AN ASYNCHRONOUS PROGRAMMING LANGUAGE AND COMPUTING MACHINE*

by

Arvind

Kim P. Gostelow

Wil Plouffe

Department of Information and Computer Science
University of California, Irvine
Irvine, California 92717

December 8, 1978

---

*This work was supported in part by NSF Grant MCS76-12460: The UCI Dataflow Architecture Project.

Z
699
C3
no. 114A

ABSTRACT

Dataflow is presented as an alternative to the von Neumann model as the basis for computer system design. The need for a new semantic basis is supported by current research both in software methodologies and in computer architecture. Dataflow systems emphasize asynchronous and functional computation. We present a high-level dataflow language Id, and its companion base language. Id supports programming with streams, programmer-defined data types, and facilities for nondeterministic programming. The base language when interpreted by the unfolding interpreter generates a potentially large number of independent activites which can be executed concurrently by a dataflow machine. The unfolding interpreter seems very promising for implementation on a machine composed of large numbers of LSI processors.

Key Words and Phrases: Dataflow, high-level language, machine language, streams, resource management, abstract data types, parallel processing, computer architecture, program schemata.

CR Categories: 4.20, 4.21, 4.32, 4.35, 5.24, 6.21

1. Introduction

It is well known that LSI technology is capable of economically producing large numbers of similar, small and complex devices. It is equally clear that use of LSI technology has not yet provided a breakthrough in the computing power available in a single system. Rather, the best that has been accomplished is simple reduction in the physical size of all too familiar systems.

Our goal is to exploit LSI technology by making these complex devices the building blocks of a large general-purpose computer. Such a machine, comprising hundreds, perhaps thousands of small processors, must allow these processors to operate asynchronously and (almost) independently. Each processor will accept and perform a small task generated by a program, produce partial results, and then send these partial results to other processors in the system. Thus many processors would cooperate towards the common goal of completing the overall computation. [illegible] A natural concomitant effect of such behavior would be the potential for increasing speeds of computation as new processor modules are added to the machine.

Many computer architects have imagined machines that might exhibit such behavior. However we are convinced that such machines cannot be successfully constructed simply by devising some appropriate bus and machine interconnection scheme, or by designing a machine which, for example, can efficiently manipulate arrays or interchange numbers. Rather, the difficulties are due to one of the fundamental bases of computer design: the von Neumann model. Indeed, more than 30 years have passed since John von Neumann first laid down the model that virtually all machines and languages have adopted ever since. The von Neumann model has become so ingrained in our thinking that we rarely even consider it, let alone question it, but this is precisely what has prevented the creation of the kind of machine just described.

Two particularly troublesome attributes of the von Neumann model are [Dennis73, GIMT74, Backus78] sequential control, and memory cells. Sequential control is troublesome since it prohibits the asynchronous behavior and distributed control that we consider essential to the machine we wish to design. It also burdens the programmer with the need to explicitly specify (or to employ an analyzer to determine) exactly where concurrency may occur. The concept of a memory cell, along with the idea of assigning a value, presents a difficulty since its existence forces the programmer to consider not only what value is being computed, but also where that value is to be kept. This places additional burden on the programmer and presents particularly thorny problems in program verification. Furthermore, the use of memory cells to coordinate asynchronous processes causes serious problems in a distributed machine. Such coordination calls for rather complex synchronization controls to ensure orderly use of shared variables. These controls are difficult to design into a machine and may be very costly in execution time. They are also tedious for programmers to use, especially where large numbers of activities are to be coordinated.

We contend that the above two cornerstone principles of the von Neumann model (sequential control and the memory cell) must be rejected if we are to realize our goal. We offer evidence in the rest of this paper in support of this contention. In place of these two principles, we adopt a language that is asynchronous except where synchronization is explicitly specified (i.e., no sequential control), and in which values are the subject of computation rather than the locations where those values are kept (i.e., no memory addresses). An asynchronous language assumes computations are unrelated, and thus concurrent, unless otherwise specified. The absence of memory cells ensures that only simple control mechanisms are needed to coordinate access to data, since races to "store" data will never occur. Such a semantic basis should work well with a machine composed of many asynchronous cooperating processors.

The principal arguments against the von Neumann model are not original to these authors and have been noted by several researchers [Dennis73, GIMT74, Backus78]. The approach is one of avoiding the difficulties currently plaguing multiprocessor design rather than suffering with them. Rejecting von Neumann's model may at first seem a radical approach. However, a brief survey of much of the current work in programming language design and software methodology reveals that this is, in fact, taking place already, albeit in a much disguised form and at a very slow pace. For instance, structured programming can be viewed as an attempt to produce programs that are more functional in their operation. According to modern programming methodology, a procedure which produces results by modification of shared variables is less desirable than a procedure that returns values as the result of a function call. The fact that writing such a function-type procedure is not even possible in many languages (particularly if more than one result or an array of results is to be returned) is not the fault of the functional approach, rather it is the fault of language restrictions that do not allow the returning of such values. We can also give several examples of the movement away from the von Neumann model in the field of programming language design. Note that EUCLID has imposed many restrictions on PASCAL that make variables inaccessible to procedures when those variables are declared outside those procedures, the effect of which is to force procedures closer to the ideal of a mathematical function. Also, the current interest in (abstract) data types [Guttag77, LSAS77, SWL77] points away from the semantic base implied by a von Neumann machine, since functionality (information hiding) appears essential to both data and program abstraction. Finally, we can observe the recent work on the linguistic aspects of resource control [Jones77] as a movement away from arbitrary specification of program synchronization using semaphores to more highly controlled and encapsulated specifications such as monitors [Brinch-Hansen72, Hoare74]. This movement is in the direction of providing the programmer with a more functional view of a computation that involves resources. However, resource management is one of the areas in programming language design which has not yet seen solutions that go far enough in the direction of functionality to provide hard evidence of this movement. We hope to convince the reader of what can be accomplished with an even more functional approach [AGP77] by giving a concrete example later in this paper (Section 5). Lastly, we mention program verification, where some researchers have noted the potential benefits of a language with semantics more closely tied to mathematical languages. The concept of a memory cell is not natural to mathematics, and can often complicate what otherwise would be a simple proof of correctness [Guttag77, Ashcroft & Wadge76, Kahn74, Kahn & MacQueen77].

The chief thrust of the above argument is that a proposal to replace von Neumann's model with a new semantic model is not capricious. All too often studies on the high cost of software have ignored the architectural base on which software and software tools have been developed and continue to exist. The unstated assumption is that von Neumann semantics will remain. Our position is that if we are going to fully utilize LSI technology, we cannot retain the von Neumann basis. We also believe that the required semantic foundation coincides with the principles now evolving in software engineering and programming language design. However, by beginning with a new semantic base rather than continuing to develop "restrictions" on the old, we see a much smaller and more elegant semantics resulting [Kahn74, Arvind & Gostelow77a, Kosinski76, 78] -- an essential for future development.

One system that has been proposed in the past and which incorporates new principles more compatible with the needs we see, is <u>dataflow</u> [Dennis73, Bährs72, Kosinski73, Arvind & Gostelow77b]. (Pure LISP [McCarthy60] and FFP languages [Backus73, 78] were not chosen for adoption because, even though their semantic bases are elegant and functional, neither caters to asynchronous and history sensitive operation.) A dataflow program is a set of partially ordered operations on operand values where the partial order is determined solely and explicitly by the need for intermediate results; operationally:

1. a dataflow operation executes when and only when all of the required operands become available, and

2. a dataflow operation is purely functional and produces <u>no</u> side-effects as a result of its execution.*

Arguments in the past against dataflow have centered around the lack of a high-level language, the questioned ability of people to program in such a language (were it to exist), the inability to handle database problems, and efficiency. In this paper we provide definite answers to some of these objections. We present a complete high-level dataflow language incorporating all of the usual programming concepts, as well as some new concepts not usually found in contemporary languages (for example, streams, functionals, and nondeterministic programming). Also, the implementation of these concepts (both old and new) is often easier in dataflow than in conventional languages due to the simplicity of dataflow semantics. In this category we include definition and manipulation of procedures, programmer-defined data types, and operator extensionality. Our language also has the capability to handle resource (database) problems, a capability which current applicative languages do not have. Regarding "efficiency," one should not evaluate dataflow in terms of a von Neumann implementation, for dataflow not only allows a new kind of machine design but in fact <u>requires</u> it. It is most important that dataflow languages be considered in their own terms and not be forced to fit into measures valid only for conventional systems.

Two languages will be described here: a high-level language

---

*In this respect dataflow models are very similar to applicative programming models.

Id (for Irvine dataflow), and a base machine language that serves as the semantic language of Id. The syntax of Id is not important beyond the fact that some syntax is needed to communicate the ideas. However, several compilers which accept the syntax presented in this report are currently in use. In Section 2 we show how to write elementary Id programs and we explain the meaning of these programs in terms of their base language translations. In Section 3, we give more details on how base language programs are interpreted by a machine and how these programs achieve highly concurrent operation. Streams are introduced in Section 4, while issues concerning indeterminacy and resource managers are discussed in Section 5. Programmer-defined data types and functionals are presented in Section 6, a brief discussion of dataflow architectures is contained in Section 7, while Section 8 summarizes the work and presents our conclusions.

2. Elementary Programming in Dataflow

Id (for Irvine dataflow) is a block-structured expression-oriented single-assignment language. Throughout this paper Id syntax is explained through examples and the semantics by the corresponding base language schemata. The base language is used not only to define and explain Id, but is also the machine language to be directly executed by our dataflow computer. Since we allow a program to be written only in Id, we are less concerned with how the base language operators behave in isolation than how they behave in concert with one another in building a basic schemata. A program in Id is a list of expressions. In this section, values and the four most basic expressions -- blocks, conditionals, loops, and procedure applications -- are explained.

2.1 Values

Id variables are not typed. The internal representation of values is simply self-identifying and type is thus associated with a value and not with a variable. However, the primitives necessary to test the type of a value, and to coerce a value to a different type are assumed available.

There are ten primitive types of Id values: integer, real, boolean, string, structure, procedure definition, manager definition, manager object, pdt (programmer-defined data type), and error. The first four need no discussion; structure values are discussed below; procedure definition values will be discussed in Section 2.5; manager definitions and manager objects will be discussed in Section 5; while pdts will be discussed in Section 6. Error values are not discussed at all (see [Plouffe78]).

A structure value is either the distinguished empty structure Λ or a set of <selector:value> ordered pairs. A selector is an integer or string value; a value is any Id value. An example of a structure is shown in Figure 2.1a where "name", "height", "weight", and "age" are string selectors (string selectors are not quoted when used in figures). There are exactly two operators defined on structure values: SELECT and APPEND. If t is the structure value in Figure 2.1a, then values can be selected from t, for example, by writing t["weight"] and t["height"][1] (giving 175 and 6, respectively). The APPEND operator is somewhat more complex. Given a structure, a selector, and a value to be associated with that selector, APPEND creates a new structure. For example, Figures 2.1b and 2.1c are the results of the append operations t+["sex"]"M" $\triangleq$ APPEND(t,"sex","M") and t+["sex"]"M"+["weight"]180 $\triangleq$ APPEND(APPEND (t,"sex","M"),"weight",180), respectively. Most importantly, the structure created by an APPEND is neither the original structure t nor any modified version of t. Rather each APPEND creates a new and logically distinct structure, and the old structure (t in the examples of Figure 2.1b and 2.1c) has an existence of its own, possibly concurrent with the new structures. This means that the value of t may be referenced by some other expression in the program even after the APPENDs have been completed. SELECT and APPEND may be summarized by the following equations (where "-" a variant of APPEND means to delete a selector)

$$\begin{aligned}(t+[s]v)[s'] &= (\underline{\text{if}}\ s'=s\ \underline{\text{then}}\ v\ \underline{\text{else}}\ t[s'])\\ \Lambda[s'] &= \underline{\text{error}}\\ (t-[s])[s'] &= (\underline{\text{if}}\ s'=s\ \underline{\text{then}}\ \underline{\text{error}}\ \underline{\text{else}}\ t[s'])\\ \Lambda-[s'] &= \underline{\text{error}}\end{aligned}$$

Some syntactic shorthands are available in Id for manipulating structure values. The expression x[1,2] means (x[1])[2], and in the case of string selectors, the notation x.weight can be used instead of the more cumbersome x["weight"]. The angle-bracket notation

<height:<6,5>, weight:175, age:33>

can be used for construction instead of

Λ + ["height"](Λ + [1]6 + [2]5) + ["weight"]175 + ["age"]33

where <6,5> refers to a structure with 2 values hanging by the integer selectors 1 and 2 respectively.

2.2 Block expressions

To evaluate the two roots of a quadratic equation we can write the following Id program (i.e., an expression) that contains two expressions:

```
        ((-b+sqrt(b↑2-4*a*c))/(2*a),
              (-b-sqrt(b↑2-4*a*c))/(2*a))                    (2.1)
```

However it is often more efficient and convenient to first compute several intermediate results as shown in the following *block expression*:

```
        ( x ← sqrt(b↑2-4*a*c);
          y ← 2*a
          return (-b+x)/y, (-b-x)/y )                        (2.2)
```

Expressions (2.1) and (2.2) each require three inputs (a, b, and c) and produce two (ordered) outputs. Expression (2.2) compiles into the base language schema shown in Figure 2.2. Note that an assignment statement simply names the output(s) of an expression. The name itself is called a *variable* and is used to specify interconnections among operators (the boxes in Figure 2.2). Assignment statements in a block are separated by semicolons and can always be commuted without affecting the result(s). The inputs to a block expression are exactly those variables referenced but not assigned within the block. The return clause is the last item and specifies the (ordered) outputs.

Assignment in Id is not an operator as it is in other languages, rather it is a specification to the compiler to label an output. The scoping rules of Id are similar to Algol with some important exceptions. Since there are no explicit declarations in Id, assignment is equivalent to declaration. Thus assignment to a variable name which is defined in an outer block is equivalent to defining a new variable. Hence, variables x and y are not visible outside the block expression (2.2); and if the same names x and y were also assigned outside (2.2) they would not be visible inside (2.2). There is a further restriction that an Id variable is assigned (defined) exactly once in a block. This single assignment rule [Chamberlin71] makes the connection shown in Figure 2.3 impossible.

Values in the base language are carried by tokens that flow along lines. Note that a constant in Id does not represent a value but rather a constant function which produces that value when triggered by a token carrying any value (e.g., "4" is triggered by "a" in Figure 2.2). Whenever a token encounters a fork while traversing a line, it replicates and follows all out-branches of the fork (figure 2.4). Notice that a token that carries a structure logically carries the whole strucuture as its value. In an actual machine this is impractical where large structure values are involved. However, the fact that structures are acyclic and that dataflow operators are pure functions has allowed Dennis [Dennis73] to devise a technique whereby a memory may be used to store the actual structure while only pointers to the structures are actually carried by the tokens. That is, the underlying implementation of structure values in dataflow may use pointers, share common substructures, employ reference count garbage collection, and use many other techniques in order to reduce overhead [Dennis74, Newell & Tonge60]. We do not discuss memory mechanisms in detail here, but it is important to emphasize that any such memory system that may be used to implement dataflow is never seen by the programmer. A memory system would be present only to reduce the amount of information that would otherwise be carried by a token.

According to the first principle of dataflow, an operator may execute when and only when all its required input tokens have arrived. The operator executes by absorbing all input tokens, computing a result, and producing an output token that carries that result as its value (Figure 2.5). Note that the operators internal to the block expression of Figure 2.2 will start executing as soon as any tokens on lines a, b, or c arrive.* Thus an Id expression is asynchronous because all of its subexpressions are independent of one another unless otherwise constrained by an explicit need for intermediate results. This approach to expressing asynchrony is in contrast to the usual method where sequencing is

----
*Please note the use of triggers for constant functions in Figure 2.2. The choice of which line to use as a trigger affects only the time of execution of an expression and not the final results.

the default and parallelism is explicitly specified by special programming constructs such as *cobegin*-*coend* which often have side effects. The need for synchronization in computing intermediate results is removed by the single-assignment rule that eliminates the possibility of a race, and furthermore implies that Id programs are determinate (unless, of course, some operator is used which is internally nondeterministic, as in Section 5)[Patil70, Kosinski76, 78, Arvind & Gostelow77a].

2.3 Conditional expressions

Consider the Id conditional expression

$$(\underline{\text{if}}\ p(x)\ \underline{\text{then}}\ f(x)\ \underline{\text{else}}\ g(x)) \qquad (2.4)$$

and its base language translation in Figure 2.6. Whenever a token arrives on line x the predicate p is evaluated to produce a boolean value. If the predicate is *true* then the token from x is sent by the SWITCH operator to schema f, otherwise it is sent to schema g. The symbol $\otimes$ is used in base language schemata to indicate a legal merging of two lines, as in this case where only one of the two lines will actually receive a value. The notion of a legal schema is made more precise in Section 3.

To execute, a conditional expression requires a token on each of its input lines regardless of the branch to be taken. For example, the expression

$$(\underline{\text{if}}\ p(x)\ \underline{\text{then}}\ f(x)\ \underline{\text{else}}\ g(x,y)) \qquad (2.5)$$

always takes an input token from y (Figure 2.7), but whenever p(x) is *true* that token is simply absorbed and is not used. The proper order of tokens flowing along all lines is thereby maintained, and regardless of whether an Id expression is a block, a conditional, or any other kind of expression, one token is absorbed from each input and one token is produced for each output on each execution of that Id expression. Note that an entire expression behaves similar to a primitive operator.

In a conditional expression the *then* clause and the *else* clause must contain an equal number of expressions. Thus the following is illegal:

```
y,z ← (if p(x) then f(x),1  else  g(x))   **illegal**
```

2.4 Loop expressions

Loop expressions in Id are provided essentially as a convenience for writing programs as they can be regarded as a special case of procedures. However, we will show in Section 3 that our implementation of loops provides more asynchrony than procedures. A loop expression comprises four parts: an *initial* part, a predicate to decide further iteration, a loop body, and a list of expressions to return values from the loop. A loop expression to compute $\sum_{i=1}^{n} f(i)$ is

```
( initial i ← 1;
          sum ← 0
  while i≤n do
       new i ← i+1;
       new sum ← sum+f(i)
  return sum)                              (2.6)
```

An Id loop is a set of first degree recurrence equations. For example, a set of recurrence equations for computing the above values of i and sum are

$$\begin{aligned} i_{j+1} &= i_j + 1 & \text{where } i_1 &= 1 \\ sum_{j+1} &= sum_j + f(i_j) & sum_1 &= 0 \end{aligned}$$

The loop body specifies that two values called new i and new sum are to be created at each iteration. However, any reference to a recurrence variable in the body of a loop refers to the "old" value of that variable unless the reference is preceded by the word "new". Thus the i in line 5 of (2.6) does not refer to the value new i computed in line 4. (The value of new i could be referenced in line 5 by writing new i instead of just i.) The translation of expression (2.6) into the base language is given in Figure 2.8. Note that changing the order of statements in the loop body affects neither the results nor the base language translation. (The reader will have to wait until Section 3 to understand the meanings of the D, D^{-1}, L, and L^{-1} operators. These operators do not affect the values of the tokens passing through them, and for now we treat them as identity functions.)

