THE (PRELIMINARY) Id REPORT: AN ASYNCHRONOUS PROGRAMMING LANGUAGE AND COMPUTING MACHINE*

by

Arvind
Kim P. Gostelow
Wil Plouffe

Technical Report #114

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
May 10, 1978

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

Preface

This report has been a long time in the making, and as a result several modifications (and some extensive improvements) have accumulated in our personal notes. However, to include them now and to re-integrate the language would only delay this report further.

So in the interest of providing some document on Id, these changes and improvements will be reserved for an updated version of the language. At that point we will be able to remove the qualifier "Preliminary" from the title.

Your comments and criticisms are welcomed.

A
KG
WP

Table of Contents

Preface i

Introduction 1

Elementary Programming in Dataflow 6

The Base Language and the Unravelling Interpreter 34

Programming with Streams 51

Resource Managers 94

Programmer-defined Data Types, Extensionality,
and Environments 110

References 121

1. Introduction

The purpose of this work is to capture what one intuitively feels is the enormous potential of LSI technology to produce large numbers of small processors to be the building blocks for a large general-purpose computer. A characterization of the kind of computer we have in mind is the following: The machine would consist of a large number (possibly hundreds or even thousands) of small asynchronously operating processors. Each processor accepts and performs a small task generated by a program, produces partial results, and then sends these partial results to other processors in the system. Thus the many processors would cooperate towards the common goal of completing the overall computation. A natural concomitant effect of such behavior would be increasing speeds of computation as new processor modules are added to the machine.

Many computer architects have imagined machines that might exhibit such behavior and thereby utilize this new technology. But in trying to discover why no such machine has as yet demonstrated real success in this endeavor, we became convinced that the real problems are not related simply to devising an appropriate bus and machine interconnection scheme, or to 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 taken ever since. The von Neumann model has become so ingrained in our thinking that we rarely even consider it, let alone question it, but it is precisely here that we find the source of the real problems that have prevented the creation of the kind of machine just described.

Two particularly troublesome attributes of the von Neumann model are [Dennis73, GIMT74, Sutnerland77, etc]:

1. centralized sequential control
2. 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 devise. It also burdens the programmer with the need to explicitly specify (or to employ an

analyzer to determine) exactly where concurrency is to occur. The second point, the memory cell, 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 introduction of asnychrony into a programming system makes memory cells even less tolerable; we illustrate why this is so by an extreme case: the global variable. We imagine a situation where some otherwise asynchronous processor modules are busy executing tasks, but these tasks require coordination through a common cell. This calls for rather complex synchronization controls to ensure orderly use of the global variable. Such controls are difficult to design into a machine and may be very costly in execution time. Synchronization controls 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 in order to obtain a useful general-purpose system composed of large numbers of small processors. 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 everywhere asynchronous except where synchronization is explicitly specified (i.e., no sequential control), and where values are the subject of computation rather than the places where those values are kept (i.e., no memory cells). An asynchronous language assumes computations are unrelated, and thus concurrent, unless otherwise specified. A sequential or von Neumann language on the other hand requires explicit specification (either by the programmer or by analysis of the program) to identify those places where concurrent processing may be initiated. 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, Sutherland77, etc.]. Essentially, 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. Consider the drive towards structured programming. Often it can be viewed as an attempt to produce programs that are more functional (as opposed to procedural) in their operation. As an example, modern programming practice suggests that returning results by modification of global variables shared among subprograms is less desirable than writing the subroutine as a function-type subprogram and returning values as the result of a function call. The fact that this 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 programing language design: We 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 [Shaw, Liskov, Guttag, etc.] points in a direction 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 past few years work on the linguistic aspects of resource control [Jones] where as we move from semaphores, to conditional critical regions, and to monitors [Brinch-Hansen, Hoare74], we see movement away from arbitrary specification of program synchronization (semaphores) to more highly controlled and encapsulated specifications (monitors). 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 a 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 akin 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 [Guttag, Ashcroft & Wadge76].

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 in studies related to the above examples in language design, remedies for reducing 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 the new technology of LSI in the manner described above, we cannot retain the von Neumann base. We furthermore believe that the semantic foundation we require coincides in the long run with the natural culmination of much of the evolutionary movement ongoing 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 [Arvind & Gostelow77a] -- 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> [Arvind & Gostelow77b, Dennis73, Kosinski73]. (Pure LISP [McCarthy60] and Red languages [Backus73] were not chosen for adoption because, even though their semantic bases are elegant and functional, neither caters to asynchronous operation.) The major deviation of "data flow" semantics from von Neumann's principles of "control flow" is that dataflow is <u>asynchronous</u>, and it has no memory cells -- only <u>values</u> are the results of computations. 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 no side-effects as a result of its execution.

Arguments in the past against dataflow have centered around the lack of a higher-level language, the 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 higher-level dataflow language that incorporates all of the usual programming concepts, as well as some new concepts not usually found in contemporary languages (for example, streams, functionals, and non-deterministic 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 procedure definition and manipulation, programmer-defined data types, and operator extensionality. The ability to handle resource (database) problems is also a capability of the language. Concerning the above noted objection of "efficiency", one must first of all not evaluate dataflow in terms of a von Neumann implementation, for dataflow not only allows a new kind of machine design but in fact requires it. It is most important when considering dataflow languages that they be considered in their own terms and not be forced to fit into measures valid only for other systems.

