Chapter 4

Computational Shape

The goal of slivers is to decompose monolithic recursive procedures into networks of mix-and-match parts that preserve the operational character of the original procedure. But what is meant by the "operational character" of a recursive procedure?

Intuitively, operational character describes how the computations specified by a procedure unfold over time. Fine-grained aspects of this unfolding include the relative order of particular operations and the profile of the storage consumed by the computation as a function of time. But unfoldings also display coarser-grained patterns that appear again and again. The iterative nature of computations generated by a tail-recursive procedure are one example of such a pattern. Other patterns include nested loops, mutual recursions, and different classes of tree walks. These patterns are important enough to programmers that they will spend often expend considerable energy to construct a program that generates a desirable computational pattern.

This chapter describes the computational patterns generated by recursive procedures. I call these patterns *computational shapes.* Computational shapes provide a basis for describing some of the important operational characteristics of computations that are preserved by the sliver technique.

4.1 Linear Shapes

4.1.1 Linear Tiles

Figure 4.1 shows the abstract structure of a simple linear recursion. It is a stack of repeated boxes that grow increasingly shorter on the way down. I call this ladder-like structure a *linear trellis*. Although the figure suggests an infinite regress, any terminating computation will populate only a finite prefix of the trellis.

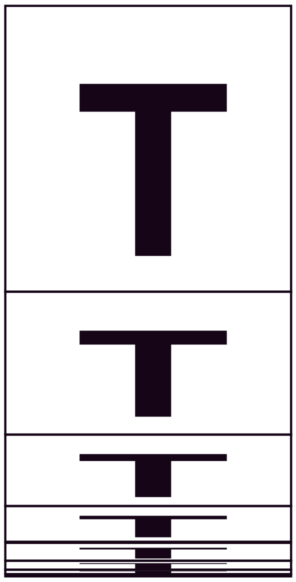


Figure 4.1: Trellis that depicts the structure of a simple linear recursion.

Each **T**-labeled box in the trellis, called a *linear tile*, represents the computation performed by one layer of the recursion. The interface to a tile is shown in Figure 4.2. The top line of a tile, called the *call boundary*, corresponds to the recursive procedure call that initiates the computation in the tile. The bottom line of a tile, called the *subcall boundary*, corresponds to the single recursive *subcall* that a tile computation is allowed to make.[1] Within a linear trellis, the subcall boundary of one tile coincides with the call boundary of the tile below it.

[1]A tile computation might not make any subcall; in fact, a linear recursion will terminate only if some tile in the trellis does not make a subcall. Also, a tile computation is allowed to have several potential subcalls as long as it is guaranteed to actually call at most one of them.

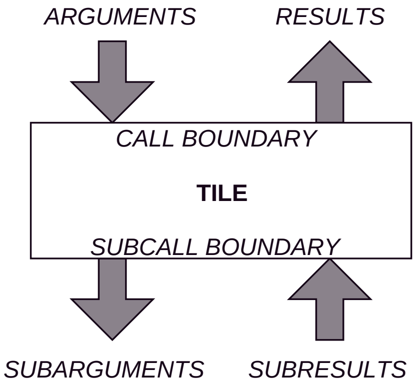


Figure 4.2: Interface to a linear tile. Each shaded arrow designates the transmission of zero or more values.

A tile receives *arguments* and passes *results* across the call boundary. Additionally, it may pass *subarguments* and receive *subresults* across the subcall boundary. Each of these actions (receiving/passing arguments/results) corresponds to an event that the tile computation may participate in:

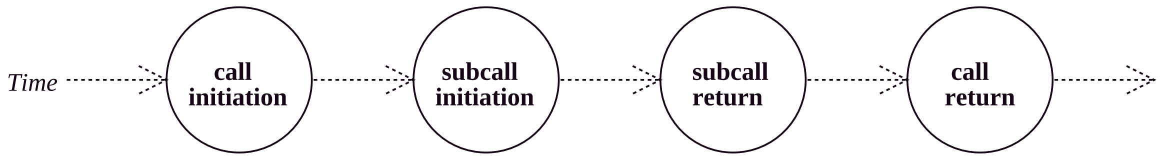
1. *Call initiation* marks the beginning of the tile computation. Since all argument values must be available before call initiation, the call boundary is said to be *strict*.

2. *Subcall initiation* marks the beginning of the subcall's computation. Due to the strictness of the subcall boundary, all subargument values must be computed before subcall initiation.

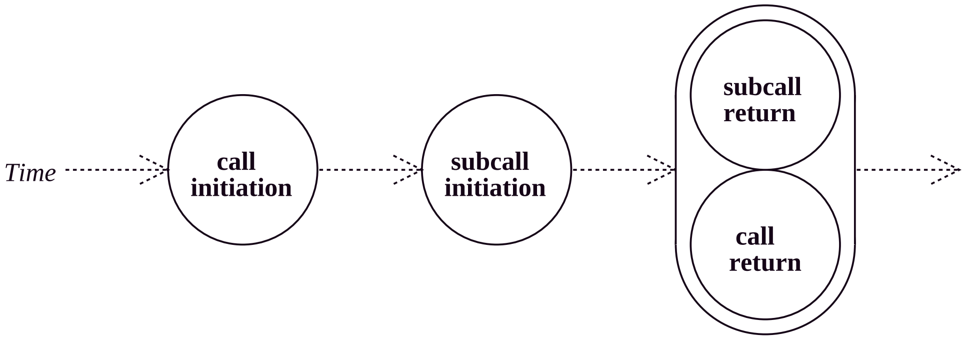
3. *Subcall return* marks the end of the subcall's computation. After this point, the result values of the subcall are available.

4. *Call return* marks the end of the tile computation. The result values returned by the call must be available before this point.

These four events are ordered linearly in time (Figure 4.3). A tile computation that peforms no operations between the subcall return and call return events is said to be *tail-recursive*. In this case, the two events are considered to temporally coincide.



(a) General tile



(b) Tail-recursive tile

Figure 4.3: Time ordering of linear tile events.

4.1.2 Linear Orientation

Operations performed by a tile computation can be classified according to when they can be performed with respect to the subcall:

1. A *down* operation must be performed between call initiation and subcall initiation.

2. An *up* operation must be performed between subcall return and call return.

3. An *across* operation is unordered with respect to subcall initiation and return. It must be performed within the call, but may be performed before, during, or after the subcall.

This classification will be called the *orientation* of an operator. The timing constraints on the three orientations are summarized by Figure 4.4.

4.1.3 Linear Shards

Because the timing constraints are mutually exclusive, they partition a tile's computation into into disjoint collections of operations, which I will call *shards*. The partitioning of a tile

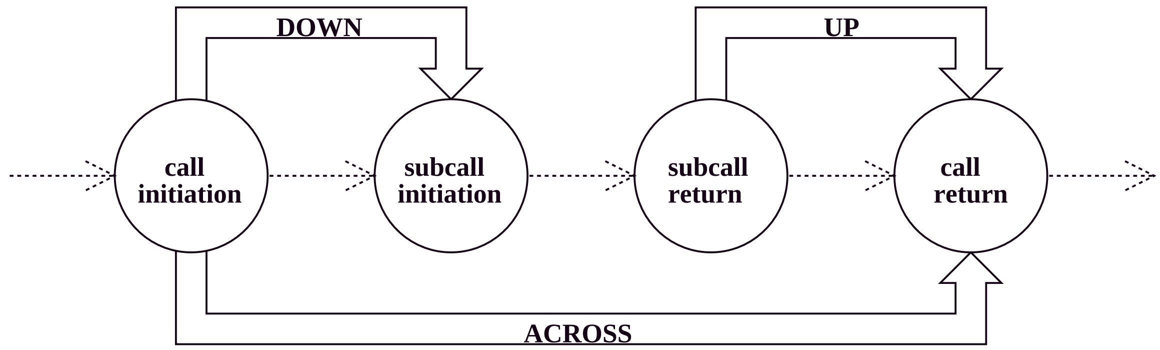


Figure 4.4: Timing constraints that define the orientation of an operation within in a linear tile.

into shards is graphically depicted in Figure 4.5. The *down* shard computes subarguments

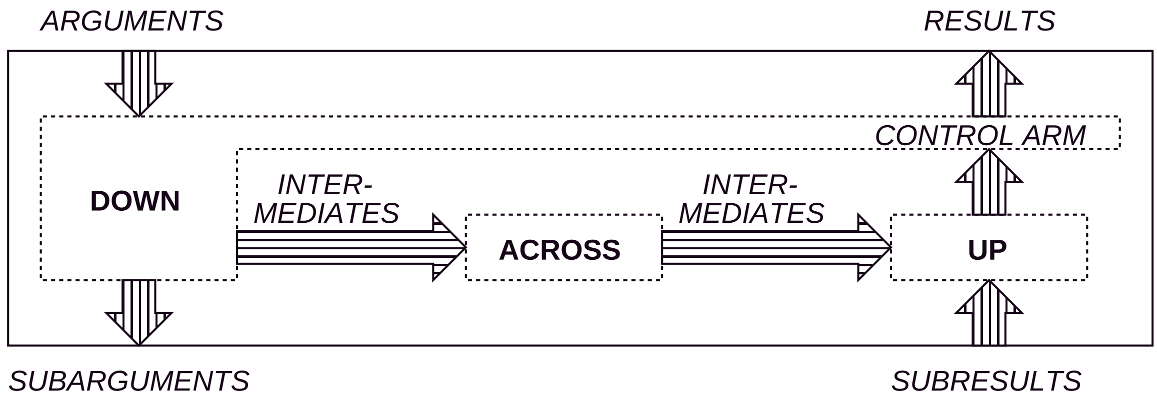
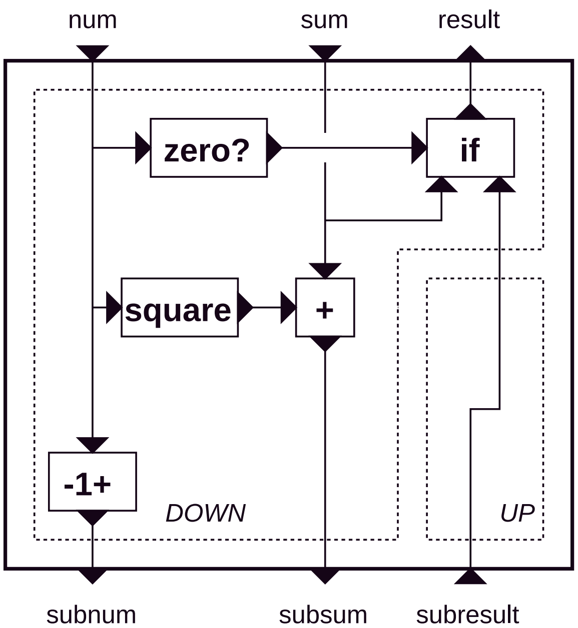


Figure 4.5: Computational shards within a linear tile.