Now let us briefly consider the execution of (2.6). Suppose function f of line 5 takes a long time to execute. The loop predicate $i \le n$, however, does not depend upon the evaluation of f(i). Therefore it is possible for several tokens to accumulate on line i going into function box f, since generating tokens with values from 1 to n is a relatively fast process. Now if i were treated as a memory cell then the notion that i might be several values at the same instant would be meaningless. We will show in Section 3 that the machine's interpretation of the base language is such that instead of simply accumulating tokens on line i, many instantiations of function f may proceed concurrently. This greatly increases the apparent asynchrony and concurrency of loop expressions.

Id supports many different loop constructs such as for-loops, repeat-until-loops, and for-while-loops. The semantics of all loops (except for those involving streams*) are encompassed within the general while-loop construct given in expression (2.7).

```
( initial x ← f(a)
  while p(x,c) do
        y ← g(x,c);
        new x ← h(x,y,c)
  return r(x,c) )                                    (2.7)
```

In an actual loop expression there might be more than one variable in category a, x, y, or c. Variables assigned in the initial part (x variables) circulate in the loop and thus have both old and new values. Variables not assigned in the initial part but that are assigned in the loop body (y variables) are simply intermediate results and can be used only within the body; a y variable never circulates. Variables referenced but not assigned in a loop (or assigned only in the initial part) are loop constants (c variables). A c variable behaves exactly like an x variable in that it circulates (see n in Figure 2.8) and one can assume that a statement new c ← c exists in the loop body. All variables referenced on the right-hand side of assignments in the initial part (a variables) are treated as inputs to the loop expression and must originate outside that loop expression. Hence, x ← f(x) appearing in the initial part of a loop expression would be a valid assignment statement only if x were defined outside the loop expression. Assignments of the

----
*Streams are discussed in Section 4.

type x<-x may be omitted from the initial part without ambiguity. Finally, the assignment

new x[i] ← v

stands for the APPEND operation new x← x+[i]v.

2.5 Procedure applications and definitions

Figure 2.2 showed the Id sqrt function implemented by the machine primitive SQRT. If sqrt were instead a procedure application, then the SQRT box would be replaced by the schema of Figure 2.9. The APPLY operator expects a token carrying a procedure definition value and another token carrying the argument value. It applies the procedure definition to the argument when both have been received. Note that sqrt is the name of a line, and we would now say that expression (2.2) needs sqrt in addition to a, b, and c as inputs. The line sqrt is connected to a box (a constant function) that outputs the following procedure definition value (or a compiled encoding) whenever triggered.

```
procedure(a)(initial x ← a/2
             while abs(x↑2-a)>.000001 do
                   new x ← (x↑2+a)/2*x
             return x)                          (2.8)
```

That is, just as the Id constant 5 actually represents a constant function that produces 5 as its value, so does a procedure definition imply a function that produces that procedure as its value. In all other respects, variable sqrt is like any other variable and can be passed as an argument to a procedure, appended to a structure, or operated on by any operator defined on the type of value carried by sqrt.

The APPLY operator accepts a procedure and one argument, and produces exactly one result. However, Id syntax permits procedure definitions and applications with multiple arguments and multiple results. For example let f be the following procedure definition for multiplying two matrices a and b of sizes ℓ by m and m by n respectively

```
    procedure (a,b,ℓ,m,n)
     (initial c ← Λ
      for i from 1 to ℓ do
         new c[i] ←(initial d ← Λ
                    for j from 1 to n do
                      new d[j]← (initial s<-0
                                 for k from 1 to m do
                                     new s←s+a[i,k]*b[k,j]
                                 return s)
                    return d)
      return c)                                          (2.9)
```

Procedure f is applied by writing

f(x,y,2,3,2)

which means

APPLY(f,<x,y,2,3,2>)

Every Id procedure definition is translated to expect one structure parameter α (with integer selectors) that contains all the arguments. For example expression (2.9) becomes

```
          f ← procedure (α)
                (a ← α[1]; b ← α[2]; ℓ ← α[3];
                 m ← α[4]; n ← α[5]
                 return (initial
                                   ⋮           ))
```

Note that if there are more actual than formal arguments, the extra actual arguments will be ignored; if insufficient actual arguments are sent, error values will be produced for some parameters.

Multiple results are handled in a similar way. A result structure β is formed inside the procedure and taken apart in the environment where the results are used. For example,

$$\langle x,y\rangle \leftarrow f(a)$$

means

$$\beta \leftarrow f(a);\ x \leftarrow \beta[1];\ y \leftarrow \beta[2]$$

Both $x \leftarrow f(a)$ and $\langle x\rangle \leftarrow f(a)$ mean that x receives the value $\beta[1]$.

It is also possible to give a name to an Id procedure which can be very useful in writing recursive programs. Below, the named procedure f calculates the factorial function

```
    y ← procedure f(n) (if n=0 then 1 else n*f(n-1))   (2.10)
```

as does the following

```
    z ← procedure (n,f) (if n=0 then 1 else n*f(n-1)) (2.11)
```

Note that z(3,z) = y(3).

Another operator defined on procedure values is COMPOSE. This operator is actually a simple but very powerful functional. COMPOSE takes the input procedure value and "freezes" one or more of the procedure's formal parameters to particular actual values, and then removes the parameters that were frozen from the formal parameter list and outputs the resulting procedure. For example, we can freeze the parameters in position 3, 4, and 5 (i.e., parameters ℓ, m, and n) in the matrix multiply procedure of expression

(2.9) to the values 2, 3, and 2, respectively, by

```
r ← compose(f,<<3,2>,<4,3>,<5,2>>)
```

The procedure value assigned to r, when applied, behaves as if the programmer had written

```
r ← procedure(a,b)(ℓ←2; m ← 3;  n ← 2
                   return (          ))
```

The argument of compose is a list of subarguments of the form <formal-parameter-position, value>.

As another example, we use z from (2.11) and write

```
w ← compose(z,<<2,z>>)
```

so that w(3)=z(3,z)=3! . In fact, named procedures are implemented by COMPOSE. Thus statement (2.10) actually translates into

```
y ← (f′ ← procedure(f,n)
           (if n=0 then 1
                   else n*compose(f,<<1,f>>)(n-1))
      return compose (f′,<<1,f′>>))
```

so that any reference to f within y, regardless of how y is further composed, always refers to the original definition of f. A named procedure definition f is translated according to the following steps:

1. Construct a new procedure definition f′:
   - insert the original procedure's name f as the first parameter of the argument list; and
   - alter the code inside the original procedure f by replacing every occurrence of f by the code compose (f,<<1,f>>).
2. Return the procedure compose(f′,<<1,f′>>).

The COMPOSE operator is useful for tailoring a procedure to special forms by freezing certain parameters. COMPOSE is used extensively in Section 6 for implementing programmer-defined data types. It furthermore provides a way in which (dynamic) program linking can be performed, since such linking is actually just the freezing of certain formal parameters to actual parameters, where the actual parameters would generally be subprograms.

2.6 Examples

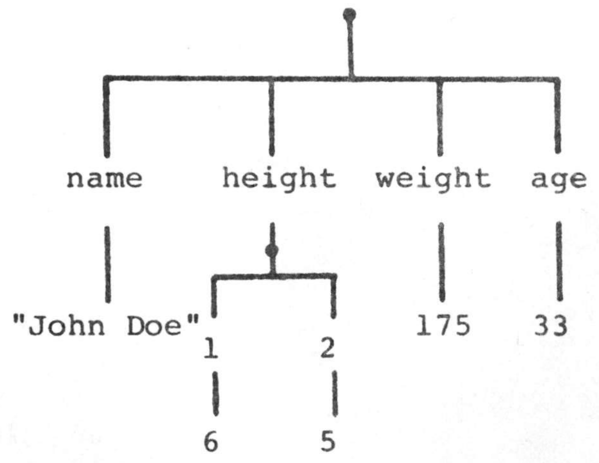
Hoare's quicksort written in Id is presented in expression (2.12). We have chosen a conventional algorithm for two reasons. First of all, we want to show that a programmer proficient in Algol-60 would have no difficulty writing dataflow programs. Second, we want to show that even conventional algorithms often contain significant concurrency when expressed in Id, though a complete discussion of this point must wait until Section 3 because it is related to the unraveling interpreter of the base language.

```
procedure quicksort(a,n)
     (middle ← a[1];
      below,j,above,k ←
              (initial below ← Λ; j ← 0;
                       above ← Λ; k ← 0
               for i from 2 to n do
                   new below,new j,new above,new k ←
                       (if a[i]<middle
                           then below+[j+1]a[i],j+1,above,k
                           else below,j,above+[k+1]a[i],k+1)
               return (if j>1 then quicksort(below,j) else below),j,
                      (if k>1 then quicksort(above,k) else above),k)
      return (initial t ← below+[j+1]middle
              for i from 1 to k do
                  new t ← t+[i+j+1]above[i]
              return t))                                    (2.12)
```

On a single-processor machine, quicksort takes an average of $O(n \log n)$ time and in the worst case $O(n^2)$ time. The above Id counterpart, when compiled into the base language and executed under the unraveling interpreter, has an average of $O(n)$ and a worst case behavior of $O(n^2)$, but requires an average of $O(n)$ processors. The time complexity is reduced because of the possibility of executing the recursive procedure calls in parallel. Given sufficient processor resources, this will occur automatically and without any analysis of the program. The mechanism which accomplishes this is the unraveling interpreter discussed in Section 3.

Consider briefly the matrix multiply procedure given in expression (2.9). It executes in $O(\ell+m+n)$ time utilizing in the worst case $O(\ell mn)$ processors and in the best case $O(\ell n)$ processors. The unraveling interpreter will try to execute all of the multiplications and ℓn of the ℓmn additions in parallel, thus reducing the usual time complexity of $O(\ell mn)$ to $O(\ell+m+n)$.

Id is not just another idiosyncratic language. It is a high-level dataflow language which has an asynchronous control structure. This is required for the development of the base language (Section 3) and a machine that can utilize a large number of processors (Section 7).



gure 2.1a A structure value t with string and integer selectors

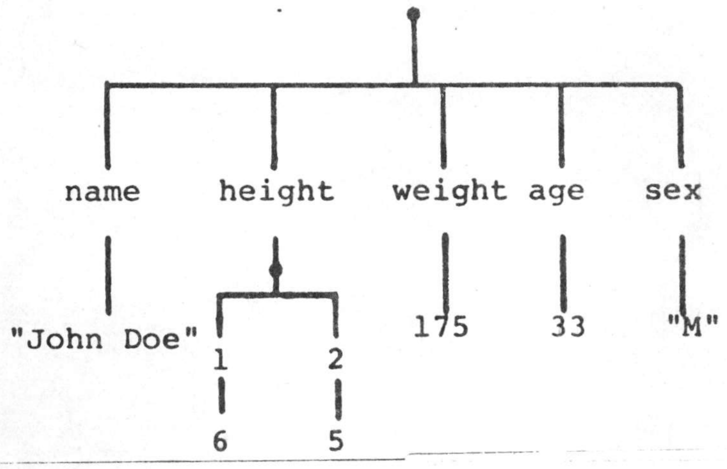


Figure 2.1b The structure t + ["sex"] "M"

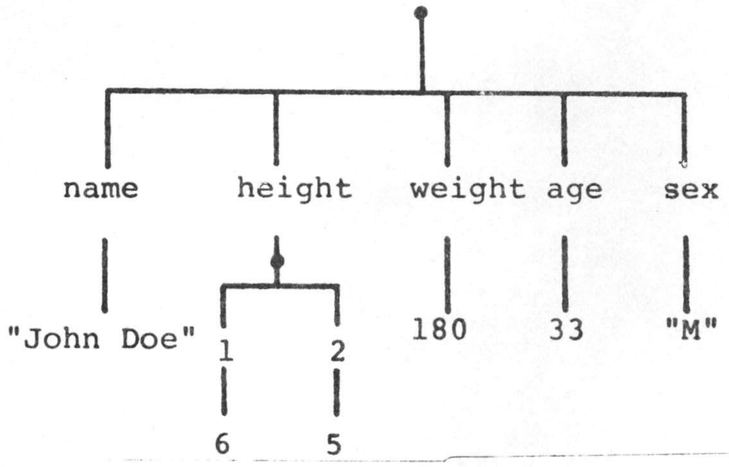


Figure 2.1c The structure (t + ["sex"] "M") + ["weight"] 180

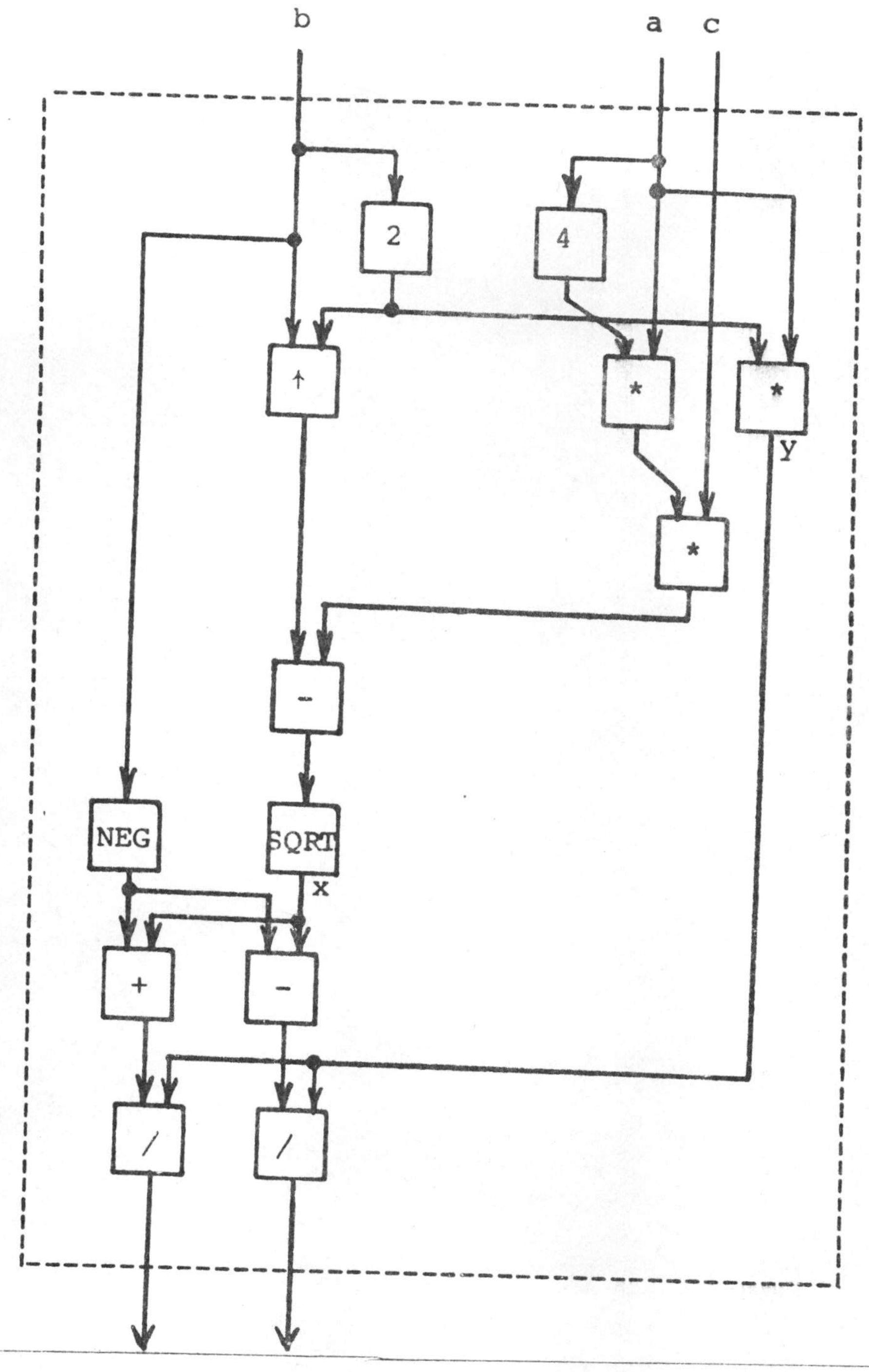


Figure 2.2 Compilation of the block expression (2.2)

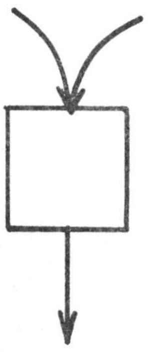


Figure 2.3 An illegal connection

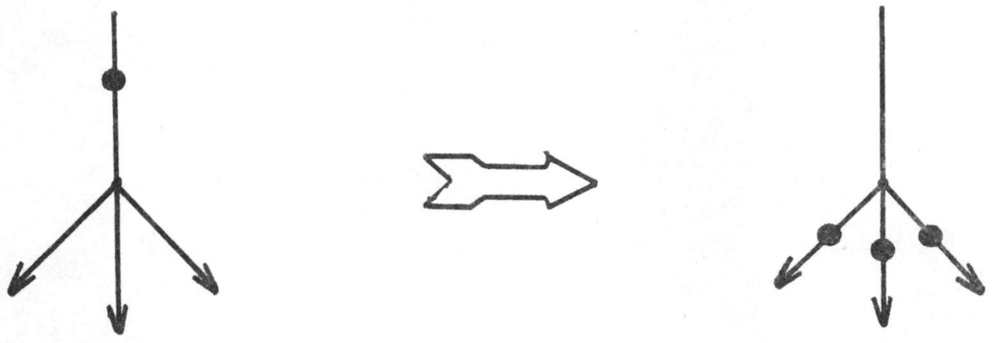


Figure 2.4 Behavior of a fork

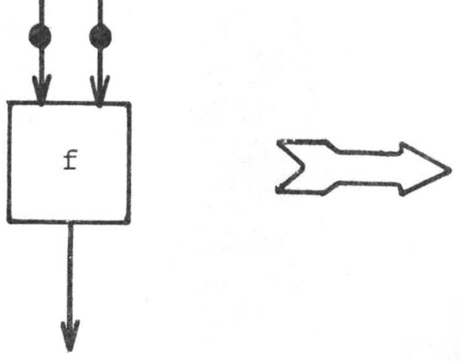


Figure 2.5 Execution of a dataflow operator

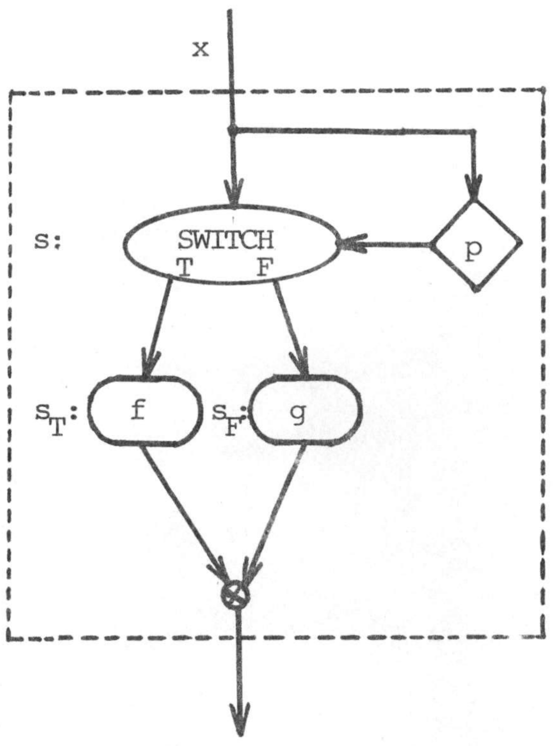


Figure 2.6 Compilation of condition expression (2.4)

