Chapter 6

SYNAPSE: Programming with Slivers and Slags

SYNAPSE[1] is a language that provides a collection of operators (slivers) for manipulating synchronized lazy lists and trees (slags). The language shows the utility of slivers and slags while suppressing their implementation details. While SYNAPSE does not exhibit the full range of expressiveness implied by the sliver technique, it is rich enough for investigating the power and limitations of slivers.

According to Abelson and Sussman, a language has three parts: primitives, means of combination, and means of abstraction [ASS85]. For SYNAPSE, the primitives are a set of higher-order procedures that manipulate synchronized lazy lists and trees. These procedures are carefully implemented so that they obey all of the sliver requirements discussed in Section 5.4.2. The means of combination and means of abstraction for synapse are simply inherited from OPERA, the dialect of Scheme in which SYNAPSE is embedded. Thus, the SYNAPSE "language" is just a library of OPERA procedures.

Ideally, we would like to characterize synchronized lazy lists and trees as abstract data types. An *abstract data type* is a method of defining a data structure in terms of the functions that manipulate it. For example, a pair data type can be defined in terms of a constructor `cons` that builds a pair from two components; selectors `car` and `cdr` that

[1] *SYN*chronized *A*ggregate *P*rogramming *S*tructures *E*nvironment.

extract the left and right elements of a pair; and a *pair?* predicate that determines whether a given entity is a pair. The functions defining an abstract data type are usually required to satisfy certain relationships. For example, for any version of pairing it should be the case that `(car (cons` A B`))` is equivalent to A, `(cdr (cons` A B`))` is equivalent to B, and `(pair? (cons` A B`))` is equivalent to `#t`.

Unfortunately, the *functional* nature of abstract data types is at odds with the *operational* nature of synchronized lazy aggregates. Because their operational handling is intricately intertwined with their structure, synchronized lazy lists and trees resist being characterized as abstract data types. I have been unable to distill the essence of slags into an concise axiomatic theory. Instead, I have taken a more experimental approach. I have developed a suite of higher order slag procedures (slivers) that have proven useful in a wide variety of simple applications. Below, I describe these slivers and illustrate their use.

All of the examples described here are written in OPERA, a dialect of Scheme supporting the concurrency, synchronization, and laziness features motivated in Section 5.1.3. For the most part, it is not necessary to understand the details of OPERA in order to appreciate the examples; for this reason, the discussion of OPERA is deferred until Chapter 7. The one detail that *is* important to the present discussion is that the default evaluation strategy of OPERA evaluates all subexpressions of a procedure call in parallel.[2] This parallel evaluation strategy accounts for the concurrency exhibited by the examples.

6.1 Linear Computations

SYNAPSE supports linear computations through a suite of slivers (procedures) that manipulate synquences (synchronized lazy lists). The core procedures are summarized in Figures 6.1 and 6.2. The core is neither minimal (some slivers can be defined in terms of the others) nor complete (not all useful synquence slivers can be defined in terms of the core). Rather, the core is a set of standard slivers that are useful in many situations. The higher order nature of many of the slivers (i.e., they accept functional arguments) allows them to be tailored to a wide variety of problems. The choice of core procedures was influenced by the

[2]This evaluation strategy is not allowed in standard Scheme [CR+91], which requires that the subexpressions be evaluated in some sequential order.

list operations of Lisp [Ste90, CR+91] and Haskell[3] [HJW+92], the array operations of APL [Ive87], the vector operations of FX [GJSO92], and Waters's series operations [Wat90].

In the remainder of this section, we will employ the core procedures to illustrate various properties of sliver decomposition within SYNAPSE.

6.1.1 Iteration vs. Recursion

Using the slivers in Figures 6.1 and 6.2, it is easy to define both iterative and recursive procedures in a modular fashion. A combination of slivers behaves iteratively if all of its components have *down* or *across* shape. If any component has *up* shape, the computation may behave recursively. We now consider a series of examples that illustrate the basic behavior of slivers.

Factorial

We begin with the time worn, but still trusty, factorial function. Here are iterative and recursive versions of a factorial procedure expressed in SYNAPSE:

```
(define (fact-iter num)
  (downQ 1 * (genQ num -1+ zero?)))

(define (fact-rec num)
  (upQ 1 * (genQ num -1+ zero?)))
```

Both definitions use a generator that delivers the numbers from the input down to (but not including) zero. The definitions differ only in the shape of the product accumulator.

The iterative nature of `fact-iter` is demonstrated by the following trace of a sample `fact-iter` application:[4]

---

[3] It is worth noting that SYNAPSE's `upQ` and `up-scanQ` are equivalent to Haskell's `foldr` and `scanr`, but that SYNAPSE's `downQ` and `down-scanQ` are *not* equivalent to Haskell's `foldl` and `scanl`. In particular, the *op* parameter to `downQ` and `down-scanQ` takes its arguments in the opposite order from the corresponding parameter of the Haskell functions. The reason for the discrepency is that the higher order procedural arguments to the SYNAPSE procedures all take their arguments in the same order; this facilitates changing between *down* and *up* accumulators and scanners. The argument order for the higher order procedure in Haskell are intended to faciliate transformations and proofs.

[4] This and all subsequent traces were automatically produced verbatim by the OPERA interpreter. In some cases, illustrating certain behavior required tuning the relative probabilities of particular operators before running the example. (See the discussion of task selection strategies in Section 9.1.1.) Different examples were executed with different tuning parameters. Tuning affects only the probability of observing a particular trace; it does not change the set of possible traces.

Generators:

(`genQ` *init next done?*) [*down*]
Generates a synquence whose first element is *init* and each of whose subsequent elements is determined from the previous by *next* function. The synquence terminates when *done?* function is true of the current element (which is not included in the synquence).

(`produceQ` *initial-state handler*) [*down*]
Generates a synquence whose element values are computed from a "state" that is passed down the synquence in a *down* fashion. At every synquence node, *handler* is applied to three arguments: a current state, a binary *yielding* procedure, and a nullary *terminating* procedure. Calling the yielding procedure on a value v and a new state s produces v as the current synquence element and uses s as the next state. Calling the terminating procedure terminates the synquence.

Reducers:

(`downQ` *init op synq*) [*down*]
Returns the result of iteratively accumulating the elements of *synq* using the combiner *op* and the initial value *init*. If *synq* has elements $e_1 \ldots e_n$, `downQ` computes:

$$(op\ e_n\ (\ \ldots (op\ e_2\ (op\ e_1\ init)\ \ldots)))$$

(`upQ` *init op synq*) [*up*]
Returns the result of recursively accumulating the elements of *synq* using the combiner *op* and the initial value *init*. If *synq* has elements $e_1 \ldots e_n$, `upQ` computes:

$$(op\ e_1\ (op\ e_2 \ldots (op\ e_n\ init)\ \ldots))$$

(`glom-stream` *synq*) [*down*]
Returns an OPERA stream of the non-gap elements iteratively collected from *synq*. (Note: this cannot be defined in terms of `downQ`.)

(`down-nthQ` *index synq*) [*down*]
Returns the result of iteratively calculating the *index*th element (0-based) of a synquence. An error is signalled if *index* is greater than the length of the synquence.

(`up-nthQ` *index synq*) [*up*]
Returns the result of recursively calculating the *index*th element (0-based) of a synquence. An error is signalled if *index* is greater than the length of the synquence.

(`down-lastQ` *index synq*) [*down*]
Returns the last (non-gap) element of a synquence. An error is signalled if *synq* contains no elements.

(`up-firstQ` *index synq*) [*up*]
Returns the first (non-gap) element of a synquence. An error is signalled if *synq* contains no elements.

Figure 6.1: A summary of SYNAPSE's core synquence generating and reducing slivers. The shape of each sliver appears in brackets to the right of each form.

Transducers:

(`mapQ` *fun synq*) [*across*]
Returns the synquence resulting from the elementwise application of *fun* to *synq*.

(`map2Q` *fun synq1 synq2*) [*across*]
Returns the synquence resulting from the elementwise application of *fun* to corresponding elements of *synq1* and *synq2*. An output element is a gap if either input element is a gap. The length of the output synquence is the shorter of the length of the two input synquences.

(`filterQ` *pred synq*) [*across*]
Returns a synquence with the same structure as *synq* in which every element satisfying *pred* is mapped to itself and all other elements are mapped to gaps.

(`reifyQ` *obj synq*) [*across*]
Returns a synquence that maps every gap of *synq* to *obj* and every ungapped element to itself.

(`unreifyQ` *obj synq*) [*across*]
Returns a synquence that maps every instance of *obj* in *synq* to a gap and every other element to itself.

(`down-scanQ` *init op synq*) [*down*]
Returns the synquence of intermediate accumulated values in the iterative accumulation of *synq* using combiner *op* and initial value *init*.

(`up-scanQ` *init op synq*) [*up*]
Returns the synquence of intermediate accumulated values in the recursive accumulation of *synq* using combiner *op* and initial value *init*.

(`truncateQe` *pred synq*) [*down*]
Returns the prefix of *synq* up to, but not including, the first element for which *pred* is true. (The *e* at the end of the name stands for "exclusive".)

(`truncateQi` *pred synq*) [*down*]
Returns the prefix of *synq* up to, and including, the first element for which *pred* is true. (The *i* at the end of the name stands for "inclusive".)

(`prefixQ` *len synq*) [*down*]
Returns a synquence consisting of the first *len* elements of *synq*. An error is signalled if *len* is greater than the length of *synq*

(`down-shiftQ` *init synq*) [*down*]
Returns a synquence with the same size and gap locations as *synq*, but in which every element has been shifted down to the next ungapped element location. The resulting synquence uses *init* for the first ungapped element location.

(`up-shiftQ` *init synq*) [*up*]
Returns a synquence with the same size and gap locations as *synq*, but in which every element has been shifted up to the next ungapped element location. The resulting synquence uses *init* for the last ungapped element location.

(`appendQ` *synq1 synq2*) [*down*]
Returns a synquence formed by appending the elements of *synq1* to those of *synq2*.

Figure 6.2: A summary of SYNAPSE's core synquence transducing slivers.

```
OPERA> (fact-iter 3)
-----------------------:down[A,0]
(zero? 3) --> ()
(* 3 1) --> 3
(-1+ 3) --> 2
-----------------------:down[A,1]
(zero? 2) --> ()
(-1+ 2) --> 1
(* 2 3) --> 6
-----------------------:down[A,2]
(zero? 1) --> ()
(* 1 6) --> 6
(-1+ 1) --> 0
-----------------------:down[A,3]
(zero? 0) --> #t
; Value: 6
```

The trace begins with the expression and ends with its value. The intervening lines of text correspond to various events in the program's execution. Lines of the form

(*primop arg1* ... *argn*) `-->` *result*

are *primop events* that indicate the performance of the primitive operation *primop* on the listed args.[5] A dotted line is a *barrier event* that represents a rendezvous of all the slivers in a lock step component at the down or up barrier of a barrion. In this case, the fact that all dotted lines are labelled `:down` indicates that the computation is iterative — it never returns through any up barriers. The information following the `:down` label (e.g., `[A,2]`) consists of a *generator label* (`A`) and a *barrion index* (`2`). All barrions generated by the same call to `genQ` share the same generator label, but have a unique barrion index. These automatically generated pieces of information make it easier for humans to parse complex traces into meaningful parts.

Barrier events partition traces into *episodes* of primop events. In the `fact-iter` example, each episode contains a zero test and decrement from the `genQ` sliver and a multiplication from the `downQ` sliver.[6] The interleaving of primitive operations from `genQ` and `downQ` illustrates the concurrent evaluation that is fundamental to the sliver technique.

[5] Only operators that have been declared *traceable* produce printed operator events. SYNAPSE has a mechanism for specifying which operators are traceable. In the examples, only the "interesting" operators are traced.

[6] The `zero?` procedure is always performed first, because the other operations will only be performed if the test fails. However, as suggested by the trace, the `*` and `-1+` are unrelated by data dependence and may be performed in either order.

The barriers are important in this example because they they guarantee that `fact-iter` behaves like a monolithic iterative factorial procedure even though it is constructed out of two slivers. The corresponding monolithic factorial could be written as:

```
(define (fact-iter-mono num)
  (define (loop n acc)
    (if (zero? n)
        acc
        (loop (-1+ n) (* n acc))))
  (loop num 1))
```

Each of the barriers depicted in the `fact-iter` trace corresponds to one of the calls to `loop` in the computation of `(fact-iter-mono 3)`. Except for the absence of dotted lines, the following sample trace of `fact-iter-mono` is indistinguishable from a possible trace of `fact-iter`:

```
OPERA> (fact-iter-mono 3)
(zero? 3) --> ()
(-1+ 3) --> 2
(* 3 1) --> 3
(zero? 2) --> ()
(-1+ 2) --> 1
(* 2 3) --> 6
(zero? 1) --> ()
(* 1 6) --> 6
(-1+ 1) --> 0
(zero? 0) --> #t
; Value: 6
```

This is not just a lucky coincidence. Barriers *guarantee* that the possible traces of `fact-iter` on any given input are *exactly* the same as the possible traces of `fact-iter-mono`. In fact, the whole purpose of the barriers is to simulate the behavior of strict, monolithic procedure calls in a computation distributed over several slivers.

The `fact-rec` procedure gives rise to a computation with a very different trace:

```
OPERA> (fact-rec 3)
-----------------------:down[A,0]
(zero? 3) --> ()
(-1+ 3) --> 2
-----------------------:down[A,1]
(zero? 2) --> ()
(-1+ 2) --> 1
-----------------------:down[A,2]
(zero? 1) --> ()
(-1+ 1) --> 0
-----------------------:down[A,3]
(zero? 0) --> #t
-----------------------:up[A,3]
(* 1 1) --> 1
-----------------------:up[A,2]
(* 2 1) --> 2
-----------------------:up[A,1]
(* 3 2) --> 6
-----------------------:up[A,0]
; Value: 6
```

The presence of up barriers indicates that the computation is not an iteration, but a general recursion. Zero tests and decrements are still performed in the *down* phase (because `genQ` has *down* shape), but all the multiplications are performed in the *up* phase – exactly what we expect for a recursive factorial computation. The barrion indices help us match an up barrier with the corresponding down barrier; barriers with the same index are from the same barrion.

The components of the factorial procedures are likely to be used again, so it is a good idea to name them:

```
(define (to-1 num) (genq num -1+ zero?))

(define (down-* synq) (downQ 1 * synq))

(define (up-* synq) (upQ 1 * synq))
```

The above factorial procedures can be rewritten in terms of these new abstractions without changing the behavior exhibited by the traces:

```
(define (fact-iter num) (down-* (to-1 num)))

(define (fact-rec num) (up-* (to-1 num)))
```

It is also possible to represent factorial procedures that count up towards the input number rather than counting down from it:

```
(define (fact-iter2 num) (down-* (from-to 1 num)))

(define (fact-rec2 num) (up-* (from-to 1 num)))

(define (from-to start stop)
  (genQ start 1+ (lambda (n) (> n stop))))
```

These versions *would* yield traces different from the ones illustrated above.