Two languages will be described here: a higher-level language Id (for Irvine dataflow), and a base machine language that serves as the semantic language of Id. 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, while Section 7 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. A program in Id is a list of expressions. In this section we explain the four most basic kinds of Id expressions -- blocks, conditionals, loops, and procedure applications -- by giving examples of each, and by giving their translation into the base language. We use the base language both to define and to explain Id; the base language is also used as the machine language to be directly executed by a dataflow computer. Programs, however, are written only in Id. Thus the base dataflow language is discussed only in the context of how that language supports the higher-level Id constructs. We are less concerned with how the base language operators behave in isolation than how they behave in concert with one another in building these Id constructs.

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. This matter is discussed later in this section where we detail two particular value types: structure values and procedure definition values.

2.1 Block expressions: To evaluate the two roots of a quadratic equation we can write the following list of expressions or program:

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

Any program can be written as a list of expressions in Id, however it is often more convenient for a programmer to break a computation into pieces, identify certain partial results, and then use those partial results to compute a final answer. Thus we can rewrite (2.1) as the 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 expression shown in Figure 2.1. The reader should refer to Figure 2.1 and to expression (2.2) while noting the following: An assignment statement serves to name the

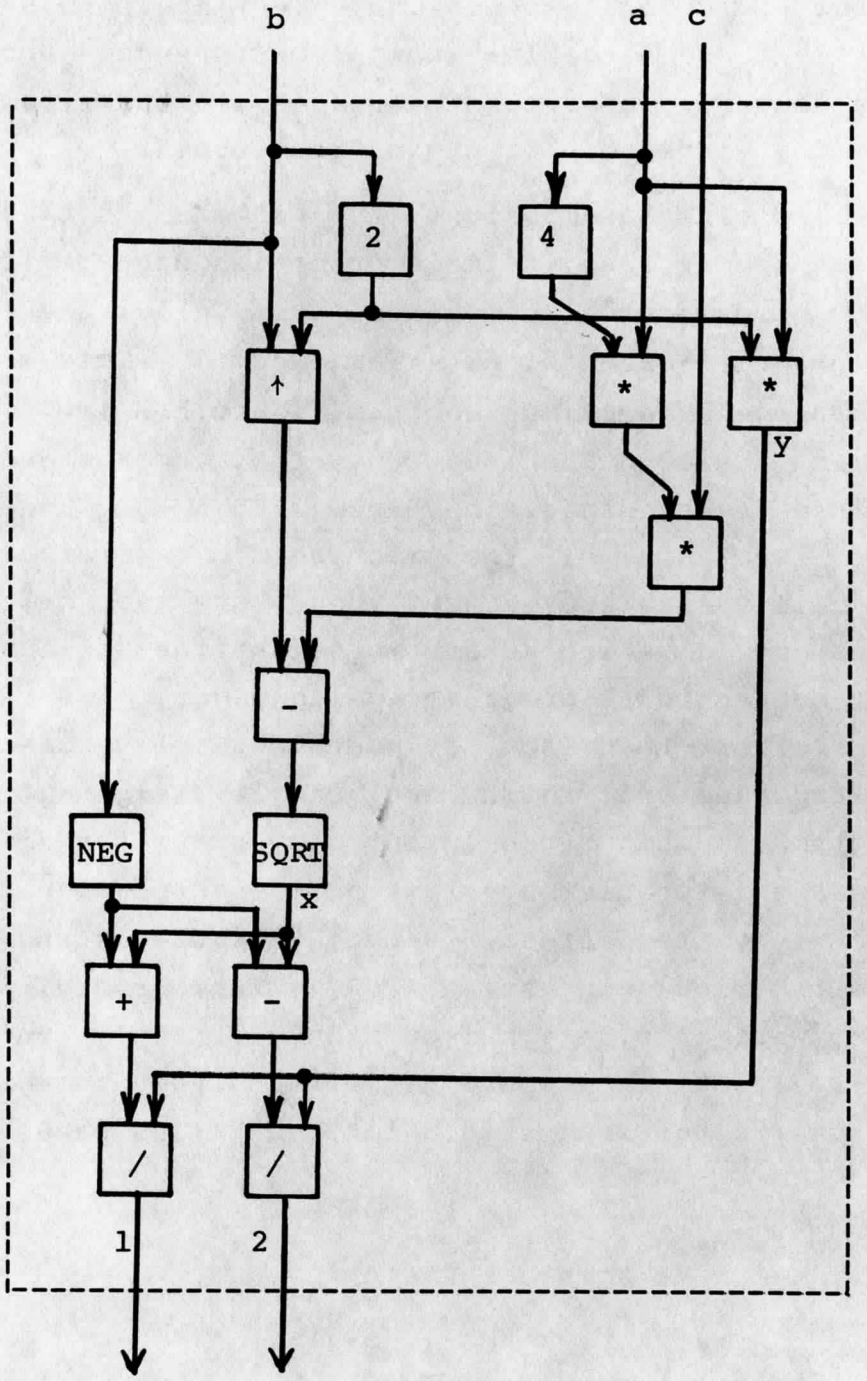

Figure 2.1
Compilation of the block expression (2.2)

output(s) of an expression. The name itself is called an Id variable. Variables are used to specify interconnections among operators (the boxes in Figure 2.1). Assignment statements in a block are separated by semicolons and can always be commuted without affecting the result of the expression. The inputs to a block expression are exactly those variables that are referenced but not assigned within that block. The return clause is the last item in a block and specifies the (ordered) outputs of that block.