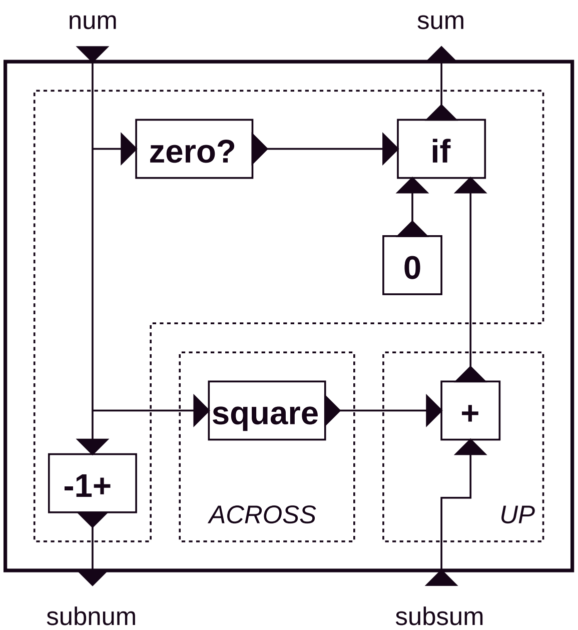
from arguments, while the *up* shard computes results from subresults. Arguments and intermediate values may be transmitted via so-called *intermediates* from the *down* shard, through the *across* shard, and to the *up* shard. In a tail-recursive tile, both the *across* and *up* shards are trivial — they contain no operations. The *down* shard usually has a *control arm* interposed between the *up* shard and the results; this will be explained shortly.

Tiles for some simple computations are shown in Figure 4.6.

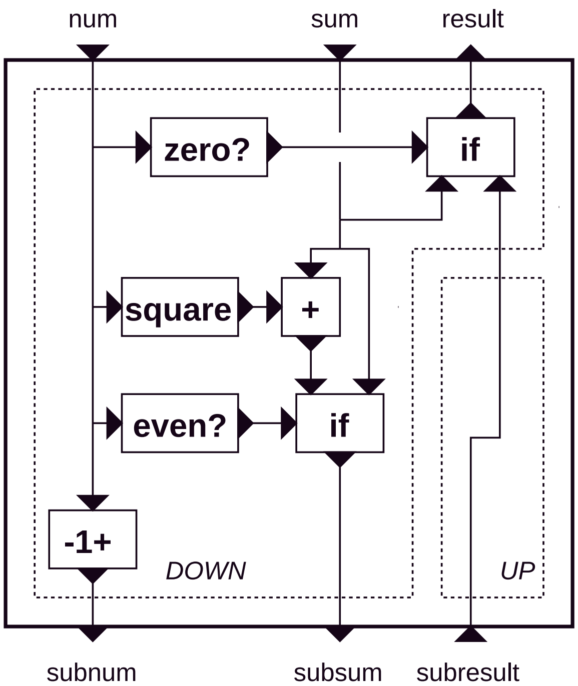
The tiles are arranged in pairs of iterative (left) and recursive (right) approaches to the same problem. Tiles **(a)** and **(b)** calculate the sum of the squares of the integers between



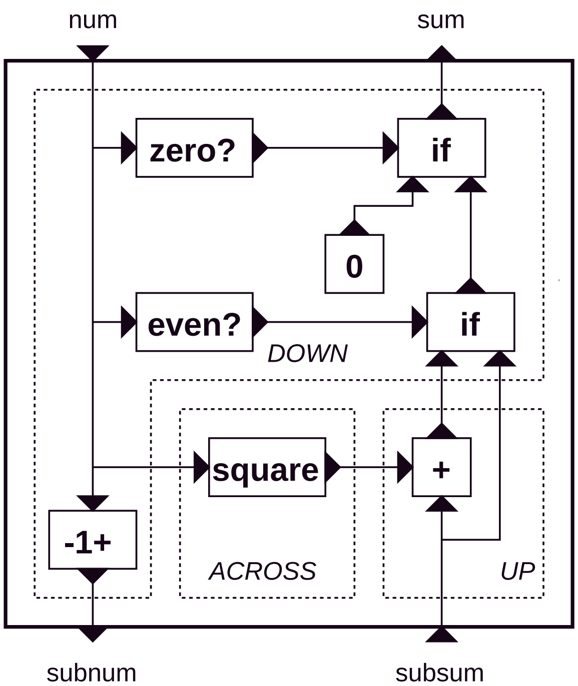
(a) Iterative sum of squares



(b) Recursive sum of squares

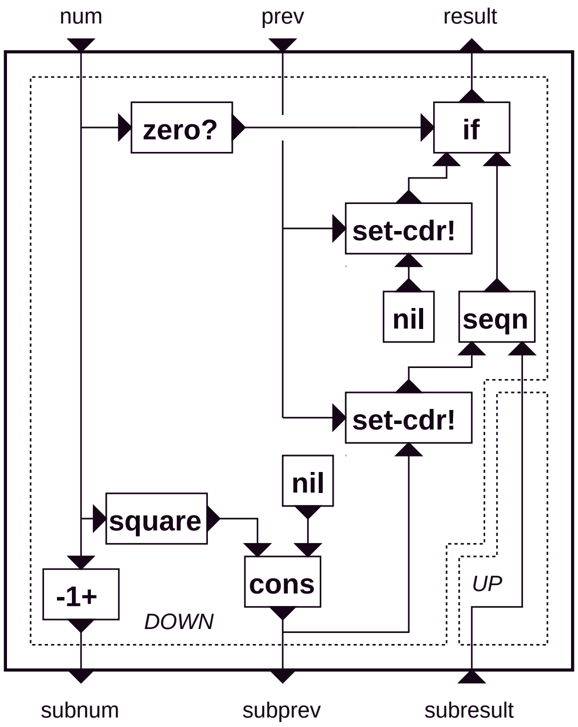


(c) Iterative sum of squared evens

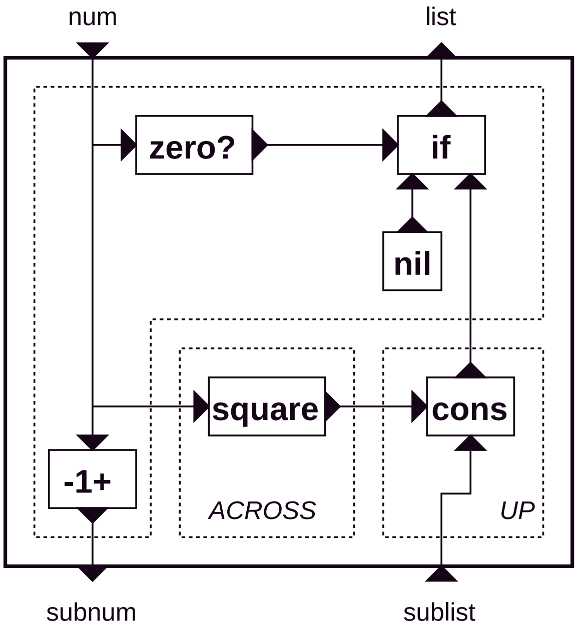


(d) Recursive sum of squared evens

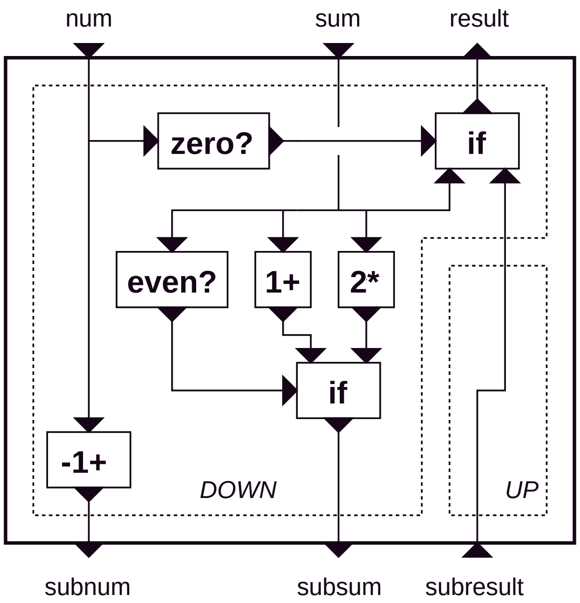
Figure 4.6: (Part I) Sample tiles from some linear recursive computations.



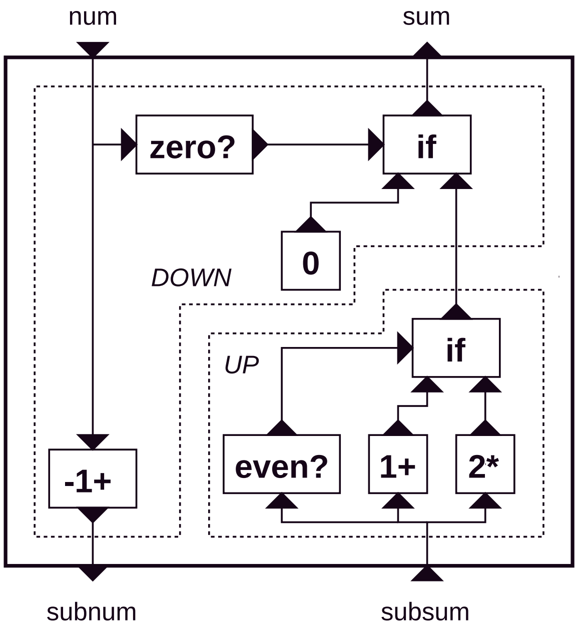
(e) Iterative list of squares



(f) Recursive list of squares



(g) Iterative even/odd



(h) Recursive even/odd

Figure 4.6: (Part II) Sample tiles from some linear recursive computations.

1 and n. Tiles **(c)** and **(d)** calculate the sum of the squares of the even integers between 1 and n. Tiles **(e)** and **(f)** list the squares of the integers between 1 and n. Tiles **(g)** and **(h)** calculate the nth term of the *even-odd* sequence eo defined as follows:

$$
\begin{aligned}
eo_0 &= 0 \\
eo_i &= \begin{cases} 1 + eo_{i-1} & \text{if } i > 0,\ eo_{i-1} \text{ even} \\ 2eo_{i-1} & \text{if } i > 0,\ eo_{i-1} \text{ odd} \end{cases}
\end{aligned}
$$

In each case, the tile is the computational unit repeated by the corresponding loop or recursion. For now, issues of initialization (how arguments for the top-level call are specified) and finalization (what is done with the results from the top-level call) will be ignored.

4.1.4 An Operational Model

Building intuitions about tiles and shards requires some understanding of how their computational innards work. The precise details concerning the interpretation of computational elements like those pictured in Figure 4.6 won't be spelled out until the exposition of the EDGAR model in Chapter 8. For now, assume that computation proceeds in a demand-driven fashion as follows:

- Computation within in a tile is initiated by the request of its results. The computation returns only when values are available for all results.

- A requested operator node (e.g. `zero?`, `-1+`, `set-cdr!`) requests all of its subnodes, which are then evaluated concurrently. When values for all subnodes are available, the operator is *performed* (computes a result).