Sum of Squares

As another simple example, consider SYNAPSE programs for summing the squares of the elements in a list:

```
(define (sum-squares-iter lst)
  (down-+ (mapQ square (splay-list lst))))

(define (sum-squares-rec lst)
  (up-+ (mapQ square (splay-list lst))))

;; Utilities
(define (splay-list lst)
  (mapQ car (splay-sublists lst)))

(define (splay-sublists lst)
  (genQ lst cdr null?))

(define (down-+ synq) (downQ 0 + synq))

(define (up-+ synq) (upQ 0 + synq))
```

Here, `splay-sublists` generates the successive sublists of a list, while `splay-list` generates the successive elements of a list. The two versions of the summing program are identical except for the shape of the accumulator.

Sample traces for these two programs appear in Figure 6.3. As expected, the first trace shows how shared barriers and *down*-shaped slivers force `sum-squares-iter` to behave as if it were a monolithic loop. In the second trace, the `+` operations in the *up* phase are expected, but `car` and `square` operations in the *up* phase might seem a little surprising. Since both of these are mapping operations that occur between a *down* sliver and an *up* sliver, they have an *across* orientation (see Section 5.2.3). Recall that it is permissible to perform an *across* operator any time after call initiation but before call return. The sample trace obeys these constraints.

```
OPERA> (sum-squares-iter '(2 3 5))
-----------------------:down[A,0]
(null? (2 3 5)) --> ()
(cdr (2 3 5)) --> (3 5)
(car (2 3 5)) --> 2
(square 2) --> 4
(+ 4 0) --> 4
-----------------------:down[A,1]
(null? (3 5)) --> ()
(cdr (3 5)) --> (5)
(car (3 5)) --> 3
(square 3) --> 9
(+ 9 4) --> 13
-----------------------:down[A,2]
(null? (5)) --> ()
(cdr (5)) --> ()
(car (5)) --> 5
(square 5) --> 25
(+ 25 13) --> 38
-----------------------:down[A,3]
(null? ()) --> #t
; Value: 38


OPERA> (sum-squares-rec '(2 3 5))
-----------------------:down[A,0]
(null? (2 3 5)) --> ()
(cdr (2 3 5)) --> (3 5)
-----------------------:down[A,1]
(null? (3 5)) --> ()
(cdr (3 5)) --> (5)
-----------------------:down[A,2]
(null? (5)) --> ()
(cdr (5)) --> ()
-----------------------:down[A,3]
(null? ()) --> #t
-----------------------:up[A,3]
(car (2 3 5)) --> 2
(square 2) --> 4
(car (5)) --> 5
(square 5) --> 25
(+ 25 0) --> 25
-----------------------:up[A,2]
(car (3 5)) --> 3
(square 3) --> 9
(+ 9 25) --> 34
-----------------------:up[A,1]
(+ 4 34) --> 38
-----------------------:up[A,0] ; Value: 38
```

Figure 6.3: Traces of iterative and recursive programs for summing the squares of elements in a list.

This example underscores that `mapQ` behaves differently in different contexts. In general, this flexibility enhances the reusability of the `mapQ` sliver. (See Section 6.1.3 for specific examples.) However, in the case of `sum-squares-rec`, the fact that *across* operators can exhibit *up* behavior may be undesirable. Delaying the `car` operators until the *up* phase means that `sum-square-rec` maintains a pointer to the list argument until it returns a final result, which can unnecessarily prevent the list cells from being garbage collected sooner.

The default behavior can be overridden by explicitly specifying that the `car`s performed by `splay-list` should be *down* operators. The following `force-down` sliver uses the data dependencies implicit in `down-scanQ` to force all elements of a synquence to be computed in the *down* phase:

```
(define (force-down synq)
  (down-scanQ 'ignore
              (lambda (current previous) current)
              synq))
```

Using `force-down`, it is easy to rewrite `splay-list` so that it always performs `car` operations as early as possible, rather than as late as possible:

```
(define (splay-list lst)
  (force-down (mapq car (splay-sublists lst))))
```

The post-modification trace in Figure 6.4 shows that the change has the desired effect. A similar change could force the multiplications to happen in the *down* phase as well. The moral of this example is that slivers permit control aspects of a program to be fine-tuned in a modular fashion.

List Accumulation

It is instructive to consider computations where the direction of accumulation makes a difference to the final answer. Here are down and up accumulators for pairing, the canonical non-associative, non-commutative binary operator:[7]

```
(define (down-cons synq) (downQ '() tcons synq))

(define (up-cons synq) (upQ '() tcons synq))
```

[7] `Tcons` is a variant of the usual pairing procedure, `cons`. It is use here for a technical reason explained in Section 6.1.3.

```
OPERA> (sum-squares-rec '(2 3 5))
-----------------------:down[A,0]
(null? (2 3 5)) --> ()
(cdr (2 3 5)) --> (3 5)
(car (2 3 5)) --> 2
-----------------------:down[A,1]
(null? (3 5)) --> ()
(car (3 5)) --> 3
(cdr (3 5)) --> (5)
-----------------------:down[A,2]
(null? (5)) --> ()
(cdr (5)) --> ()
(car (5)) --> 5
-----------------------:down[A,3]
(null? ()) --> #t
-----------------------:up[A,3]
(* 5 5) --> 25
(+ 25 0) --> 25
-----------------------:up[A,2]
(* 3 3) --> 9
(+ 9 25) --> 34
-----------------------:up[A,1]
(* 2 2) --> 4
(+ 4 34) --> 38
-----------------------:up[A,0]
; Value: 38
```

Figure 6.4: Sample trace of `sum-squares-rec` after `splay-list` has been modified to perform all `car`s in the *down* phase.

Figure 6.5 shows traces of computations in which these slivers are applied to the synquence produced by `(from-to 1 3)`. The traces show that `up-cons` returns a list of the synquence elements in the order in which they were generated (smallest to largest), while `down-cons` returns the elements in the reverse order. As indicated by the down and up barriers, the order-preserving version requires a linear control stack, while the order-reversing version uses only constant control space.

The fact that `up-cons` preserves the order of elements is one of the main motivations to use recursion (vs. iteration) in procedures that produce lists. One alternate approach to preserving element order is to use `down-cons` to produce the list, and then iteratively reverse this result. Another alternative, as discussed in Section 2.1.5, is to employ a `cdr`-bashing technique that maintains the element order *and* uses constant control space.[8] Figure 6.6 shows the definition a `cdr`-bashing pairing accumulator and a sample trace of its execution. The ability to code `down-cons!` as an independent sliver that has the "right" behavior when composed with other slivers illustrates the power of SYNAPSE.

Iterated Scans

We conclude this section with a less trivial example. The problem is to compute from a list its *nth down* (or *up*) *scan*. The zeroth scan of a list is the list itself; the first scan is a list of the intermediate results from accumulating the list in the specified direction; the second scan is the result of scanning the first scan; and so on. The goal is to write general nth scanning procedures that simulate the behavior of individually crafted monolithic verions for each n.

Figure 6.7 gives sliver-based solutions to the problem. The solutions employ a higher-order `repeated` procedure that returns the n-fold composition of a given procedure `f`. As indicated by the sample traces in Figures 6.8 and 6.9, successive scans are synchronized with each other even though they are dynamically glued together by the higher-order `repeated` procedure. Note how the use of `down-cons!` ensures that `nth-down-scan` generates an

[8] Waters has recently shown that the reversal technique and the `cdr`-bashing technique are practically indistinguishable in terms of efficiency [Wat]. In light of this result, he argues for using the simpler one — i.e., the reversal approach. However, in a language like SYNAPSE, where the `cdr`-bashing technique can be expressed as a modular component, `cdr`-bashing may actually the "simpler" approach.

```
OPERA> (down-cons (from-to 1 3))
-----------------------:down[A,0]
(> 1 3) --> ()
(1+ 1) --> 2
(tcons 1 ()) --> (1)
-----------------------:down[A,1]
(> 2 3) --> ()
(1+ 2) --> 3
(tcons 2 (1)) --> (2 1)
-----------------------:down[A,2]
(> 3 3) --> ()
(1+ 3) --> 4
(tcons 3 (2 1)) --> (3 2 1)
-----------------------:down[A,3]
(> 4 3) --> #t
; Value: (3 2 1)

OPERA> (up-cons (from-to 1 3))
-----------------------:down[A,0]
(> 1 3) --> ()
(1+ 1) --> 2
-----------------------:down[A,1]
(> 2 3) --> ()
(1+ 2) --> 3
-----------------------:down[A,2]
(> 3 3) --> ()
(1+ 3) --> 4
-----------------------:down[A,3]
(> 4 3) --> #t
-----------------------:up[A,3]
(tcons 3 ()) --> (3)
-----------------------:up[A,2]
(tcons 2 (3)) --> (2 3)
-----------------------:up[A,1]
(tcons 1 (2 3)) --> (1 2 3)
-----------------------:up[A,0]
; Value: (1 2 3)
```

Figure 6.5: Sample `down` and `up` accumulations of a pairing operator.

```
(define (down-cons! default-cdr synq)
  (let ((dummy (cons 'ignore default-cdr)))
    (begin
      (downQ dummy
             (lambda (val pair)
               (let ((new (tcons val default-cdr)))
                 (begin
                   (set-cdr! pair new)
                   new)))
             synq)
      (cdr dummy))))


OPERA> (down-cons! '() (from-to 1 3))
(cons ignore ()) --> (ignore)
-----------------------:down[A,0]
(> 1 3) --> ()
(1+ 1) --> 2
(tcons 1 ()) --> (1)
(set-cdr! (ignore) (1)) --> #t
-----------------------:down[A,1]
(> 2 3) --> ()
(1+ 2) --> 3
(tcons 2 ()) --> (2)
(set-cdr! (1) (2)) --> #t
-----------------------:down[A,2]
(> 3 3) --> ()
(1+ 3) --> 4
(tcons 3 ()) --> (3)
(set-cdr! (2) (3)) --> #t
-----------------------:down[A,3]
(> 4 3) --> #t
(cdr (ignore 1 2 3)) --> (1 2 3)
; Value: (1 2 3)
```

Figure 6.6: A `cdr`-bashing list accumulator, `down-cons!`, along with a sample execution trace.

```
(define (nth-down-scan num init combiner list)
  (down-cons! ((nth-down-scanQ num init combiner) (splay-list list))))

(define (nth-down-scanQ num init combiner)
  (repeated (lambda (synq) (down-scanQ init combiner synq))
            num))

(define (nth-up-scan num init combiner list)
  (up-cons ((nth-up-scanQ num init combiner) (splay-list list))))

(define (nth-up-scanQ num init combiner)
  (repeated (lambda (synq) (up-scanQ init combiner synq))
            num))

(define (repeated f n)
  (if (= n 0)
      identity
      (compose f (repeated f (- n 1)))))

(define (compose f g)
  (lambda (x) (f (g x))))

(define (identity x) x)
```

Figure 6.7: Sliver-based code for computing the nth iterated down and up scans of a list.

iterative computation. For `nth-up-scan`, the *up* orientation of the accumulation operations requires an *up* list collector (`up-cons`).

6.1.2 Expressive Power

Before we investigate more advanced synquence features, it is worthwhile to step back for a moment and comment on how slivers and synquences enrich the expressive power of a language. Consider the sliver implementations of some standard list utilities (see Figure 6.10). It may seem somewhat odd to convert lists into synquences just to perform some simple synquence manipulations. And in fact, in straightforward implementations of slivers, synquence manipulation would incur sufficient overhead to discourage programmers from actually writing the list utilities as indicated in Figure 6.10. However, as I will argue later, it is likely that techniques similar to those used by Waters in his series package [Wat91] can be used to compile such definitions into efficient code. More important, the kinds of decompositions shown in Figure 6.10 have many expressiveness advantages that transcend issues of efficiency:

- Synquences are a common currency in which to express a wide range of linear iterations and recursions. Using a standard collection of synquence slivers in conjunction with conversion routines makes it unnecessary to reimplement common functionality for different datatypes. For example, mapping a function over a vector can be expressed by sandwiching `mapQ` between vector analogs of `splay-list` and `up-cons`.

- Synchronized lazy lists are a more suitable common currency than other standard linear structures (lists, vectors, files, lazy lists) for expressing operational notions like lock step processing and the distinction between iteration and recursion. The synchronized nature of synquences guarantees that each of the modular procedures defined in Figure 6.10 behaves like the standard monolithic implementations. Furthermore, the use of `downQ` in the definitions of `reverse` and `reverse!` indicates that these generate iterative processes, while the `upQ` in the other definitions indicates recursive processes.