From the above, it is most important to realize that an Id assignment is not an operator as it is in other languages; it is a specification to the compiler to label an output. In Id, a variable can be assigned exactly once within its scope. This single assignment rule makes the connection shown in Figure 2.2 impossible. The single-assignment rule guarantees to us that once defined, an instance of a variable never changes in value. This obviates the need for disparate sections of the machine to coordinate the updating of memory cell variables, since there are no cells to update. In this respect Id variables are more like the variables of mathematics than of conventional programming languages, i.e., they stand for values rather than act as containers of values. One problem with most languages that follow the single-assignment rule is that the programmer must often invent many names for distinct variables. Id avoids this problem by limiting the scope of names to the block in which they are defined. Hence, variables x and y are not visible outside the block expression (2.2). Furthermore, if the same names x and y were also assigned outside (2.2) they would not be visible inside (2.2) and hence would not affect the computation within that block. Expression (2.3) further illustrates this scoping rule.

```
( a <- 1; b <- 1; c <- -2;
  x,y <- (x <- sqrt(b^2-4*a*c);
          y <- 2*a
          return (-b+x)/y,(-b-x)/y)
  return x,y )                                    (2.3)
```

When expression (2.3) is evaluated the values of x and y in the inner block will be 3 and 2, respectively, while the values of x and y in the outer block will be 1 and -2. Unlike ALGOL, the scoping rules of Id prevent assignments to global variables from an inner block since assignment to a variable automatically defines its scope

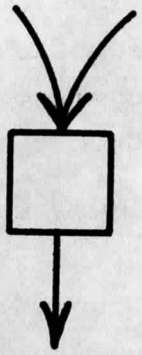

Figure 2.2
An illegal connection

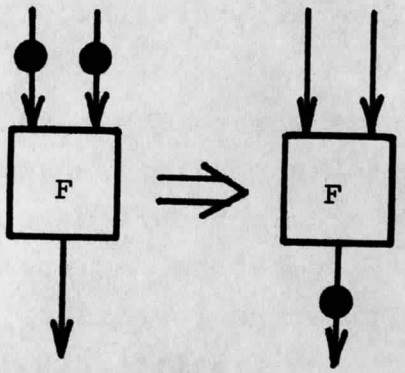

Figure 2.3a
Execution of a dataflow operator

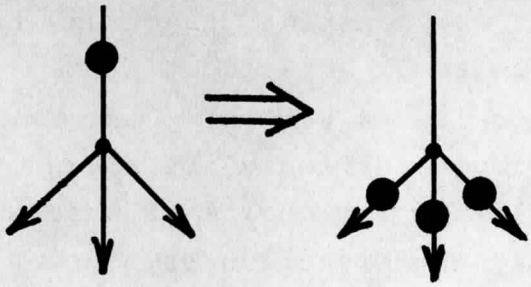

Figure 2.3b
Behavior of a fork

to be that of the innermost block in which that assignment occurs, and assignment can occur only once in that scope. Use of the same variable name in two different blocks is not a violation of the single-assignment rule since the scopes do not overlap -- one name is simply being used to label two distinct outputs.

values in the base language are carried by tokens that flow along lines. According to the first principle of dataflow, an operator may execute when and only when all its required input tokens have arrived. An operator executes by absorbing all input tokens, computing a result, and producing an output token that carries that result as its value. Operator execution is illustrated in Figure 2.3a. Note that the operators internal to the block expression of Figure 2.1 will start executing as soon as any tokens on lines a, b, or c arrive.

Figure 2.3b shows that whenever a token encounters a fork while traversing a line, the token replicates and follows all branches of the fork. In this way a single result may be sent asynchronously as input to many different operators. Clearly, several operators may be enabled at any given time and the order of execution of those operators does not affect the final result, that is, the computation is determinate [Patil70, Arvind & Gostelow77a].

The reader should note that a constant in Id does not represent a value, but rather a function that produces that value as its output regardless of the value of its input. But since a dataflow operator will not execute until its input is present, some token must always be sent to a constant function to indicate that an output value is needed. Any input line used for signaling a constant function is called a trigger. The value of a token on a trigger line is unimportant, only its presence is significant. As will become clearer later, the choice of which line to use as a trigger affects only the execution asynchrony of an expression and not the final results.

Reflecting on the basic principles of dataflow given in Section 1, we see that Id is asynchronous because (sub-) expressions are independent of one another and may be executed in any order, or at the same time, unless otherwise constrained by an explicit need for partial results. Expressing the need for partial results is easy:

just write the variable name. This approach is the inverse of the usual method of obtaining asynchrony or parallelism in conventional languages which requires either an analysis of the problem or the use of parallel programming constructs (e.g., cobegin-coend). Also, Id does not present the concept of a memory cell to the programmer since the value of a variable cannot be updated. The purpose of a variable is solely to allow the programmer to name partial results which he may later reference in other expressions. It is the single-assignment rule that removes the possiblity of a race and implies that Id programs are determinate (unless, of course, some operator is used which internally is nondeterministic).

2.2 Conditional expressions: Consider the Id conditional expression

(if p(x) then f(x) else g(x)) (2.4)

and its base language translation in Figure 2.4. 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 box f, otherwise it is sent to box g. Figure 2.5a shows the behavior of the SWITCH operator for the case of a true valued boolean input. Also, Figure 2.5b shows that if one of the output lines of a SWITCH is not used, and if according to the boolean input that branch is to be taken, then the input token is simply absorbed. The ⊗ operator is used in base language program graphs in exactly three situations: to mark the merging of two branches of a conditional expression (as in Figure 2.4), to mark the formation of a loop (Section 2.3), and to produce stream output from a loop (Section 4). As with the SWITCH and ⊗, we will see that many base language operators do not result in legal programs if interconnected arbitrarily. Since we intend to program in Id and not in the base language, we do not give rules for forming all possible legal base language programs. The emphasis here is on specifying the semantics of syntactically correct Id programs in terms of legal base language programs. The operational meaning of a legal base language program will be defined more precisely in Section 3.