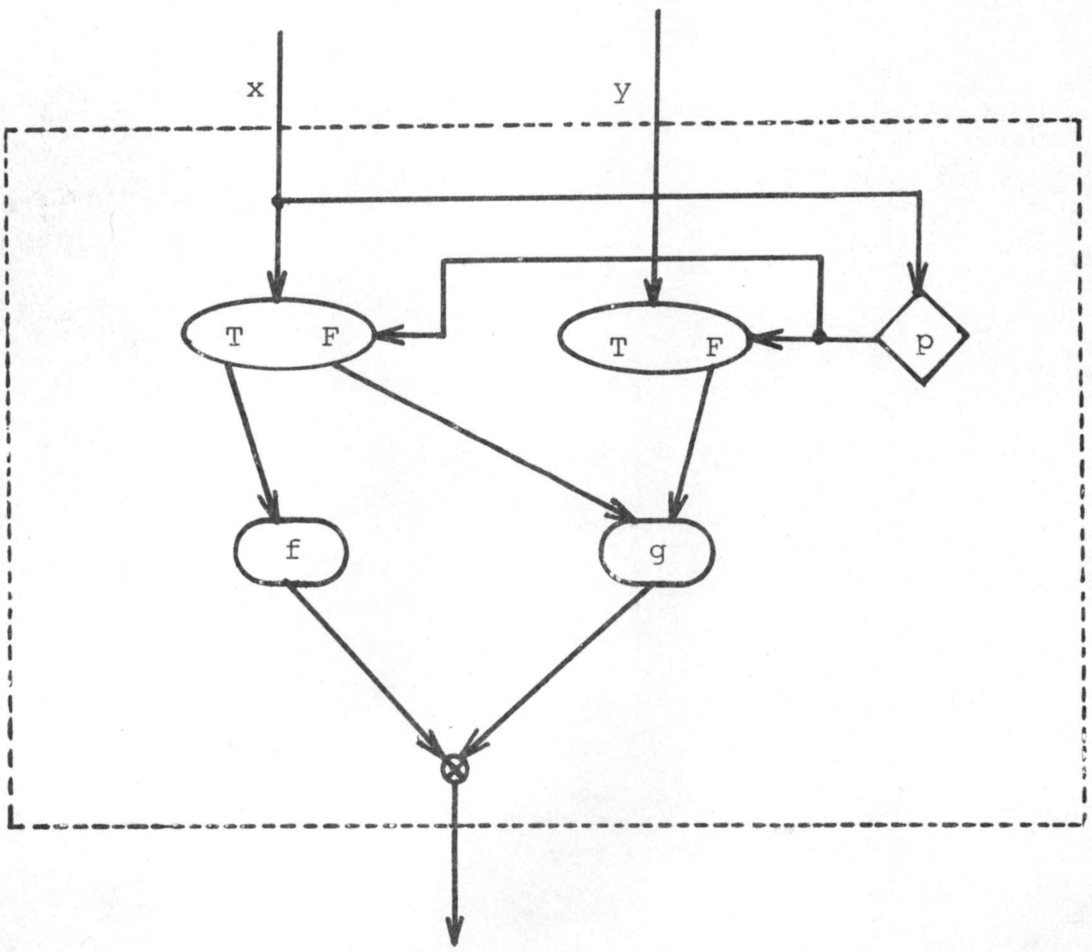


Figure 2.7 Compilation of conditional expression (2.5)

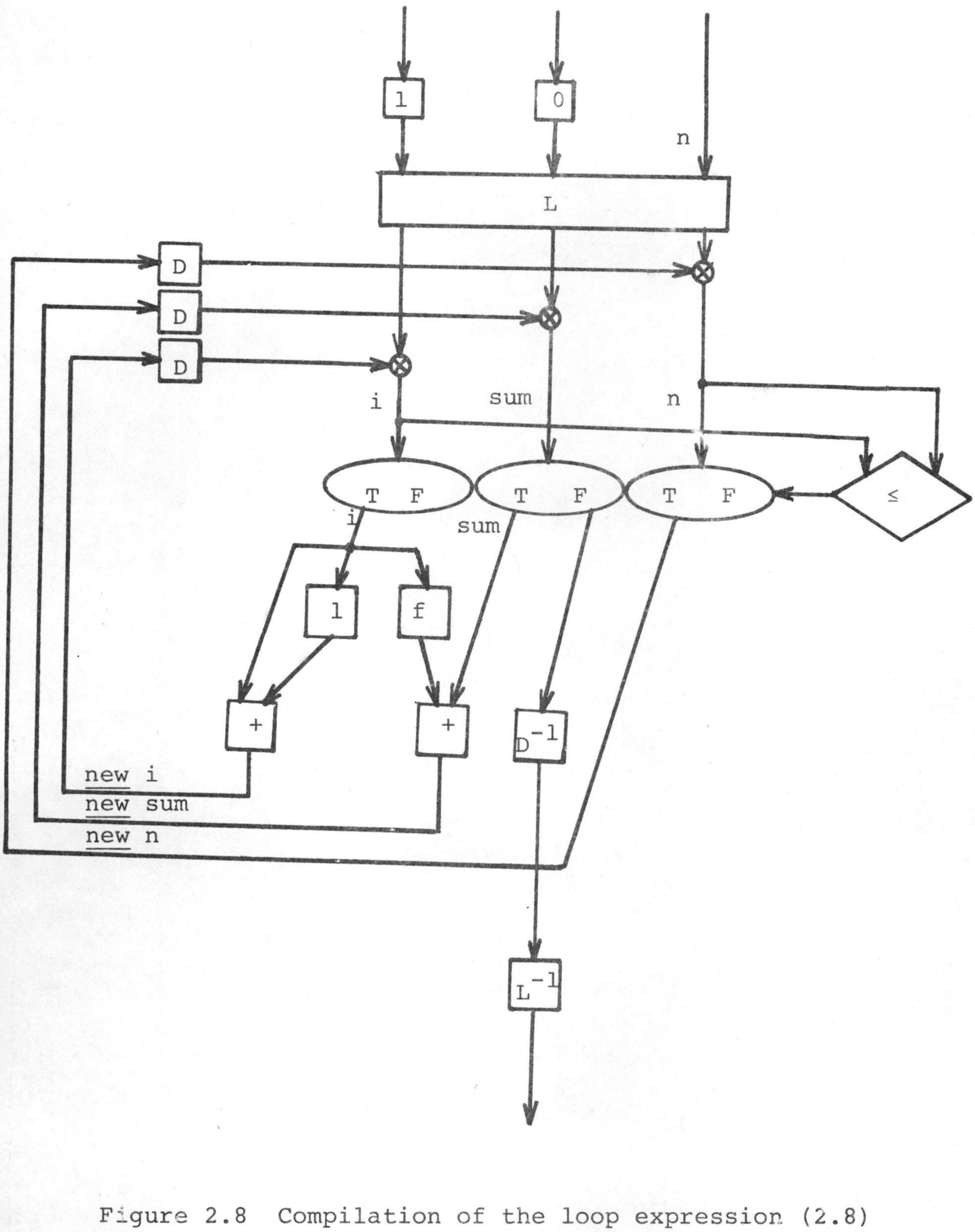


Figure 2.8 Compilation of the loop expression (2.8)

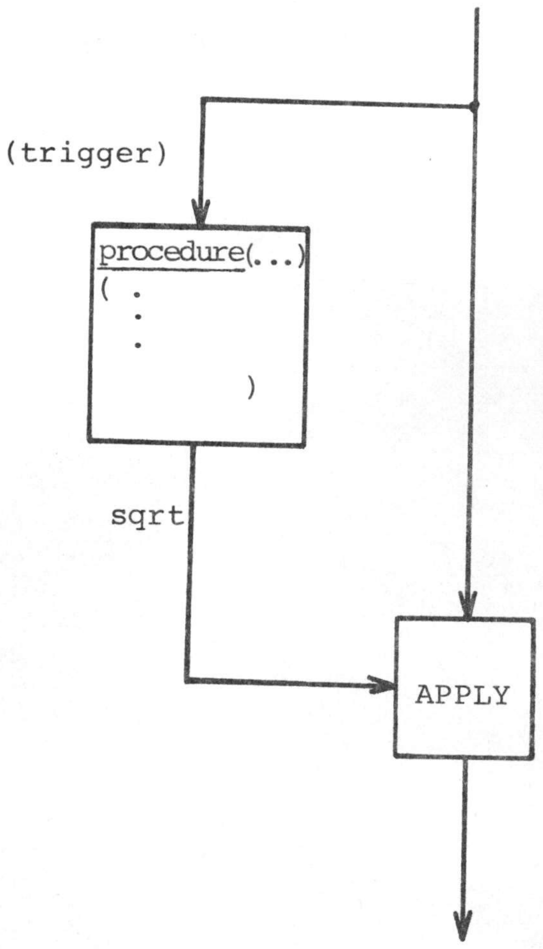


Figure 2.9 Applying a procedure

3. <u>The Base Language and the Unraveling Interpreter</u>

Let each distinct execution of an operator be termed an <u>activity</u>. Now imagine the operator f of Figure 3.1a in the body of a loop at a time when two complete sets of input values are ready for processing. Since both input values x_1 and y_1 are present, the first principle of dataflow permits the first activity (execution of the operator) to take place (Figure 3.1b). Immediately after that the second activity can occur (Figure 3.1c). However, the second principle of dataflow (freedom from side-effects) implies that the second activity actually need not wait for the first and in fact both may be executed currently.

The purpose of the unraveling interpreter is to generate as many such activities as possible. Of course, the main problem is to keep the different sets of tokens from being mixed, since it is essential that x_i be matched only with y_i. This is accomplished by tagging each token with the name of its destination activity. Each activity is assigned a unique <u>activity name</u> whereby:

> Activity x accepts all and only those tokens that carry destination activity name x.

Within a dataflow procedure, each operator is uniquely labeled and has some specific number of input and output <u>ports</u>. Each token thus specifies its destination as a particular input port of a particular activity and is represented as a subscripted ordered pair*

$$\langle \text{data, destination activity name} \rangle_p$$

* This use of angle brackets in describing tokens is not related to Id syntax for structures.

where the "destination activity name" is of the form u.c.s.i and p is the destination input port. The specifications c,s, and p specify that the token is moving along a line of the program graph connected to input port p of operator s in procedure c. This implies, of course, that the activity that produced that token must be a predecessor of operator s in procedure c consistent with the static structure of the program. The remaining fields u and i give, respectively, the <u>context</u> (for example, the procedure application context) and the <u>iteration</u> <u>count</u> (for example, this is the i^{th} iteration of the loop) consistent with the dynamics of program execution. All tokens carrying the activity name u.c.s.i in their destination field comprise the <u>input token set</u> to the activity named u.c.s.i .

Below we specify in detail the semantics of dataflow operators where we often make use of definition by case. That is,

$$(\underline{a}\text{->}\underline{b};\ \underline{c}\text{->}\underline{d};\ \ldots;\ \underline{e}\text{->}\underline{f};\ \underline{g})$$

means that if $\underline{a}$ holds then token $\underline{b}$ is the result; otherwise, if $\underline{c}$ holds then token $\underline{d}$ is the result, etc.; finally, if no condition holds then token $\underline{g}$ is the result. If no result token is specified (i.e. ϕ), then none is produced.

3.1 <u>Block schema (functions and predicates)</u>

This includes all arithmetic, boolean, and relational operators, as well as the SELECT and APPEND operators. The binary function f typically specifies the operators of this class:

$$\text{input token set} = \{<x,u.c.s.i>_1, <y,u.c.s.i>_2\}$$
$$\text{output token set} = \{<f(x,y),u.c.s'.i>_p\}$$

In general, we will use primes to denote successor operator labels. We also move each fork back to the output port to which it is connected and make the output port perform the token replication task. Hence, even an operator with only one output port may actually produce more than one token. For simplicity we will quite often neither write port numbers nor indicate that an operator might have more than one successor.

We define the history of a line (i.e., tokens with fields c, s, and p common in their activity names) to be valid if in a given context (i.e., u) no two tokens have the same iteration count. A schema is defined to be valid if given valid input histories it produces valid output histories, and corresponding to the input token set with iteration count i there is one and only one output token set with iteration count i.

Since functions and predicates do not affect the iteration count, they are clearly valid schemata. A block schema comprising acyclic interconnections of functions and predicates, and other valid schemata, is also valid because no two lines ever converge on a single input port. A rigorous proof of the validity of block schema and other schemata presented in this section can be derived using the formalism given in [Arvind & Gostelow77a].

As noted earlier, the activity name generation rule given here does not require two initiations of a block schema to initiate or terminate in any particular order. In fact, activities corresponding to the initiations can execute quite independently of each other.

3.2 Conditional schema

The SWITCH operator needed to implement the conditional schema (Figure 2.6) may be described by

$$\text{input} = \{<x,u.c.s.i>_{\text{data}},\ <b,u.c.s.i>_{\text{control}}\}$$

$$\text{output} = (b=\underline{\text{true}} \rightarrow \{<x,u.c.s_T.i>\};\ b=\underline{\text{false}} \rightarrow \{<x,u.c.s_F.i>\};\phi)$$

Note that exactly one of the successor operators s_T or s_F receives a token, and that the total number of successors need not be the same on both sides of the SWITCH. If the history of line x is valid then so will be the history of lines going to schemata f and g. However, due to the SWITCH, the iteration counts of the tokens going into f and g will be mutually exclusive. If f and g are valid schemata then the iteration counts of the tokens on the output lines of f and g will also be mutually exclusive. Hence only a valid history will result after merging the two lines via $\otimes$. Therefore the conditional schema is a valid schema.

3.3 Loop schema

A simplified loop schema is shown in Figure 3.2 where the corresponding Id expression is

```
(while p(x) do
    new x <- f(x)
 return x )                                   (3.1)
```

A loop needs operators D, D^{-1}, L, and L^{-1}, as well as a SWITCH. None of these operators affects the data portion of the tokens passing through them. They do, however, affect activity names. An execution of a loop expression can receive information only from tokens explicitly input to it because Id loops have no memory. Thus in the case of nested loops it is quite possible that the input tokens for several instantiations of an inner loop may be available at the same time. It is the L operator (in conjunction with the L^{-1}, D and D^{-1} operators) which capitalize on this fact by creating a new context u´ for each instantiation of a loop. An L operator may be described as

```
 input = {<x,u.c.s.i>}
output = {<x,u´.c.t´.1>} where u´ = (u.s.i)
```

The iteration count of a token in a loop has to be incremented every time a token goes around the loop. Corresponding to every new x type variable in a loop, a D operator is present which accomplishes this as follows:

```
 input = {<x,u´.c.t.j>}
output = {<x,u´.c.t´.j+1>}
```

If after n-1 iterations the loop predicate p turns false, the loop terminates, and sends a token (i.e., the value in the return clause) with iteration count n to the D^{-1} operator, which changes it to 1.

```
 input = {<x,u´.c.w.n>}
output = {<x,u´.c.w´.1>}
```

The L^{-1} operator returns its input tokens to the context before loop initiation. An L^{-1} operator behaves as follows:

```
 input = {<x,u´.c.w´.1>}   where u´ = (u.s.i)
output = {<x,u.c.s´.i>}
```

where s´ is the successor of the L^{-1} operator.

Note that the L operator generates exactly one set of input tokens for a loop schema with a given context (u′ = (u.s.i)). Therefore the input lines of f, D, and SWITCH have valid histories provided f is a valid schema. Clearly the D^{-1} operator never receives more than one token, and hence L^{-1} also receives and produces exactly one token on each line. The L^{-1} operator unstacks the context part stacked by the corresponding L operator and hence the tokens produced by L^{-1} have an iteration count equal to that of the input to the L operator. Hence, the loop schema is a valid schema.

Logically, activity names can become arbitrarily long because the context field is recursive. For terminating computations, names can physically be kept within bounds by proper encoding of the information. For example, an L operator can send the new context u′ = (u.s.i) on a special "dummy" token directly to its mate L^{-1} operator, and use a small unique integer (essentially equal in size to u) as a tag in place of u′. With the help of the dummy token the L^{-1} operator will be able to generate the proper output.

All activities belonging to a particular instantiation of a loop are said to constitute a loop domain and can proceed independent of activities outside the loop domain including those of the nested loop. It is interesting to note that tokens need not go around a loop in any particular order unless constrained by the need for intermediate results. This situation was illustrated earlier by the program in Figure 2.8 where several initiations of f could execute concurrently. Even if the j+1st execution of f terminates before the jth execution, no mismatch of activity names can result. Automatic unraveling of loops, constrained only by those data dependencies that are actually present, greatly increase the asynchrony of programs (for example if f were another nested loop), many of which would otherwise be considered completely sequential.

3.4 Procedure application schema

Figure 3.3 demonstrates the elements of procedure application, where APPLY actually comprises the two operators A and A^{-1}. Also shown is the fact that each procedure is prefixed by a BEGIN operator and suffixed by an END operator. The A operator must (1) create a new context u´ within which the procedure on line q may execute, and (2) pass the argument value on line α to that context. The A operator is described by

$$\begin{aligned} \text{input token set} &= \{<q,u.c.s_A.i>_{proc},<\alpha,u.c.s_A.i>_{arg}\} \\ \text{output token set} &= \{<\alpha,u'.c_q.begin.1>\} \text{ where } u' = (u.c.s_T.i) \\ &\quad \text{with } s_T \text{ the } A^{-1} \text{ mate of operator } s_A. \end{aligned}$$

That is, the "return address" $u.c.s_T.i$ is stacked and u´ becomes the new context in which procedure q is executed. The output of A goes to the BEGIN operator, a member of the class of functions and predicates discussed earlier (Section 3.1). BEGIN is very simple and serves only to replicate tokens for the fork in its output line. The END operator is more complex. It returns the result back to the caller by unstacking the "return address":

$$\begin{aligned} \text{input token set} &= \{<\beta,u'.c_q.end.1>\} \quad \text{where } u' = (u.c.s_T.i) \\ \text{output token set} &= \{<\beta,u.c.s_T.i>\} \end{aligned}$$

Finally, the A^{-1} operator is straightforward as it too, just like the BEGIN operator, serves only to replicate its output for its successors. If the procedure on line q is syntactically correct, hence is a valid schema, then the END operator will receive exactly one token. Thus if A receives only valid histories, A^{-1} will receive from END and will produce only valid histories. Hence APPLY is a valid schema.

All the activities belonging to a procedure invocation constitute a procedure domain, and can proceed independent of other activities. Similarities between a procedure application schema and a loop schema go beyond the idea of domains. A loop schema is like a procedure without a name which is invoked from exactly one place. Hence, creation of a new context for a loop instantiation requires less information (i.e., u.s.i as opposed to u.c.s.i). Procedures are generally less asynchronous than loops because in our implementation all actual parameters of a procedure must be present (i.e., in the α structure) before it can be applied. One can look at the functions of the D and D^{-1} operators as an inexpensive way of creating new activity names. In systems based on activity names, the difference between a cyclic and a noncyclic schema is not very fundamental because the concept of a line does not manifest itself in the dynamic behavior. Hence, we can treat Id loops as nameless procedures that have somewhat more efficient and asynchronous implementations than general procedures.