- Sliver-based decompositions highlight the structural similarities and differences between procedures. For example, Figure 6.10 suggests insights like the following:

```
OPERA> (nth-down-scan 4 0 + '(1 20 300))
(= 4 0) --> ()
(- 4 1) --> 3
(= 3 0) --> ()
(- 3 1) --> 2
(= 2 0) --> ()
(- 2 1) --> 1
(= 1 0) --> ()
(- 1 1) --> 0
(= 0 0) --> #t
(tcons ignore ()) --> (ignore)
----------------------:down[A,0]
(null? (1 20 300)) --> ()
(car (1 20 300)) --> 1
(cdr (1 20 300)) --> (20 300)
(+ 1 0) --> 1
(+ 1 0) --> 1
(+ 1 0) --> 1
(+ 1 0) --> 1
(tcons 1 ()) --> (1)
(set-cdr! (ignore) (1)) --> #t
----------------------:down[A,1]
(null? (20 300)) --> ()
(car (20 300)) --> 20
(+ 20 1) --> 21
(+ 21 1) --> 22
(+ 22 1) --> 23
(+ 23 1) --> 24
(cdr (20 300)) --> (300)
(tcons 24 ()) --> (24)
(set-cdr! (1) (24)) --> #t
----------------------:down[A,2]
(null? (300)) --> ()
(car (300)) --> 300
(+ 300 21) --> 321
(+ 321 22) --> 343
(+ 343 23) --> 366
(+ 366 24) --> 390
(cdr (300)) --> ()
(tcons 390 ()) --> (390)
(set-cdr! (24) (390)) --> #t
----------------------:down[A,3]
(null? ()) --> #t
(cdr (ignore 1 24 390)) --> (1 24 390)
; Value: (1 24 390)
```

Figure 6.8: Sample trace of the fourth additive iterative scan of the list `(1 20 300)`.

```
OPERA> (nth-up-scan 4 0 + '(1 20 300))
(= 4 0) --> ()
(- 4 1) --> 3
(= 3 0) --> ()
(- 3 1) --> 2
(= 2 0) --> ()
(- 2 1) --> 1
(= 1 0) --> ()
(- 1 1) --> 0
(= 0 0) --> #t
-----------------------:down[A,0]
(null? (1 20 300)) --> ()
(cdr (1 20 300)) --> (20 300)
(car (1 20 300)) --> 1
-----------------------:down[A,1]
(null? (20 300)) --> ()
(cdr (20 300)) --> (300)
(car (20 300)) --> 20
-----------------------:down[A,2]
(null? (300)) --> ()
(cdr (300)) --> ()
(car (300)) --> 300
-----------------------:down[A,3]
(null? ()) --> #t
-----------------------:up[A,3]
(+ 300 0) --> 300
(+ 300 0) --> 300
(+ 300 0) --> 300
(+ 300 0) --> 300
(tcons 300 ()) --> (300)
-----------------------:up[A,2]
(+ 20 300) --> 320
(+ 320 300) --> 620
(+ 620 300) --> 920
(+ 920 300) --> 1220
(tcons 1220 (300)) --> (1220 300)
-----------------------:up[A,1]
(+ 1 320) --> 321
(+ 321 620) --> 941
(+ 941 920) --> 1861
(+ 1861 1220) --> 3081
(tcons 3081 (1220 300)) --> (3081 1220 300)
-----------------------:up[A,0]
; Value: (3081 1220 300)
```

Figure 6.9: Sample trace of the fourth additive iterative scan of the list `(1 20 300)`.

```
;;; A recursive computation of the length of a list
(define (length lst)
  (upQ 0
       (lambda (ignore count) (1+ count))
       (splay-list lst)))

;;; A recursive computation that returns a list containing (in order) the
;;; elements of the two given lists.  LST1 is copied; LST2 is not.
(define (append lst1 lst2)
  (upQ lst2 tcons (splay-list lst1)))

;;; A recursive computation returning a new list with the same elements
;;; as the given one.
(define (copy lst)
  (up-cons (splay-list lst)))

;;; An iterative computation that returns a new list whose
;;; elements are in reverse order from the given list
(define (reverse lst)
  (down-cons (splay-list lst)))

;;; An iterative computation that performs a destructive
;;; reversal of the pairs in a given list.
(define (reverse! lst)
  (downQ '()
         (lambda (pair prev)
           (set-cdr! pair prev)
           pair)
        (splay-sublists lst)))

;;; An iterative computation that returns the last pair of LST; or nil
;;; if it's empty
(define (last-pair lst)
  (downQ '()
         (lambda (pair prev) pair)
         (splay-sublists lst)))

;;; A recursive computation that returns a new list each of whose elements
;;; is the result of applying F to the corresponding element of the original.
(define (map f lst)
  (up-cons (mapQ f (splay-list lst))))

;;; A recursive computation that returns a new list each of whose elements
;;; is the result of applying F to the corresponding sublist of the original.
(define (map-list f lst)
  (up-cons (mapQ f (splay-sublists lst))))
```

Figure 6.10: Expressing some standard list utilities in terms of slivers.

  - `Append`, `copy`, and `reverse` are all cons accumulations on a list; `copy` differs from `reverse` only in the direction of accumulation, while it differs from `append` only in the initial value.
  - `Reverse!` and `last-pair` procedures are surprisingly similar; both iterate over the sublists of the given list and return the last pair, but `reverse!` additionally resets `cdr` pointers on the way down.
  - `Map` can naturally be viewed as a generalization of `copy`.
  - `Map` and `map-list` differ only in the choice of generator.

- Expressing procedures in terms of slivers suggests alternate ways to implement a procedure. For example, various classes of `up` accumulations can be changed to `down` accumulations. For any binary accumulator f such that $(fa(fbc)) = (fb(fac))$ for all a, b, and c, it is always possible to replace `upQ` by `downQ`. In the `length` procedure, the function denoted by `(lambda (elt count) (1+ count))` satisfies this condition, so `length` can be changed to an iterative procedure by substituting `downQ` for `upQ`. Another valid recursive to iterative transformation is to use the `cdr`-bashing trick to replace `(upQ` *init* `tcons` *synq*`)` by `(down-cons!` *init* *synq*`)`. This technique could be used to make iterative versions of `append`, `copy`, `map`, and `map-list`. A host of other transformations from the literature on program transformation [DR76, Dar82, Bac78, Bel86, Bir89a, Bir86, Bir88, Coh83, Str71, WS73, Pet84] find a convenient expression in terms of slivers.

The above points indicate that, regardless of their appropriateness as an implementation language, slivers are a powerful way to design, specify, reason about, and explain programs. They form a basis of "subprocedural particles" into which many common procedures can be easily decomposed; as such, they have excellent prospects for reusability. Although other aggregate structures share many of these advantages, slivers additionally allow the programmer to express finer-grained control over the resultant computation when it's important to do so. In this sense, it is possible to view the definitions in Figure 6.10 as specifications for the operational behavior of the procedures; that they are executable specifications is an added bonus.

6.1.3 Laziness

In the case of synquences, the 'lazy" in "synchronized lazy aggregates" means that neither the element nor the tail of a synquence is computed until it is actually needed. (In comparison, Scheme streams have lazy tails but not lazy elements.) Additionally, both elements and tails are *memoized* [ASS85, Hug85] so that they are only computed once if they are computed at all.

Laziness leads to two important features of synquences:

1. Synquences may be conceptually infinite.

2. An element is not computed if it never used.

Infinite Synquences

An easy way to generate an infinite synquence is to use `genQ` with a *done?* predicate that is never true. The following sliver, which produces the successive powers of a base (starting with the zeroth power), is based on this idea:

```
(define (powers base)
  (genQ 1
        (lambda (prev) (* prev base))
        (lambda (prev) #f)))
```

For example, the expression `(powers 3)` produces a conceptually infinite synquence with elements 1, 3, 9, 27, 81, ...

In order to observe the initial elements of an infinite synquence, it is handy to define a `down-print` sliver that prints out the successive elements of a synquence:

```
(define (down-print synq)
  (for-eachQ output synq))

(define (for-eachQ proc synq)
  (downQ 'ignore
         (lambda (current ignore) (proc current))
         synq))
```

`For-eachQ` is a higher-order sliver for iteratively performing a given procedure `proc` on every element of a synquence. `Output` is an OPERA primitive for printing a given element on a separate line preceded with `OUTPUT:`. Here is a sample use of `down-print` on the powers-of-three synquence:

```
OPERA> (down-print (powers 3))
-----------------------:down[A,0]
(* 1 3) --> 3
OUTPUT: 1
-----------------------:down[A,1]
OUTPUT: 3
(* 3 3) --> 9
-----------------------:down[A,2]
OUTPUT: 9
(* 9 3) --> 27
-----------------------:down[A,3]
OUTPUT: 27
(* 27 3) --> 81
-----------------------:down[A,4]
(* 81 3) --> 243
OUTPUT: 81
-----------------------:down[A,5]
(* 243 3) --> 729
           ⋮
```

Within each episode, the current power is printed and the next one is computed (in some arbitrary order).

Most practical manipulation of infinite synquences requires some means of truncating the synquence or selecting elements from it. Here are some simple examples of infinite synquence manipulation using some of the truncaters and selectors defined in Figures 6.1 and 6.2 (tracing has been turned off in these examples):

```
OPERA> (down-+ (truncateQe (lambda (n) (> n 10)) (powers 3)))
; Value: 13

OPERA> (down-+ (truncateQi (lambda (n) (> n 10)) (powers 3)))
; Value: 40

OPERA> (down-+ (prefixQ 4 (powers 2)))
; Value: 15

OPERA> (down-nthQ 10 (powers 2))
; Value: 1024
```

Infinite synquences can help us use the `nth-down-scanQ` procedure from Section 6.1.1 to compute Pascal's triangle. As depicted in Figure 6.11, the elements of Pascal's triangle can be generated as the intermediate results of iteratively scanning an infinite sequence of ones. Here is a procedure that produces the element at a given position in the array of elements suggested by the figure:

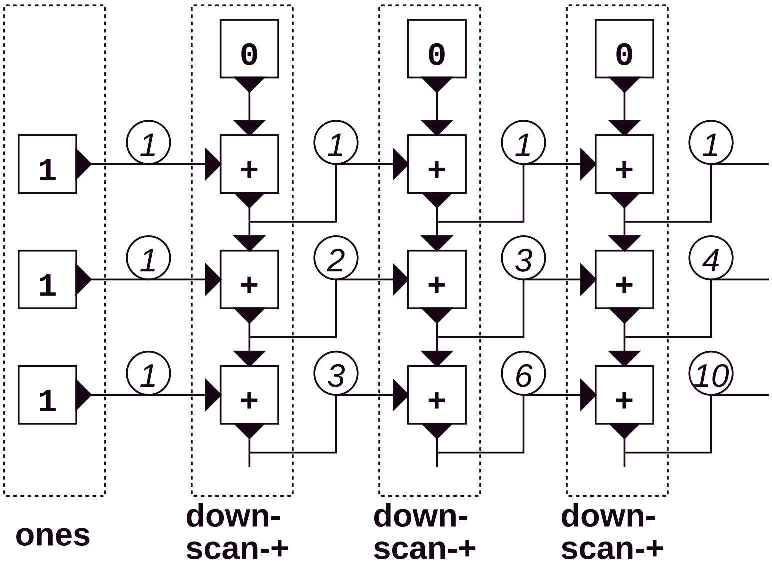


Figure 6.11: One way to organize the computation of the elements of Pascal's triangle. The circled numbers are the computed elements.

```
(define (pascal row column)
  ;; ROW and COLUMN are zero-based indices
  (down-nthQ row
           ((nth-down-scanQ column 0 +) (ones))))

(define (ones)
  (genQ 1 (lambda (x) x) (lambda (x) #f)))
```

Lazy Elements

The second feature of laziness (lazy elements) is more subtle. The computation of an element is delayed until it is needed; if it is never required, it is never computed.

Consider the `contrived-sum` sliver in Figure 6.12, which collects the current element if the accumulated value is even but ignores it (and adds 1) if the accumulated value is odd. As shown in the accompanying trace, the squaring operation is not performed on the even numbers generated by `(to-1 5)` because that value is not required by the accumulator. Although the savings is minimal in this case, it can be much larger if the mapped function is expensive to compute.

Lazy elements serve the goal of operational faithfulness by modelling the kinds of con-

```
(define (contrived-sum synq)
  (downQ 0
         (lambda (current sum)
           (if (even? sum)
               (+ current sum)
               (1+ sum)))
         synq))
```

```
OPERA> (contrived-sum (mapq square (to-1 5)))
-----------------------:down[A,0]
(zero? 5) --> ()
(-1+ 5) --> 4
(square 5) --> 25
(+ 25 0) --> 25
-----------------------:down[A,1]
(zero? 4) --> ()
(-1+ 4) --> 3
(1+ 25) --> 26
-----------------------:down[A,2]
(zero? 3) --> ()
(-1+ 3) --> 2
(square 3) --> 9
(+ 9 26) --> 35
-----------------------:down[A,3]
(zero? 2) --> ()
(-1+ 2) --> 1
(1+ 35) --> 36
-----------------------:down[A,4]
(zero? 1) --> ()
(-1+ 1) --> 0
(square 1) --> 1
(+ 1 36) --> 37
-----------------------:down[A,5]
(zero? 0) --> #t
; Value: 37
```

Figure 6.12: An example illustrating the laziness of sliver elements. Note that `square` is not performed on even elements.

ditional computations found in monolithic recursions. The above example corresponds to the following monolithic procedure:

```
(define (contrived-sum-mono n)
  (define (loop num sum)
    (if (zero? num)
        sum
        (if (even? sum)
            (loop (-1+ num) (+ (square num) sum))
            (loop (-1+ num) (1+ sum)))))
  (loop n sum))
```

In this procedure, the evaluation of `(square num)` is controlled by the `(even? sum)` test. In conjunction with a demand-driven model of evaluation, lazy elements make it possible to express this conditionalization in a modular fashion.

Sometimes it is desirable to override the default laziness of elements. Consider what happens if we collect the results of a map using `cons`:

```
OPERA> (downQ '() cons (mapQ square (to-1 2)))
-----------------------:down[A,0]
(zero? 2) --> ()
(-1+ 2) --> 1
(cons lazon:50 ()) --> (lazon:50)
-----------------------:down[A,1]
(zero? 1) --> ()
(-1+ 1) --> 0
(cons lazon:51 (lazon:50)) --> (lazon:51 lazon:50)
-----------------------:down[A,2]
(zero? 0) --> #t
; Value: (lazon:51 lazon:50)
```

In OPERA, `cons` does not require the values of its operands. If an operand to `cons` is a lazy element, it appears as a *lazon* object that represents the delayed computation. Each of the lazons in the above example represents a suspended squaring computation. If our goal is to model a loop in which a squaring is performed each iteration, then the above computation fails to meet this goal.

The computation suspended within a lazon is initiated by *touching* the lazon. A lazon is automatically touched if it appears in a context that requires the value of the suspended computation.[9] The `touch` procedure can be used to explicitly touch a lazon. Here is

[9] Automatic touching distinguishes lazons from Scheme's delayed objects [ASS85], whose suspended computations can only be explicitly touched via the `force` procedure.

a modified version of the list collection example in which the first argument to `cons` is explicitly touched:

```
OPERA> (downQ '()
              (lambda (elt lst)
                (cons (touch elt) lst))
              (mapQ square (to-1 2)))
-----------------------:down[A,0]
(zero? 2) --> ()
(-1+ 2) --> 1
(square 2) --> 4
(cons 4 ()) --> (4)
-----------------------:down[A,1]
(zero? 1) --> ()
(-1+ 1) --> 0
(square 1) --> 1
(cons 1 (4)) --> (1 4)
-----------------------:down[A,2]
(zero? 0) --> #t
; Value: (1 4)
```

Because of the `touch`, the squarings are performed in lock step with the creation of the resulting list. The `tcons` procedure introduced in Section 6.1.1 is just a synonym for the `lambda` expression in the above example.

We emphasize that the synquence processing model uses laziness and strictness to handle different kinds of communication between program parts. Communication of arguments and results across the barrier between a sliver call and its subcalls is strict by default. This strictness models the behavior of strict call within a monolithic recursion. Communication of elements between one sliver and another is lazy by default to model the demand-driven computation associated with one layer of a monolithic recursion. In the terminology of Chapter 5, argument/result communication is strict, while consump/product communication is lazy (see Figure 5.4). Since these default mechanisms are not appropriate for every situation, there are mechanisms for overriding the defaults.

6.1.4 Fan-in

The slivers examined up to this point take at most one input synquence. Many situations require slivers that accept multiple input synquences. Such slivers are said to exhibit *fan-in*. An inner product sliver that computes the sum of the pairwise products of two synquences is an example of a sliver with fan-in. According to the barrion preservation requirement (see Section 5.4.2), a sliver with two synquence inputs must somehow combine each pair of corresponding input barrions into a single output barrion. None of the slivers introduced so far has this capability.