A conditional expression needs all of its inputs for execution regardless of the branch to be taken. For example, the expression

(if p(x) then f(x) else g(x,y)) (2.5)

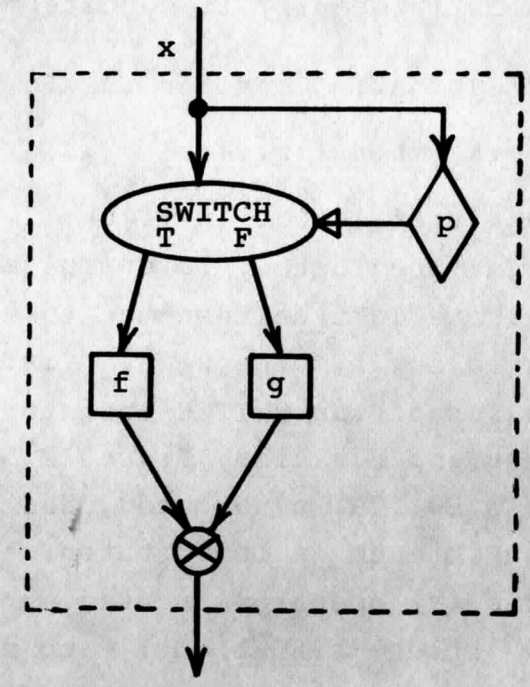

Figure 2.4
Compilation of the conditional expression (2.4)

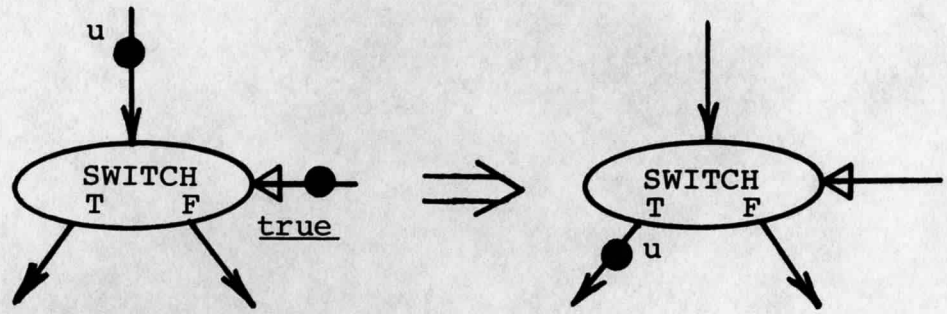

Figure 2.5a
The SWITCH operator for a true input

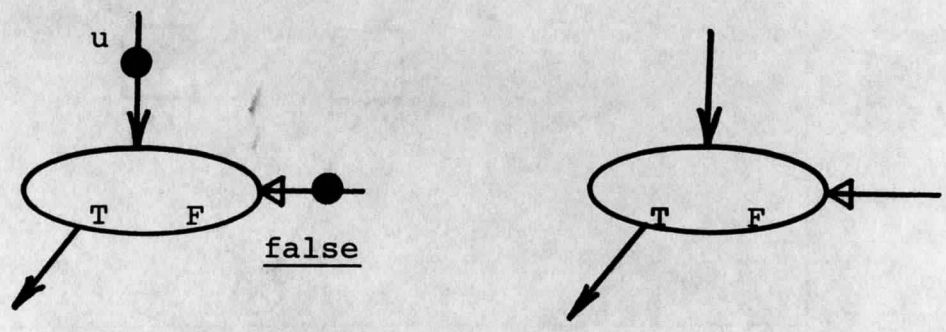

Figure 2.5b
The case for a SWITCH whose output
is not utilized

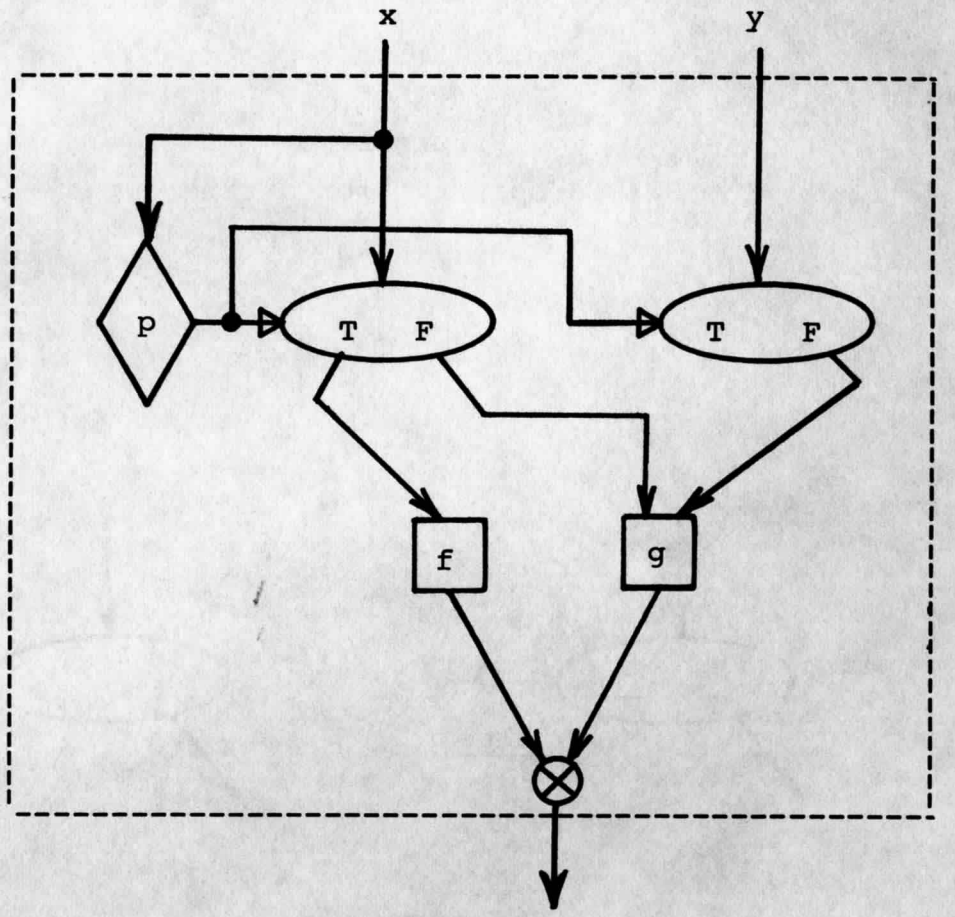

Figure 2.6
Compilation of the conditional expression (2.5)