3.5 Asynchrony in a sequential algorithm

Here we analyze the procedure given in Section 2.5 for multiplying two matrices. The procedure of expression (2.9) assumes a matrix is represented by a structure, such that a[i] is the structure value representing the ith row of matrix a. While discussing loop schema in Section 3.3, we showed that the unraveling interpreter exploits loop asynchrony in two ways: by unraveling, and by permitting concurrent invocations of the same loop. Asynchrony in matrix multiply depends on both. The innermost dot product will unravel; for if the structure selections and the multiplications take longer than generating m values for k, the a[i,k]*b[k,j] operations will overlap. But the addition in the innermost loop of (2.9) must be done serially, so it will take O(m) time to generate each dot product s (each element of row d).

Even though the value of <u>new</u> d depends upon the old value of d (just like s above), many initiations of the innermost loop might execute concurrently because values of j can be generated faster than a complete execution of the innermost loop. Hence the time complexity of the j loop is determined by the sequentiality of the append operations as opposed to the time to generate each element of d. Since the first append operation on d cannot begin until the innermost loop produces an answer, the total time to generate a row d is O(m+n). Similar arguments can be made for the loop with index i to show that the total time complexity of the matrix multiply program is O(ℓ+m+n). Note that the total number of multiplications, i.e., the total work, has not changed from that of a purely sequential execution of the same program -- only the overlapping of operations in time has changed. The importance of the unraveling interpreter lies in the fact that it does not recognize unnecessary data dependencies and thereby exploits the semantics of Id programs to enhance the attainable asynchrony.

The above analysis was done assuming unlimited processors. On a machine with p processors the time complexity would be

$$\max \{O(\ell+m+n),\ O(\ell mn/p)\}$$

which is the minimum for this matrix multiply algorithm.

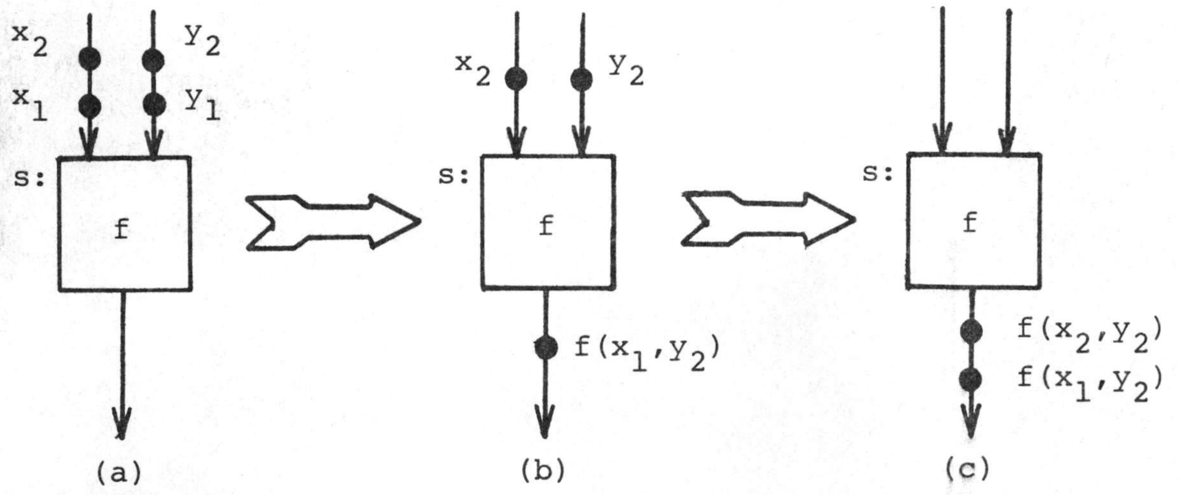


Figure 3.1 Two executions of a dataflow operator

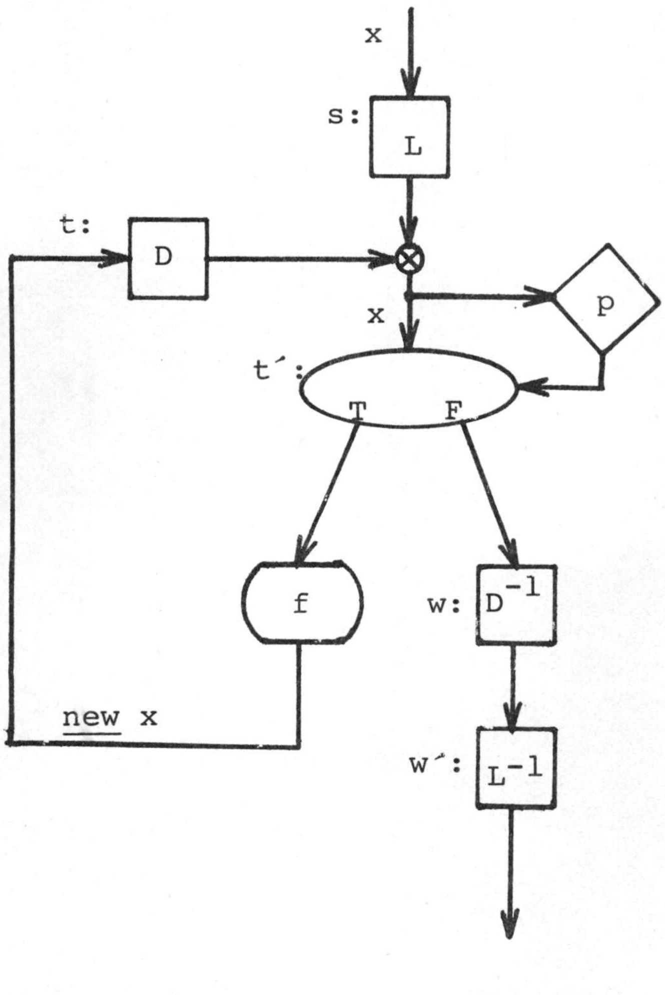


Figure 3.2 A simplified loop schema corresponding to expression (3.1)

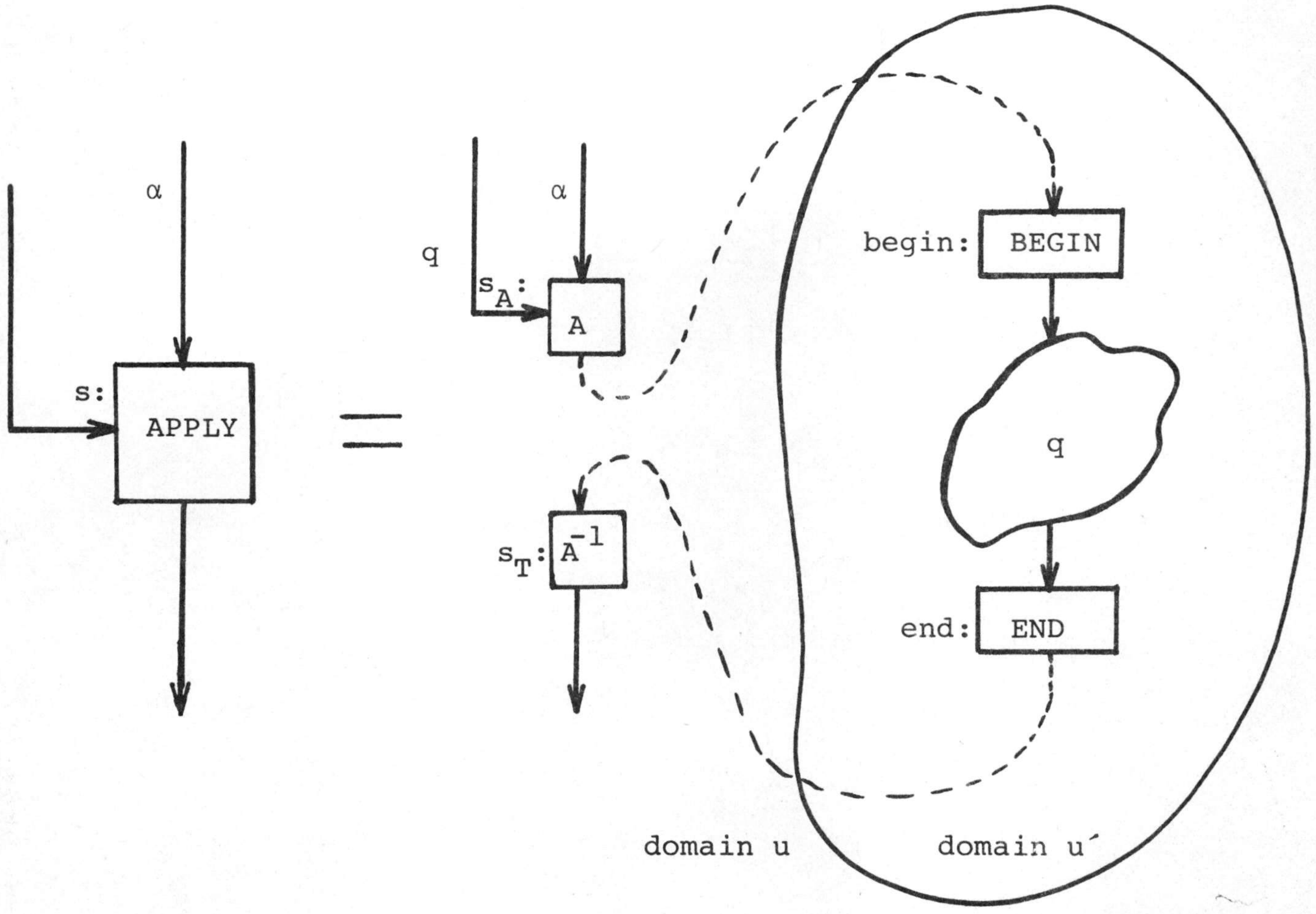


Figure 3.3 The APPLY operator and its execution

4. Programming with Streams

4.1 Introduction to streams

The values discussed in Sections 2 and 3 are simple values (e.g., integer, structure); any simple value is represented in its entirety on a single token. A stream value is represented by a (possibly unbounded) ordered sequence of tokens, each token carrying a simple value, and where the last token in the sequence carries the special value est (which is not accessible to the programmer), meaning end-of-stream. The stream value constructor for streams of length n is written $[x_1,...,x_n]$ in Id, so the constant stream value [1,1,2,3,5] of length five is represented by six tokens as shown in Figure 4.1. The empty stream (consisting of exactly the est token) is written [].

Id variables are typed as simple or stream. In this paper, stream variables are written in upper-case letters. To give variable A the stream value [1,1,2,3,5] we write

$$A \leftarrow [1,1,2,3,5]$$

Two primitive stream functions are first and rest. For example, the Id statements

```
a ← first(A);
B ← rest(A);
```

would give a = 1 and B = [1,2,3,5]. That is, the entire sequence of tokens in A was input to exactly two activities. The activity carrying out first produced a simple result (a single token), while the activity carrying out rest produced a stream (a sequence of output tokens). Input and output of streams through an activity may be asynchronous, so that not all input tokens need be defined before some output can be produced. Thus an activity executing rest, given only the input token in position k ($k \neq 1$), can immediately output the token in position k-1; that activity must also remain in existence until all input tokens have been received and all output tokens have been produced.

The remainder of this subsection will illustrate a looping construct on streams to give the reader some understanding of streams before going into details. Consider the following statement to generate stream A comprising the first k Fibonacci numbers:

```
A ← (initial i ← 1; nexti ← 1
     for counter from 1 to k do
          new i ← nexti;
          new nexti ← i + nexti
     return all i)
```

The clause return all i outputs, in order, the sequence of all values assumed by i as long as the loop predicate is true. If k<1 initially, then A will receive the empty stream.

As a second example, consider the electrical circuit shown in Figure 4.2. Let the state of an electrical line be represented by a voltage stream consisting of structured values of the form <time:t,voltage:v> giving the voltage and the time when that voltage is said to exist on that electrical line. If the input voltage v varies in discrete steps then the output voltage w may be described at times of input change by the equation

$$w_{i+1} = w_i + (v_i - w_i)(1-e^{-\tau(t_{i+1}-t_i)})$$

where τ is the circuit time constant (see Figure 4.2b). The Id statement (4.1) below can model the circuit with arbitrary precision if sufficiently fine time steps are used.

```
W ← (initial v←v₀;w←w₀;t←t₀
     for each status in V do
         new t ← status.time;
         new v ← status.voltage;
         new w ← w+(v-w)*(1-e↑(-τ*(new t-t)))
     return all <time:new t, voltage:new w>)              (4.1)
```

The for each loop above is a new construct which for each token arriving on stream line V places that token on the simple line called status, and executes the body of the loop once. As before, the all construct causes one token to be contributed to the output voltage stream W on each iteration. Execution of (4.1) terminates on receiving the end-of-stream token est on input V. This example demonstrates not only asynchronous input and output of streams, but also the operation of a "history-sensitive" function. That is, the output produced for a given input depends upon input tokens previously received.

Viewing the inputs and outputs of some operating system routines (e.g., I/O drivers) as streams is not new. Streams have also been used by Landin [Landin65] in describing the applicative semantics of loops in Algol-60. There Landin defined a stream as a list (i.e., structure) with some special properties regarding the sequencing of evaluation. Essentially, the elements of a stream (here as well as Landin's) have a total linear ordering and are not required to exist simultaneously. Thus the sequence of values assumed by a loop variable in Algol can be modeled by a stream. However streams also have practical advantages over structures especially when subjected to a pipeline of processes. For example, in applicative languages (such as Id) streams enable one to perform operations on lists without using an item-by-item representation of the intermediate resulting lists. More interesting is the fact that streams enable one to postpone the evaluation of the expressions that produce the items of a list until those items are actually needed. Friedman and Wise have exploited these ideas in pure LISP and other related languages [Friedman & Wise76a,76b]. Streams also form an integral part of the language for networks of parallel processes developed by Kahn and MacQueen [Kahn & MacQueen77].

Streams were first introduced into dataflow by Weng in [Weng75] where he gave formal rules for constructing "well-formed" dataflow schemas with streams in Dennis' Dataflow Language [Dennis73]. We have extended Weng's ideas by incorporating streams into loops, and by the <u>for each</u> construct introduced above. Streams in dataflow are essential for doing applicative programming of history-sensitive functions such as updating a data base (see Section 5). However, streams are also interesting because they introduce still another level of asynchrony which can be very significant in exploiting machine concurrency.

4.2 Extending the semantics of control operators for streams

Extending the notation of Section 3, we denote the kth token in stream X carrying the value X_k to input port p of activity u.c.s.i by

$$<\{X_k, k\},\ u.c.s.i>_p$$

We may denote an entire stream A by the set

$$\{<\{X_k, k\}, \text{u.c.s.i}>_p \mid 1\leq k\leq \#A\}$$

where $X_{\#A}=\underline{est}$.

The basic rules given in Section 3 for generating activity names are also valid for streams. Even though an activity may input or output more than one token through a port, no two tokens will have identical stream positions. Hence, each token in the input set of an activity is still uniquely identified. Furthermore, a stream never has a missing token position (that is, for stream A a token is defined for each stream position k such that $1\leq k\leq \#A$).

The semantics of the control operators SWITCH, D, D^{-1}, L, and L^{-1} are trivially extended to deal with streams since none of these operators affects the value or the position of any token. Thus all fields in activity names are manipulated according to the rules already explained in Section 3. We illustrate the idea with the SWITCH operator.

The SWITCH operator still expects a simple (non-stream) boolean token at the control input port. Depending upon the boolean it switches the entire stream at the data input port to either the T or the F output port:

$$\text{input} = \{<\{X_k,k\},\text{u.c.s.i}>_{\text{data}} \mid 1\leq k\leq \#X\} \cup \{<b,\text{u.c.s.i}>_{\text{control}}\}$$
$$\text{output} = (b=\underline{true} \rightarrow \{<\{X_k,k\},\text{u.c.s}_T.\text{i}> \mid 1\leq k\leq \#X\};$$
$$b=\underline{false} \rightarrow \{<\{X_k,k\},\text{u.c.s}_F.\text{i}> \mid 1\leq k\leq \#X\};\phi)$$

Note that if the boolean has already arrived then each token in the input data stream can be immediately output without waiting for further input. Also note that due to token communication delays it is possible that a stream token may appear as input to an activity at a time which is out of phase with its physical stream position. This latter point causes no difficulty and is in fact essential to resource manager operation (Section 5).

Extending the semantics of procedures to permit stream arguments and stream results is not so straightforward. Due to the asynchronous nature of streams, the A operator cannot wait for all the tokens of a stream argument to arrive before passing them on to the corresponding BEGIN operator. In other words, a stream argument cannot be part of the structure α explained in Section 2.5. The problem is solved by creating a separate port for every stream parameter which then works asynchronously with respect to the simple port or the other stream ports. The extended semantics of the A operator can now be given as follows:

$$\begin{aligned}\text{input} &= \{<q,u.c.s_A.i>_{proc}, <\alpha,u.c.s_A.i>_{arg}\} \quad \cup \\ &\quad \{<\{Z_k,k\},u.c.s_A.i>_{stream} \mid 1 \le k \le \#Z\} \\ \text{output} &= \{<\alpha,u'.c_q.begin.1>_{arg}\} \quad \cup \\ &\quad \{<\{Z_k,k\},u'.c_q.begin.1>_{stream} \mid 1 \le k \le \#Z\} \\ &\qquad \text{where } u' = (u.c.s_T.i)\end{aligned}$$

The semantics of BEGIN, END, and A^{-1} can also be extended in a similar way. Finally, concerning the problem of mismatched parameters, the A operator simply ignores (i.e., absorbs) any extra input streams after examining the definition of the received procedure value. In case a BEGIN operator has more stream ports than the corresponding A operator, the A operator sends <u>error</u> tokens to the appropriate ports of the BEGIN operator. Similar rules can be specified for the END and A^{-1} operators.

4.3 Some new functions on streams

Several functions and predicates are defined below on streams, of which the first five are primitive in the base language. A few of these functions and predicates will be used in implementing various Id constructs to be discussed later.

(a) [ ] (generate an empty stream): This function produces an empty stream on receiving a (simple) trigger.

$$\text{input} = \{\langle x, u.c.s.i\rangle\}$$
$$\text{output} = \{\langle \{\!\!\{\underline{est}, 1\}\!\!\}, u.c.s'.i\rangle\}$$

(b) empty(A): This predicate produces a boolean token true if A = [ ], otherwise a false token is produced.

$$\text{input} = \{\langle \{\!\!\{A_k, k\}\!\!\}, u.c.s.i\rangle \mid 1 \le k \le \#A\}$$
$$\text{output} = (\#A = 1 \rightarrow \{\langle \underline{true}, u.c.s'.i\rangle\}; \{\langle \underline{false}, u.c.s'.i\rangle\})$$

(c) first(A): This function outputs the first token of stream A provided stream A is not empty.

$$\text{input} = \{\langle \{\!\!\{A_k, k\}\!\!\}, u.c.s.i\rangle \mid 1 \le k \le \#A\}$$
$$\text{output} = (\#A = 1 \rightarrow \{\langle \underline{error}, u.c.s'.i\rangle\}; \{\langle A_1, u.c.s'.i\rangle\})$$

(d) rest(A): The result stream is all but the first member of stream A.

$$\text{input} = \{\langle \{\!\!\{A_k, k\}\!\!\}, u.c.s.i\rangle \mid 1 \le k \le \#A\}$$
$$\text{output} = (\#A = 1 \rightarrow \{\langle \{\!\!\{\underline{est}, 1\}\!\!\}, u.c.s'.i\rangle\}; \{\langle \{\!\!\{A_{k+1}, k\}\!\!\}, u.c.s'.i\rangle \mid 1 \le k \le \#A-1\})$$

(e) cons(x,A): The output stream has x as the first number and A as the rest, i.e., if X represents the output stream then X = cons(first(X), rest(X)).

$$\text{input} = \{\langle x, u.c.s.i\rangle_1\} \cup \{\langle \{\!\!\{A_k, k\}\!\!\}, u.c.s.i\rangle_{\text{stream}} \mid 1 \le k \le \#A\}$$
$$\text{output} = \{\langle \{\!\!\{x, 1\}\!\!\}, u.c.s'.i\rangle\} \cup \{\langle \{\!\!\{A_{k-1}, k\}\!\!\}, u.c.s'.i\rangle \mid 2 \le k \le \#A+1\}$$

(f) consℓ(A,x): This function is similar to cons except that the input x appears at the end of the output stream.