Fortunately, a single higher-order sliver is able to handle most forms of fan-in. This sliver, `map2Q`, is just a generalization of `mapQ` that applies a given binary function to the respective elements of *two* input synquences. The length of the result is the length of the shorter input synquence. This property is particularly helpful when one synquence is finite but the other is infinite.

With `map2Q`, an inner product sliver can be defined as:

```
(define (inner-product synq1 synq2)
  (down-+ (map2Q * synq1 synq2)))
```

Figure 6.13 shows an execution trace for an application of `inner-product` in which the first argument is finite and the second is infinite. The unmatched elements of the second synquence are ignored because of the truncation property of `map2Q`.

Even in the presence of fan-in, each epsiode is still framed by a single pair of down barriers. Each of the two generators (`splay-list` and `powers`) creates independent barrions for each synchronization event, but `map2Q` unifies these into a single barrion. The lock step processing of the slivers in the inner product example crucially depends on the sharing of synchronization information entailed by barrion propagation and unification.

Like `mapQ`, `map2Q` has an *across* shape because its behavior is determined by context. In the `inner-product` example, `map2Q` participates in an iteration, but it works in other configurations as well. For example, Figure 6.14 illustrates how the mapped function is forced to happen in the up phase when `map2Q` is wedged between two *up* scanners and an *up* accumulator. It is important to note that truncation property of `map2Q` limits the length of the result synquence, but does not prevent the extra *up* processing that must be done on the longer input synquence. In a monolithic recursion, the same behavior could be obtained by breaking the computation up into two stages: one in which the corresponding elements of both inputs are processed, and one in which the "overflow" elements of the second inputs are processed. The sliver approach is a decidedly simpler way of expressing this sort of complex, staged recursion.

```
OPERA> (inner-product (splay-list '(2 3 4)) (powers 10))
-----------------------:down[A,0]
(null? (2 3 4)) --> ()
(* 1 10) --> 10
(car (2 3 4)) --> 2
(* 2 1) --> 2
(+ 2 0) --> 2
(cdr (2 3 4)) --> (3 4)
-----------------------:down[A,1]
(null? (3 4)) --> ()
(* 10 10) --> 100
(cdr (3 4)) --> (4)
(car (3 4)) --> 3
(* 3 10) --> 30
(+ 30 2) --> 32
-----------------------:down[A,2]
(null? (4)) --> ()
(* 100 10) --> 1000
(car (4)) --> 4
(* 4 100) --> 400
(+ 400 32) --> 432
(cdr (4)) --> ()
-----------------------:down[A,3]
(null? ()) --> #t
; Value: 432
```

Figure 6.13: Sample execution trace of `list-inner-product`, a procedure that obtains fan-in via `map2Q`.

```
OPERA> (up-cons (map2Q max
                       (up-scanQ 0 + (to-1 4))
                       (up-scanQ 1 * (splay-list '(7 5)))))
-----------------------:down[A,0]
(null? (7 5)) --> ()
(zero? 4) --> ()
(-1+ 4) --> 3
(cdr (7 5)) --> (5)
(car (7 5)) --> 7
-----------------------:down[A,1]
(zero? 3) --> ()
(null? (5)) --> ()
(-1+ 3) --> 2
(car (5)) --> 5
(cdr (5)) --> ()
-----------------------:down[A,2]
(null? ()) --> #t
(zero? 2) --> ()
(-1+ 2) --> 1
-----------------------:down[A,3]
(zero? 1) --> ()
(-1+ 1) --> 0
-----------------------:down[A,4]
(zero? 0) --> #t
-----------------------:up[A,4]
(+ 1 0) --> 1
-----------------------:up[A,3]
(+ 2 1) --> 3
-----------------------:up[A,2]
(+ 3 3) --> 6
(* 5 1) --> 5
(max 6 5) --> 6
(tcons 6 ()) --> (6)
-----------------------:up[A,1]
(+ 4 6) --> 10
(* 7 5) --> 35
(max 10 35) --> 35
(tcons 35 (6)) --> (35 6)
-----------------------:up[A,0]
; Value: (35 6)
```

Figure 6.14: An example underscoring the *across* nature of `map2Q`. It is challenging to write a monolithic recursion that exhibits this behavior.

```
;;; Return a synquence of pairs of the corresponding elements of inputs.
;;; This is handy for constructing other slivers with fan-in.
(define (zipQ synq1 synq2)
  (map2Q cons synq1 synq2))

;;; Mapper on three input synquences.
(define (map3Q f synq1 synq2 synq3)
  (map2Q (lambda (elt1 elt2&elt3)
           (f elt1 (car elt2&elt3) (cdr elt2&elt3)))
         synq1
         (zipQ cons synq2 synq3)))

;;; Iterative accumulator over two input synquences. FUN is a ternary function
;;; that maps two elements and an accumulator value to a new accumulator value.
(define (down2Q fun synq1 synq2)
  (downQ (lambda (pair acc)
           (f (car pair) (cdr pair) acc))
         (zipQ synq1 synq2)))
```

Figure 6.15: Many slivers that exhibit fan-in can be defined in terms of `map2Q`.

Figure 6.15 shows how `map2Q` can be used to construct some other slivers that accept multiple synquence arguments. With the `down2Q` sliver defined there, `inner-product` could be rewritten as:

```
(define (inner-product synq1 synq2)
  (down2Q 0
          (lambda (elt1 elt2 sum) (+ (* elt1 elt2) sum))
          synq1
          synq2))
```

Not all slivers exhibiting fan-in can be defined in terms of `map2Q`. Consider the `appendQ` sliver, whose behavior is exhibited in Figure 6.16. `AppendQ` only begins processing its second input synquence after completely processing the first. Note how the barrion label associated with the synquence generated by `to-1` (`A`) differs from the label associated with synquence generated by `splay-list` (`B`). Because the two synquences are not processed in lock step, the corresponding barrions are never unified. Because `map2Q` processes both inputs in lock step, it cannot be used to define `appendQ` or other slivers with special processing requirements for multiple synquence inputs.

```
OPERA> (down-+ (appendq (to-1 2) (splay-list '(7 5))))
-----------------------:down[A,0]
(zero? 2) --> ()
(-1+ 2) --> 1
(+ 2 0) --> 2
-----------------------:down[A,1]
(zero? 1) --> ()
(-1+ 1) --> 0
(+ 1 2) --> 3
-----------------------:down[A,2]
(zero? 0) --> #t
-----------------------:down[B,0]
(null? (7 5)) --> ()
(cdr (7 5)) --> (5)
(car (7 5)) --> 7
(+ 7 3) --> 10
-----------------------:down[B,1]
(null? (5)) --> ()
(cdr (5)) --> ()
(car (5)) --> 5
(+ 5 10) --> 15
-----------------------:down[B,2]
(null? ()) --> #t
; Value: 15
```

Figure 6.16: A sample trace of `appendQ`.

6.1.5 Fan-out

The lock step behavior and laziness illustrated in many of the examples up to this point is not very remarkable. In most cases, similar behavior could be obtained using standard lazy data structures (e.g., Scheme streams). The advantages of synquences over lazy lists are best illustrated by sliver programs that exhibit fan-out. In this case, fan-out means either that (1) a single synquence produced by one sliver is consumed by multiple slivers elsewhere in a sliver network or (2) a sliver produces more than one synquence.

A simple example of fan-out is a program that iteratively computes the average of a list of numbers (see the sliver diagram in Figure 5.18 on page 5.18). Each of the three slivers maintains one state variable of the iteration (current list, current sum, and current count). The diagram exhibits fan-out because the synquence output of `SPLAY-LIST` is use as an input to both `DOWN-SUM` and `DOWN-COUNT`.

A Synapse program for the list averaging problem, along with a sample execution trace, appears in Figure 6.17. The trace, which corresonds to the snapshot movie in Figure 5.19, clearly shows that the network as a whole behaves like a monolithic iteration. In contrast, a version using lazy lists rather than synquences could build up space linear in the size of the input list. The synchronization information propagated by slags guarantees that all the slivers work in concert; one cannot race ahead or lag behind the others. In this respect, synquences resemble interprocess communication channels with bounded buffering. Waters [Wat91] and Hughes [Hug84] also extend the aggregate data approach with forms of synchronization in order to achieve this effect.

One advantage of slivers over the channel-based approaches and the other aggregate data extensions is the ease with which recursion is handled in networks exhibiting fan-out. (A more important advantage is the handling of tree-structured compuations, to be discussed in Section 6.2.) For example, the list-averaging example can be turned into a recursive process by changing one or both of the accumulating slivers to have *up* shape. Figure 6.18 contains an example that corresponds to the snapshot movie in Figure 5.20. The up barriers guarantee that the accumulations of the *up* phase proceed in lock step. This accurately models the behavior of a monolithic recursion that takes a list and returns

```
(define (average-list-iter lst)
  (let ((nums (splay-list lst)))
    (/ (down-+ nums)
       (down-length nums))))

; DOWN-+ as before

(define (down-length synq)
  (downQ 0
         (lambda (ignore count) (1+ count))
         synq))

OPERA> (average-list-iter '(7 2 6))
-----------------------:down[A,0]
(null? (7 2 6)) --> ()
(cdr (7 2 6)) --> (2 6)
(1+ 0) --> 1
(car (7 2 6)) --> 7
(+ 7 0) --> 7
-----------------------:down[A,1]
(null? (2 6)) --> ()
(cdr (2 6)) --> (6)
(car (2 6)) --> 2
(+ 2 7) --> 9
(1+ 1) --> 2
-----------------------:down[A,2]
(null? (6)) --> ()
(cdr (6)) --> ()
(car (6)) --> 6
(+ 6 9) --> 15
(1+ 2) --> 3
-----------------------:down[A,3]
(null? ()) --> #t
(/ 15 3) --> 5
; Value: 5
```

Figure 6.17: An iterative list averaging program and sample trace. The trace corresponds to the snapshot movie in Figure 5.19.

both its sum and its length.[10]

The lock step behavior of slivers extends to any shape-compatible network of slivers arranged in a directed acyclic graph (DAG). Figure 6.19 shows a complex network exhibiting fan-in, fan-out, and undirected cycles (i.e., slivers between which there are multiple paths). The corresponding trace in Figure 6.20 illustrates the iterative behavior of the network.

6.1.6 Deadlock

In all of the examples presented above, slivers networks behaved like monolithic recursions only because the slivers were carefully connected in shape-compatible ways. But not all arrangements of slivers are shape-compatible. Consider the following example, which is based on the arrangement shown earlier in Figure 5.10:

```
OPERA> (down-+ (mapQ square (up-scanQ 1 * (splay-list '(7 2 6)))))
----------------------:down[A,0]
(null? (7 2 6)) --> ()
(car (7 2 6)) --> 7
(cdr (7 2 6)) --> (2 6)
++++++++++++++++++++++++++++++++
DEADLOCK! -- pcall:38
++++++++++++++++++++++++++++++++
```

The `down-+` sliver requires synquence elements in the *down* phase of the computation, but the `upQ` sliver does not produce any elements until the *up* phase. The upshot is that computation reaches a deadlock state from which no progress can be made. In the above trace, this state is marked by the `DEADLOCK!` indicator. (The information following the indicator is debugging information that we will ignore.)

SYNAPSE employs a dynamic deadlock detection strategy. This means that a computation may be able to make some progress before deadlock is signalled. In the above example, the `splay-list` generator is able to perform some list operations before a deadlock state is reached. Dynamic deadlock detection also means that a program may deadlock for some inputs but not for others. If we make the input list empty in the above example, the resulting computation will terminate normally because the circular dependency is never discovered:

[10] Lock step processing in the *up* phase is not always desirable. It is possible to design slivers that synchronize in the *down* phase but not in the *up* phase, but I do not include examples of such slivers in this report.

```
(define (average-list-rec lst)
  (let ((nums (splay-list lst)))
    (/ (up-+ nums)
       (up-length nums))))

(define (up-+ nums)
  (upQ 0 + nums))

(define (up-length nums)
  (upQ 0
       (lambda (ignore count) (1+ count))
       synq))

OPERA> (average-list-rec '(7 2 6))
-----------------------:down[A,0]
(null? (7 2 6)) --> ()
(cdr (7 2 6)) --> (2 6)
(car (7 2 6)) --> 7
-----------------------:down[A,1]
(null? (2 6)) --> ()
(cdr (2 6)) --> (6)
(car (2 6)) --> 2
-----------------------:down[A,2]
(null? (6)) --> ()
(car (6)) --> 6
(cdr (6)) --> ()
-----------------------:down[A,3]
(null? ()) --> #t
-----------------------:up[A,3]
(1+ 0) --> 1
(+ 6 0) --> 6
-----------------------:up[A,2]
(+ 2 6) --> 8
(1+ 1) --> 2
-----------------------:up[A,1]
(+ 7 8) --> 15
(1+ 2) --> 3
-----------------------:up[A,0]
(/ 15 3) --> 5
; Value: 5
```

Figure 6.18: A recursive list averaging program and sample trace. The trace corresponds to the snapshot movie in figure 5.20.