- A requested `if` node first requests the value of its test subnode. When the test value is available, the `if` is *performed* by rerouting the request for its value to either its *then* subnode or its *else* subnode, as appropriate.

- A requested `seqn` node first requests the value of its left subnode. When the left value becomes available the `seqn` is *performed* by rerouting the request for its value to its right subnode. (The value of the left subnode is ignored).

- A request for a subresult produced by a subcall first propagates requests to all of the subargument nodes, which are then evaluated concurrently. Subcall initiation occurs only after *all* subresults have been requested and *all* subargument values are available; i.e., the subcall is strict in all of its arguments. At this point, the subresult requests are allowed to propagate across the subcall boundary to the computation of the tile below.

Based on the above operational model in mind, here are some observations about the sample tiles in Figure 4.6:

- The demand-driven handling of conditionals accounts for the control arm of the *down* shard. Any `if`s that appear in this arm are performed before the subcall, so they belong in the *down* shard. (I assume that an `if` is always performed in the same shard as the operation that determines its test value.) The name "control arm" derives from the fact that these conditionals control the rest of the tile's computation. An `if` can appear outside the control arm when its test is determined by an *up* operation, as in tile **(h)**.

- The operator nodes and subcall boundary represent only potential computation, not actual computation. For example, when the upper-left `if` in any of the sample tiles tests true, the subcall and most of the tile nodes are not performed.

- The *up* shard may be a trivial computation that contains wires but no operations (e.g., tiles **(a)**, **(c)**, **(e)**, and **(g)**). This signifies a tail-recursive tile computation. Intuitively, a tail-recursive tile never returns any results because there is no *up* computation to be performed. Instead, requests for the results of a tail-recursive tile are simply rerouted to become requests for the subresults.

- Some tiles (e.g., tile **(d)**) have both trivial and non-trivial paths through an *up* shard. I call such tiles *conditionally tail-recursive*. Since the tail-recursive property must be known before subcall initiation, any condition on which tail-recursion depends must be tested in the *down* shard.

- The `square` operations in tiles **(b)**, **(d)**, and **(f)** are *across* operations because they must be performed during the tile computation but can be performed before, during, or after the subcall. In contrast, the corresponding `square` operations in tiles **(a)**, **(c)**, and **(e)** are *down* operations because subargument evaluation requires their results. Due to such dependencies, tail-recursive tiles cannot have an *across* shard. *Across* shards that are empty or trivial (i.e., only wires) are simply omitted from tile diagrams.

- In tile **(g)**, the `seqn` node serves to control the order of the side-effects performed by the `set-cdr!` operator. The value returned by the tile doesn't actually matter because the finalizer for the tile (not shown) is responsible for maintaining a pointer to the result.

4.1.5 Linear Tile Shapes

It is useful to characterize tiles according to their shards. In the case of linear tiles, the main distinguishing feature is whether or not the *up* and *across* shards are trivial. A tile with trivial *up* and *across* shards will be said to have a *down shape* because it consists of only a *down* component. A computation with a non-trivial *up* component will be said to have an *up shape*. For now, "*down* shape" and "*up* shape" can be treated as synonyms for "tail-recursive" and "non-tail-recursive". But we will soon encounter other computational patterns that extend the shape notion beyond tail-recursion.

4.1.6 Linear Computations

Individual tiles are stacked together to represent a whole computation. For now, I will only consider computations that can be expressed by replicating the same tile throughout the linear trellis of Figure 4.1. This defines the class of *unitilable* computations.

For example, a recursive sum-of-squares computation on 3 would be constructed out of four instances of tile **(b)** from Figure 4.6 (three for the non-zero cases, and one for the zero case). Operationally, we can imagine that new tiles are dynamically appended to the bottom of a stack only when a subcall is initiated.

A terminating computation uses only a finite number of tiles. Some computations, however, conceptually require an infinite number of tiles. For example, a sum-of-squares computation on a negative input would require unlimited instances of tile **(b)** because the `zero?` test would never be true. Some tiles can *only* generate infinite computations; for example, both tiles in Figure 4.7 give rise to computations that are guaranteed never to return, regardless of their inputs.

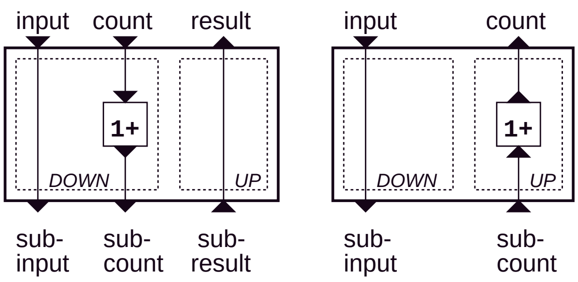


Figure 4.7: Two tiles that generate infinite computations regardless of input.

A particular computation can be classified according to the timing relationships induced among all the call events that occur during the computation. Figure 4.8 shows some timing configurations for linear computations. Each pair of circles represents the initiation (`I`) and return (`R`) events for a single call boundary. A solid directed line from event A to event B indicates that A must happen before B. It is also assumed that every initiation event must occur before its paired return event. A dotted line connecting two events indicates that they occur simultaneously; this is used to represent tail-recursion.

The configurations in the Figure differ in the relationships among the return events. All returns in configuration **(a)** occur at the same time, indicating that all of its calls are tail-recursive. In contrast, in configuration **(c)**, every subcall return occurs before the corresponding call return; this means that no calls are tail-recursive. Configuration **(b)** exhibits both tail-recursive and non-tail-recursive calls.

The timing diagrams suggest that the computation performed within a linear trellis can naturally be divided into *down* and *up* phases. In the *down phase*, the *down* shards of the tiles are performed in a top to bottom order. In the *up phase*, the stack of pending *up* shards are performed in bottom to top order. *Across* operations may happen at any time (subject to data dependencies) between the point where the down phase enters the tile computation

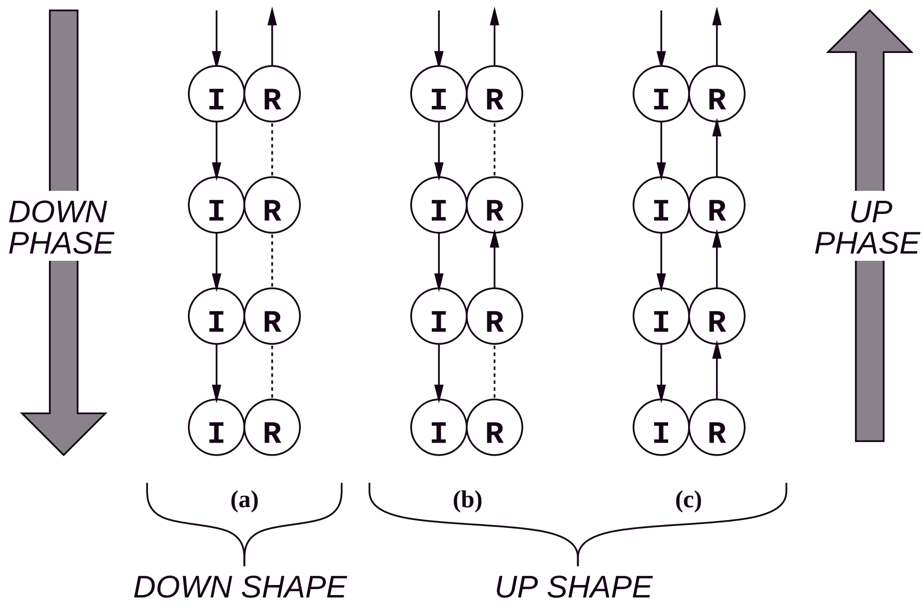


Figure 4.8: Three possible timing configurations for a linear computation.

and the point where the up phase leaves it.

If all the tiles in a linear computation prove to be tail-recursive, the *up* phase will be trivial (i.e., will perform no operations). In this case, the entire linear computation is a pure iteration and is said to have a *linear down shape*. If a linear computation has one or more tiles that turns out not be tail-recursive, then it exhibits some stack-pushing behavior and is said to have a *linear up shape*.

These computational shapes are dynamic properties of running computations. In contrast, the tile shapes defined earlier are static properties of tile structure. The shape of a tile gives a conservative approximation of the shape of a computation generated by that tile. A tile with *down* shape necessarily generates a computation with *down* shape. However, a tile with *up* shape doesn't necessarily generate an *up*-shaped computation. For example, a conditionally tail-recursive tile has *up* shape, but for some inputs may generate iterative computations. In fact, there are *up*-shaped tiles that *always* generate *down*-shaped computations; Figure 4.9 gives one such example. Nevertheless, in practice, tile shape tends to be a good predictor computation shape.

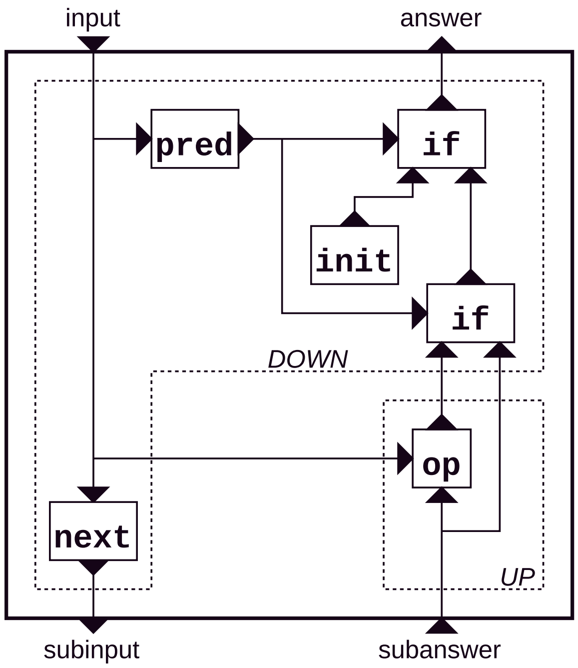


Figure 4.9: A tile with *up* shape that always generates a computation with *down* shape. Because the result of `pred` is tested by both `if` nodes, the *then* branch of the lower `if` node can never be taken.