will always require an input token from y, but whenever p(x) is true that token is simply absorbed and is not used. Figure 2.6 gives the base language translation of expression (2.5) and clearly shows that a token is always absorbed from y. The reason a token is always absorbed is to maintain the proper order of tokens flowing along all lines, 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 than an entire expression behaves similar to a primitive box.

Id also provides syntax for writing conditionals in statement form. The meaning of the conditional statement (2.6) is given by the conditional expression (2.7).

```
(if p(x) then y <- f(x); z <- 1  else y <- g(x); z <- ∅)    (2.6)
      y,z <- (if p(x) then  f(x),1  else  g(x),∅)           (2.7)
```

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**
```

A case-expression (case-statement) may be written as a nest of if-then-else expressions (statements).

2.3 Loop expressions: Like most conventional languages, looping constructs are important for writing interesting programs in Id. All looping constructs in Id are expressions consisting of four parts: an initial part, a predicate to decide further iteration, a loop body, and a list of expressions to be returned as the value of the loop. Consider the loop expression (2.8). It represents a program to compute the smallest i such that the sum of all integers from 1 through i exceeds some number s.

```
1   ( initial i <- 1;
2             sum <- 1
3     while sum<s do
4           new i <- i+1;
5           new sum <- sum+i
6     return i )                                            (2.8)
```

An Id loop is a set of recurrence relations, where new values are specified in terms of old values and initial values. For

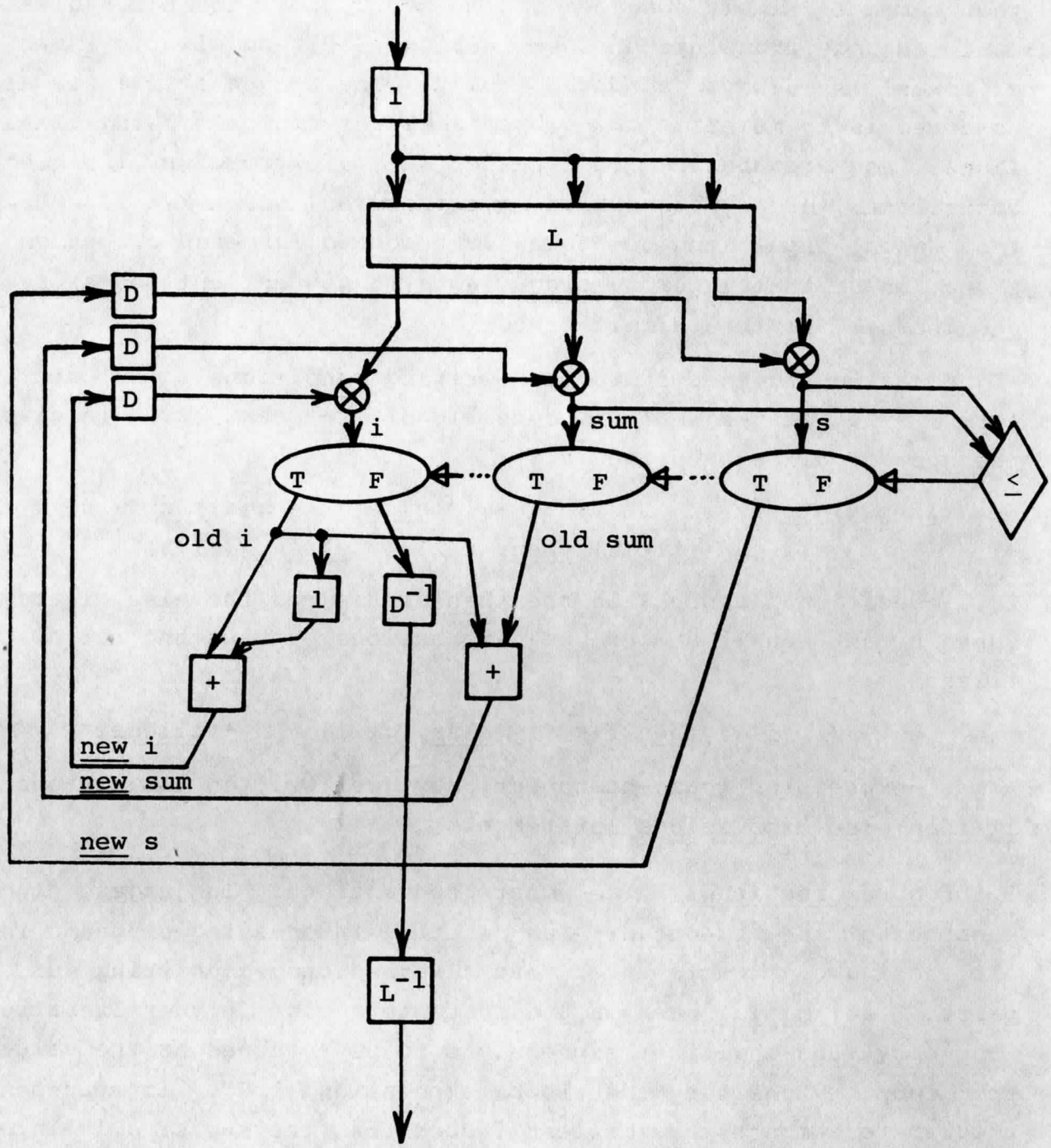

Figure 2.7
Compilation of the loop expression (2.8)

example, a set of recurrence equations for computing the above values of i and sum are

$$\begin{aligned} i_{j+1} &= i_j + 1 \qquad \underline{\text{where}}\ i_1 = 1 \\ sum_{j+1} &= sum_j + i_j \qquad\qquad sum_1 = 1 \end{aligned}$$

Id differs only in that a stopping condition is specified and those final values of interest are written in the return clause of the loop. Thus the statements in the loop body specify that new value instances of i and 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 (2.8) 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 (2.8) into the base language is given in Figure 2.7. 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 the sake of dicussion at this point we can treat these operators as as identity functions.)

We would like to emphasize that the assignment statements new i<-i+1 and new sum<-sum+i do not violate the single-assignment rule. Since i on the left-hand side is new i and the i on the right-hand side is old i, we really have two distinct lines in the base language program. One is tempted to think of i and sum as memory cells whose values are being updated, but as pointed out earlier, a dataflow variable never represents a memory cell. Also, any assignment statement whose left-hand side is a variable that is not assigned an initial value is not considered a recurrence statement. Such variables are actually partial results used within a given iteration but not carried over to any subsequent iteration. Expression (2.9) and its translation given in Figure 2.8 further clarify these points.