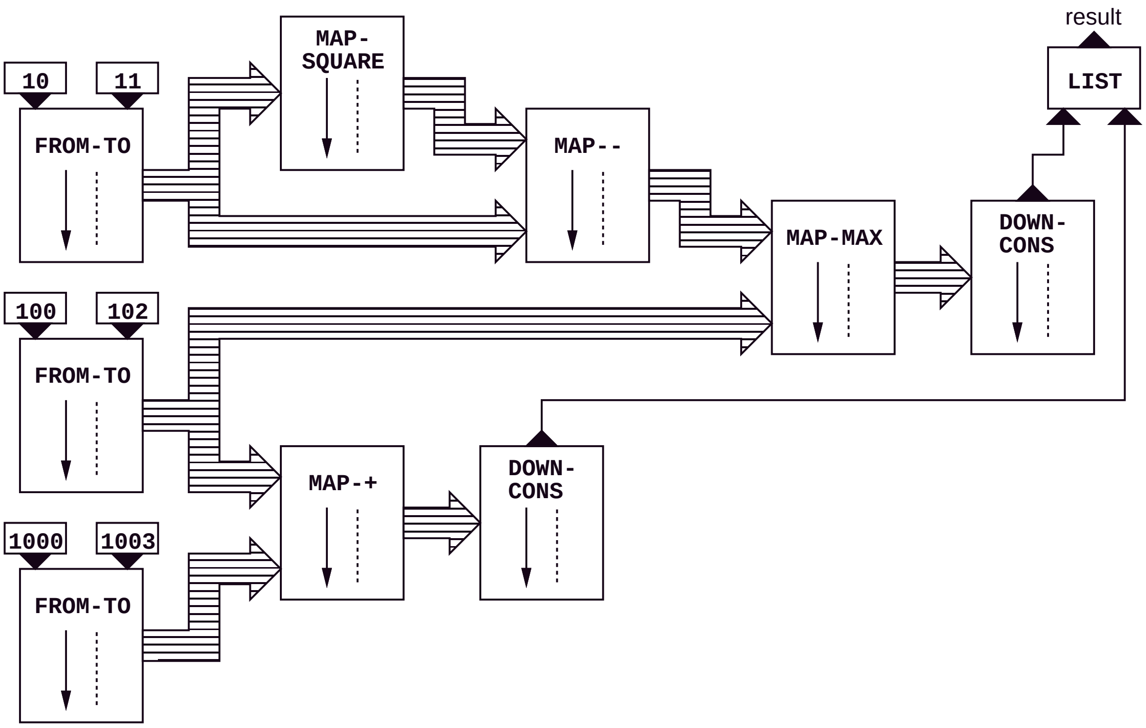


Figure 6.19: Complex sliver network exhibiting fan-in, fan-out, and undirected cycles. Since all the components are iterative, the network will generate an iterative computation.

```
OPERA> (let ((nums1 (from-to 10 11))
             (nums2 (from-to 100 102))
             (nums3 (from-to 1000 1003)))
         (list (down-cons (map2Q max
                                 (map2Q -
                                        (mapQ square nums1)
                                        nums1)
                                 nums2))
               (down-cons (map2Q + nums2 nums3))))
-----------------------:down[A,0]
(> 100 102) --> ()
(> 1000 1003) --> ()
(> 10 11) --> ()
(1+ 1000) --> 1001
(1+ 100) --> 101
(+ 100 1000) --> 1100
(tcons 1100 ()) --> (1100)
(1+ 10) --> 11
(square 10) --> 100
(- 100 10) --> 90
(max 90 100) --> 100
(tcons 100 ()) --> (100)
-----------------------:down[A,1]
(> 11 11) --> ()
(> 1001 1003) --> ()
(> 101 102) --> ()
(1+ 101) --> 102
(1+ 1001) --> 1002
(+ 101 1001) --> 1102
(tcons 1102 (1100)) --> (1102 1100)
(1+ 11) --> 12
(square 11) --> 121
(- 121 11) --> 110
(max 110 101) --> 110
(tcons 110 (100)) --> (110 100)
-----------------------:down[A,2]
(> 1002 1003) --> ()
(> 12 11) --> #t
(> 102 102) --> ()
(1+ 102) --> 103
(1+ 1002) --> 1003
(+ 102 1002) --> 1104
(tcons 1104 (1102 1100)) --> (1104 1102 1100)
-----------------------:down[A,3]
(> 1003 1003) --> ()
(> 103 102) --> #t
(cons (1104 1102 1100) ()) --> ((1104 1102 1100))
(cons (110 100) ((1104 1102 1100))) --> ((110 100) (1104 1102 1100))
; Value: ((110 100) (1104 1102 1100))
```

Figure 6.20: Sample trace generated by a computation specified by a complex sliver network.

```
OPERA> (down-+ (mapQ square (up-scanQ 1 * (splay-list '()))))
-----------------------:down[A,0]
(null? ()) --> #t
-----------------------:up[A,0]
; Value: 0
```

From a high-level perspective, the source of the deadlock in the first example above is that the computation defined by any sliver network is only allowed to buffer information in a single stack. In the *down* phase, the computation is certainly capable of buffering the outputs of `splay-list` and stacking the pending `*` operations. But the program additionally requires the *up* phase of the computation to buffer the outputs of the `up-scanQ` sliver and stack the pending `+` and `square` operations. Intuitively, this behavior requires *two* stacks, and cannot be expressed with the single stack that characterizes a monolithic recursion. The deadlock of this example is not a blemish, but is dictated by the goal of operational faithfulness.

Since the order of summation does not matter in the example, deadlock can easily be avoided by using `up-+` rather than `down-+` to accumulate the sum. More generally, however, deadlock cannot be fixed with so simple a change. Consider the following `down-up-scan-+` procedure:

```
(define (down-up-scan-+ lst)
  (down-cons! '()
              (down-scanQ 0 +
                          (up-scanQ 0 +
                                    (splay-list lst)))))
```

A correct result for this procedure requires that the order of the up and down summations are preserved. However, as written, the procedure will clearly deadlock on a non-empty input.

It is almost always possible to circumvent deadlocks in sliver networks by explicitly buffering intermediate data.[11] Here is the definition of a simple buffer that first collects all of the elements of a synquence into a list and then generates a new synquence from that list:

```
(define (list-buffer synq)
  (splay-list (up-cons synq)))
```

[11] Buffering techniques cannot circumvent deadlocks due to *unresolvable circularities* in which the input to a primitive operator fundamentally depends on its output.

Figure 6.21 shows how a `list-buffer` sliver can be used to avoid deadlock by breaking the single lock step component of `down-up-scan-+` into two such components. Each component to an independent lock step processing stage of the computation. The non-deadlocking version of `down-up-scan-+` and a sample trace appear in Figure 6.22.

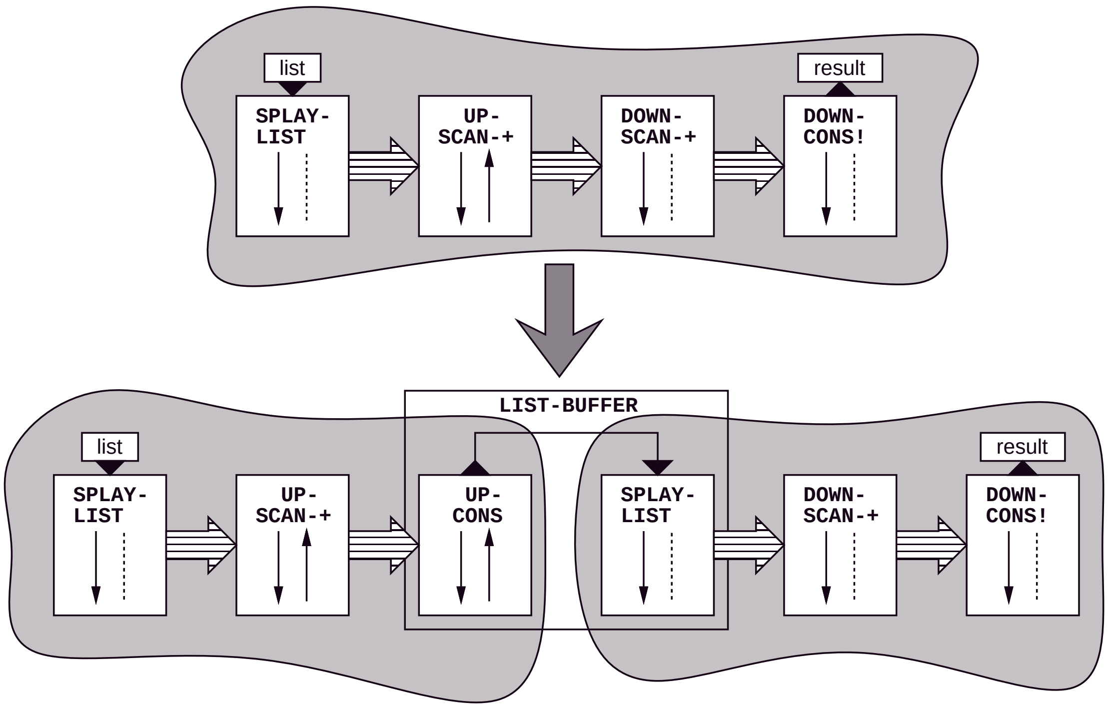


Figure 6.21: A `LIST-BUFFER` sliver can be used to split a single lock step component (shaded blob) into two lock step components.

Specifying explicit buffers may seem annoying, but it is the cost of the space consumption guarantees provided by sliver decomposition. The implicit buffering provided by other aggregate data approaches makes it hard to control the storage profiles of modular programs. Existing list and tree removal techniques (such as listlessness [Wad84, Wad85], deforestation [Wad88, Chi92, GLJ93], and other transformations [Bel86, Bir89a]) automatically remove some intermediate data structures, but they either are limited to restricted network topologies or do not guarantee that all intermediate structures go away. In con-

```
;;; A non-deadlocking version of the DOWN-UP-SCAN-+ presented in the text.
(define (down-up-scan-+ lst)
  (down-cons! '()
              (down-scanQ 0 +
                          (list-buffer
                           (up-scanQ 0 +
                                     (splay-list lst))))))

OPERA> (down-up-scan-+ '(1 20))
-----------------------:down[A,0]
(null? (1 20)) --> ()
(car (1 20)) --> 1
(cdr (1 20)) --> (20)
-----------------------:down[A,1]
(null? (20)) --> ()
(car (20)) --> 20
(cdr (20)) --> ()
-----------------------:down[A,2]
(null? ()) --> #t
-----------------------:up[A,2]
(+ 20 0) --> 20
(tcons 20 ()) --> (20)
-----------------------:up[A,1]
(+ 1 20) --> 21
(tcons 21 (20)) --> (21 20)
-----------------------:up[A,0]
(cons ignore ()) --> (ignore)
-----------------------:down[B,0]
(null? (21 20)) --> ()
(car (21 20)) --> 21
(+ 21 0) --> 21
(cdr (21 20)) --> (20)
(tcons 21 ()) --> (21)
(set-cdr! (ignore) (21)) --> #t
-----------------------:down[B,1]
(null? (20)) --> ()
(car (20)) --> 20
(+ 20 21) --> 41
(cdr (20)) --> ()
(tcons 41 ()) --> (41)
(set-cdr! (21) (41)) --> #t
-----------------------:down[B,2]
(null? ()) --> #t
(cdr (ignore 21 41)) --> (21 41)
; Value: (21 41)
```

Figure 6.22: Deadlock can often be avoided by using explicit buffering. Here `list-buffer` resolves the shape incompatibility between `up-scanQ` and `down-scanQ`.

trast, slivers provide a framework in which programmers manipulate shaped components to avoid or insert buffering. This is an approach similar to that embraced by Waters's series package, except that series only support linear, iterative computations whereas slivers support recursions and tree-shaped computations.

In linear computations, deadlock is typically induced by an attempt to use in the *down* phase a result computed in the *up* phase. But there are other dependencies that can lead to deadlock. For instance, since `appendQ` begins processing its second argument only after it completes processing of its first argument, a deadlock will be induced if the two arguments share synchronization information. The following expression deadlocks because the `to-1` generator can't both immediately produce elements to feed the first argument of `appendQ` but delay producing the same elements until the second argument is needed (see Figure 6.23):

```
(let ((nums (to-1 3)))
  (down-+ (appendQ nums nums)))
```

This deadlock can be removed by wrapping a `list-buffer` around the second argument.

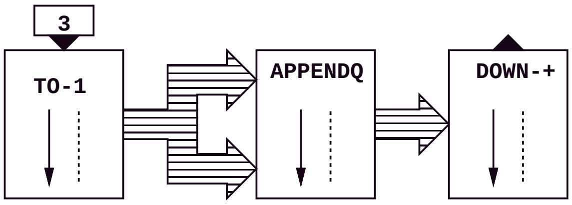


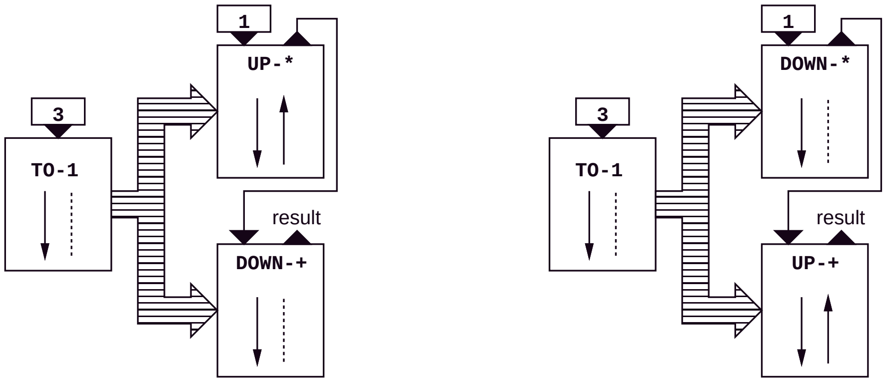
Figure 6.23: An example of deadlock involving `appendQ`.

Another source of deadlock is wiring the non-slag output of one reducer to the non-slag input of another sliver in the same network. Here are two expressions of this sort that give rise to deadlock (see Figure 6.24):

```
(let ((nums (to-1 3)))
  (downQ (upQ 1 *  nums) + nums))

(let ((nums (to-1 3)))
  (upQ (downQ 1 * nums) + nums))
```

Due to the strict nature of call boundaries, both `downQ` and `upQ` require values for all arguments before proceeding. But each example requires the final value of processing a slag to be availble before processing on the slag begins! This is a recipe for deadlock. Explicit buffering can be used to avoid deadlock in both cases.



(a) Non-slag dataflow from *up* to *down*. **(b)** Non-slag dataflow from *down* to *up*.

Figure 6.24: Two examples of deadlock caused by non-slag dataflow.

The deadlocking network in Figure 6.24(**b**) can be fixed by a non-buffering method. Since the initial value of the `upQ` isn't really required until the beginning of the *up* phase, there's no intrinsic reason why that value can't be supplied by a *down* computation (which is guaranteed to return before the *up* phase begins). The deadlock in this example is actually a spurious one caused by the strictness convention of call boundaries. In this case, it is safe to override the convention by introducing explicit non-strictness. Figure 6.25 shows how wrapping an `eagon` around the result of the `downQ` allows the computation to proceed without deadlock. `Eagon` is an OPERA special form that starts computing its argument expression but immediately returns a placeholder for its result.[12] While its value is being computed, the placeholder can be passed into and returned from procedures, and stored into and selected from data structures. Any operation that actually needs to examine the placeholder's value must wait for its computation to complete. Using eagon in the example "fools" the call boundary into "believing" that a value is available when it really isn't. But this is fine, since the placeholder's value isn't required until after the *down* phase is complete, by which time the placeholder value will be computed.

[12] Those familiar with *futures* will recognize that eagon is just another name for future. See the discussion of eagons in Chapter 7.

```
OPERA> (let ((nums (to-1 3)))
         (upQ (eagon (downQ 1 * nums)) + nums))
-----------------------:down[A,0]
(zero? 3) --> ()
(-1+ 3) --> 2
(* 3 1) --> 3
-----------------------:down[A,1]
(zero? 2) --> ()
(-1+ 2) --> 1
(* 2 3) --> 6
-----------------------:down[A,2]
(zero? 1) --> ()
(-1+ 1) --> 0
(* 1 6) --> 6
-----------------------:down[A,3]
(zero? 0) --> #t
-----------------------:up[A,3]
(+ 1 6) --> 7
-----------------------:up[A,2]
(+ 2 7) --> 9
-----------------------:up[A,1]
(+ 3 9) --> 12
-----------------------:up[A,0]
; Value: 12
```

Figure 6.25: OPERA's non-strict `eagon` form can be used to fix a spurious deadlock.