4.1.7 Wrinkles

Deadlock

There are a few extra issues to discuss before leaving linear shapes. First is the issue of computational fate. So far, we have studied computations that terminate with a result and computations that don't terminate. There is a third possibility: computations that get stuck. Figure 4.10 depicts a tile that generates such a computation. The problem is that `op`

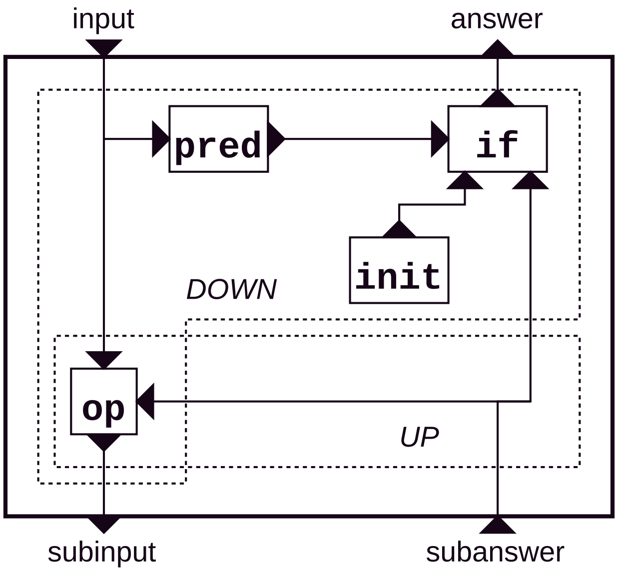


Figure 4.10: A tile that gives rise to a computation that deadlocks.

requires the subresult value in order to compute the subargument value. But the subresult value is not available until the subcall returns, and the subcall cannot be initiated until the subargument is available. This cyclic dependency halts the computation dead in its tracks; the resulting state is called *deadlock*. Deadlock will play an important role in the lock step processing model developed in Chapter 5.

As with computational shape, it is often possible to predict whether or not a computation will deadlock based on the structure of its generating tile. An operation that can be classified in both the *down* and *up* components of a tile (like `op` in the example) is a leading candidate for causing deadlock. But, like most interesting properties, deadlock is uncomputable in general. As an added complication, it turns out that in the presence of non-strict operators, not all cyclic dependencies result in deadlock. I will return to these issues later.

Multiple Potential Subcalls

Second, it is worth mentioning that not all tile diagrams are as tidy as the ones in Figure 4.6. For example, consider a tile (Figure 4.11) that models the following exponentiation procedure:

```
(define (fast-expt_rec base power)
  (if (zero? power)
      1
      (if (even? power)
          (square (fast-expt_rec base (/ power 2)))
          (* base (fast-expt_rec base (- power 1))))))
```

A different `power` argument is computed in each of the branches determined by the `even?` test. So there are two potential subcalls, at most one of which can be taken. This is represented in the tile diagram by two subcall boundaries with an **OR** separator. This is an ad hoc and awkward way of "sharing" several subcalls along the subcall boundary lines. This situation could be improved by decomposing the conditional into *split* and *join* operations (e.g., see [Den75]) that would enable the tile to share a single subcall boundary among several potential calls. However, this "problem" is purely an issue of visual presentation; in terms of the computational model, all that matters is that a tile computation make at most one subcall.

Initialization and Finalization

Third, the notion that many recursive computations can be constructed by replicating a single tile ignores important issues of initialization and finalization. For example, the *down*-shaped tiles **(a)**, **(c)**, and **(e)** from Figure 4.6 nowhere specify that the initial sum accumulation value should be 0. Similarly, the cdr-bashing list accumulation tile (**(g)**) does not spell out the important details of how to start and finish the computation. For these purposes, I will assume the existence of unreplicated *interface tiles* that sit between trellis tiles and the rest of a computation. Figure 4.12 illustrates interface tiles for the summation and cdr-bashing examples.

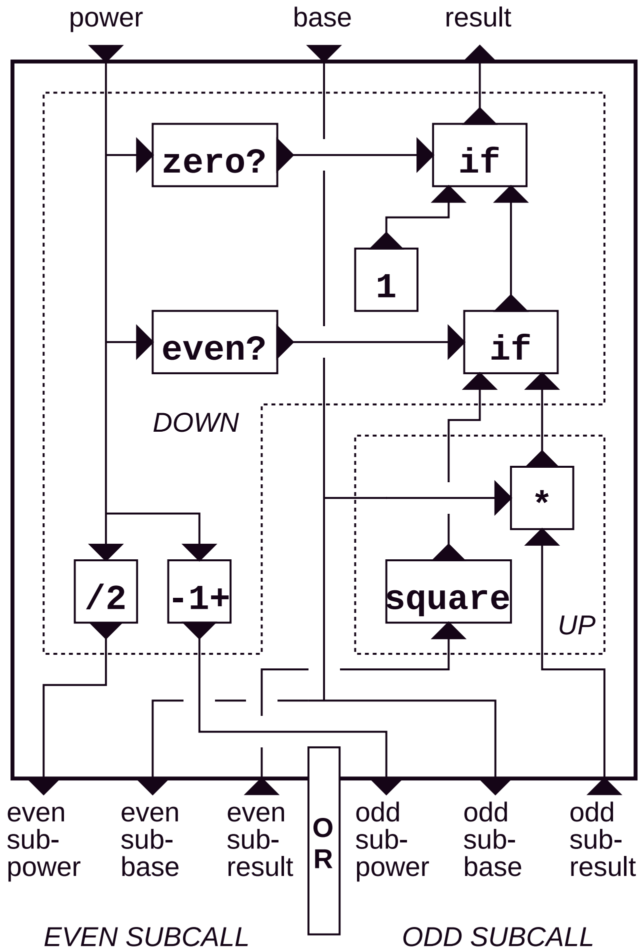
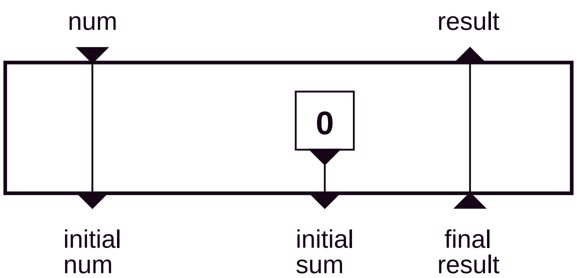
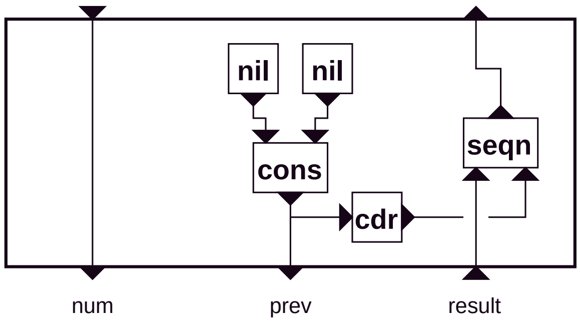


Figure 4.11: A tile with two potential subcalls, at most one of which can be initiated.



(a) Interface tile for iterative summing tiles.



(b) Interface tile for the cdr-bashing tile.

Figure 4.12: Interface tiles to a summation iteration and a cdr-bashing copying routine.

More Complex Recursion Patterns

Finally, I have only described a very narrow range of linear recursions — namely those that can be represented by the replication of a single tile. More complex linear computations, like mutual recursions and nested loops, can have structures that are composed out of several different kinds of tiles (see Figure 4.13). For now, I will continue to ignore these complexities and stick to the simple unitilable case. Nevertheless, the sliver technique based on this theory of shape will be able to handle more complex patterns of recursion.

4.2 Tree Shapes

The shape concepts developed for linear computations generalize to tree-structured computations. For simplicity, I will only consider binary computation trees for now. However, the concepts developed here can be extended to more general computation trees in which the branching factor is a non-uniform function of each tree node.

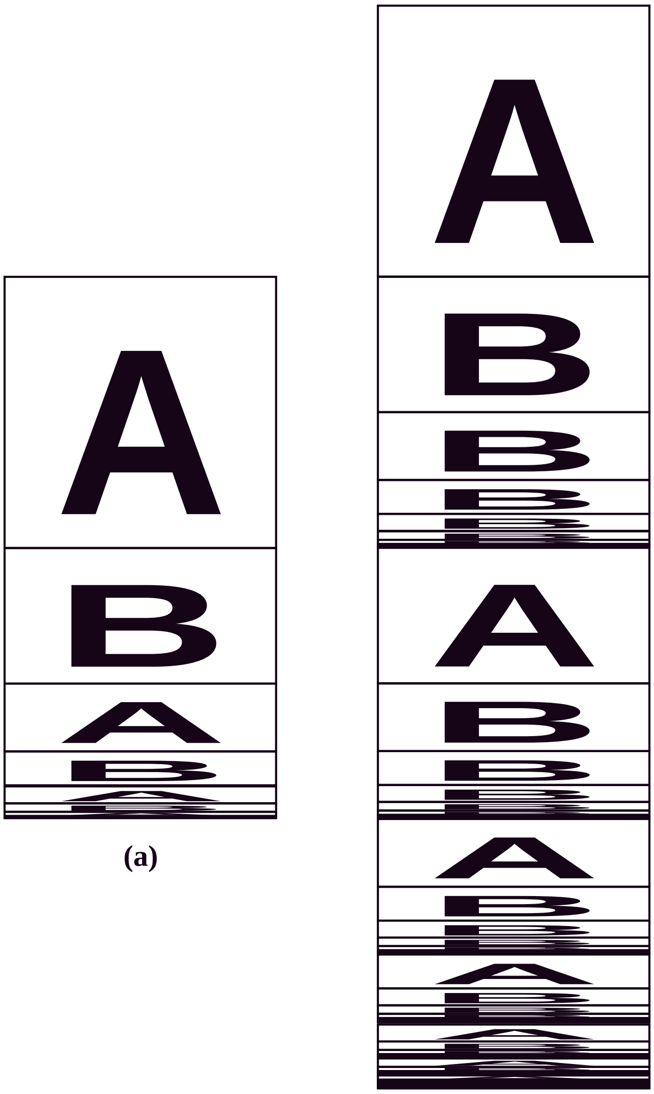
4.2.1 Binary Tiles

Figure 4.14 depicts a *binary trellis* that shows the structure of a simple recursive tree computation populated with instances of the *binary tile* labelled **T**. The interface to a binary tile is shown in Figure 4.15. The chief difference between binary tiles and linear tiles is that a binary tile may make up to two subcalls whereas a linear tile can make at most one. This leads to an interface with *left* and *right* subcall boundaries. The **AND** separator in Figure 4.15 indicates that both subcalls may be initiated. In contrast, the **OR** separator mentioned in the previous section indicates that at most one subcall may be initiated.

Associated with the computation of a binary tile are initiation and return events for the call boundary and each of the two subcall boundaries. These six events exhibit the branching time partial order illustrated in Figure 4.16.

4.2.2 Binary Orientation

Since an operation has a *down*, *across*, or *up* orientation with respect to each of the two subcall boundaries, there are nine distinct orientations for a binary tile operator. These are



(a)

(b)

Figure 4.13: Some linear trellises constructed from two tiles. (a) is intended to suggest mutual recursion between **A** and **B**, while (b) is intended to suggest an inner loop of **B**s nested within an outer loop of **A**s.

