putation orders instruction execution according to the availability of data. A dataflow computer continuously processes *tokens* that deliver data to the instructions. On Monsoon, each token contains: (1) a *value*; (2) an *ip*, pointer to the destination instruction; (3) a *port*, specifying left or right input; and (4) an *fp*, the frame pointer. An instruction is executed during the processing of a token if that token delivers the last piece of input data needed by the instruction. When an instruction executes, zero or more tokens are produced. Program execution starts with the injection of tokens carrying the arguments and ends when there is no more token to be processed.

2.1 Matching and Frames

Central to this execution model is keeping track of which input tokens have arrived at each instruction, and the values carried on these tokens. The process of grouping together all the input operands of a particular instruction, which then enables the instruction for execution is called *matching.* We will further assume that there are at most two inputs to each instruction[1].

A matching mechanism needs to have the following two features: (1) a way of finding out whether the other operand has arrived and if so, fetch that operand; (2) a place for storing the first operand that arrives. In order to support re-entrant code, the mechanism must distinguish between tokens belonging to different invocations of the same piece of code.

On Monsoon, matching is achieved by using *frame* memory which is local to each processing element (PE). Each instruction that requires matching[2] is assigned a unique slot of frame memory which has two fields: (1) a *presence bits* field indicating the presence or absence of each operand; and (2) a *data* field. The former keeps track of whether any one of the operands has arrived, while the latter provides the space for storing the operand that arrives first.

In compiling Id code for Monsoon, we group instructions belonging to the same procedure into a *code block.* The matching slots of instructions belonging to the same code block are assigned contiguous frame memory locations which are collectively called a *frame.* During the processing of a token, the *fp* (base pointer to a frame) on the token and the offset encoded in the destination instruction allows the matching frame slot to be determined. Matching then proceeds according to the state of the presence bits at the frame slot.

Monsoon supports re-entrant procedure invocation by allocating a new frame for each procedure call[3]. By so doing, tokens from different invocations will have different *fp*'s and hence matching under the scheme described above will be able to distinguish them.

2.2 Loops in Id

We start by introducing the syntax with an example that computes $\sum_{i=1}^{n} i$ when called with `n`. The code is shown in Figure 1. This procedure starts by initializing `s` to 0 and then iterates through a loop `n` times. During each iteration, it "updates" `s` by adding the current value of `i` to the value of `s`. Updating a variable presents a problem in Id because

[1] The exception to this is a signal-tree which we will see later.

[2] Not all instructions need matching. Unary operations, for example, have only one input and do not require matching.

```
def sum_1_to n = {s = 0;
                  in
                     {for i <- 1 to n do
                         (next s) = i+s;
                      finally s}};
```

Figure 1: An Id procedure that computes $\sum_{i=1}^{n} i$ when called with n.

Id is basically a functional language. In our example, `s` is a *nextified variable* of the `for` loop with an assignment made to `(next s)` during each iteration of the loop. In the same iteration, `s` and `(next s)` are different identifiers. However, the `(next s)` computed in an iteration automatically becomes the value of `s` in the next iteration. The scope of a nextified variable extends to the `finally` part of the loop. Thus the `for` loop in the example returns the value of `(next s)` computed during the last iteration of the loop.

3 Past Implementations of Sequential Loops

Implementing sequential loops on Monsoon turns out to be very tricky because by using only one frame to execute the entire loop, the *fp* is the *same* for tokens belonging to different iterations. Loops in general have to be re-entrant but the default way of allowing re-entrant code on Monsoon requires using a new frame for each iteration. With only one frame, unless the compiler compiles in mechanisms to separate tokens belonging to different iterations of the loop, tokens from different iterations destined for the same instruction can erroneously meet under the hardware matching scheme. We call such an event, a *conflict.*

In this section, we describe past attempts at solving this problem. We begin with an "obvious" implementation that uses a barrier at the end of each iteration. This is, unfortunately, incorrect. We next look at a straight forward fix which we call the brute-force 2-barrier approach. Finally, we look at a schema found in Culler[5] which was the previous best schema.

3.1 An Incorrect, One-barrier Sequential Loop Schema

A first attempt at solving the conflict problem has the compiler insert a *barrier* at the end of each iteration. The barrier separates the current iteration from the next one and ensures that the next iteration does not begin execution until the current one has completed. Such a barrier can be implemented as shown at the bottom of Figure 2.

Figure 2 is the dataflow diagram for the code in Figure 3 which computes the n^{th} Fibonacci number. We can discern four regions in the dataflow diagram. At the top is a shaded oval blob labeled "Loop Predicate", which computes a boolean value indicating whether another iteration should be executed. Below this blob is a row of ovals, each with a "T" and an "F" output. Each oval is a switch that directs the value coming in on top to either the "T" or "F" output depending on the boolean value that comes in by the side. Next comes a big grey region labeled "loop body" where the bulk of the computation of each iteration is carried out. At the bottom is the barrier which consists of two parts: (1) A *signal tree* (the iteration termination signal tree) that collects the `next` values of the nextified variables.