6.1.7 Filtering

Synquence slivers are designed to handle filtering in a reusable manner (see Section 5.5.3). Gaps are introduced into a synquence by the `filterQ` sliver and are propagated by all transducers except for scanners and `reifyQ`.

The following trace shows a simple example of a filtering:

```
OPERA> (downQ 0 + (mapQ square (filterQ even? (to-1 3))))
-----------------------:down[A,1]
(zero? 3) --> ()
(even? 3) --> ()
(-1+ 3) --> 2
-----------------------:down[A,2]
(zero? 2) --> ()
(even? 2) --> #t
(square 2) --> 4
(+ 4 0) --> 4
(-1+ 2) --> 1
-----------------------:down[A,3]
(zero? 1) --> ()
(even? 1) --> ()
(-1+ 1) --> 0
-----------------------:down[A,4]
(zero? 0) --> #t
```

The computation sums the squares of all the even numbers in a given synquence. As indicated by the trace, the mapper and summer ignore the slots associated with the odd inputs.

A scanner fills a gapped slot with the current value of its accumulator:

```
OPERA> (downQ 0 + (down-scanQ 1 max (mapQ square (filterQ even? (to-1 3)))))
-----------------------:down[A,1]
(zero? 3) --> ()
(even? 3) --> ()
(+ 1 0) --> 1
(-1+ 3) --> 2
-----------------------:down[A,2]
(zero? 2) --> ()
(even? 2) --> #t
(square 2) --> 4
(max 4 1) --> 4
(+ 4 1) --> 5
(-1+ 2) --> 1
-----------------------:down[A,3]
(zero? 1) --> ()
(even? 1) --> ()
(+ 4 5) --> 9
(-1+ 1) --> 0
-----------------------:down[A,4]
(zero? 0) --> #t
; Value: 9
```

Although the `max` operation of the scanner is only performed in the second slot, the output of the scanner is a synquence of the elements `1`, `4`, and `4`. These are summed by `downQ` to yield a `9`.

Sometimes it is desirable for a scanner to preserve the gaps of its inputs. This can be accomplished with the help of the handy `preserving-gaps` procedure:

```
(define (preserving-gaps scanner)
  (lambda (init op synq)
    (map2Q (lambda (a b) a)
           (scanner init op synq)
           synq)))
```

`Preserving-gaps` maps a given scanner into a gap-preserving scanner using the gap-preserving properties of the `map2Q` sliver (see Figure 5.22). If this procedure is used in the above example, the synquence consumed by the summer will contain a `4` wedged between two gaps:

```
OPERA> (downQ 0 +
              ((preserving-gaps down-scanQ)
               1
               max
               (mapQ square (filterQ even? (to-1 3)))))
-----------------------:down[A,1]
(zero? 3) --> ()
(even? 3) --> ()
(-1+ 3) --> 2
-----------------------:down[A,2]
(zero? 2) --> ()
(even? 2) --> #t
(square 2) --> 4
(max 4 1) --> 4
(+ 4 0) --> 4
(-1+ 2) --> 1
-----------------------:down[A,3]
(zero? 1) --> ()
(even? 1) --> ()
(-1+ 1) --> 0
-----------------------:down[A,4]
(zero? 0) --> #t
; Value: 4
```

Up accumulators and scanners exhibit the desirable behavior of aggresively testing for gaps in the down direction. In the following trace, there is no pending stack frame corresponding to the the `2` input because a `upQ` effectively makes a tail call for that input:

```
OPERA> (upQ 0 + (filterQ odd? (to-1 3)))
-----------------------:down[A,1]
(zero? 3) --> ()
(odd? 3) --> #t
(-1+ 3) --> 2
-----------------------:down[A,2]
(zero? 2) --> ()
(odd? 2) --> ()
(-1+ 2) --> 1
-----------------------:down[A,3]
(zero? 1) --> ()
(odd? 1) --> #t
(-1+ 1) --> 0
-----------------------:down[A,4]
(zero? 0) --> #t
-----------------------:up[A,4]
(+ 1 0) --> 1
-----------------------:up[A,2]  ; *** Missing frame here
(+ 3 1) --> 4
-----------------------:up[A,1]
; Value: 4
```

This desirable behavior is even exhibited by parallel combinations of slivers that independently filter a shared input. For example, the computation in Figure 6.26 is a parallel combination of a filtered up sum accumulator and a filtered up product accumulator. Note a stack frame is pushed in the resulting computation only if one of the accumulators has a pending operation to perform in that frame. No frame is pushed for slots in which both accumulators process a gap (e.g. the slot for which `to-1` produces `2`).

```
OPERA> (let ((nums (to-1 4)))
         (cons (upQ 0 + (filterQ odd? nums))
               (upQ 1 * (filterQ (lambda (x) (> x 2)) nums))))
-----------------------:down[A,1]
(zero? 4) --> ()
(> 4 2) --> #t
(odd? 4) --> ()
(-1+ 4) --> 3
-----------------------:down[A,2]
(zero? 3) --> ()
(> 3 2) --> #t
(odd? 3) --> #t
(-1+ 3) --> 2
-----------------------:down[A,3]
(zero? 2) --> ()
(odd? 2) --> ()
(> 2 2) --> ()
(-1+ 2) --> 1
-----------------------:down[A,4]
(zero? 1) --> ()
(odd? 1) --> #t
(> 1 2) --> ()
(-1+ 1) --> 0
-----------------------:down[A,5]
(zero? 0) --> #t
-----------------------:up[A,5]
(+ 1 0) --> 1
-----------------------:up[A,3]  ; *** Missing frame here
(+ 3 1) --> 4
(* 3 1) --> 3
-----------------------:up[A,2]
(* 4 3) --> 12
-----------------------:up[A,1]
(cons 4 12) --> (4 . 12)
; Value: (4 . 12)
```

Figure 6.26: Trace illustrating the proper interleaving of independently filtered parallel up computations. Note that there is no up frame for the slot where both filter tests fail.

Despite the apparent aggressive testing in the *down* phase, `upQ` and `up-scanQ` still deal appropriately with gaps that are determined in the *up* phase. Figure 6.27 shows a trace of a computation with filtering in both the *down* and *up* phases (the computation is similar to that of the sliver diagram depicted in Figure 5.26). The `odd?` tests that occur in the *down* phase determined that there are only three stacked frames. The `even?` tests are required to occur in the *up* phase. Only one `*` operation is performed because only one output of the scanner is even.

Achieving these kinds of expected fine-grained operational behavior for filtered synquences is one of the technical triumphs of SYNAPSE. See Section 7.2.4 for an explanation of how it all works.

6.2 Tree Computations

SYNAPSE supports tree computations through a suite of slivers that manipulate syndrites (synchronized lazy trees). These are summarized in Figures 6.28 – 6.30.

Syndrite slivers are natural extensions to the synquence slivers presented in the previous section. Syndrite generators, mappers, and filters are just tree-shaped versions of the corresponding synquence operators. However, syndrite accumulators and scanners support a much richer shape vocabulary than their linear cousins. For example, tree nodes can be processed in parallel either down or up the branches of a tree; or they can be processed in a wide variety of sequential traversals.

We will only consider sequential traversals that follow the same node-processing strategy at each node. Such traversals can be characterized by two properties:

1. The *order* of a sequential traversal specifies when the element of a node is processed relative to the processing of its children. There are three classes of orders:

   (a) *pre-order*: an element of a node is processed before any of its children's elements.

   (b) *in-order*: an element of a node is processed after some of its children's elements but before other of its children's elements.

   (c) *post-order*: an element of a node is processed after all of its children's elements.

```
OPERA> (upQ 1 * (filterQ even? ((preserving-gaps up-scanQ)
                                0
                                +
                                (filterQ odd? (to-1 5)))))
-----------------------:down[A,1]
(zero? 5) --> ()
(odd? 5) --> #t
(-1+ 5) --> 4
-----------------------:down[A,2]
(zero? 4) --> ()
(odd? 4) --> ()
(-1+ 4) --> 3
-----------------------:down[A,3]
(zero? 3) --> ()
(odd? 3) --> #t
(-1+ 3) --> 2
-----------------------:down[A,4]
(zero? 2) --> ()
(odd? 2) --> ()
(-1+ 2) --> 1
-----------------------:down[A,5]
(zero? 1) --> ()
(odd? 1) --> #t
(-1+ 1) --> 0
-----------------------:down[A,6]
(zero? 0) --> #t
-----------------------:up[A,6]
(+ 1 0) --> 1
(even? 1) --> ()
-----------------------:up[A,4]
(+ 3 1) --> 4
(even? 4) --> #t
(* 4 1) --> 4
-----------------------:up[A,2]
(+ 5 4) --> 9
(even? 9) --> ()
-----------------------:up[A,1]
; Value: 4
```

Figure 6.27: Trace illustrating the desirable properties of synquence filtering.

Generators:

(`genDi` *initial-state element&next-states*) [*parallel down*]
: Generates a syndrite whose element values are computed from a "state" that is passed down the tree in a *parallel down* fashion. At every syndrite node, the node element and child states are determined by the *element&next-states* procedure, which takes the current state of the node and a two-argument receiver, and calls the receiver on the desired element and a list of child states. A syndrite branch terminates when the list of child states is empty.

(`genDe` *initial-state element&next-states done?*) [*parallel down*]
: Like `genDi`, except applies *done?* to the node state before calling *element&next-states*. If *done?* returns true, then the current branch is terminated with a gap element; otherwise *element&next-states* is used to compute the element and next states.

Figure 6.28: A summary of SYNAPSE's syndrite generating slivers.

The *pre* and *post* sliver shapes indicate the pre-order and post-order strategies, respectively. The *in* shape of syndrite slivers corresponds to processing a node element after the processing the first child but before processing the rest of the children. This is only one instance of the in-order strategy, but it has the nice property that for binary syndrites it corresponds to the traditional notion of in-order processing on binary trees [Knu73].

2. The *direction* of a sequential traversal specifies the order in which a node's children are processed. If the children are assumed to have a default left-to-right ordering, then the direction can be specified as a permutation on this ordering. Any permutation is allowed, but in our examples, we will only consider the identity permuation (a *left-to-right* traversal) and a reversing permutation (a *right-to-left* traversal).

   In SYNAPSE, the *direction* argument to various sequential transducers and reducers is expected to be a *pair* of procedures: a list permutation and its corresponding inverse permutation. The inverse permutation is used by transducers to reconstruct the original ordering of children after permuting them. Here's how the two standard directions are defined:

Reducers:

(`upD` *op gap-op synd*) [*parallel up*]
: Recursively accumulates the elements of *synd* using *op* and *gap-op*. At every node whose element is not a gap, *op* is applied to the node element and a list of the accumulated results for each child. At a node whose element is a gap, *gap-op* is applied to a list of the accumulated results for each child.

(`preD` *direction init op synd*) [*direction pre*]
: Starting with *init*, uses *op* to accumulate the elements of *synd* in a pre-order walk over the syndrite. A pre-order walk processes the element at a node before processing and children. At every node, *op* is applied to the element and the current accumulated value. *Direction* specifies the direction of the walk (e.g., left-to-right, right-to-left); see the text to see how directions are specified.

(`inD` *direction init op synd*) [*direction in*]
: An in-order sequential accumulation over *synd*. An in-order walk processes the element at a node after processing the first child but before processing the other children.

(`postD` *direction init op synd*) [*direction post*]
: A post-order sequential accumulation over *synd*. A post-order walk processes the element at a node after processing all of the children.

(`up-firstD` *synd*) [*parallel up*]
: Returns the top element of a syndrite. Signals an error if the top element is a gap.

(`pre-lastD` *direction synd*) [*direction pre*]
: Returns the last (non-gap) element processed in a *direction* pre-order walk over *synd*. Signals an error if no element is found.

(`in-lastD` *direction synd*) [*direction in*]
: In-order version of `pre-lastD`.

(`post-lastD` *direction synd*) [*direction post*]
: Post-order version of `pre-lastD`.

(`pre-nthD` *direction index synd*) [*direction pre*]
: Returns the *index*th (non-gap) element processed in a *direction* pre-order walk over *synd*. Signals an error if *synd* does not contain *index* elements.

(`in-nthD` *direction index synd*) [*direction in*]
: In-order version of `pre-nthD`.

(`post-nthD` *direction index synd*) [*direction post*]
: Post-order version of `pre-nthD`.

Figure 6.29: A summary of SYNAPSE's syndrite reducing slivers.

Transducers:

(`mapD` *fun synd*) [*across*]
Returns the syndrite resulting from the elementwise application of *fun* to *synd*.

(`map2D` *fun synd1 synd2*) [*across*]
Returns the syndrite resulting from the elementwise application of *fun* to corresponding elements of *synd1* and *synd2*. An output element is a gap if either input element is a gap.

(`filterD` *pred synd*) [*across*]
Returns a syndrite with the same structure as *synd* in which every element satisfying *pred* is mapped to itself and all other elements are mapped to gaps.

(`reifyD` *obj synd*) [*across*]
Returns a syndrite that maps every gap of *synd* to *obj* and every ungapped element to itself.

(`unreifyD` *obj synd*) [*across*]
Returns a syndrite that maps every instance of *obj* in *synd* to a gap and every other element to itself.

(`down-scanD` *init op synd*) [*parallel down*]
Returns the syndrite of intermediate accumulated values in an iterative accumulation of each branch of *synd* using combiner *op* and initial value *init*.

(`up-scanD` *op gap-op synd*) [*parallel up*]
Returns the syndrite of intermediate accumulated values in a parallel recursive accumulation of *synd* using the combiner *op* at ungapped nodes and the combiner *gap-op* at gapped nodes.

(`pre-scanD` *direction init op synd*) [*direction pre*]
Returns the syndrite of intermediate accumulated values in a sequential *direction* pre-order accumulation of *synd* using combiner *op* and initial value *init*.

(`in-scanD` *direction init op synd*) [*direction in*]
In-order version of `pre-scanD`.

(`post-scanD` *direction init op synd*) [*direction post*]
Post-order version of `pre-scanD`.

(`down-shiftD` *init synd*) [*parallel down*]
Returns a syndrite with the same structure at *synd*, but in which every ungapped element location has the value of the first ungapped element found on the path from the parent of the location to the root. If there is no ungapped element on the path to the root, *init* is used.