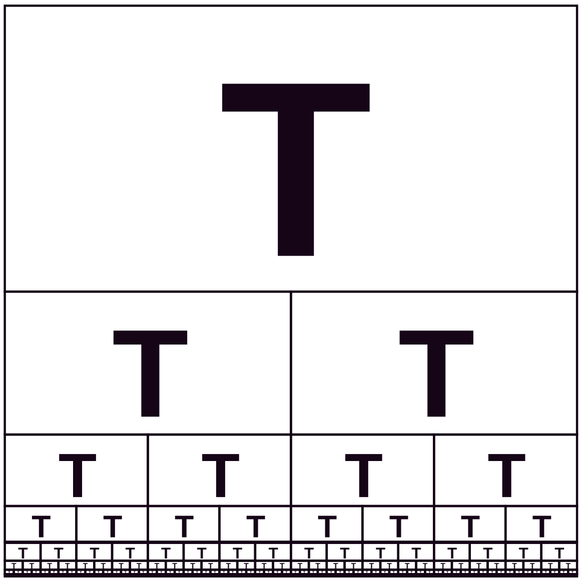


Figure 4.14: Trellis that depicts the structure of a simple tree recursion.

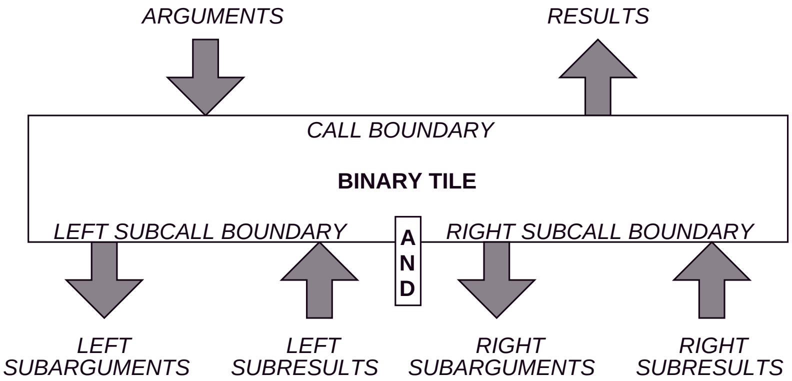


Figure 4.15: The interface to a binary tile.

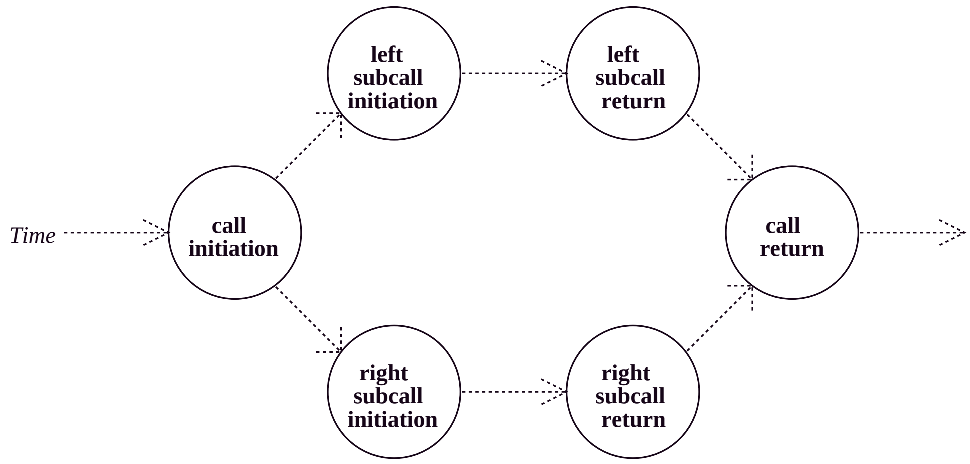


Figure 4.16: Time ordering of binary tile events.

enumerated in table 4.1. Distinguishing nine orientations may seem like overkill, but we will see that it provides us with a terse vocabulary for describing some important operational traits.

Figure 4.17 graphically summarizes the timing constraints for the nine binary orientations. The *down-both* orientation involves three events because an operation with this orientation must occur after call initiation but before both subcall initiations. The *up-both* orientation is similar in this regard. In contrast, the *down-left* orientation involves only two events. This indicates that a *down-left* operation not only must occur before initiation of the left subcall, but also must be unordered with respect to the right subcall. Otherwise it would be classified as *down-both* or *between-RL*. Similar remarks hold for the *down-right*, *up-left*, and *up-right* orientations. The binary *across* orientation has the same meaning as the linear one; an *across* operation can be performed at any point within the duration of the tile computation.

Sometimes it is convenient to use a less detailed vocabulary for talking about binary orientations. Here are a few helpful abstractions:

- *between-LR* and *between-RL* will be classified as *between* orientations.
- *down-both*, *down-left*, and *down-right* will be classified as *down* orientations.

Left Subcall Orientation	Right Subcall Orientation	Binary Orientation
down	*down*	*down-both*
down	*across*	*down-left*
down	*up*	*between-RL*
across	*down*	*down-right*
across	*across*	*across*
across	*up*	*up-left*
up	*down*	*between-LR*
up	*across*	*up-right*
up	*up*	*up-both*

Table 4.1: The nine orientations of an operation within a binary tile computation.

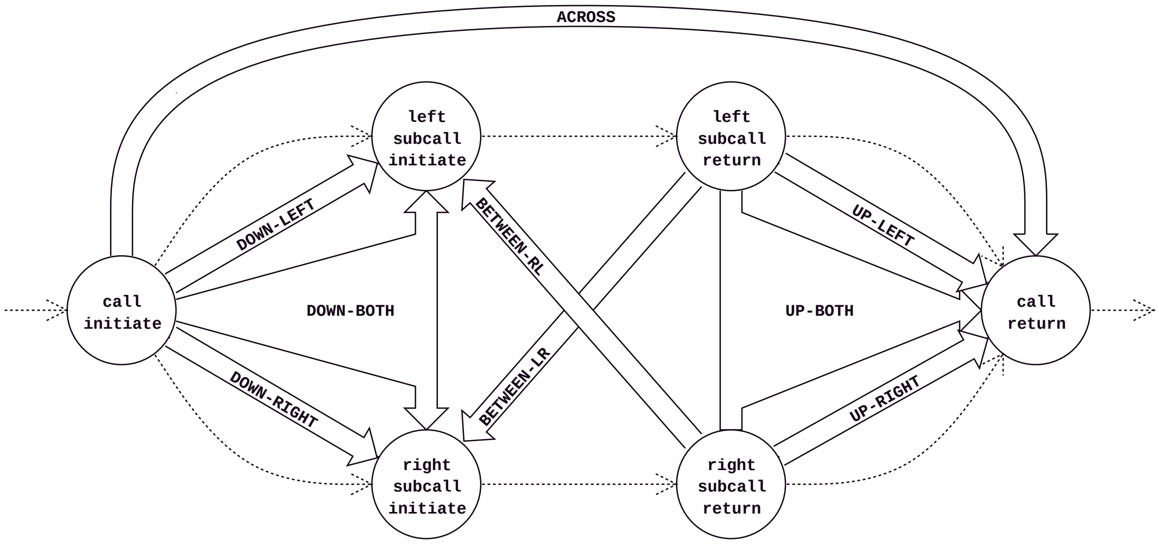


Figure 4.17: Timing constraints that define the orientation of an operation within in a binary tile. The two-headed *down-both* and two-tailed *up-both* arrows indicate constraints that involve three events.

- *up-both*, *up-left*, and *up-right* will be classified as *up* orientations.

Down and *up* for tree orientations can be viewed as generalizing their meaning for linear orientations. Context will distinguish whether linear or tree orientations are intended.

4.2.3 Binary Shards

As in a linear tile, the operations within a binary tile can be partitioned into shards according to their orientation. Different possibilities for dataflow between the two subcall boundaries lead to three patterns for binary tile computations (Figure 4.18):

1. A *parallel* tile allows no dataflow between subcalls. Due to the concurrency inherent in the computational model sketched earlier, the subcalls can be evaluated in parallel.

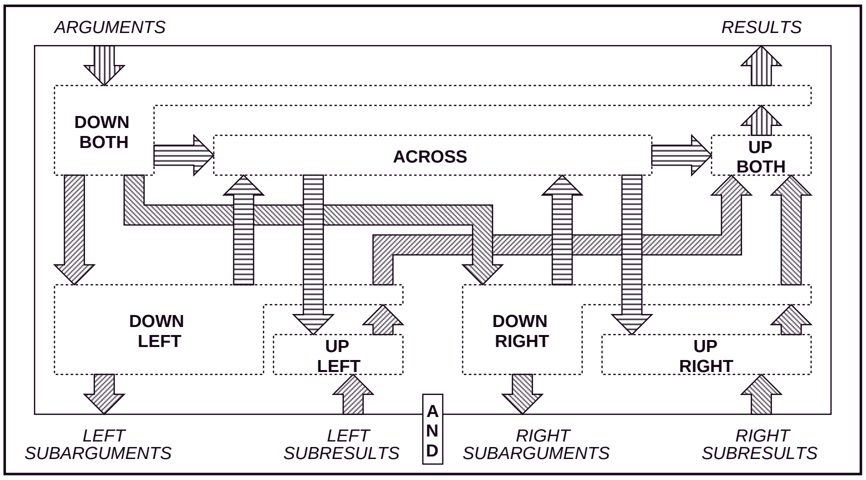
2. A *left-to-right* tile allows the right subarguments to depend on the left subresults. Due to the strictness of a subcall boundary, this means that the left subcall must return before the right subcall is initiated.

3. A *right-to-left* tile is symmetric with the left-to-right one.

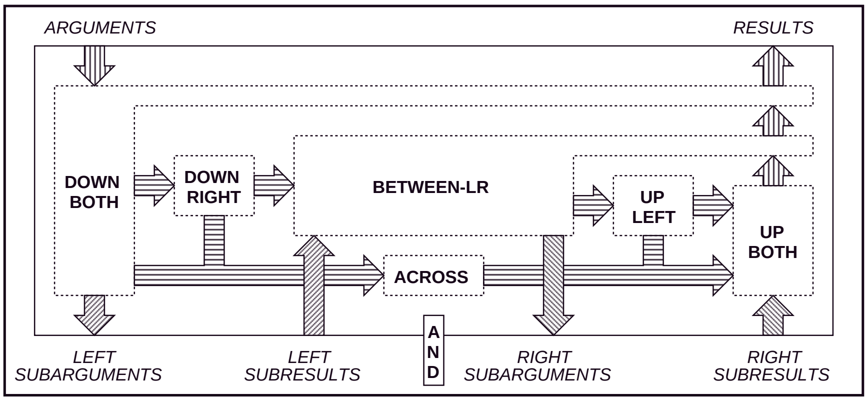
The fourth possibility, a *mutually dependent* tile in which the left and right subcalls depend on each other, is disallowed because the cyclic dependencies would lead to deadlock.[2] The left-to-right and right-to-left tiles are classified as *sequential* tiles because they force one subcall to return before the other is initiated.