```
1  ( initial i <- 1;  sum <- ∅
2    while i≤n do
3       new i <- i+1;
4       y <- f(i);
5       new sum <- sum+y
6    return sum )                                    (2.9)
```

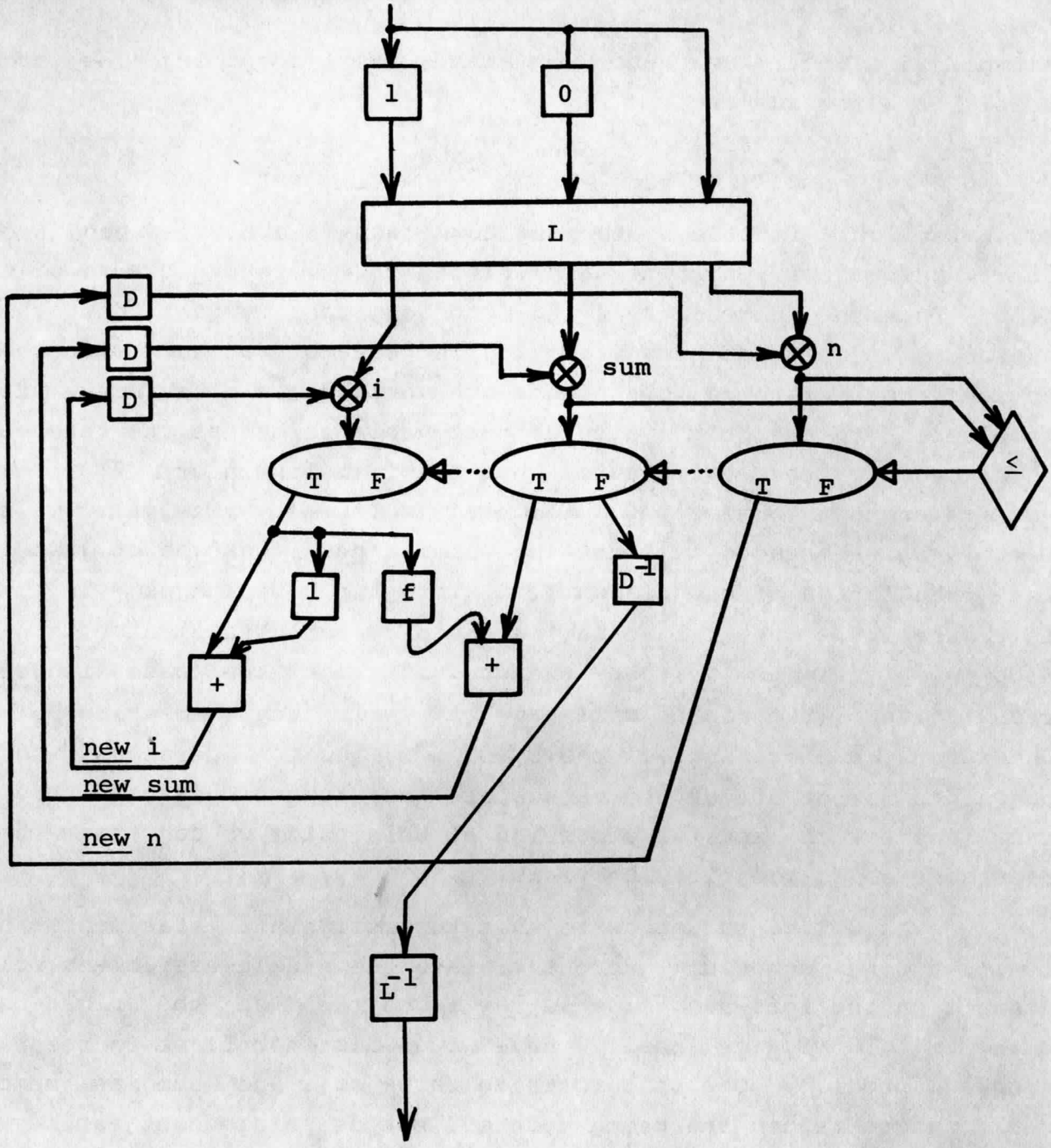

Figure 2.8
Compilation of the loop expression (2.9) where
many instantiations of f may procede simultaneously

Here statements 3 and 5 are recurrence statements (new i and new sum are being computed, both variables having been given initial values), while statement 4 is simply a partial result used in statement 5 (variable y was not given an initial value).

Now let us briefly consider the execution of (2.9). Suppose function f of line 4 takes a long time to execute. The loop predicate i≤n, however, does not depend upon the evaluation of f(i). Therefore it is possible for several tokens to accumulate on the line i going into the function box f since these tokens are the values of i counting from 1 to n, 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 of time would be meaningless. We will show in Section 3 that the machine's interpretation of the base language is such that instead of accumulating tokens on the line leading from output i to box f, 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, but the semantics of all loops (except for those involving streams*) are encompassed within the general while-loop construct given in expression (2.10).