Figure 6.30: A summary of SYNAPSE's syndrite transducing slivers.

```
;;; The direction abstraction
(define (make-direction permute unpermute)
  (cons permuter unpermuter))

(define permuter car)
(define unpermuter car)

;;; Left-to-right direction
(define l->r
  (make-direction (lambda (lst) lst) (lambda (lst) lst)))

;;; Right-to-left direction
(define r->l
  (make-direction reverse reverse))
```

6.2.1 Simple Examples

Before investigating applications of the syndrite slivers, we first exercise them in some simple situations. The examples will employ the following routines, which represent one way to convert between s-expressions (parenthesized tree notation) and syndrites:

```
(define (splay-tree top-sexp)
  (genDi top-sexp
         (lambda (sexp return)
           (if (pair? sexp)
               (return (car sexp) (cdr sexp))
               (return sexp '())))))

(define (glom-tree synd)
  (upD (lambda (elt subtrees)
         (if (null? subtrees)
             elt
             (cons elt subtrees)))
       (lambda (subtrees)
         (if (null? subtrees)
             '*gap*
             (cons '*gap* subtrees)))
       synd))
```

Let A stand for an atom (non-pair) and S_i stand for any s-expression. Then `splay-tree` treats A as a syndrite leaf node whose element is A and (S_0 S_1 $\ldots S_n$) as an syndrite intermediate node whose element is S_0 and whose children are the recursive results of converting S_1 $\ldots$ S_n. The element structure of the syndrite resulting from applying `splay-tree` to `(a (b c d) e (f (g h)))` is shown in Figure 6.31. `Glom-tree` performs the inverse conversion, but fills any gapped positions with the symbol `*gap*`.

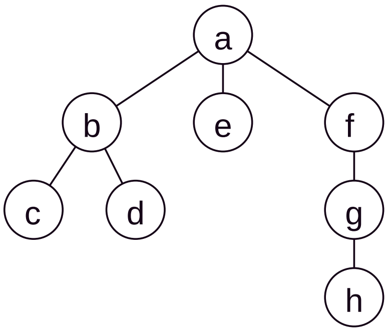


Figure 6.31: The tree corresponding to the syndrite created by `(splay-tree '(a (b c d) e (f (g h))))`.

`Splay-tree` gives one example of the use of a syndrite generator. Here are two more examples, which underscore the difference between `genDi` and `genDe`:

```
(glom-tree (genDi 1
                  (lambda (n return)
                    (if (> n 4)
                        (return n '())
                        (return n (list (* 2 n) (+ 1 (* 2 n))))))))
; Value: (1 (2 (4 8 9) 5) (3 6 7))

(glom-tree (genDe 1
                  (lambda (n return)
                    (return n (list (* 2 n) (+ 1 (* 2 n)))))
                  (lambda (n) (> n 4))))
; Value: (1 (2 (4 *gap* *gap*) *gap*) (3 *gap* *gap*))
```

Both examples produce a breadth index tree (see page 155), in which the number at each node is doubled before being passed to the left subtree, and is doubled and incremented before being passed to the right subtree. The `genDi` version terminates the tree when a number larger than 4 is encountered, but includes the number in the tree. The `genDe` version excludes numbers larger than 4 by putting gaps in their positions.

Figure 6.32 illustrates the different syndrite accumulation strategies by performing `cons` accumulations over the syndrite created by `(splay-tree '(a b c))`. A `cons` accumulation with `upD` is similar to `glom-tree` except that the leaves of the resulting tree are parenthesized. (Also, the `no-gaps` gap accumulator is used to indicate that no gaps are expected in the input.) Left-to-right and right-to-left versions of `preD`, `inD`, and `postD` give all six possible permutations of a three element list. In the sequential accumulation examples, the numbers are listed in the *reverse* order that they are processed because later elements are prepended to the list of elements processed earlier. By using the `cdr`-bashing technique, it is possible for the accumulators to construct a list of the elements in the order visited.

```
OPERA> (define (sample-syndrite) (splay-tree '(a b c)))

OPERA> (define (no-gaps lst)
         (error "NO-GAPS: found a gap where none expected."))

OPERA> (upD cons no-gaps (abc))
; Value: (a (b) (c))

OPERA> (preD l->r '() cons (abc))
; Value: (c b a)

OPERA> (preD r->l '() cons (abc))
; Value: (b c a)

OPERA> (inD l->r '() cons (abc))
; Value: (c a b)

OPERA> (inD r->l '() cons (abc))
; Value: (b a c)

OPERA> (postD l->r '() cons (abc))
; Value: (a c b)

OPERA> (postD r->l '() cons (abc))
; Value: (a b c)
```

Figure 6.32: Examples of different shapes of syndrite accumulation.

Examples of syndrite scanning appear in Figure 6.33. In each case, the result is a tree of intermediate `cons` accumulations. The length of the list in each tree position indicates the order in which the nodes are visited. For example, the result `((a c b) (b) (c b))` indicates that the `b` node was visited first, the `c` node was visited second, and the `a` node was visited last.

Syndrite mapping and filtering operations are *across* operators that simply perform elementwise application:

```
OPERA> (glom-tree (down-scanD '() cons (abc)))
; Value: ((a) (b a) (c a))

OPERA> (glom-tree (up-scanD cons no-gaps (abc)))
; Value: ((a (b) (c)) (b) (c))

OPERA> (glom-tree (pre-scanD l->r '() cons (abc)))
; Value: ((a) (b a) (c b a))

OPERA> (glom-tree (pre-scanD r->l '() cons (abc)))
; Value: ((a) (b c a) (c a))

OPERA> (glom-tree (in-scanD l->r '() cons (abc)))
; Value: ((a b) (b) (c a b))

OPERA> (glom-tree (in-scanD r->l '() cons (abc)))
; Value: ((a c) (b a c) (c))

OPERA> (glom-tree (post-scanD l->r '() cons (abc)))
; Value: ((a c b) (b) (c b))

OPERA> (glom-tree (post-scanD r->l '() cons (abc)))
; Value: ((a b c) (b c) (c))
```

Figure 6.33: Examples of different shapes of syndrite scanning.

```
(define (sample) (splay-tree '(1 (2 3 4) 5 (6 (7 8)))))

OPERA> (glom-tree (mapD square (sample)))
; Value: (1 (4 9 16) 25 (36 (49 64)))

OPERA> (glom-tree (filterD even? (sample)))
; Value: (*gap* (2 *gap* 4) *gap* (6 (*gap* 8)))
```

In an ungapped tree, syndrite shifting moves each node element one level down every branch from the node. In the presence of gaps, an element replaces the next element accessible down each branch:

```
OPERA> (glom-tree (down-shiftD 17 (big-synd)))
; Value: (17 (1 2 2) 1 (1 (6 7)))

OPERA> (glom-tree (down-shiftD 17 (filterD even? (big-synd))))
; Value: (*gap* (17 *gap* 2) *gap* (17 (*gap* 6)))
```

6.2.2 Shape Combinations

Perhaps the most important feature of syndrite slivers is that their process shapes combine in a reasonable way. We will present a series concrete examples of traces to help build intuitions about these combinations.

We consider operations on a simple binary tree:

```
(define (binary) (splay-tree '(1 (2 3 4) (5 6 7))))
```

Figure 6.34 shows sample traces of left-to-right pre-order and post-order accumulations on `binary`.[13] Because of the way the tree is structured, the barrion index labelling a dotted barrier corresponds to the numeric element held by the syndrite with that barrion. For the pre-order case, the barriers indicate that accumulation of element n always happens immediately after the down rendezvous at the barrier labelled n. In the post-order case, the accumulation of element n occurs directly before the up rendezvous at the barrier labelled n.

The barriers also indicate that the pre-order accumulator makes some tail calls while the post-order accumulator does not. In the post-order case, every down barrier has a matching up barrier, signifying a post-order tree walker must return from every call. However, in

[13] The traces only show instances of the `+` and `*` operators.

```
OPERA> (preD l->r 1 * (binary))
-----------------------:down[A,1]
(* 1 1) --> 1
-----------------------:down[A,2]
(* 2 1) --> 2
-----------------------:down[A,3]
(* 3 2) --> 6
-----------------------:up[A,3]
-----------------------:down[A,4]
(* 4 6) --> 24
-----------------------:up[A,2]
-----------------------:down[A,5]
(* 5 24) --> 120
-----------------------:down[A,6]
(* 6 120) --> 720
-----------------------:up[A,6]
-----------------------:down[A,7]
(* 7 720) --> 5040
; Value: 5040

OPERA> (postD l->r 0 + (binary))
-----------------------:down[A,1]
-----------------------:down[A,2]
-----------------------:down[A,3]
(+ 3 0) --> 3
-----------------------:up[A,3]
-----------------------:down[A,4]
(+ 4 3) --> 7
-----------------------:up[A,4]
(+ 2 7) --> 9
-----------------------:up[A,2]
-----------------------:down[A,5]
-----------------------:down[A,6]
(+ 6 9) --> 15
-----------------------:up[A,6]
-----------------------:down[A,7]
(+ 7 15) --> 22
-----------------------:up[A,7]
(+ 5 22) --> 27
-----------------------:up[A,5]
(+ 1 27) --> 28
-----------------------:up[A,1]
; Value: 28
```

Figure 6.34: Traces of left-to-right pre-order and post-order accumulations of a binary tree.

the pre-order case, there are only three up barriers for the seven down barriers. These up barriers correspond to cases where the tree walker must return from a call, while the missing ones indicate calls for which the tree-walker does not return. In particular, the pre-order accumulator returns to top-level immediately after processing 7 because there is no work left to be done. In constrast, the post-order accumulator still needs to pop a stack of two pending accumulations after it processes 7.

Combining the above two processes in parallel retains the character of the individual processes. Figure 6.35 shows the trace of a sample parallel combination. Every operation still appears at the same relative location. The combined process returns from every call because one of the components does. If we instead combine two pre-order accumulators, the result is guaranteed to look like a single pre-order process (Figure 6.36). If we change the directions of the two accumulators to be incompatible, a deadlock results:

```
OPERA> (let ((bin (binary)))
         (cons (preD l->r 1 * bin)
               (postD r->l 0 + bin)))
-----------------------:down[A,1]
(* 1 1) --> 1
++++++++++++++++++++++++++++++++
DEADLOCK! -- prim-cons:52
++++++++++++++++++++++++++++++++
```

Using scanners, different tree walking processes can be combined in series. Figure 6.37 shows the trace of a series combination of a pre-order scanner and a post-order accumulator. The trace inherits its post-order shape from the accumulator. In contrast, a pre-order shape results from the series combination of a pre-order scanner and a pre-order accumulator (Figure 6.38). As expected, incompatible series combinations will lead to deadlock. For example, a pre-order accumulator would require the result of a post-order scanner sooner than the scanner could produce it:

```
OPERA> (preD l->r 1 *
             (post-scanD l->r 0 +
                         (binary)))
-----------------------:down[A,1]
++++++++++++++++++++++++++++++++
DEADLOCK! -- pcall:54
++++++++++++++++++++++++++++++++
```

```
OPERA> (let ((bin (binary)))
         (cons (preD l->r 1 * bin)
               (postD l->r 0 + bin)))


-----------------------:down[A,1]
(* 1 1) --> 1
-----------------------:down[A,2]
(* 2 1) --> 2
-----------------------:down[A,3]
(* 3 2) --> 6
(+ 3 0) --> 3
-----------------------:up[A,3]
-----------------------:down[A,4]
(+ 4 3) --> 7
(* 4 6) --> 24
-----------------------:up[A,4]
(+ 2 7) --> 9
-----------------------:up[A,2]
-----------------------:down[A,5]
(* 5 24) --> 120
-----------------------:down[A,6]
(+ 6 9) --> 15
(* 6 120) --> 720
-----------------------:up[A,6]
-----------------------:down[A,7]
(* 7 720) --> 5040
(+ 7 15) --> 22
-----------------------:up[A,7]
(+ 5 22) --> 27
-----------------------:up[A,5]
(+ 1 27) --> 28
-----------------------:up[A,1]
; Value: (5040 . 28)
```

Figure 6.35: Trace of a parallel combination of pre-order and post-order accumulators.

```
OPERA> (let ((bin (binary)))
         (cons (preD l->r 1 * bin)
               (preD l->r 0 + bin)))
-----------------------:down[A,1]
(+ 1 0) --> 1
(* 1 1) --> 1
-----------------------:down[A,2]
(+ 2 1) --> 3
(* 2 1) --> 2
-----------------------:down[A,3]
(* 3 2) --> 6
(+ 3 3) --> 6
-----------------------:up[A,3]
-----------------------:down[A,4]
(+ 4 6) --> 10
(* 4 6) --> 24
-----------------------:up[A,2]
-----------------------:down[A,5]
(* 5 24) --> 120
(+ 5 10) --> 15
-----------------------:down[A,6]
(* 6 120) --> 720
(+ 6 15) --> 21
-----------------------:up[A,6]
-----------------------:down[A,7]
(+ 7 21) --> 28
(* 7 720) --> 5040
; Value: (5040 . 28)
```

Figure 6.36: Trace of a parallel combination of two pre-order accumulators.

```
OPERA> (postD l->r 0 +
              (pre-scanD l->r 1 *
                         (binary)))
-----------------------:down[A,1]
(* 1 1) --> 1
-----------------------:down[A,2]
(* 2 1) --> 2
-----------------------:down[A,3]
(* 3 2) --> 6
(+ 6 0) --> 6
-----------------------:up[A,3]
-----------------------:down[A,4]
(* 4 6) --> 24
(+ 24 6) --> 30
-----------------------:up[A,4]
(+ 2 30) --> 32
-----------------------:up[A,2]
-----------------------:down[A,5]
(* 5 24) --> 120
-----------------------:down[A,6]
(* 6 120) --> 720
(+ 720 32) --> 752
-----------------------:up[A,6]
-----------------------:down[A,7]
(* 7 720) --> 5040
(+ 5040 752) --> 5792
-----------------------:up[A,7]
(+ 120 5792) --> 5912
-----------------------:up[A,5]
(+ 1 5912) --> 5913
-----------------------:up[A,1]
; Value: 5913
```

Figure 6.37: Trace of a series combination of a pre-order scanner and a post-order accumulator.

```
OPERA> (preD l->r 0 +
             (pre-scanD l->r 1 *
                        (binary)))
-----------------------:down[A,1]
(* 1 1) --> 1
(+ 1 0) --> 1
-----------------------:down[A,2]
(* 2 1) --> 2
(+ 2 1) --> 3
-----------------------:down[A,3]
(* 3 2) --> 6
(+ 6 3) --> 9
-----------------------:up[A,3]
-----------------------:down[A,4]
(* 4 6) --> 24
(+ 24 9) --> 33
-----------------------:up[A,2]
-----------------------:down[A,5]
(* 5 24) --> 120
(+ 120 33) --> 153
-----------------------:down[A,6]
(* 6 120) --> 720
(+ 720 153) --> 873
-----------------------:up[A,6]
-----------------------:down[A,7]
(* 7 720) --> 5040
(+ 5040 873) --> 5913
; Value: 5913
```

Figure 6.38: Trace of a series combination of a pre-order scanner and a pre-order accumulator.