Note that each of the three patterns in Figure 4.18 is necessarily missing several shards. The *between-LR* and *between-RL* shards are mutually exclusive because they imply that the subcalls happen in different orders. Clearly, no tile can contain both of these shards; and a parallel tile, whose subcalls must be unordered, can contain neither of them. A left-to-right tile invariably forces every operation that happens before the left subcall to also happen before the right subcall as well. In this case, a *down-left* shard is impossible because its

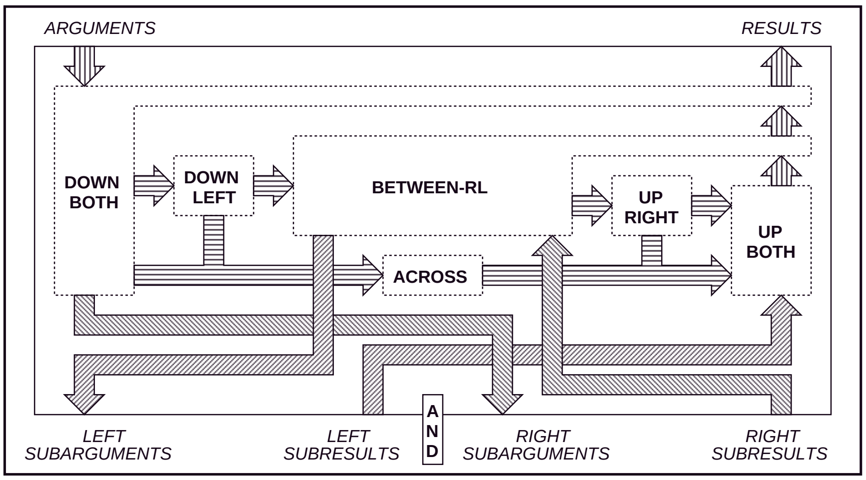
[2] I am assuming here that all operations performed in a tile computation are themselves strict. In the presence of lazy operators, cyclic dependencies are not only possible but often desirable (e.g., see [Bir84], [Joh87]). Later, I will introduce a form of laziness that, if used indiscriminantly, would invalidate the claims made about the impossibility of mutually dependent tiles and the execution order of sequential tiles. However, I will carefully restrict laziness in order to preserve these claims.



(a) Parallel Tile



(b) Left-to-right Tile



(c) Right-to-left Tile

Figure 4.18: Three general shard patterns for a binary tile computation.

operations would have to be unordered with respect to the right subcall; an *up-right* shard can similarly be discounted. Symmetric remarks hold for the right-to-left tile.

Some sample binary tiles appear in Figure 4.19. All of them share a generating fragment (consisting of the doubler `2*` and the incrementer `1+`) that creates a "virtual" binary tree of a given size in which each non-leaf node is numbered with its position in a left-to-right breadth-first traversal. I will call this a *breadth index tree* (see Figure 4.20).

The breadth index tree is virtual in the sense that it never exists as a bona fide data structure; rather, its elements are created and used by the tree-structured computation. Tile **(a)** is a parallel accumulator that collects the virtual tree elements into a tree-shaped data structure in which every node is represented as a list of its element and its left and right subtrees; `nil` represents the empty subtree. Tile **(b)** (**(c)**, **(d)**) collects a list of elements visited during a left-to-right pre-order (in-order, post-order) walk of the virtual tree.[3]

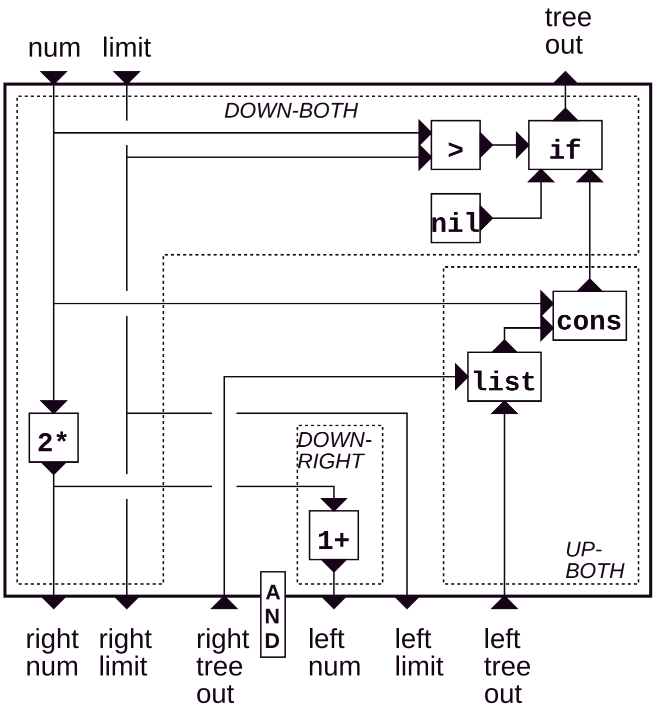
Several shards in the tile patterns of Figure 4.18 have control arms. In general, a control arm can be exhibited by any shard that must occur before some subcall — i.e., *down-both*, *down-left*, *down-right*, *between-LR*, and *between-RL* shards. For example, Figure 4.21 shows a tile with a control arm on both the *down-both* and *between-LR* shards. These allow the accumulation to terminate on entry to the call as well as between the two subcalls. A subtle constraint is that while *down-left* and *down-right* shards may have control arms in a parallel tile, they may not have control arms in a sequential tile.

4.2.4 Binary Tile Shapes

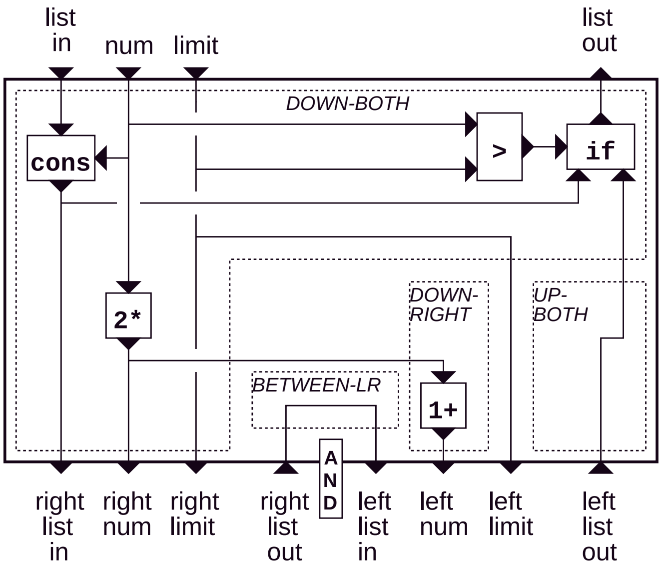
It is helpful to give names to special cases of the binary tile patterns introduced above. As in the linear case, these will be called shapes. A listing of important binary shapes appears in table 4.2.

Binary up tiles are a generalization of *linear up* tiles. These are tiles that simply combine the results of two independent subcalls; Figure 4.19**(a)** is an example. *Binary down* tiles

[3] Note that because `cons` builds the list from the end, the resulting list contains the elements in *reverse* left-to-right pre-order. A left-to-right pre-order list of tree elements can be generated by `cons`-accumulating elements in a right-to-left post-order walk of the tree. Alternately, the cdr-bashing strategy can be used to collect the list in pre-order during a pre-order walk. Similar remarks hold for the in-order and post-order cases.

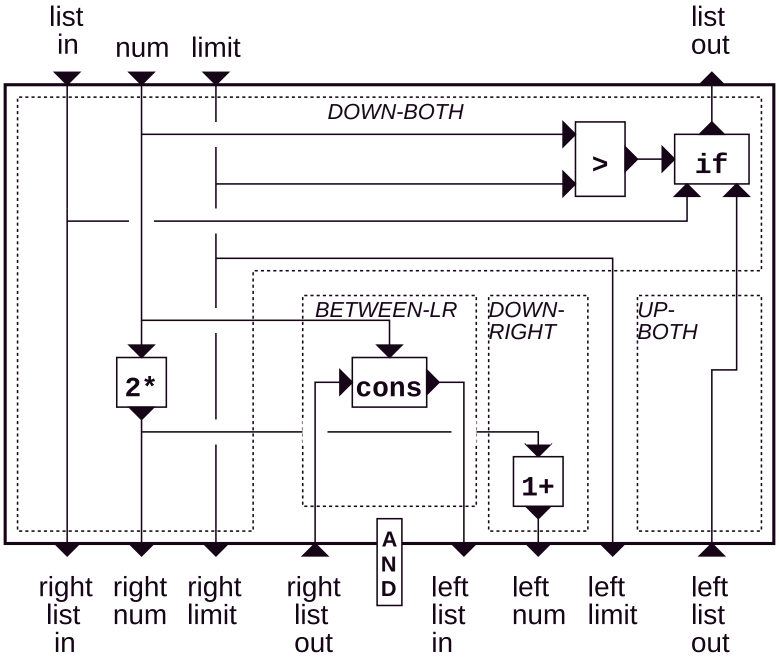


(a) A tile collecting the elements of a virtual tree into a tree-shaped data structure.

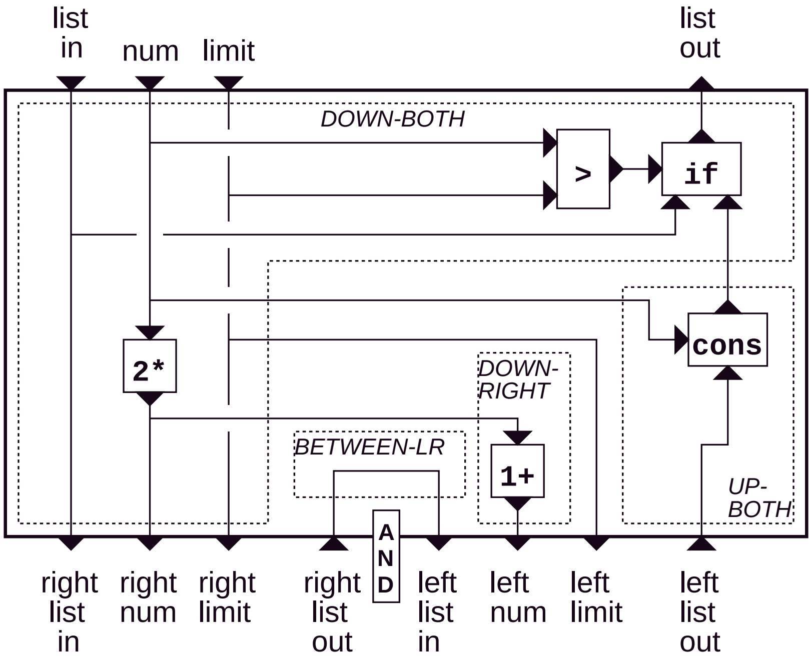


(b) A tile collecting a list of elements during a pre-order walk over a virtual tree.

Figure 4.19: (Part I) Sample binary tiles.



(c) A tile collecting a list of elements during a in-order walk over a virtual tree.



(d) A tile collecting a list of elements during a post-order walk over a virtual tree.

Figure 4.19: (Part II) Sample binary tiles.