(g) concatenate(A,B): The output is a stream with the tokens of A (except the est token) preceding the tokens of B.

(h) filter(x,A): This function produces two output streams. The stream on output port 1 contains all those tokens of A that are not equal to x, while the stream on output port 2 specifies the input stream position of those tokens selected to appear at output port 1.

(i) equalize(A,B): This function outputs two equal length streams formed from input streams A and B by truncating the longer of A and B to the length of the shorter. The truncated portions of A and B are also output as remainders (at least one of these two output remainder streams will be empty by definition).

(j) extend(A,x,B,y): This function also outputs two streams of equal size, formed from the input streams A and B. However, the length of the output streams is equal to the longer of streams A and B. The shorter stream is extended by x or by y depending upon which is the shorter input stream.

(k) exif(A,B,x): This operator means "extend A to the length of B by x, if necessary." It produces a stream equal in length to stream B. In case A is longer than B, the output stream contains the first #B-1 tokens of A. Otherwise enough x tokens are added behind stream A to extend its length to that of stream B. The remainder of stream A is also produced on a separate output port. It can be implemented as

```
( AEQ,BEQ,AREM,BREM ←equalize (A,B,);
  AEX, BEX ←extend (AEQ, x,B,x)
  return AEX,AREM )
```

(ℓ) size(A): It produces a simple token containing the value #A-1.

As previously shown [a,b,c] is a syntactic shorthand for cons(a,cons(b,cons(c,[]))).

4.4 Some new constructs on streams and their implementation

4.4.1 The for each-while construct

Consider the for each loop in statement (4.1). The implementation appears in Figure 4.3, where the unimportant details are subsumed by schemata f and g. The E and ≠est operators implement the for each construct. The E operator takes a single stream of tokens as input and produces a sequence of simple tokens for distinct initiations (activities) of its successor operator:

$$\text{input} = \{<\{X_k, k\},\text{u.c.s.1}> \mid 1 \le k \le \#X\}$$
$$\text{output} = \{<X_k, \text{u.c.s}'.\text{k}> \mid 1 \le k \le \#X\}$$

Since an E operator is always preceded by an L operator the input stream to E always has an iteration count of 1; the iteration counts of the output tokens correspond exactly to the positions of those tokens in the input stream. The predicate ≠est tests a simple token for the special est value to terminate the loop. (This predicate is not accessible to the Id programmer.) The all construct is implemented by taking output tokens from the T side of the loop SWITCHes, and feeding them to the E^{-1} operator to form a single output stream. It places tokens in proper order by using their iteration count as their position in the output stream:

$$\text{input} = \{<x,u.c.s.i>\}$$
$$\text{output} = \{<\{x,i\},u.c.s'.1>\}$$

Note that every initiation of operator E^{-1} with activity name u.c.s.i contributes to the production of the same stream, and that this output stream always has an iteration count of 1. The final token to be inserted into the output stream is triggered by a token from the F side of a loop SWITCH. It is an est token and is produced by the est operator (which is also used for generating an empty stream). Note that the est operator is triggered only once for each execution of the entire loop. The E^{-1} and est operators work together to create a valid output stream with no missing tokens.

A program equivalent to expression (4.1) written without using either the for each or the all construct is:

```
  procedure RC(v,w,t,V)
      (if empty(V) then []
        else (status ← first(V);
              t′ ← status.time;
              v′ ← status.voltage;
              w′ ← w+(v-w)*(1-e↑(-τ*(t′-t)))
              return cons (<time:t′,voltage:w′>,
                           RC(v′,w′,t′,rest(V)))))          (4.2)
```

However, procedure RC generates a much larger number of tokens. This is due to handling of streams using the _first_, the _rest_, and the _cons_ operators instead of the E and E^{-1} operators. Even though only one token from the input stream V is used in each procedure invocation, the whole stream V has to be input. Similarly, _cons_ requires the whole stream constructed at the later recursive calls of RC to be returned to the earlier invocations. If there are n tokens in the input stream procedure, RC internally generates $O(n^2)$ tokens instead of the O(n) tokens generated by expression (4.1).

We can now explain the most general construct involving loops in Id. The base language schemata generated by the _for each_ construct and the _while_ construct (discussed in Section 2.4) are both particular cases of the base language schema generated by the _for each-while_ loop [Kathail78]. Consider the following expression using the _for each-while_ construct:

```
                (initial x ← a
                 for each b in B while p(b,x) do
                     new x ← f(x,b)
                 return x, all x)                                   (4.3)
```

The code following _do_ is executed for each token of B only as long as predicate p holds. The major difficulty in implementing such a loop is to make it self-cleaning so no tokens from B remain in the loop after termination to clutter up the machine. One way to solve this problem is to release tokens of stream B on demand. That is, let a token from B enter the loop only when it is needed. To do this, we translate a loop such as (4.3) into the form given by (4.4).*

----
*We are grateful to Vinod Kathail for this observation.

```
    (x,X,S´ ← (initial x ← a
               for each b in B´ while(if b=ε then false else p(b,x))do
                   new x ← f(b,x)
               return x, all x, all true);
     S ← cons(true,S´);
     B´ ← exif(B,S,ε)
     return x,X)                                                (4.4)
```

where ε is a special signal value not available to the Id programmer. In this form, the loop within (4.4) can be implemented by only a slight modification of the implementation shown in Figure 4.3. The actual implementation of (4.4), and hence of (4.3) is given in Figure 4.4. A new operator, delete est (shown as $\boxed{\cancel{est}}$) has been included just after the E operator to remove the last token. To explain how it all works, note that stream S and hence stream B´ always have at least one non-est token each. We consider the case when stream B runs out of tokens before p(b,x) turns false. When this occurs, an ε token is generated for B´ which shuts off the loop, generating no more tokens in S´. Since B´ has exactly one more token than S´, only the est token of B´ will enter the loop after b=ε. This last est token will be absorbed by the delete est operator. Now consider the case when p(b,x) turns false and there are one or more tokens (including the est token) left in stream B. Shutting off the loop, as before, prevents any more tokens from going into stream S´. This in turn generates the est tokens for both S and B´. The extra tokens of B are simply absorbed by the exif operator, and the est token of B´ is absorbed by the delete est.

Id also allows us to write return all X in a loop return clause, in which case the stream returned is the ordered concatenation of all the X streams generated in the loop (this is used in Section 5.4). It is important to note that an ordinary stream, as opposed to a stream of streams, is produced. This construct is very useful but unfortunately our implementation of it is inelegant and is not discussed here.

4.4.2 The *but* construct

This construct is used in conjunction with the *all* construct to withhold some tokens from a stream. For example, if the *return* clause of a loop expression is

return *all* x *but* a

then only those values of x that are not equal to a will be returned. Its implementation is straightforward using the *filter* function.

4.4.3 The *remainder* stream

In expression (4.3), if we write *remainder* B in the return clause, then a stream containing all those elements of B that were left over when p(b,x) became false is generated. This stream is produced easily by *cons*ing the last token to enter the loop to the remainder stream produced by the *exif* operator.

4.4.4 The parallel looping construct

Several *for each* x *in* X type of clauses can be combined to form a parallel looping construct as follows:

for *each* x *in* X; y *in* Y *while* p(x,y) *do*

where the *while* clause is optional. The loop terminates as soon as either stream X or Y runs out of tokens, or if p(x,y) turns false. The base language translation is straightforward to derive by treating the while predicate as (if x=est ∨ y=est then false else p(x,y)) and generating the appropriate signal stream. Two exif operators are used to control X and Y with the help of the signal stream.

4.5 Pipelining effect in stream programs

We illustrate the natural cascading effect of streams by the program in (4.5) to generate primes according to the Sieve of Eratosthenes algorithm. A recursive version of this procedure is given in [Weng75].

```
procedure SIEVE(LIST)
    (while not empty(LIST) do
        prime ← first(LIST);
        new LIST ← (!delete all the multiples of prime from LIST!
                     for each item in LIST do
                     a ← (if mod(item, prime) = 0 then λ else item)
                     return all a but λ)
        return all prime)                                    (4.5)
```

The above procedure, when applied to a stream of integers from 2 through n, will iteratively create sieves, each of which filters out multiples of the first item of the LIST input to it. Each iteration of the outer loop generates one prime number and a new LIST. Sifting of the LIST produces a stream of integers for the next sieve if that stream is not empty (see Figure 4.5). The predicate empty(LIST) can be decided by examining any token of stream LIST. Therefore the next iteration of the loop will begin as soon as any token of the stream new LIST is produced. Since the LIST gets smaller after every sifting, it is possible that many sieves may work simultaneously. The amount of time it takes to do the i^{th} iteration of the outer loop is $O(s_i)$ where s_i is the number of tokens in the LIST for the i^{th} iteration. (Note that filtering is a completely sequential operation.) However, due to the pipelining of sieves the total time to execute procedure SIEVE will also be O(s) where s is the size of the largest LIST. Obviously the size of the initial LIST is the largest and thus procedure SIEVE will take O(n) time (assuming an unlimited number of processors is available).

In order to illustrate the asynchrony of this stream procedure we compare it with the following non-stream version of the Sieve of Eratosthenes:

```
procedure sieve(list, s)   !s is the number of elements in the list!
    (initial p ← Λ; i ← 1
    while s≠0 do
        new p[i] ← list[1];
        new i ← i+1;
        new list, new s ←
                (initial a ← Λ;k ← 0; prime ← list[1]
                 for j from 2 to s do
                    new a, new k ← (if mod(list[j],prime)≠0
                                    then a+[k+1]list[j],k+1
                                    else a, k)
                 return a, k)
    return p)                                                  (4.6)
```

Even though each sieve still takes $O(s_i)$ time, this procedure takes $O(\sum_{i=1}^{m} s_i)$ time, assuming there are m primes in the first n numbers. Since a complete new list has to be produced before the next iteration begins, no overlapping of the sieves is possible.

We again want to emphasize the fact that these significant speedups of programs take place automatically. Dataflow programs generally are more asynchronous than their counterparts in sequential languages, and dataflow programs with streams are even more asynchronous than comparable dataflow programs without streams. It should also be noted that the actual number of processors or their configuration are of no concern in writing programs. All dataflow programs will naturally run slower if there is a lack of resources.

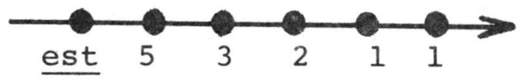


Figure 4.1 A stream on line A

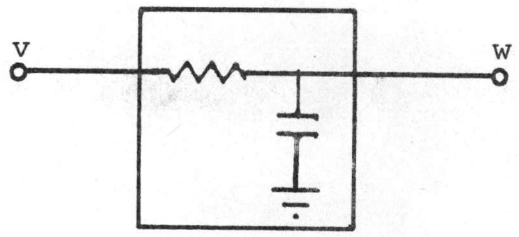


Figure 4.2a A RC circuit for expression (4.1)

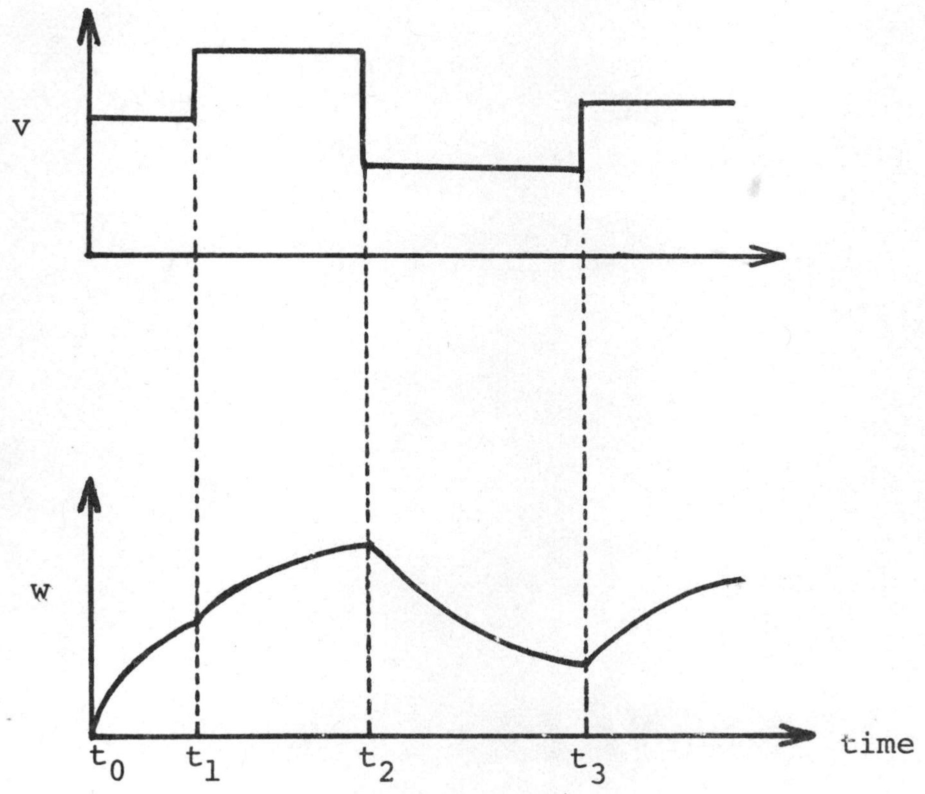


Figure 4.2b Input/Output of the circuit in Figure 4.2a

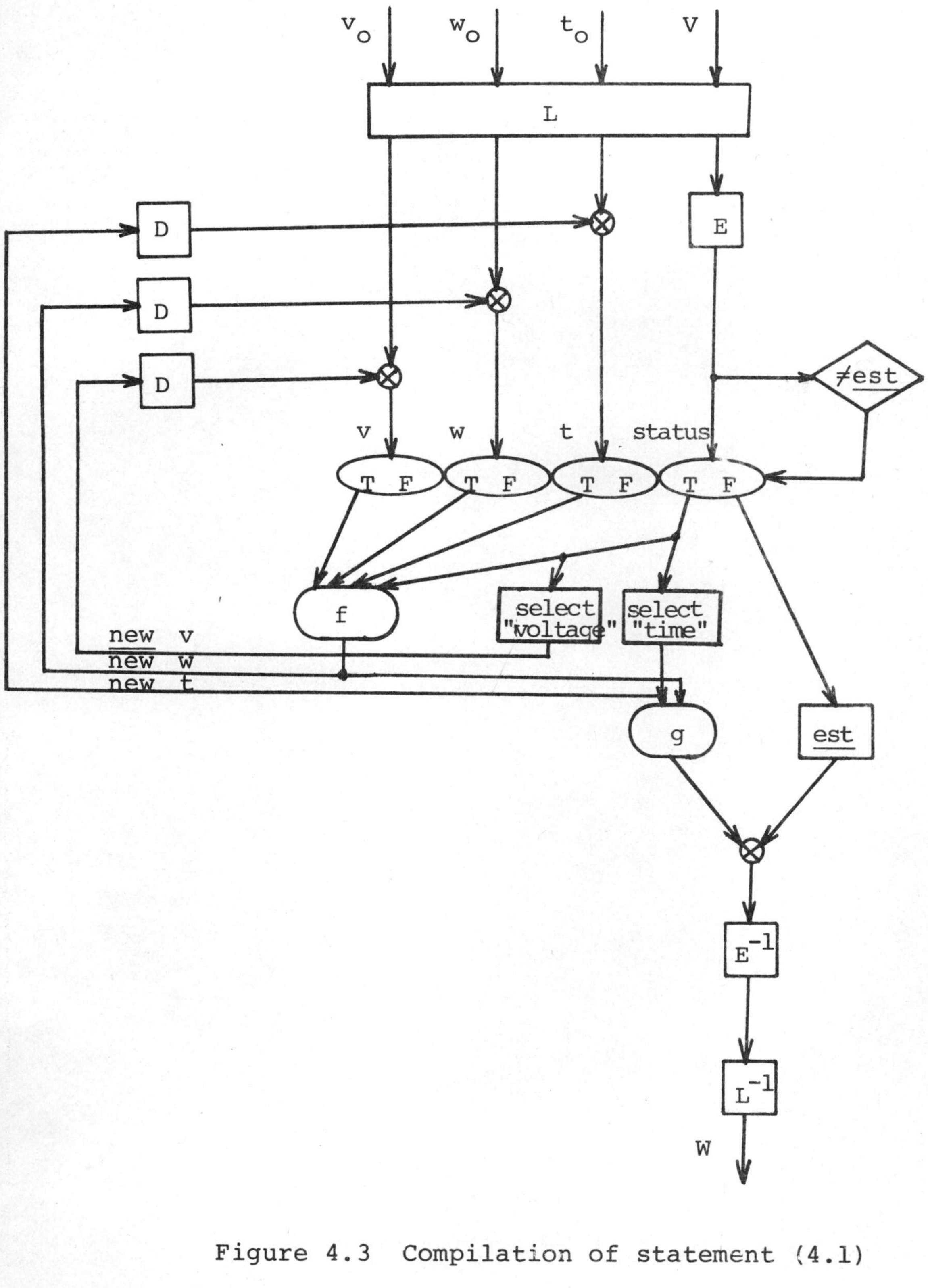


Figure 4.3 Compilation of statement (4.1)