Mappers, filters, up accumulators, and parallel down and up scanners have malleable shapes that depend on the context in which they are used. For instance, Figure 6.39 shows how a down scanner conforms to shape constraints supplied by other slivers. When used in conjunction with a left-to-right accumulator, `down-scanD` is guaranteed to perform accumulations in a left-to-right order. However, with a right-to-left accumulator, `down-scanD` will perform its accumulations in a right-to-left order. If combined with a parallel up accumulator, the operations of `down-scanD` can happen in any order consistent with the data dependencies.

The final trace in Figure 6.39 illustrates an important fact about parallel tree shapes. If a lock step component contains only slivers with parallel down and up shapes, then the entire component has a parallel shape. This means that independent branches can be processed concurrently. For example, the sample trace shows that the processing of the `2` and `3` elements is interleaved. While such parallel processing can lead to more efficient execution times in the presence of multiple processors, it can also lead to much greater storage consumption. In the uniprocessor environment we have been assuming, the time efficiency vanishes but the space inefficiency remains. A programmer wishing to control space behavior must explicity add sequentiality to a parallel-shaped tree process in order to obtain desirable space behavior.

6.2.3 Extended Example: Alpha Renaming

As a practical example of syndrites, we show how the monolithic alpha renaming program introduced in Section 2.2 can be expressed as a modular SYNAPSE program. We also consider some extensions that are easily supported by the modular program.

A Sliver-based Alpha Renamer

Figure 6.40 is a sliver diagram for alpha renaming. `TERM->SYNDRITE` and `SYNDRITE->TERM` convert terms represented as s-expressions into syndrites. `FILTER-FORMALS` is a filter that passes only the formal parameters of abstraction nodes. `UNIQUE-NAMES` returns a tree of names such that the ungapped positions of its input map to unique outputs. Given a tree of names and a tree of values, `BIND` returns a tree of environments, where each environment

```
OPERA> (define (tiny) (splay-tree '(1 2 3))) ; Value: tiny

OPERA> (preD l->r 1 * (down-scanD 4 + (tiny)))
-----------------------:down[A,1]
(+ 1 4) --> 5
(* 5 1) --> 5
-----------------------:down[A,2]
(+ 2 5) --> 7
(* 7 5) --> 35
-----------------------:up[A,2]
-----------------------:down[A,3]
(+ 3 5) --> 8
(* 8 35) --> 280
; Value: 280

OPERA> (preD r->l 1 * (down-scanD 4 + (tiny)))
-----------------------:down[A,1]
(+ 1 4) --> 5
(* 5 1) --> 5
-----------------------:down[A,2]
(+ 3 5) --> 8
(* 8 5) --> 40
-----------------------:up[A,2]
-----------------------:down[A,3]
(+ 2 5) --> 7
(* 7 40) --> 280
; Value: 280

OPERA> (upD (lambda (elt lst) (* elt (*-list lst)))
            no-gaps
            (down-scanD 4 + (tiny)))
-----------------------:down[A,1]
(+ 1 4) --> 5
-----------------------:down[A,2]
-----------------------:down[A,3]
(+ 2 5) --> 7
(* 7 1) --> 7
(+ 3 5) --> 8
-----------------------:up[A,2]
(* 8 1) --> 8
-----------------------:up[A,3]
(* 8 1) --> 8
(* 7 8) --> 56
(* 5 56) --> 280
-----------------------:up[A,1]
; Value: 280
```

Figure 6.39: Traces illustrating the malleable nature of `down-scan`.

is a set of name/value bindings. At every position, an environment extends the bindings of its parent with the name and value at that position. RENAME takes a syntax tree and a tree of environments and returns a similarly shaped syntax tree in which each occurrence of a name (formal parameter and variable reference) is replaced by the corresponding value in the environment at the same position.

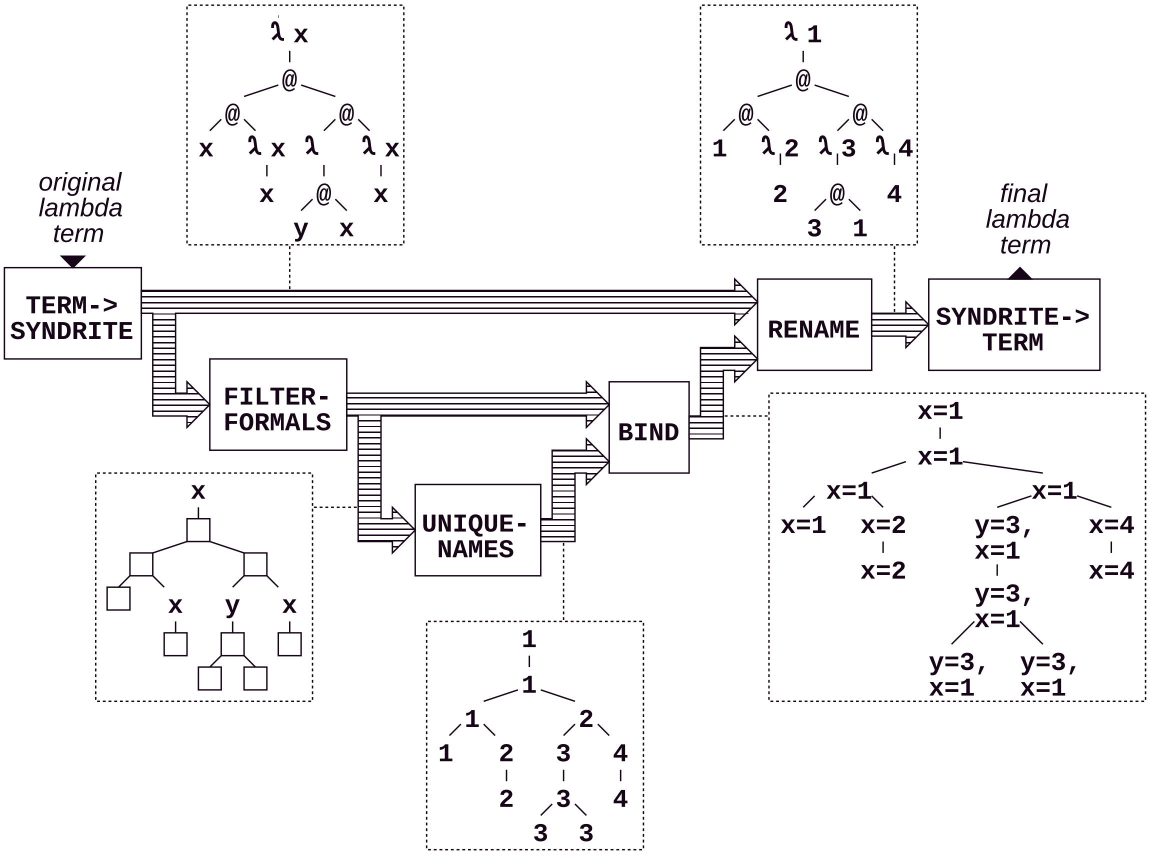


Figure 6.40: A signal processing style view of alpha renaming.

In the figure, each of the slags is annotated with a tree that represents the syndrite transmitted by the slag for sample input:

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

The empty boxes in the output of `FILTER-FORMALS` indicate gaps. The names produced by `UNIQUE-NAMES` in this case are numbers that result from a left-to-right pre-order scan of an incrementer over the tree of formals.

The structure of the sliver diagram translates directly into a SYNAPSE program:

```
(define (alpha-rename term)
  (let ((nodes (term->syndrite term)))
    (let ((formals (filter-formals lambdrite)))
      (syndrite->term
       (rename nodes
               (bind formals
                     (unique-names formals)))))))
```

Each of the slivers used by `alpha-rename` can be easily constructed out of the SYNAPSE primitives. `Term->syndrite` transforms a term represented as an s-expression into a syndrite of nodes tagged with the type and contents of the corresponding position in the abstract syntax tree of the term:

```
(define (term->syndrite term)
  (genDi term
         (lambda (tm return)
           (cond
            ((variable? tm)
             (return (var-node (name tm))
                     '()))
            ((abstraction? tm)
             (return (lambda-node (formal tm))
                     (list (body tm))))
            ((application? tm)
             (return (call-node)
                     (list (rator tm) (rand tm))))))))
```

`Name`, `formal`, `rator`, etc. are selectors on terms, while `var-node`, `lambda-node`, and `call-node` are constructors for the syndrite elements. The inverse transformation is performed by `syndrite->term`:

```
(define (syndrite->term nodes)
  (upD
   (lambda (node subaccs)
     (cond
      ((var-node? node)
       (variable (node-name node)))
      ((lambda-node? node)
       (abstraction (node-formal node) (first subaccs)))
      ((call-node? node)
       (application (first subaccs) (second subaccs)))))
   no-gaps
   nodes))

(define (no-gaps lst) (error "Shouldn't happen"))
```

Here, `node-name` and `node-formal` are selectors on the syndrite nodes, while `variable`, `abstraction`, and `application` are term constructors.

Converting between terms and syndrites is the messy part; the slivers at the heart of the alpha renamer are remarkably elegant. `Filter-formals` is just a simple application of filtering and mapping:

```
(define (filter-formals nodes)
  (mapD node-formal
        (filterD lambda-node? nodes)))
```

`Unique-names` is easily implemented as a pre-order scanner that increments the accumulator every time it encounters a name in its input tree:

```
(define (unique-names names)
  (pre-scanD left-to-right
             0
             (lambda (x n) (1+ n))
             names))
```

`Bind` is a trivial application of a two-argument down scanner:

```
(define (bind names vals)
  (down-scan2D env-empty
               env-bind
               names
               vals))

(define (down-scan2D init combine synd1 synd2)
  (down-scanD init
              (lambda (pair acc)
                (combine (car pair) (cdr pair) acc))
              (map2D cons synd1 synd2)))
```

Here, `env-empty` is an empty environment and `env-bind` is a procedure that extends a given environment with a name/value binding. Finally, `rename` can be implemented as a two-input mapper that discriminates on node type:

```
(define (rename nodes envs)
  (map2D (lambda (node env)
           (cond
            ((var-node? node)
             (var-node (env-lookup (node-name node) env)))
            ((lambda-node? node)
             (lambda-node (env-lookup (node-formal node) env)))
            (else node)))
        nodes
        envs))
```

Behaviorally, the important feature of `alpha-rename` is its space consumption. Define the *working space* of an alpha renaming process to be the maximal space required by the process *excluding* the space required by the input and output terms. Let n be the depth of abstract syntax tree associated with the input term. Then `alpha-rename` has working space *linear* in n. This is the same order of growth as the working space required by the monolithic versions presented in Section 2.2. In contrast, traditional aggregate data implementations of `alpha-rename` would exhibit space *exponential* in n. (See Section 9.2 for experimental results confirming this claim.)

Moreover, disregarding the slag management operations, the fine-grained operational behavior of `alpha-rename` closely resembles that of the functional monolithic version. The lock step nature of SYNAPSE interweaves the processing of the individual slivers into a single left-to-right traversal of the input term. A minor difference between the modular and monolithic processes is that the modular version does an environment lookup for formal parameters as well as variable references; the modular version only performs the lookup for variable references. This difference could be removed by directly wiring a slag from `unique-names` to `rename` and modifying `rename` to use the additional input in the case of lambda nodes.

The Benefits of Modularity

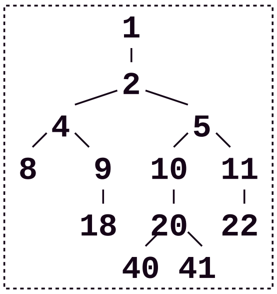
While the sliver-based alpha renamer exhibits important operational characteristics of the monolithic version, it also supports the advantages of modularity. Without these advantages, the work needed to construct all the little pieces that comprise the alpha renamer would be wasted. The real benefit of modularity is that it encourages us to explore alternatives and extensions in a way that the monolithic approach never would. We will now explore a few examples of how modularity and the notion of shape free us to think about the alpha renaming problem from fresh perspectives.

What changes would be required to `alpha-rename` in order to handle lambda terms with extended syntax? The case of handling an `if` construct (see Section 2.2) would be particularly easy: only `term->syndrite` and `syndrite->term` would have to extended with clauses for the new term. `Rename` would not have to be changed since its `else` clause is the appropriate default for simple expressions like `if`. Adding a naming construct like `let` or `letrec` is more challenging. In these cases, `rename` would have to be modified. In addition, the nodes would have to provide `bind` with information indicating in which subtrees the declared name(s) are bound.

The existence of the `unique-names` sliver in the sliver diagrams suggest that there may be many ways to compute unique names. And indeed there are. We can trivially modify the above definition of `unique-names` to visit the formals in a right-to-left direction rather than a left-to-right one. As noted in Section 2.2, such a change can be far from trivial in a monolithic organization.

As a more interesting change, the fact that the name generator is the only sliver in the diagram without a parallel shape suggests that we should consider parallelizing it.[14] One approach to a parallel name generator is to employ the binary tree numbering technique of doubling a number down one branch and doubling and incrementing down the other. This strategy would lead to the following tree for the particular example considered above:

[14] Although we have argued before that networks of purely parallel slivers are probably not a good idea in SYNAPSE, they may be very good idea for other language models. The power of thinking in terms of shapes extends beyond their particular realization in SYNAPSE!



If the branching factor is at most two (as it is in the case of lambda terms), the numbers generated down different branches are guaranteed to be unique, and so they can be calculated in a parallel down fashion. While the `down-scan` sliver is not appropriate for this purpose, it would not be hard to construct a parallel down primitive that is.

Another approach to parallel name generation would be to use `down-scan` with an accumulator that employed a mutable name generator. Of course, the name generator would have to be carefully constructed in order to guarantee atomicity. This is the sort of strategy that is encouraged by Id's M-structures [Bar92].

The alpha renaming network is useful for program manipulations other than alpha renaming. A deBruijn numbering program labels variables in a lambda term with a number indicating the level of the lambda that introduces them [Pey87]. Such a program can be trivially obtained from the alpha renamer by changing the naming strategy from a pre-order scan to a down scan:

```
(define (debruijn names)
  (down-scanD 0
              (lambda (x n) (1+ n))
              names))
```

Furthermore, starting with the deBruijn numbering program, it is easy to obtain a lexical depth compiler. The *lexical depth* of a variable name is a number indicating how many `lambda`s to go up the abstract syntax tree in order to find the one that introduces the name. This number can be computed for each variable reference by subtracting the number found by looking up its name in the environment from the current value of the `debruijn` scanner at variable reference position in the tree. This change requires only (1) an extra slag path between the name generator and `rename` and (2) a modification of the variable reference clause in `rename`.

The fact that the structure of the alpha renamer makes it easy to envision such a wide variety of alternatives, extensions, and related programs is a testament to benefits of modularity. The hope is that the operational control provided by slivers will make such modular organizations even more alluring.