```
( initial x <- f(a)
  while p(x,c) do
        y <- g(x,c);
        new x <- h(x,y,c)
  return r(x,c) )                                    (2.10)
```

In an actual loop expression there might be more than one variable of type a, x, y, or c. Variables that are assigned in the initial part (x variables) circulate in the loop and thus have both old and new values. Variables that are not assigned in the initial part but are assigned in the loop body (y variables) are simply partial values 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

---

*Streams are discussed in Section 4.

right-hand side of assignments in the initial part (a variables) are treated as inputs to the loop expression and must originate from outside the loop expression. Hence, x <- f(x) appearing in the initial part of a loop expression would be a valid assignment statement. The x on the right-hand side would be connected to the output named x outside the loop expression, while x on the left-hand side will be the name of an output inside the loop expression. To explain further we give two equivalent loop expressions. The loop (2.11) is a for-while-loop which is implemented as if it were the while-loop (2.12).

```
( x <- g(a);
  y <- a*(initial x <- f(x)
          for i from 1 to n while p(x)do
              y <- g(i);
              new x <- f(x)+y
          return x)
  return y )                                        (2.11)

( x' <- g(a);
  y' <- a*(initial x <- f(x'); i <- 1
           while (i<n) and p(x) do
                y <- g(i);
                new x <- f(x)+y;
                new i <- i+1
           return x)
   return y' )                                      (2.12)
```

Now we describe a very useful default for conditional assignment statements. Consider expression (2.13).

```
( initial x <- x; sum <- ∅
  for i from 1 to n while p(x) do
           new x <- f(x);
           (if g(x) then new sum <- sum+1)
  return sum )                                      (2.13)
```

Normally a variable must be assigned both in the then clause and the else clause. But instead of treating the conditional assignment in expression (2.13) as illegal, we translate it as if it were the following statement:

