that defines how the operational behavior of the composite structure is determined from the operational behavior of the parts.

Exactly what is meant by "similar operational behavior" will necessarily remain vague until we present a formal model of computation (Chapter 8). But here we can at least give a few examples of what we have in mind:

- The monolithic `mean-age` and `fat-cat` procedures perform a single traversal over the database, while the monolithic `alpha-rename` procedure performs a single traversal over its argument expresssion. We would like the computation described by the corresponding sliver diagrams to maintain this single-traversal property.

- The monolithic `mean-age` procedure generates a computation that requires constant data and control space. The corresponding sliver diagram should also specify a constant-space computation.

- The monolithic `fat-cats` procedure does not compute the name for a record that does not pass the salary filter. In this case, the name computation is trivial, but it's easy to imagine cases where it is important to avoid computations on items that do not pass a filter. It is desirable for the computations specified by the sliver diagrams in both Figures 2.9 and 2.10 to avoid unnecessary operations. (The fan-out in Figure 2.10 makes it tricky to implement this behavior.)

In a truly modular approach to these operational concerns, the operational behavior of a computation should be derivable from operational behavior of its component parts. In order to capture operational nuances, it will be necessary to include operational details in the interfaces to slivers. We have done this to some extent in terms of the names of the list collection slivers; `STACKING-LIST-COLLECT` and `CDR-BASHING-LIST-COLLECT` have the same input/output behavior, but the first requires a control stack while the second does not. We'd like to have a method of reasoning about these kinds of space issues based on the structure of the slivers. Chapters 4 and 5 develop such a method. But first, in Chapter 3, we will explore what's wrong with existing methods for expressing sliver diagrams as programs.