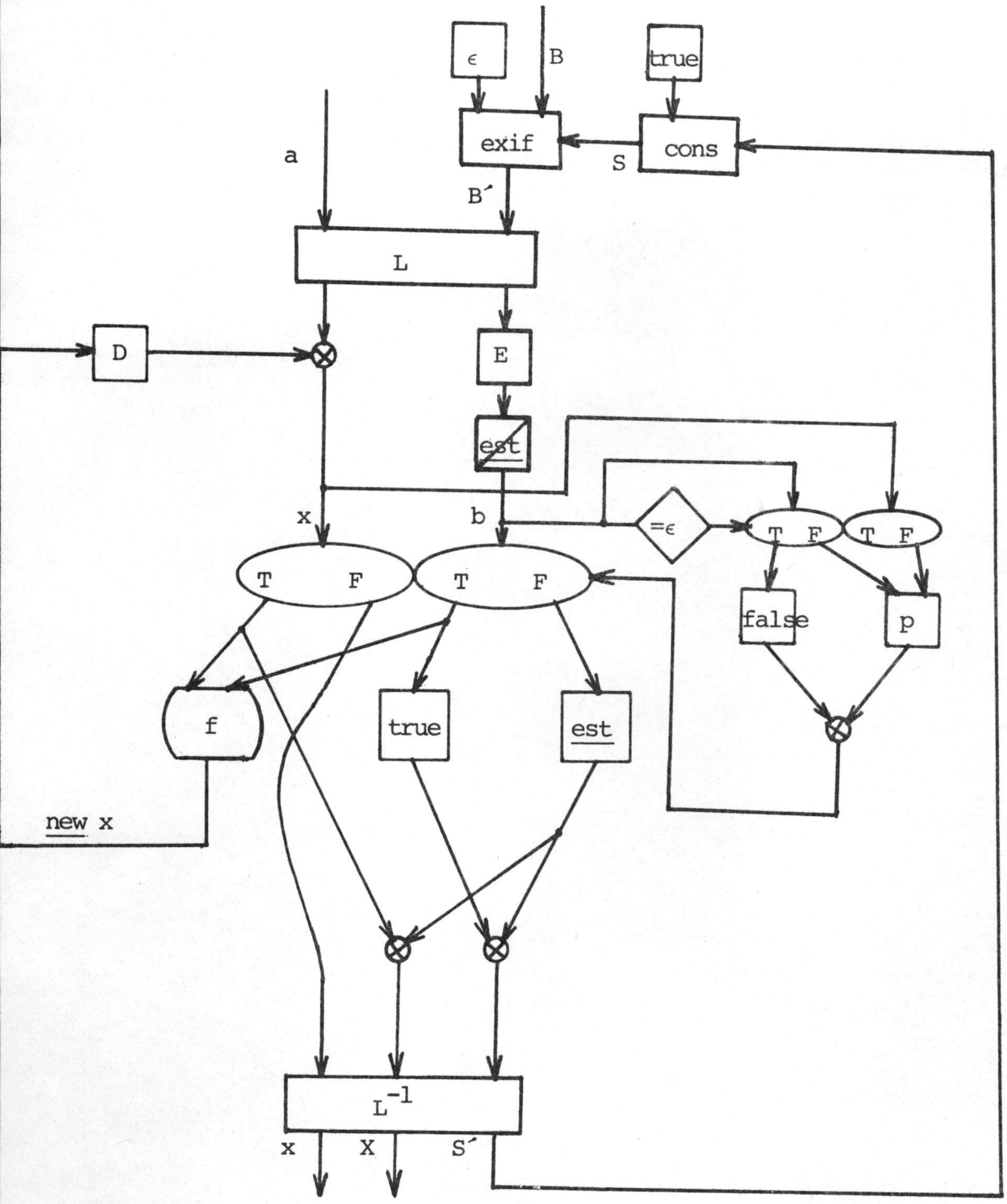


Figure 4.4 The for each-while schema

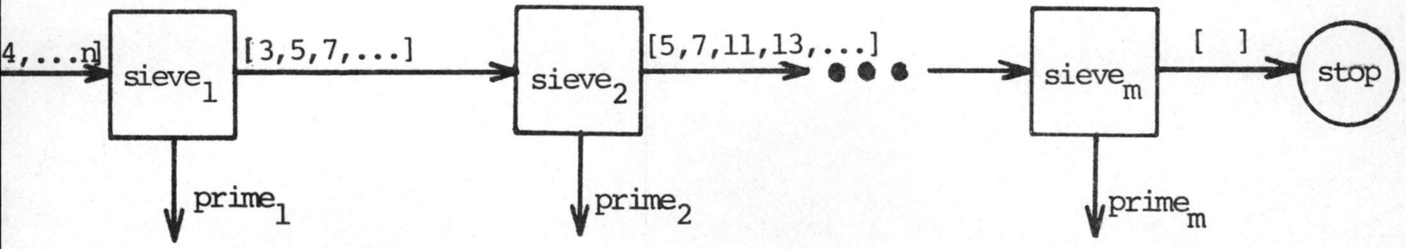


Figure 4.5 The Sieve of Eratosthenes in execution

5. Resource Managers

Id is also intended as a language for writing operating systems. Thus the concept of a resource, and mechanisms for synchronizing accesses to it, must be included. In non-applicative languages, memory cells are used to represent the state of a resource, while accesses to the resource are synchronized through appropriate reading and writing of these cells. A resource may be shared among several processes by sharing those memory cells. Id, however, utilizes quite a different model for resources and synchronization mechanisms. Furthermore, since some degree of nondeterminism is usually implied in the use of resources (for example, the order in which a computer responds to two terminals), Id also incorporates a facility for nondeterministic programming.

5.1 A primitive resource manager

Figure 5.1 outlines the body of a primitive resource manager in which the token on line s represents the current state of the resource being managed. The state s is part of a loop, so the next value of s (new s) is determined by function f acting on the current value of s and the incoming user request, each request arriving as a component of the input stream X.

In statement (5.1) below, mdl is assigned a manager definition value that describes the manager with the body of Figure 5.1.

```
mdl ← manager (s₀)
      (entry X do
         RESULT ← (initial s←s₀
                   for each x in X do
                      <new s, answer> ← f(s,x)
                   return all answer)
       exit RESULT )
                                                    (5.1)
```

A manager definition value is essentially a pattern from which many instances of manager values (i.e., managers) may be created. For example,

$$m \leftarrow \underline{\text{create}}(\text{mdl},a) \qquad (5.2)$$

makes m an instance of mdl initialized with value a for s_o (s_o is called a creation time parameter). This, of course, bears close resemblance to classes and class objects in SIMULA.

To use manager m the programmer sends an input value y (an argument) to m by writing

$$z \leftarrow \underline{\text{use}}(m,y) \qquad (5.3)$$

and the result produced by manager m is returned as the value of the use expression, much like a procedure application. The effect of (5.3) is to make the token from line y a component of stream X in manager m. However, the exact position of y in stream X cannot be determined a priori since many independent users of manager m may be sending an argument to it simultaneously. Hence the order of arrival of tokens at the entry of m is indeterminate. Entry thus performs two tasks: it changes simple tokens into stream components, and nondeterministically merges the stream components into a single stream (stream X in the case of manager m). Conversely, response tokens on line RESULT must leave the manager through exit where they are converted back to simple tokens and are then returned to the waiting use. Together, entry and exit ensure that the use activity that sends the i^{th} member of X is the use activity that receives the i^{th} member of RESULT.

For convenience, Id permits multiple entry-exit pairs to appear in a manager definition, in which case all such pairs must be labeled, for example:

```
          md2 ← manager
                    (entry name_1: X_1;
                                .
                                .
                                .
                           name_n: X_n
                     do
                        .
                        .
                        .
                     exit name_1: R_1;
                                .
                                .
                                .
                          name_n: R_n)                    (5.4)
```

Here m2 ← create(md2) returns a structure with string selectors "$name_1$" through "$name_n$" which then requires the user to specify which entry of the created manager is to be used, such as use(m2.name1,b).

5.2 Implementation of resource managers

5.2.1 Creation of managers

Manager definitions with creation time parameters are actually converted to definitions without creation time parameters by incorporating a new entry called "&parameters". Furthermore, if the original manager has only an unnamed entry, then that entry is given the name "&noname".* In the case of statement (5.1) we then have

---
*A programmer cannot write entry names beginning with &.

```
     md ← manager
          (entry &parameters: S;
                 &noname: X
          do
             s´ ← first (S);
             s_0 ← s´[1];
             RESULT ← (initial s ← s_0
                        for each x in X do
                           <new s, answer> ← f(s,x)
                        return all answer)
          exit &parameters: S;
               &noname: RESULT)
```

Then any create statement such as (5.2) actually compiles as

```
     m´  ← create(md);
   ack  ← use(m´.&parameters,<a>);
     m   ← (if m´.&noname ≠ error then m´.&noname
          else m´ - ["&parameters"] )                               (5.5)
```

where m´ is not accessible to the programmer. Note that a structure is formed using the creation time parameters for initialization. The problem of mismatched parameters is handled exactly as in procedure application. If a stream S is used as an actual argument for a creation time parameter, then the second line of (5.5) compiles into

```
     done ← (initial akw ← λ
             for each s in S do
                 new akw ← use(me´.&parameters, s) when akw
             return true when akw)
```

where the when construct is used to hold a computation in abeyance until some specified event has occurred. In dataflow such an event can only be the arrival of a token. Thus

$$f(x) \ \underline{\text{when}} \ t$$

prevents evaluation of f(x) until a token arrives on line t. The when construct is easily implemented and is necessary for programming resource managers.

Creation of a manager is accomplished by a new base language operator CREATE. Assuming the manager definition md2 of (5.4) is input the CREATE operator may be described as:

$$\begin{aligned} \text{input} &= \{<c_m, u.c.s.i>\} \\ \text{output} &= \{<<\text{name}_1\text{: } u'.c_m.\text{entry}_1.1, \\ &\quad \vdots \\ &\quad \text{name}_n\text{: } u'.c_m.\text{entry}_n.1>, u.c.s'.i>\} \end{aligned}$$

where $u' = (u.c.s.i)$ and c_m is the manager's program code.

5.2.2 Using managers

A USE, like APPLY, comprises two operators U and U^{-1}. Figure 5.2 shows the execution of the expression use(m,y) where manager m was created in (5.2). Note that m names the actual activity that is to receive the argument value y. The following details the functioning of the new operators where as before c_m is the manager's program code.

The U operator sends a token, containing the argument and the "return activity name" to the specified entry port of the manager object:

$$\begin{aligned} \text{input} &= \{<(u'.c_m.\text{entry}.1), u.c.s_A.i>_{mgr}, <y, u.c.s_A.i>_{data}\} \\ \text{output} &= \{<<y, (u.c.s_T.i)>, u'.c_m.\text{entry}.1>\} \end{aligned}$$

The ENTRY operator accepts simple tokens coming from many sources (i.e., each U operator that receives the corresponding entry name as input), changes those tokens to stream components, and merges them nondeterministically into a single stream. Two streams are output:

one stream contains the input data while the other stream contains the "return activity name" for the EXIT operator:

$$\text{input} = \{<<y,(u.c.s_T.i)>,u'.c_m.\text{entry}.1>\}$$
$$\text{output} = \{<\{y,k\},u'.c_m.s'.1>,<\{(u.c.s_T.i),k\},u'.c_m.\text{exit}.1>\}$$

where k means this is the kth such input to this ENTRY.

Note that even though many sources may be sending tokens to this one ENTRY, it is a single activity and thus can keep a count k of each token as it (nondeterministically) arrives. The <u>EXIT operator</u> returns the tokens from its data input stream, after transforming them back to simple tokens, to the activity specified in the "return activity name" input stream called RA:

$$\begin{aligned}\text{input} &= \{<\{X_k,k\},u.c_m.\text{exit}.1>_{data} \mid 1 \le k \le n\} \cup \\ &\quad \{<\{(u.c.s_T.i)_k,k\},u'.c_m.\text{exit}.1>_{ra} \mid 1 \le k \le n\} \\ \text{output} &= \{<X_k,(u.c.s_T.i)_k> \mid 1 \le k \le n\}\end{aligned}$$

The <u>U^{-1} operator</u> like A^{-1} in procedure applications, acts only to distribute results in the calling environment.

5.2.3 <u>Destruction of managers</u>

The main problem in destroying a manager is to decide when a manager is no longer in use. A manager which cannot be referenced is an obvious candidate for destruction. However, unlike structures, circular name references are possible among managers (see Section 5.5). Therefore, in a hierarchical system two managers which are referenced only by each other may be eligible for destruction. We have not been able to devise a practical scheme for the detection of circular references. This problem is still under investigation.

5.3 Nondeterministic stream merge

Let A and B be streams. Then the stream produced by merge(A,B) is the result of nondeterministically merging streams A and B subject to the restriction that the i^{th} token is taken from A only if the $i-1^{st}$ token of A has already been output. The same behavior must hold for B. An est token is output only after an est is taken from both A and B. The merge construct is implemented by a MERGE operator. Aside from entry, which can be used only in the header of a manager definition, merge is the only nondeterministic function available to the programmer.

The behavior of the MERGE operator cannot be described in terms of a stream in - stream out function since the output stream is not uniquely determined by the input streams. The semantics of MERGE also cannot be described in terms of the set of all possible streams that preserve the token order within the input streams. This latter point is rather subtle and is due to the use of cyclic schemata within a program. A semantic specification of MERGE is beyond the scope of this paper and can be found in [Kosinski78].

5.4 An example - the readers and writers problem

The problem [CHP71,Hoare74] is to devise a resource manager to allow simultaneous read access but exclusive write access to the file under its control. The manager* fmd has two logical parts: an agent which performs the actual computation on the file, and a scheduler that blocks and enables individual requests within the agent. Figure 5.3 outlines the structure of such a manager. An instance fm of fmd to control file x may be created by

fm ← create(fmd,x)

*The solution presented here is taken from [AGP77].

Thus an expression to read the file x is

<u>use</u>(fm.read,r)

Each such request enters stream READQ. A write request is made in a similar manner and enters the stream WRITEQ. Each queued request waits until matched with an enabling signal from the streams READ_ENABLE or WRITE_ENABLE generated by the scheduler. When matched a queued request is released to the access_resource routine. Proper operation of the resource manager requires that the scheduler be notified whenever a request enters the manager or completes its read or write access. Since these signals are nondeterministically generated, we <u>merge</u> them within the resource manager to form a single stream X of signals to the scheduler.

A program for the file resource manager is given in (5.6) where the scheduler state is represented by the number of active readers (ra), the number of active writers (wa), the number of waiting readers (rw), and the number of waiting writers (ww). The scheduler enables requests to leave the waiting queues by producing a stream of reader enabling tokens (RE) or one writer enabling token (we). Note that

1. wa<=1 at all times,
2. if wa=1 then ra=0,
3. if ra>0 then wa=0.

The manager (5.6) implements Hoare's version of the readers and writers problem [Hoare74]. In this version a new reader is not permitted to proceed if a writer is waiting, and all readers that are waiting when a writer completes are allowed to proceed. This prevents indefinite exclusion ("starvation") of both the readers and the writers.

```
fmd ←
 manager(file)
 (entry read: READQ;
        write: WRITEQ do

    READ_RESULT,READ_DONE ←
       (for each r in READQ; re in READ_ENABLE do
          s ← access_resource(file,r) when re
        return all s, all "read exit" when s);

    WRITE_RESULT, WRITE_DONE ←
       (for each r in WRITEQ; we in WRITE_ENABLE do
          s ← access_resource(file,r) when we
        return all s, all "write exit" when s);

    READERS ← (for each r in READQ do
               return all "reader");

    WRITERS ← (for each r in WRITEQ do
               return all "writer");

    X ← merge(READERS,WRITERS,READ_DONE,WRITE_DONE);

    READ_ENABLE,WRITE_ENABLE ←
        (initial rw, ww, ra, wa ← 0, 0, 0, 0
         for each x in X do
             new rw, new ww, new ra, new wa, RE, we ←
                (if x = "reader"
                     then (if wa=0 and ww=0
                              then rw, ww, ra+1, wa, ["go"], λ
                              else rw+1, ww, ra, wa,[], λ)
                 else if x = "writer"
                     then (if wa=0 and ra=0
                              then rw, ww, ra, 1, [], "go"
                              else rw, ww+1, ra, wa,[], λ)
                 else if x = "read exit"
                     then (if ra=1 and ww>0
                                then rw, ww-1, 0, 1, [], "go"
                                else rw, ww, ra-1, wa,[], λ)
                 else !x = "write exit"!
                          (if rw>0
                                then 0, ww, rw, 0,
                                   (for i from 1 to rw do
                                    return all "go"), λ
                                else (if ww>0
                                          then rw, ww-1, ra, wa, [], "go"
                                          else rw, ww, ra, 0, [], λ)))
         return all RE, all we but λ)

    exit read: READ_RESULT;
         write: WRITE_RESULT)                                    (5.6)
```

Two other versions of the problem appear in [CHP71] and are easily programmed by simple alteration of the scheduler.

5.5 Dataflow managers and modularity

We also wish to make two more points about resource managers in Id. The first point concerns indeterminancy, which in sequential languages is usually a secondary effect of shared variables. In dataflow, indeterminacy is provided by explicit operators (MERGE and ENTRY in the base language) which gives the programmer more convenient control over nondeterministic behavior. The second point concerns the degree to which the requesting process is separated from the manager's internal controlling mechanisms. In a sequential language, each requesting process actually controls and executes the code inside the monitor [Brinch-Hansen72, Hoare74]. This characteristic of monitors makes it difficult, for example, to replace a software resource with a hardware resource. It also makes it difficult to guarantee valid use of the resource control mechanisms within a monitor, such as enforcing conventions on the proper sequence in which procedures of the monitor are to be called. Id, however, implements a resource manager as a closed module which nondeterministically receives requests from other processes, and acts upon these requests according to the scheduler (written by the programmer) enclosed within that manager. The requesting processes have no control over, and are entirely independent of the resource manager module which is itself an independent process. Such a model completely separates the user from the resource and should make hardware/software module interchange easier to achieve. In [Jammel & Stiegler77], a model of managers is presented in the von Neumann context that incorporates some of the ideas presented here.

Managers also provide a good model for processes themselves. If two managers know about each other, then the full expressive power of Id can be used for interprocess communication. Let md3 be a manager definition that requires a manager as a creation time parameter. Then two managers that reference each other are created by:

$$m_1 \leftarrow \underline{\text{create}}(md3,m_2);$$
$$m_2 \leftarrow \underline{\text{create}}(md3,m_1);$$

This does not result in a circular reference as can be deduced by the implementation of create given in Section 5.2.1.

5.6 Use of managers in determinate computation

Consider the following manager definition

```
md4 ← manager(S)
       (entry X do
        RESULT ← (for each x in X; s in S do
                       return all s)
        exit RESULT)                                        (5.7)
```

which returns the next element of the stream S whenever it receives a token in stream X. A token in X essentially represents a request for a token from stream S. Kathail [Kathail78] has shown that (5.7) may be very useful in solving determinate problems. We illustrate his technique by a program to generate in ascending numerical order the first n elements of the set $\{2^i 3^j 5^k \mid i,j,k>0\}$ [Dijkstra76, Kahn & MacQueen77]. One method for generating this sequence uses the three queues X1, X2, and h_3. Queue X1 contains numbers which are two times the number last output, while queue X2 contains numbers which are three times the number last output. The third queue h_3 is of length one and contains five times the number last output. At any given point, the next number output is the smallest number at the head of the three queues (i.e., $\min(h_1,h_2,h_3)$ where h_i is the head of the ith queue). If the ith queue has the smallest number at its head (thus becoming the next number output), then a new element is added to every queue before the ith queue, according to the rules stated above.