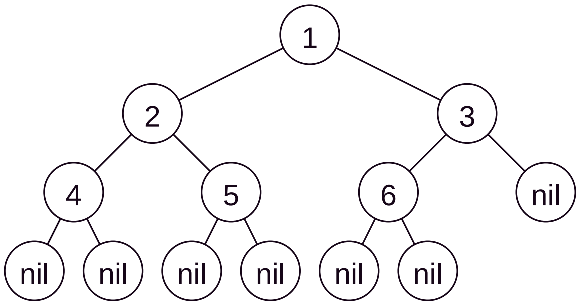


Figure 4.20: A breadth index tree with six elements.

Shape	Binary Pattern	Restrictions
binary down	parallel	*up-both*, *up-left* and *up-right* trivial
binary up	parallel	*up-both* non-trivial
LR pre	left-to-right	*between-LR* and *up-both* trivial
LR in	left-to-right	*between-LR* non-trivial and *up-both* trivial
LR post	left-to-right	*up-both* non-trivial
RL pre	right-to-left	*between-RL* and *up-right* trivial
RL in	right-to-left	*between-RL* non-trivial and *up-both* trivial
LR post	right-to-left	*up-both* non-trivial

Table 4.2: Shapes of binary tiles.

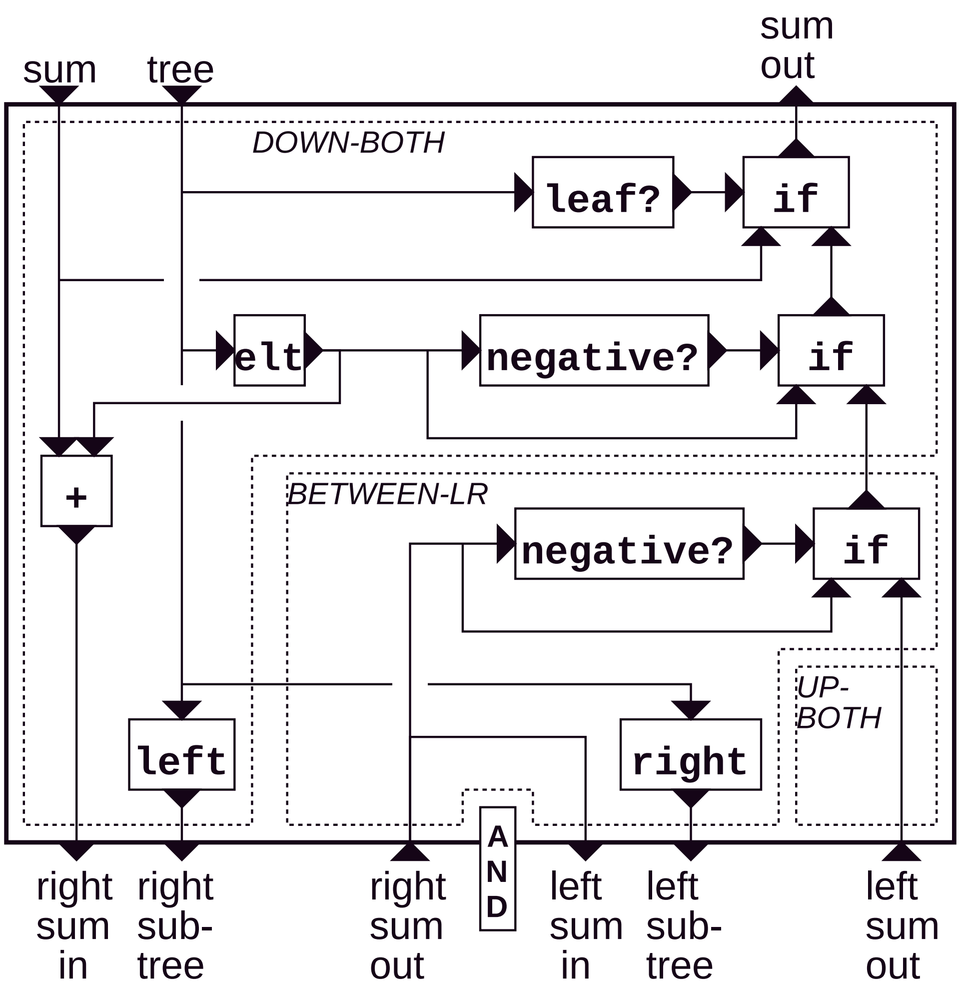


Figure 4.21: A tile in which both the *down-both* and *between-LR* shards have control arms. The `leaf?` operator tests for an elementless leaf node; the `elt`, `left`, and `right` operators return, respectively, the element, left subtree, and right subtree of a non-leaf node. If all the tree elements are non-negative, the tile generates a computation that returns their sum. However, if some of the elements are negative, the tile generates a computation that returns the first negative element encountered in a left-to-right pre-order traversal of the tree.

are more difficult to exhibit, and I postpone them for the moment.

In a *pre* tile, all operations are performed before the first subcall. An *in* tile allows operations between the two calls, while a *post* tile permits operations after the second call. Tiles **(b)**, **(c)**, and **(d)** in Figure 4.19 are examples, respectively, of *pre*, *in*, and *post* tiles. The shape names were chosen to reflect the kind of tree walk implied by the computations. Since the walks can be performed in either a left-to-right or right-to-left direction, the shape names are parameterized by this direction.

The sequential shapes can be ordered by specifity as follows:

$$\mathit{pre} < \mathit{in} < \mathit{post}$$

The shapes are defined so that every sequential tile has a unique most specific shape. These extend the simple *down* $<$ *up* ordering for linear tiles. In fact, note that both *pre* and *in* tiles are effectively tail-recursive in the second subcall.

Binary tile shapes essentially specify the dependencies between different parts of a tree computation. In this respect, they resemble *attribute grammars*, a declarative formalism in which programs can be specified in terms of the information dependencies between the nodes of a grammar-induced tree [DJL88]. (See Section 3.3.3 for a discussion of attribute grammars.)

4.2.5 Binary Down Tiles and Non-strictness

We now return to the notion of *binary down* tiles. These are tiles whose *up-both*, *up-left* and *up-right* shards are all trivial. But using the simple operations introduced so far, it is impossible to construct a binary tile with these properties! Intuitively, the only way to propagate requests to both of the subcalls is through a binary operator. But all the binary operators seen so far would have to be placed in the *up-both* shard because they cannot be peformed until their input values are available.

This problem can be circumvented by introducing a special binary node, `fork2`. A `fork2` node responds to a request by propagating requests to its two subnodes and then immediately returning the boolean true value (`#t`) without waiting for a result from either subnode. The tile in Figure 4.22 shows a simple use of `fork2` to print the elements of a tree. The `seqn` forces a node's element to be printed before any elements in either of its

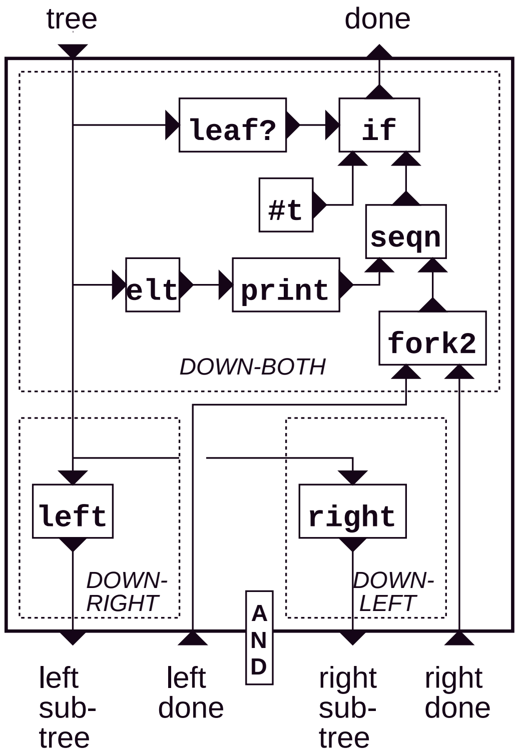


Figure 4.22: A tile with *binary down* shape that prints the elements of a tree. The `print` operator prints its input and then returns `#t`. `Seqn` forces a node's element to be printed before any element in its subtrees. `Fork2` returns after initiating both subcalls without waiting for their results.

subtrees. The `fork2` initiates the printing (in parallel) of the left and right subtrees.

`Fork2`'s strategy of evaluating arguments in parallel with returning a result is called *eager evaluation*. Eager evaluation is an example of a *non-strict* evaluation strategy, so called because the operator is performed before its arguments are completely evaluated. Another form of non-strictness is *lazy evaluation*, in which argument evaluation is suspended until it is required. The lazy data technique introduced in Section 3.1.5 is an example of the lazy evaluation strategy.

The non-strict behavior of `fork2` can change the timing constraints among the call and subcalls. A call to the tree-printing tile may return at any time with respect to the initiation or return of the subcalls (see Figure 4.23). This implies that tree elements may continue to print after the top-level call to the tree-printing tile has returned!

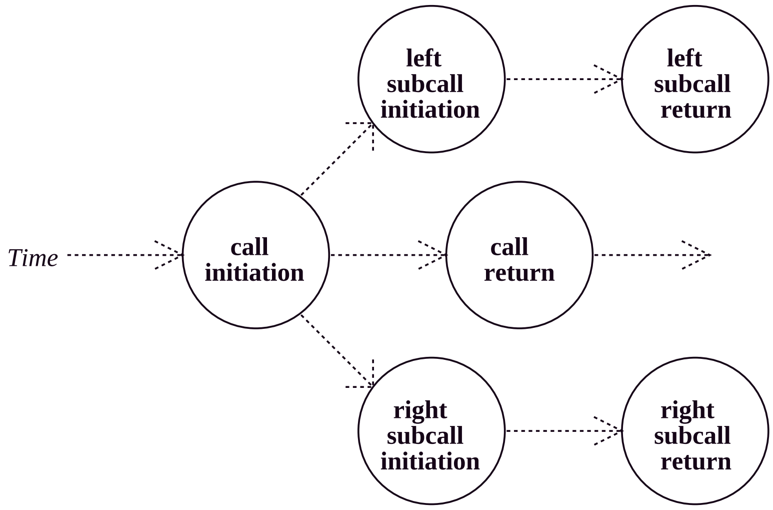


Figure 4.23: The use of the non-strict `fork2` operator changes the timing diagram for the events in a binary tile.

The ability of a non-strict computation to return before completing its computation is a powerful feature for expressing parallelism and speculative computation (see [Tra88, Hal85, Mil87]). Indeed, we will see in Chapter 7 that non-strictness is an essential technique for modularizing computations. However, because it can change timing relationships among calls and other operations, non-strictness complicates reasoning about programs (e.g., see [HA87]). For example, consider a sequential tile in which a non-strict operator links the result of one subcall to the argument of the other subcall. Then it is no longer true that the first subcall must return before the initiation of the second subcall! Due to these kinds of complications, I restrict the use of non-strict operators. Except for `if` and `seqn` (which are non-problematic instances of non-strict operators), I will assume that tile computations do not contain any non-strict operators.