The following expression produces the desired set as the output stream A. The resource manager nexth1 releases tokens from input stream X1 on demand, thus treating stream X1 as a resource.

```
(nexth1 ← create(md4,X1);
 nexth2 ← create(md4,X2);
 A,X1,X2 ← (initial h1,h2,h3 ← 2,3,5
            for i from 1 to n do
            c ← min(h1,h2,h3);
            a,x1,x2,new h1,new h2, new h3 ←
                 (if c=h1 then h1, 2*h1, λ,
                                  use(nexth1,λ), h2, h3
                  else if c=h2 then h2, 2*h2, 3*h2,
                                       h1, use(nexth2,λ), h3
                  else !c=h3! h3, 2*h3, 3*h3,
                                       h1, h2, 5*h3)

            return all a, all x1, all x2 but λ)
 return A)
```

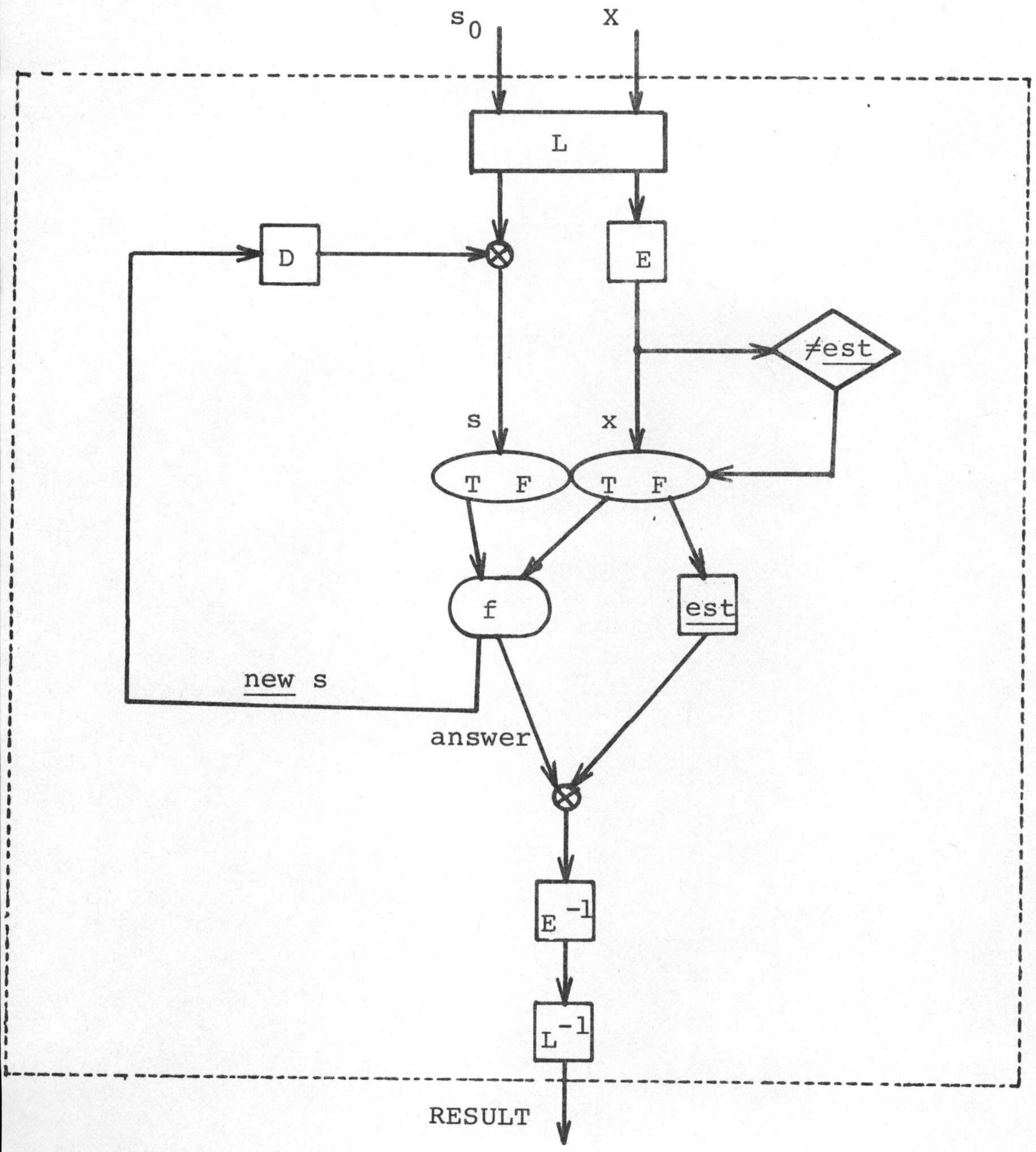


Figure 5.1 The body of a primitive resource manager

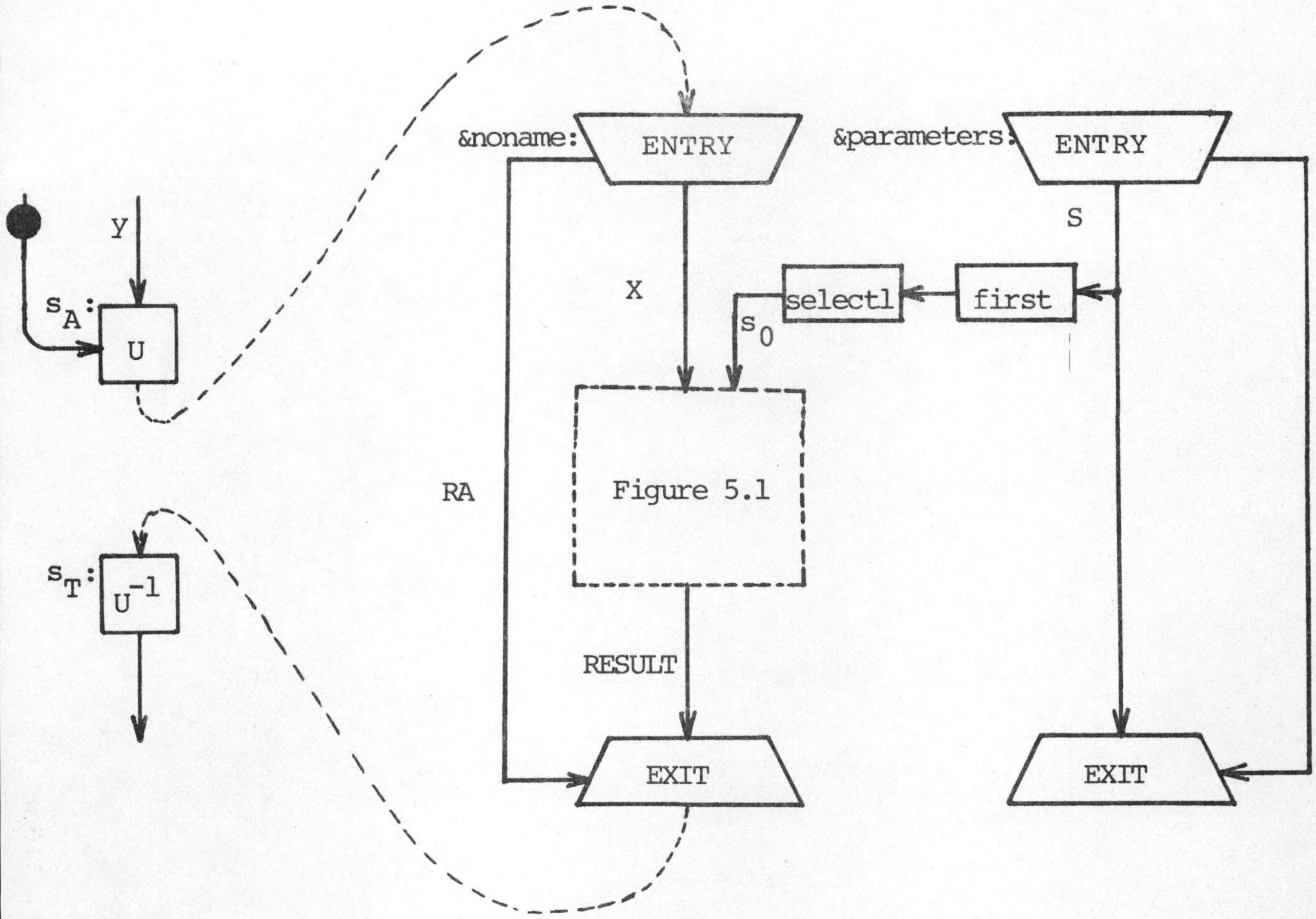


Figure 5.2 Using a resource manager

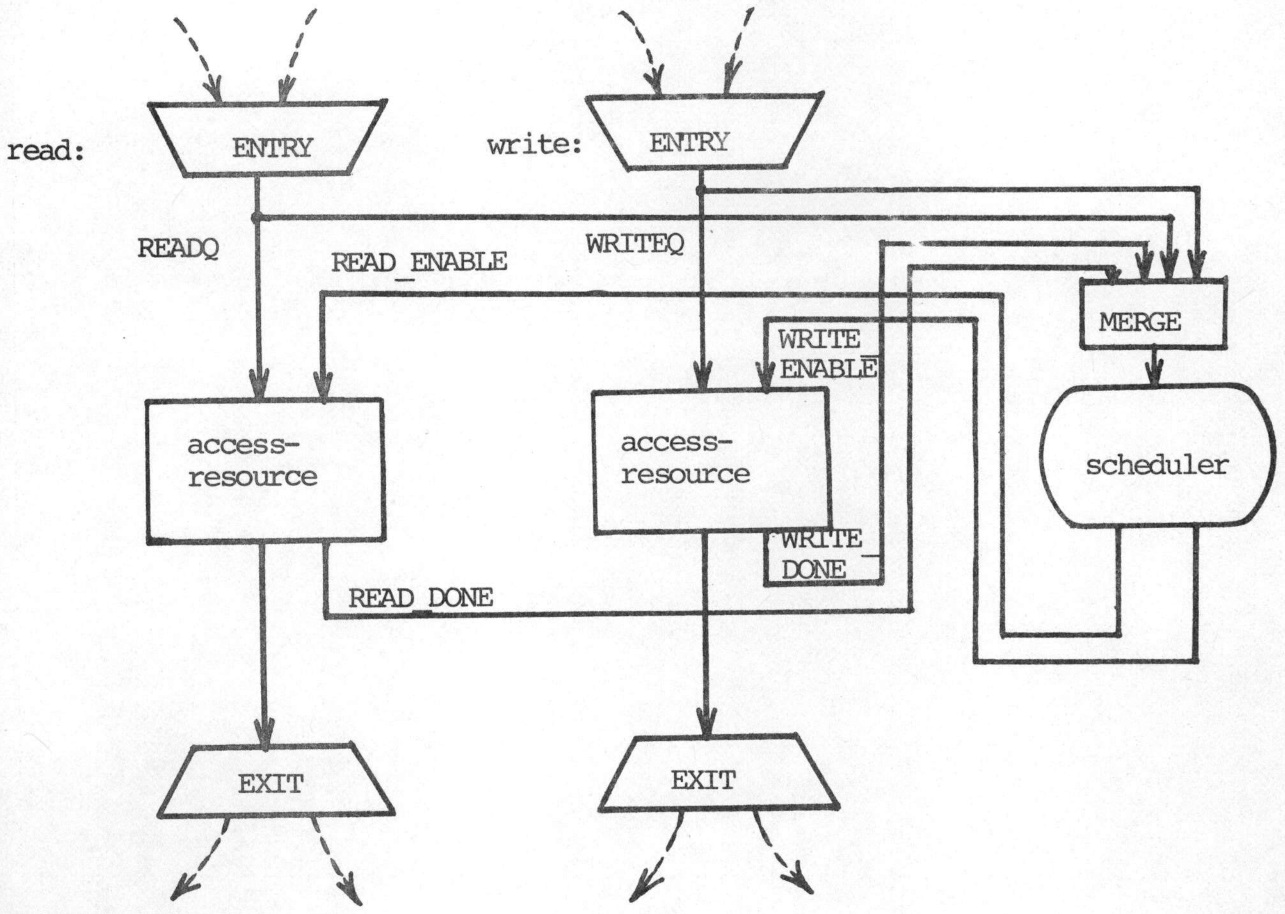


Figure 5.3 A file resource manager

6. Programmer-defined Data Types and Operator Extensionality

6.1 Data abstraction

Data abstraction deals with defining objects of a type not implemented by the system and various operations on the objects of that type. If done successfully, data abstraction relieves the user from the burden of knowing the internal representation of either the object or the implementation of the associated operations. In most languages, a special construct is used for definition of the abstraction and to serve as a template for generating instances of the new type. Generally, the operations on the data type are defined as procedures encapsulated in the definition. The class construct of SIMULA, the forms of ALPHARD [SWL77], and clusters of CLU [LSAS77] are examples of such definitional devices.

Even though the model for data abstraction in Id was not derived from any of the above languages, it has similarities to the SIMULA class concept. Consider defining a stack abstraction with the following operations:

"push":	$\text{stack} \times \text{item} \to \text{stack}$
"pop":	$\text{stack} \to \text{stack}$
"empty":	$\text{stack} \to \text{boolean}$
"top":	$\text{stack} \to \text{item}$

In SIMULA a class named stack may be written as follows:

```
class stack (s,ℓ);
      ref (item) array s;
      integer ℓ;
 begin
      procedure push (v);
      ref (item) v;
      begin
         ℓ := ℓ+1;
         s[ℓ] :- v;
      end;
```

```
          procedure pop;
             ℓ := ℓ-1;

          boolean procedure empty;
             empty := ℓ=0;

          ref (item) procedure top;
             top :- s[ℓ];
     end
```

where item is the name of another class that includes every type that may be used as stack elements. An empty stack (i.e., an object of type stack) is generated by writing

x :- new stack (a,0)

This statement gives x essentially a private integer ℓ (with initial value 0), array s (initialized to a), and the capability to access the stack operations contained in the class definition. These operations are invoked by writing x.push(b), x.pop, etc. The effect of push and pop is to change the internal state of x (i.e., s and ℓ). Hence, if the same stack instance pointed to by x is also pointed to by another variable, the value of that other variable is also changed.

In Id, data abstraction may be implemented by a procedure. The procedure definition represents the abstraction template; creating an instance of the abstraction is equivalent to freezing certain parameters of the procedures. Invoking an operation is done by applying the procedure with arguments that specify the operation to be carried out. Consider the following Id procedure which represents the stack abstraction.

```
z ← procedure stack (f,u,v,s,ℓ)
      (if f = "push" then compose(stack,<<4,s+[ℓ+1]v>,<5,ℓ+1>>)
       else if f = "pop" then compose(stack,<<4,s-[ℓ]>,<5,ℓ-1>>)
       else if f = "empty" then ℓ=0
       else if f = "top" then s[ℓ]
       else error("illegal operation on stacks"))            (6.1)
```

An empty stack is generated by writing

$$x \leftarrow \underline{\text{compose}}(z,<<4,\Lambda>,<5,0>>)$$

Note that procedure x is a copy of procedure stack with parameters s and ℓ frozen. Value b may be pushed onto stack x by writing x("push",x,b). For convenience, the syntax |f|(u,v,w) is used in Id to represent APPLY(u,<"f",u,v,w>).

A major difference between the Id and SIMULA data abstraction facilities is the removal of side effects. Procedure stack does not implement push and pop operations by altering the internal state of the stack; rather it creates a new stack on every push and pop operation. This effect can be achieved in SIMULA also by creating new stacks on every push and pop operation. However, such a stack implementation will be considerably costlier in execution time than the SIMULA stack given in this section. Distruction of an old stack in SIMULA also requires a programmer to explicitly delete references to that stack.

6.2 Operator extensionality and pdts

The stack example, and other examples in [Ravi Prakash78] suggest that current methodologies for data abstraction can be implemented by Id procedures and streams. However, we have chosen to distinguish a procedure to implement a programmer-defined data type (pdt) from other procedures. The procedure stack of

(6.1) can be made a pdt by replacing the word procedure by pdt. This distinction has been made to limit the problems encountered in operator extensibility.

There are exactly two functions defined on pdt values: ptype and compose. The operation ptype(x) returns the primitive or system type of the value x; thus ptype(5) returns "integer" and ptype(z) returns "pdt". The compose operator on pdts acts exactly like it acts on procedure values. Like any other Id value, a pdt can be passed as an argument and appended to a structure.

A pdt definition is often enclosed within a procedure to "hide" the initialization of the pdt from the programmer. For example, procedure stack_gen permits generation of only empty stacks and makes the internal state of the pdt stack (i.e., parameters s and ℓ) inaccessable to a user.

```
procedure stack_gen ( )
    (z ← pdt stack (f,u,v,s,ℓ)( ... )
     return compose(z,<<4,Λ>,<5,0>>))
```

If a user is to be allowed to generate non-empty stacks, the procedure stack_gen would perform error checking on its input parameters before performing the compose operation. Note that stack_gen does not affect the pdt stack, it only ensures the proper use of the abstraction.

Operator extensionality in Id is derived by extending the semantics of all non-control base language operators (except PTYPE and COMPOSE) as shown in Figure 6.1. Essentially an operator becomes an APPLY whenever its first argument is a pdt value. The control operator A is also extended.

This semantic extension of operators permits using "=" to test the equality of two stacks. Suppose the following code is included in the conditional expression of the pdt stack definition:

```
else if f = "="
     then (initial flag ← true
           for i from 1 to ℓ while flag ∧ not(|empty|v) do
               new v ←|pop|v;
               new flag ← s[ℓ+1-i]
           return flag ∧ i>ℓ ∧ ¬|empty|v)
```

If an element of stack v is another pdt instance then equality in s[i] = |top|v will dynamically change to an APPLY and check for equality according to the rules specified within the value s[i]. It is also worth noting that as long as "pop" and "empty" are defined, the internal representation of v is of no consequence.

To simplify type checking of pdts, one may observe the convention of including "type" as one of the operations on every pdt definition. Hence in the stack example we will include the following clause

else if f = "type" then "stack"

The predicate type(z) will respond by APPLY(z,<"type",z>) if z is a pdt value, otherwise, it will return ptype(z). (Please note that if z is a pdt and the programmer wrote z("type",z), the result would be APPLY(z,<"apply",z,<"type",z>>) because of operator extensionality).

6.3 Example - the programmer-defined data type set

We represent* a set as a boolean membership procedure which tests an element for membership in the represented set. In otherwords, the set {x | p(x)} will be represented by the boolean procedure p. We include only the following operations in our

* We are obliged to G. Ravi Prakash [Ravi Prakash78] for this representation.

definition of a set:

```
"type": set → "set"
     ε: set × value → boolean
     ∪: set × set → set
     ∩: set × set → set
```

To perform an operation like u ∪ v, we only need to generate another boolean procedure q that is the disjunction of the boolean procedures of u and v. However, one can not and should not extract the boolean procedure from v because set v may not use the same representation for sets as does u. Hence, only a legitimate operation on v, such as the membership test, should be used in creating procedure q. Consider the following definition:

```
procedure set_gen (p)   !p is a boolean procedure!
  ( z ← pdt set (f,u,v,p)
          (if f = "type" then "set"
           else if f = "ε" then p(v)
           else if f = "∪"
                then (q ← procedure(x,p,v)
                               (p(x) ∨ |ε|(v,x));
                      q´ ← compose(q,<<2,p>,<3,v>>)
                      return compose(set,<<4,q´>>))
           else if f = "∩"
                then (q ← procedure(x,p,v)
                               (p(x) ∧ |ε|(v,x));
                      q´ ← compose(q,<<2,p>,<3,v>>)
                     return compose(set,<<4,q´>>))
           else error("undefined operation on sets"))
    return (if ptype(p) = "procedure"
             then compose(z,<<4,p>>)
             else error("This set definition requires a
                         boolean procedure")))          (6.2)
```

This definition of sets works properly on finite as well as infinite sets. It does only the minimal execution needed to carry out the union and intersection operations. Again, nothing is assumed about the internal representation of the other set involved in these operations. And, in fact, different representations of the same pdt may be intermixed as long as the operators seen by the programmer are the only operations used, both by the programmer and the different representational definitions themselves.

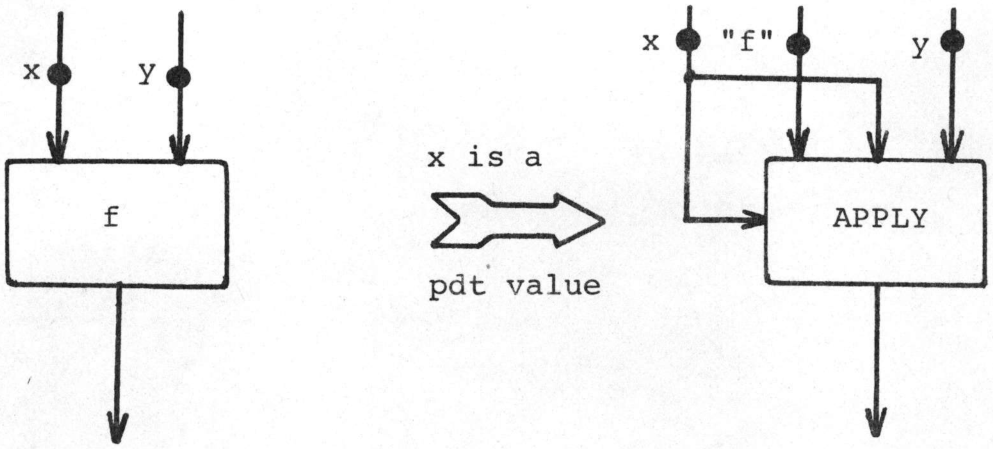


Figure 6.1. Extended semantics of non-control operators

7. An Architecture for a Dataflow Machine*

Our primary goal is to exploit parallelism present in programs by distributing activities generated during execution over a machine comprising hundreds of processors. However, unless care is exercised, the gains due to concurrent execution of activities can be overshadowed by communication costs. In this section, some general principles are stated for balancing such costs. We are not yet ready to specify a final design, but the basic unit of an architecture under study is shown in Figure 7.1. Simulation is the major tool for evaluating our ideas about architectures, and conclusions based on these results are also included where appropriate.

Simulation experiments have been conducted by executing compiled Id programs on an architecture comprising several physical subdomains shown in Figure 7.1 [Gostelow & Thomas78 . Several physical subdomains are combined into one physical domain by two disjoint bus networks: a token bus that connects the upper bus connectors (BC), and a memory bus connecting the lower BCs. We have not yet experimented with interconnection of physical domains because of experimental limitations and therefore do not suggest any particular physical domain connection scheme.

7.1 The token bus structure and locality

Each activity (the basic unit of computation) is assigned to a processing element (PE) for execution. When a PE completes execution of an activity, it manufactures the output tokens and

----
*Robert Thomas contributed much to the work reported in this Section.

evaluates an assignment function to determine the address of the destination PE for each token produced. This address is calculated using the destination activity name present on each output token. Note that any two PEs that are to send tokens to the same activity must agree on a particular assignment function.** Since an assignment function may assign more than one activity to the same PE at the same time, it is required that each PE sort all tokens received, by their activity names. Sorting capabilities may add complexity to PE design, but the alternative of scheduling PEs only when a PE is free, is either too centralized or complex.

The assignment function, instead of distributing activities over processors at random, follows a principle of locality, that is, activities logically close together are to be executed physically close together. Amongst the activities generated by the unravelling interpreter, the activities belonging to a logical domain are considered logically close. In Section 3 it was shown that activities within one logical domain do not interact with activities outside that domain (except for initial values and final results). Since activities from two domains are not usefully mixed, keeping them physically separate already effects some degree of locality.

** Physical mapping of activities on PEs is very different from the allocation schemes of our earlier architecture [Arvind & Gostelow77b]. There an activity could essentially obtain any free PE. We have rejected this undisciplined allocation of PEs on the basis of simulation results by Gostelow and Thomas [Gostelow & Thomas78].

A compiler can also enhance locality by coalescing sequentially dependent operators (such as a D operator and the operator that feeds it) into a single operator. However, coalescing also affects the "grain" of activity and thereby has a major impact on the design of a PE. We suspect that PEs should execute more complex operations that just +, -, *, etc.

Many bus structures are possible for interconnecting PEs, but the choice is affected by the particular assignment function used. We will use functions that accentuate locality (i.e., distribute messages nonuniformly). Thus an expensive bussing structure such as a cross bar switch which makes all PEs equidistant from each other is not needed. Due to high bandwidth requirements a centralized store and forward message switching facility is also unacceptable. However, between these two extremes are several hierarical bussing structures [Wittie76, Pierce72] with O(log n) delays among interconnected PEs.

Simulation results indicate that the assignment function is fundamental to balancing the need for distributing activities throughout the machine and at the same time ensuring locality. All assignment functions currently in use map each logical domain (procedure and loop instance) onto one physical sub-domain. The activities within such a logical domain are then confined to those PEs, giving locality in token transmission (and also in memory access). Since there are many such groups of locally connected PEs, and each group functions (essentially) independently, concurrency is exploited.

7.2 The memory system and redundancy

Memory in dataflow systems is needed to implement structures and to store program code. It is impractical to carry a complete structure on a token so a structure value is actually kept in memory while only a pointer to that structure is transmitted on a token. (We again emphasize that this memory system is not seen by the programmer.) For convenience we imagine all PEs share one large address space; however, memory modules (M) will be physically distributed. Every PE will have direct access to a (possibly shared) local memory module (Figure 7.1), and indirect access to memory modules local to other PEs. If the pointer on a token carries a physical memory address then a PE performing a structure operation can determine the memory module actually holding the structure. We expect each memory module to be connected to a memory controller (MC) which will be responsible for routing memory requests and for memory management tasks. An MC may store newly a newly created structure value directly in its local M or request a distant MC to do so.

In spite of similarities between token bus and memory bus requirements, there are also major differences. For example, the traffic on the memory bus will most likely involve information packets of variable size. Also, patterns of use may be quite different since every select operation implies two-way traffic (that is, (i) sending a request to the target memory module, and (ii) receiving the data from the target).

It should be noted that storing a structure in memory can have two detrimental effects. First, consider a structure that is being referenced by a large number of PEs. Since the memory module processes these requests one at a time, the gain due to distribution of activities on these PEs is essentially lost. Second, a PE referencing a structure can be physically quite distant from the memory module holding the strucutre, and hence, may have to wait a long time to receive a response. Our system follows a principle of redundancy, that is, data resident in memory may be physically present in more than one memory unit. This reduces memory conflicts as well as data communication time at the expense of memory space and copy time.

A local MC, when requested to perform an operation on a structure not stored locally, requests a copy of the level of the structure from the distant controller. After receiving a copy, the local controller updates its map of locally held structures. The local controller then performs the operation originally requested and is ready for any subsequent requests for that same structure -- a fairly probable event if the first request occurred within a loop executing in that sub-domain. Thus each memory controller acts as a local cache with respect to the rest of the memory system.

As stated earlier, the architecture under investigation is primarily for testing implementation principles. It also provides an evaluation of the effectiveness of the unfolding interpreter. We simply note that many other architectures are possible, and in this section we have concentrated on architectures that are suitable for implementing the unfolding interpreter. Several other interpreters of dataflow and their corresponding machines have been proposed [Dennis & Misunas74, Sonnenburg & Irani74, Treleaven77, Rumbaugh77, Davis78, GWG78].

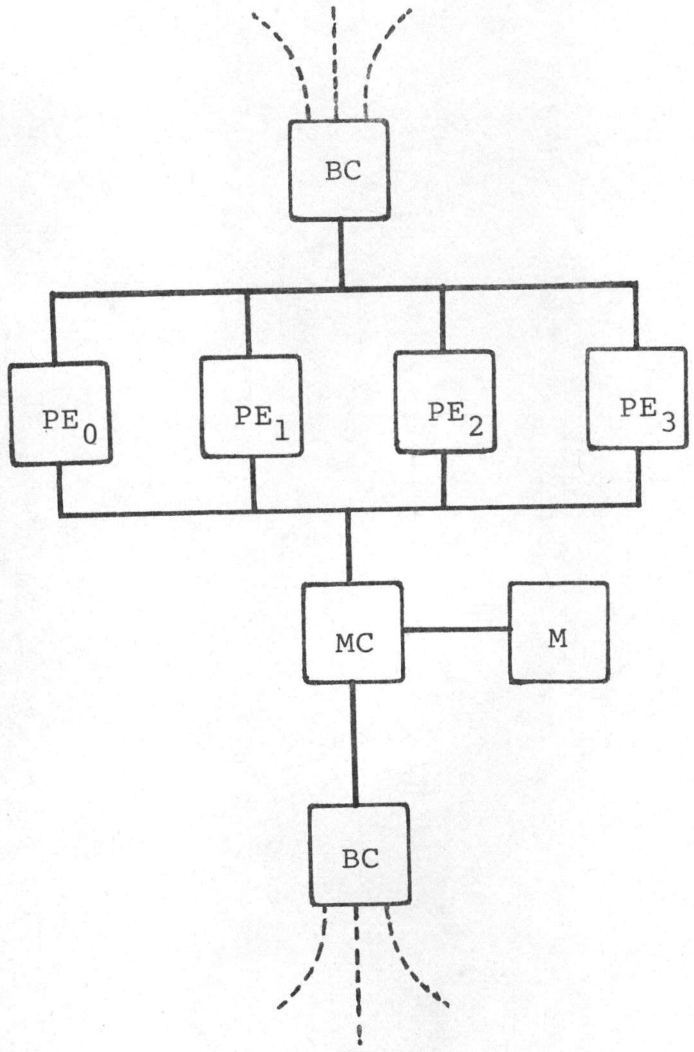


Figure 7.1 A physical sub-domain

8. Summary and Future Directions

In this paper we have presented an asynchronous computing system. The high level language, Id, and the base language are integrated and support each other. The unfolding interpreter for the base language exploits the asynchrony inherent in a program by unfolding loops and generating a potentially large number of independent activites. Many architectures have the potential to exhibit a high degree of concurrency while executing these activites.

Id not only makes it convenient for the user to program in dataflow by providing recursive procedures, loops, streams, managers, and pdts, but it also imposes very strong structure on the base language. There are only five basic schemata in the base language: blocks, conditionals, for each-while loops, procedure and pdt applications, and managers. Due to their rigid structure, the correctness and other desirable properties of these schemata can be easily proven.

Plouffe [Plouffe78] is developing an error recovery model that integrates error handling into Id with minimal expansion of the base language. In addition, Bic [Bic78] has shown that four new primitive operations, a tag field on each token, and some additional Id syntax permits solution of most of the security and protection problems found in the literature. Undoubtedly the clean semantics of the base language with the structure imposed by Id is the main reason for the success of these models.

The semantics of Id are not as elegant as that of Backus' FFP system [Backus78]. However Id is a more complete language because it permits expression of history sensitive as well as indeterminate computation. It may be worth incorporating some of these ideas into Backus' FFP systems.

The ultimate success of the system presented here will depend upon the design of machines that can execute the base language efficiently. Current experiments on an architecture suggest that the goal of designing high performance dataflow machines is not unreasonable [Gostelow & Thomas78].

Acknowledgements

We wish to acknowledge the contributions from Robert Thomas, Vinod Kathail, and G. Ravi Prakash as noted in the text. We also express our gratitude to Lubomir Bic, Tom Lipkis, Jim Neighbors, Robert Thomas, and Chuck Williams for their careful reading of this paper, and Shirley Rasmussen for her typing of many versions.

References

[Arvind & Gostelow77a] Arvind, and Gostelow, K.P. Some relationships between asynchronous interpreters of a data flow language. In Formal Description of Programming Languages, E. J. Neuhold, Ed., North-Holland, New York, 1977.

[Arvind & Gostelow77b] Arvind, and Gostelow, K.P. A computer capable of exchanging processing elements for time. In Information Processing 77, B. Gilchrist, Ed., North-Holland, New York, 1977.

[AGP77] Arvind, Gostelow, K.P., and Plouffe, W.E. Indeterminancy, monitors, and dataflow. Proc. Sixth ACM Symp. on Operating Systems Principles, Nov. 1977, pp. 159-169.

[Ashcroft & Wadge76] Ashcroft, E. A., and Wadge, W.W. LUCID -- a formal system for writing and proving programs. SIAM J. Comput. 5, 3 (Sept. 1976), 336-354.

[Backus73] Backus, J. Programming language semantics and closed applicative languages. Proc. ACM Symp. on Principles of Programming Languages, Oct. 1973, pp. 71-86.

[Backus78] Backus, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Comm. ACM 21, 8 (Aug. 1978), 613-641.

[Bährs72] Bährs, A. Operation patterns. In International Symposium on Theoretical Programming, A. Ershov and V. A. Nepomniaschy, Eds., Springer-Verlag, New York, 1974.

[Bic78] Bic, L. Protection and security in a dataflow system. Ph.D. Dissertation, Dept. of Information and Computer Science, Univ. of California, Irvine, California, Oct. 1978.

[Brinch-Hansen72] Brinch Hansen, P. Structured multiprogramming. Comm. ACM 15, 7 (July 1972), 574-578.

[Chamberlin71] Chamberlin, D. D. The "single-assignment" approach to parallel processing. Proc. AFIPS 1971 FJCC, Nov. 1971, pp. 263-269.

[CHP71] Courtois, P.J., Heymans, F., and Parnas, D.L. Concurrent control with "readers" and "writers". Comm. ACM 14, 10 (Oct. 1971), 667-668.

[Davis78] Davis, A.L. The architecture and system method of DDM1: A recursively structured data driven machine. Proc. 5th Annual Symp. on Computer Arch., Apr. 1978, pp. 210-215.

[Dennis73] Dennis, J.B. First version of a data flow procedure language. Computation Structures Group Memo 93, Lab. for Computer Science, Cambridge, Massachusetts, Nov. 1973. (revised as MAC Technical Memorandum 61, May 1975.)

[Dennis74] Dennis, J.B. On storage management for advanced programming languages. Computation Structures Group Memo 109-1, Lab. for Computer Science, Cambridge, Massachusetts, Oct. 1974.

[Dennis & Misunas75] Dennis, J.B. and Misunas, D. P. A preliminary architecture for a basic data-flow processor. Proc. 2nd Annual Symp. on Computer Arch., Jan. 1975, pp. 126-132.

[Dijkstra76] Dijkstra, E.W. A discipline of programming. Prentice-Hall, Englewood Cliffs, N.J., 1976, (Chapt. 17).

[Friedman & Wise76a] Friedman, D.P., and Wise, D.S. CONS should not evaluate its arguments. In Automata, Languages, and Programming, S. Michaelson and R. Milner, Eds., Edinburgh Univ. Press, Edinburgh, England, 1976.

[Friedman & Wise76b] Friedman, D.P., and Wise, D.S. The impact of applicative programming on multiprocessing. Proc. 1976 Intl. Conf. on Parallel Processing, Aug. 1976, pp. 263-272.

[GIMT74] Glushkov, V.M., Ignatyev, M.B., Myasnikov, V.A., and Torgashev, V.A. Recursive machines and computing technology. In Information Processing 74, J. L. Rosenfeld, Ed., North-Holland, New York, 1974.

[Gostelow & Thomas78] Gostelow, K.P. and Thomas. R.E. Per-Performance of a dataflow computer. Tech. Report 127, Dept. of Information and Computer Science, Univ. of California, Irvine, California, (in preparation)

[GWG78] Gurd, J., Watson, I., and Glauert, J. A multilayered data flow computer architecture. Dept. of Computer Science, Univ. of Manchester, Manchester, England, Jan. 1978.

[Guttag77] Guttag, J. Abstract data types and the development of data structures. Comm. ACM 20, 6 (June 1977), 396-404.

[Hoare74] Hoare, C.A.R. Monitors: An operating system structuring concept. Comm. ACM 17, 10 (Oct. 1974), 549-557.

[Jammel & Stiegler77] Jammel, A.J. and Stiegler, H.G. Managers versus monitors. In Information Processing 77, B. Gilchrist, Ed., North-Holland, New York, 1977.

[Jones77] Jones, A.K. The narrowing gap between language systems and operating systems. In Information Processing 77, B. Gilchrist, Ed., North-Holland, New York, 1977.

[Kahn74] Kahn, G. The semantics of a simple language for parallel programming. In Information Processing 74, J. L. Rosenfeld, Ed., North-Holland, New York, 1974.

[Kahn & MacQueen77] Kahn, G., and MacQueen, D. Coroutines and networks of parallel processes. In Information Processing 77, B. Gilchrist, Ed., North-Holland, New York, 1977.

[Kathail78] Kathail, V. Some extensions to Irvine dataflow language (Id). Masters Thesis, Computer Science Programme Indian Institute of Technology, Kanpur, India, July 1978.

[Kosinski73] Kosinski, P.R. A data flow language for operating systems programming. SIGPLAN Notices (ACM) 8, 9 (Sept. 1973), 89-94.

[Kosinski76] Kosinski, P.R. Mathematical semantics and data flow programming. Proc. Third ACM Symp. on Principles of Programming Languages, Jan. 1976, pp. 175-184.

[Kosinski78] Kosinski, P.R. A straightforward denotational semantics for non-determinate data flow programs. Proc. Fifth ACM Symp. on Principles of Programming Languages, Jan. 1978, pp. 214-221.

[Landin65] Landin, P.J. A correspondence between Algol 60 and Church's lambda-notation: Part I. Comm. ACM 8, 2 (Feb. 1965), 89-101.

[LSAS77] Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. Abstraction mechanisms in CLU. Comm. ACM 20, 8 (Aug. 1977), 564-576.

[McCarthy60] McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, Part I. Comm. ACM 3, 4 (Apri 1960), 184-195.

[Newell & Tonge60] Newell, A., and Tonge, F.M. An introduction to Information Programming Language V. Comm. ACM 3, 4 (Apr. 1960), 205-211.

[Patil70] Patil, S.S. Closure properties of interconnections of determinate systems. Record of the Project MAC Conf. on Concurrent Systems and Parallel Computations, June 1970, pp. 107-116.

[Pierce72] Pierce, J.R. Network for block switching of data. Bell Sys. Tech. J. 51, 6 (July-Aug. 1972), 1133-1145.

[Plouffe78] Plouffe, W.E. Exception handling and recovery in a dataflow system. Ph.D. Dissertation, Dept. of Information and Computer Science, Univ. of California, Irvine, California, (to appear).

[Ravi Prakash78] Ravi Prakash, G. Programmer-defined data types in Irvine data flow language (Id). Masters Thesis, Computer Science Programme, Indian Institute of Technology, Kanpur, India, July 1978.

[Rumbaugh77] Rumbaugh, J.E. A data flow multiprocessor. IEEE Trans. on Computers C-26, 2 (Feb. 1977), 138-146.

[SWL77] Shaw, M., Wulf, W.A., and London, R.L. Abstraction and verification in Alphard: Defining and specifying iteration and generators. Comm. ACM 20, 8 (Aug. 1977), 553-564.

[Sonnenburgh & Irani74] Sonnenburgh, C.R. and Irani, K.B. A configurable parallel computing system. Technical Report 82, Dept. of Electrical Engineering, Univ. of Michigan, Ann Arbor, Michigan, Oct. 1974.

[Treleaven77] Treleaven, P.C. Exploitation of parallelism in computer systems. Ph.D. Thesis, Dept. of Computer Science, Victoria Univ. of Manchester, Manchester, England, Feb. 1977.

[Weng75] Weng, K.S. Stream-oriented computation in recursive data flow schemes. MAC Technical Memorandum 68, Lab. for Computer Science, Cambridge, Massachusetts, Oct. 1975.

[Wittie76] Wittie, L.D. Efficient message routing in mega-micro-computer networks. Proc. 3rd Annual Symp. on Computer Arch., Jan. 1976, pp. 136-140.