4.2.6 Binary Computations

Under the unitilable assumption, a binary computation can be generated by dynamically replicating a binary tile throughout a binary trellis. As in the linear case, the resulting computations can be classified by computational shape — i.e., according to the timing relationships induced on their call events.

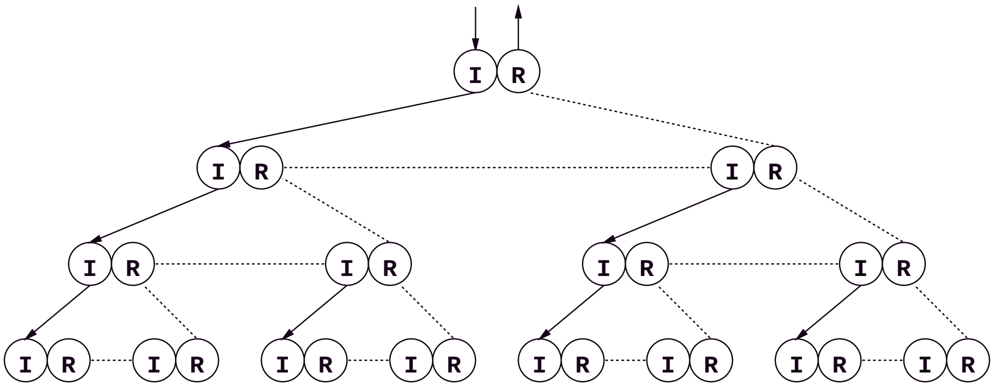
Figure 4.24 shows the three canonical shape configurations for a computation generated by a left-to-right sequential tile. These differ according to which events must happen simultaneously. Strings of events connected by dotted lines indicate paths along which no computation is allowed; all the events on the path occur at the same time. In a *pre*-shaped computation, every return must be connected by such a path to either an initiation event for a right subcall or the return even of the top-level call. A computation with *in* shape additionally forces at least one left subcall return to precede a right subcall initiation, while a *post*-shaped computation is characterized by a subcall return that precedes a call return. Due to the conditional nature of tiles, a *post* tile might actually generate any of the three computational shapes. However, a *pre* tile necessarily generates only *pre* computations, while an *in* tile can may generate only *in* and *pre* computations.

Two possible parallel shapes are illustrated in Figure 4.25. These are distinguished by the relationships among the return events. In a *binary down* computation, all return events are totally unconstrained with respect to each other, while in a *binary up* computation, some subcall returns are required to precede the return of the corresponding call. Note that the assumption that tiles do not contain non-strict operators effectively eliminates the *binary down* shape for tiles. However, by judicious use of non-strictness, it is possible to design computations having this shape.

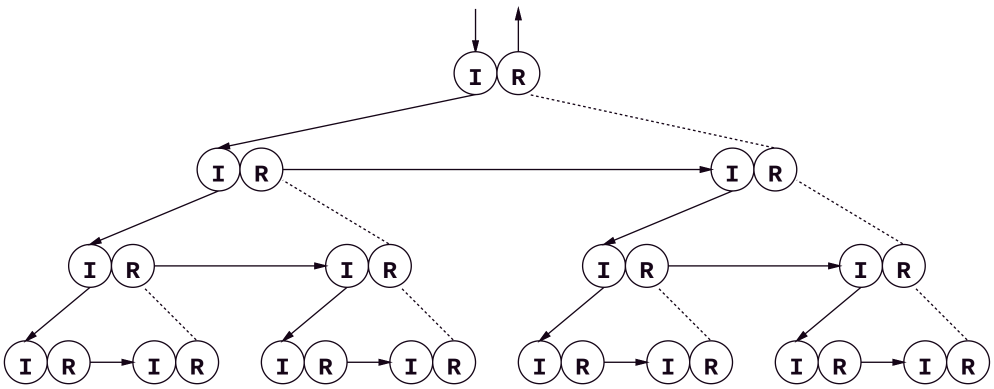
Unlike linear computations, binary computations do not decompose nicely into single *down* and *up* phases. In the sequential case, after a computation walks down one branch of the event tree, it must generally walk back up the branch in order to get to the next branch. In the parallel case, the concurrent evaluation of subcalls means that *down-both* operations in one branch may be interleaved with *up-both* operations in another branch.

The timing diagrams in Figures 4.24 and 4.25 underscore the difference between sequential and parallel computations. At the coarse-grained resolution of the timing diagrams, sequential shapes constrain time to follow a single path through the computation.[4] Because

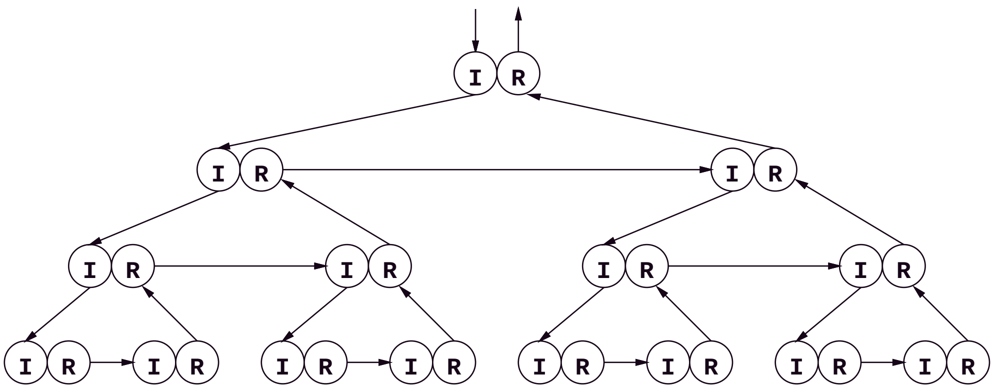
[4]A finer-grained analysis of individual operations might expose local branching of time. For example, an *across* operation in a call can be interleaved with any operations in both subcalls. The timing diagrams supress this level of detail.



(a) Computation having *pre* shape.

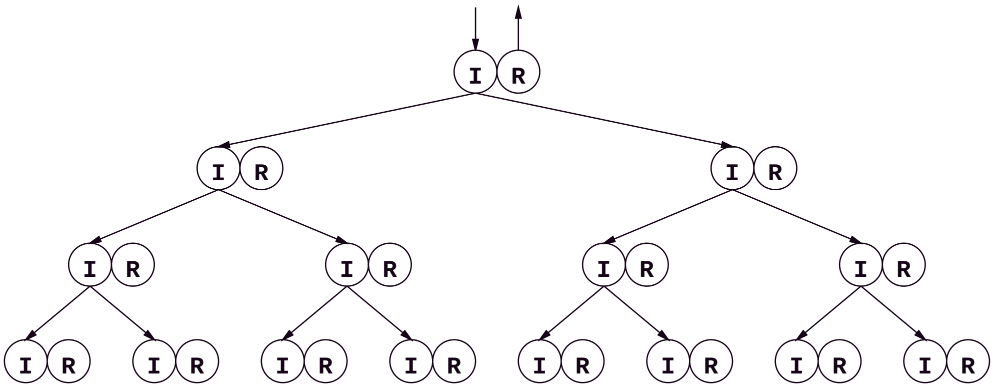


(b) Computation having *in* shape.

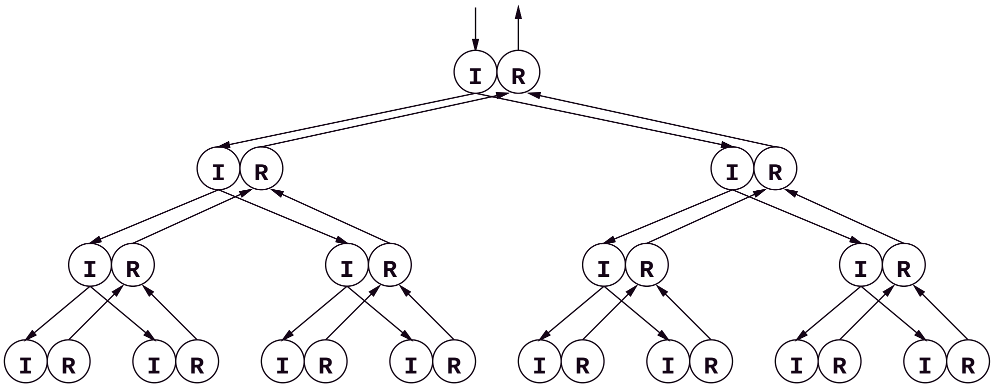


(c) Computation having *post* shape.

Figure 4.24: The timing configurations for sequential computations exhibit a single-threaded structure.



(a) Computation having *binary down* shape.



(b) Computation having *binary up* shape.

Figure 4.25: The timing configurations for parallel computations exhibit a multi-threaded structure.

such a computation effectively has a single locus of control, it is said to be *single-threaded*. Since the branching time nature of parallel computations permits multiple loci of control, they are said to be *multi-threaded*.

While the potential for evaluating subcalls in parallel makes multi-threaded computations attractive candidates for execution on parallel hardware, I will not explore this avenue. Instead, I will focus on fundamental behavioral issues that arise from the branching time nature of multi-threading. In particular, I will be concerned with the following two issues:

1. *Non-Determinism*: In the presence of side-effects, multi-threaded computations may be non-deterministic. Different interleavings of mutation operators can give rise to different results. Various atomicity and synchronization techniques (e.g., [Bir89b, Dij68, Bar92]) may be required to appropriately constrain behavior.

2. *Storage Requirements*: Multi-threaded computations have the potential for consuming much more storage than single-threaded ones. Since a single-threaded computation populates at most one branch of a binary trellis at any time, the space consumed by a single-threaded computation is linear in the depth of the trellis. In contrast, a worst-case multi-threaded computation can populate the entire trellis at once — this leads to space exponential in the depth of the trellis. This problem can be addressed by dynamic strategies (e.g., [BL93]) or by giving the programmer fine-grained control over parallelism (e.g., [GM84]).

Although they permit parallelism, multi-threaded computations do not require it. It is always possible to sequentialize a multi-threaded computation by adding additional constraints that one subcall return before another initiates. For example, many languages require procedural arguments to be evaluated in a particular order even when they have no side-effects. I will eschew these spurious constraints and only pay attention to constraints explicity specified by the programmer.

4.2.7 Discussion

Although we have only studied binary tree computations, the notions developed here generalize to tree computations where the branching factor can be node-dependent. In the more general setting, *binary down* and *binary up* shapes become *parallel down* and *parallel up* shapes. Sequential shapes need to be extended to handle the fact that any permutation of the subcalls defines a possible sequential ordering. Except for the new orderings, *pre* and *post* shapes remain essentially unchanged. However, *in* shapes must be extended to express the fact that an *in* operation may happen between any pair of subcalls consistent with a given sequential walk.

Dropping the unitilable assumption would also greatly increase the number of computations that can be expressed. Then it would be possible for a single *in* tile, say, to have *pre*, *post*, and *binary up* tiles for its subcalls. However, for reasons of simplicity, I will stick to the unitilable assumption for tree computations throughout the remainder of this report.