```
(if g(x)  then new sum <- sum+1  else new sum <- sum)
                        or
new sum <- (if g(x) then sum+1 else sum)
```

This is a default rule that applies only within a loop body.

2.4 Procedure applications: Figure 2.1 shows the Id sqrt function

implemented by the machine primitive SQRT. If sqrt were actually 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 would then, presumably, be connected to a box that outputs a procedure definition value that describes a square root function. We will elaborate on procedure definitions in Sections 2.5.2 and procedure applications in Section 3.

2.5 Data structures and procedure definitions: Variables in Id are not typed; only values are typed. Thus a variable may be assigned a value of any type at any time. The programmer may, however, write an expression to test the type of a value, coerce the value to a different type, or assert that a value must be of a particular type. We do not wish to impose a strongly-typed language on the programmer where he is required to state the type of value every variable is to acquire. In our mind this can cause a great deal of overhead for the programmer. Instead, we feel the programmer can be given sufficient tools to allow him to specify in as much detail as he wishes the information that a strongly-typed language would otherwise require; he need not, however, specify typing in those situations which are burdensome or in which it is unnecessary to state typing information.

There are nine different primitive types of Id values: integers, reals, booleans, strings, structures, procedure definitions, dataflow monitor definitions, dataflow monitor objects, and errors. The first four need no discussion while monitor definitions and monitor objects will be discussed in Section 5. Error values are not discussed at all. The following subsections concentrate on structure values and procedure definition values. A full discussion of type coercions, assertions, and programmer-defined types is deferred to Section 6.

2.5.1 Structure values: A structure value is either the distinguished empty structure Λ or a set of <selector:value> ordered

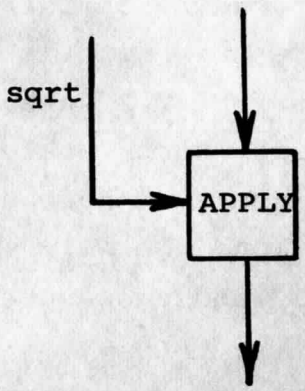

Figure 2.9
Applying a procedure

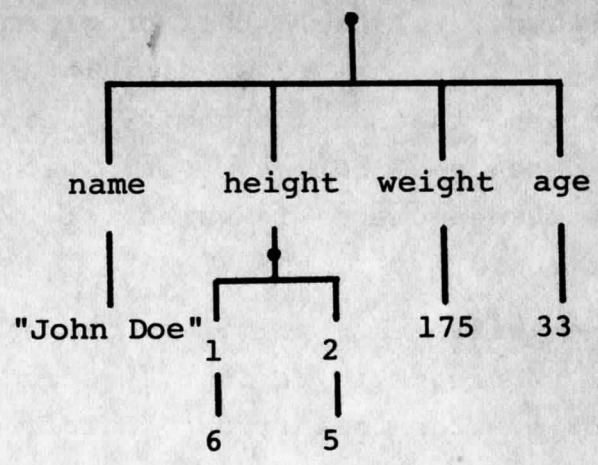

Figure 2.10a
A structure value t with string and integer selectors

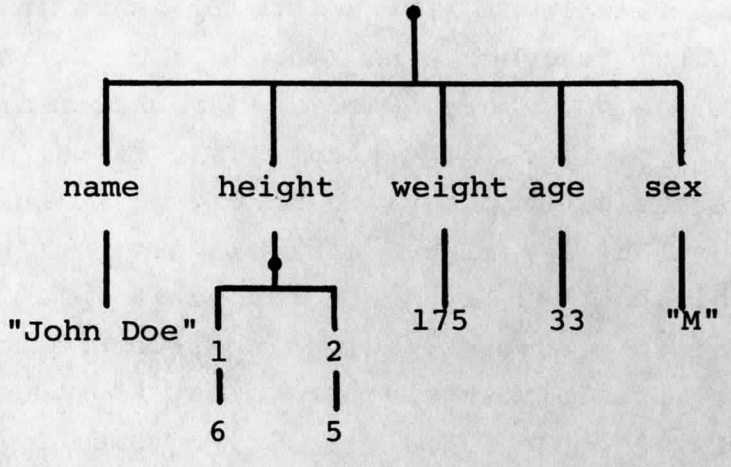

Figure 2.10b
The structure t+["sex"]"M"

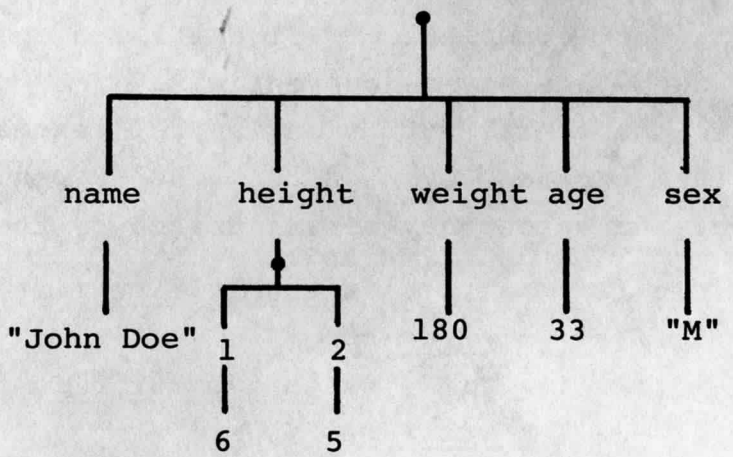

Figure 2.10.c
The structure (t+["sex"]"M")+["weight"]180

pairs. A selector is an integer, string, or boolean value; a value is any Id value. An example of a structure is shown in Figure 2.10a 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 2.10a, then values can be selected from t by writing, for example, t["weight"] and t["height"][1] (giving 175 and 6, respectively). The APPEND operator is somewhat more complex. Given a structure, a value, and a selector to be associated with that value, APPEND creates a new structure value. For example, Figures 2.10b and 2.10c are the results of the indicated appends on the structure t of Figure 2.10a (the "+" symbol means APPEND when the right-context is the symbol "["; the "-" symbol means to use an APPEND operator to remove a <selector:value> pair from the original structure). Note there are two distinct appends in Figure 2.10c. These will be done in left-to-right order with an intermediate value as indicated by the parentheses. Most importantly, the structures created as a result of an APPEND are neither the original structure t nor any modified version of t, since dataflow values cannot be modified. Rather each append creates a new and logically distinct structure, and the old structure (t in the examples of Figure 2.10b and 2.10c) has an existence of its own, possibly concurrent with the new structure. 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 equations

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

Some syntactic shorthands are available in Id for manipulating structure values. In the case of string selectors, the notation x.weight can be used instead of the more cumbersome x["weight"]. Also, to simplify the construction of structures, the angle-bracket notation

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

can be used instead of

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

In the above, the notation <6,5> means <1:6,2:5> where the values 1 and 2 are selector specifications. To explain in more detail, an integer-valued counter is associated with each angle-bracketed structure specification during compilation. The specification is scanned left-to-right and whenever a value is encountered with no associated selector specification, the value in the counter is used, and then the counter is incremented by one. Whenever an integer constant selector is specified, the counter is reset to that value plus one. No other selector specification affects the counter, which is initialized with value 1. Figure 2.11 illustrates the above with several equivalents.

We often find statements in the body of a loop that create a new structure by appending to an old structure with a change in just one component, as for example

```
new x <- x+[i]v
```

A shorthand equivalent is

```
new x[i] <- v
```

Id also allows multiple selectors in order to make references to trees and arrays more convenient, for example, the reference x[1,2] means (x[1])[2].

Notice that the programmer sees each variable that is assigned a structure value as holding that entire structure. This is reflected in the base language where a token that carries a structure logically carries the whole structure as its value. In an actual machine, this quickly becomes impractical when 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 physically carried by the tokens. That is, the underlying implementation of structure values in dataflow may use pointers, shared common substructures, reference count garbage collection, and many other techniques in order to reduce overhead [Dennis74, IPL, etc.]. Nevertheless, these aspects are completely transparent at the level of Id, and even at the level of the base

shorthand notation	definition
<v,7,w>	Λ +[1]v+[2]7+[3]w
<0:v,w,5:x,y>	Λ +[0]v+[1]w+[5]x+[6]y
<-5:v,str:w,x>	Λ +[-5]v+["str"]w+[-4]x

Figure 2.11
Shorthand structure specifications and their equivalents

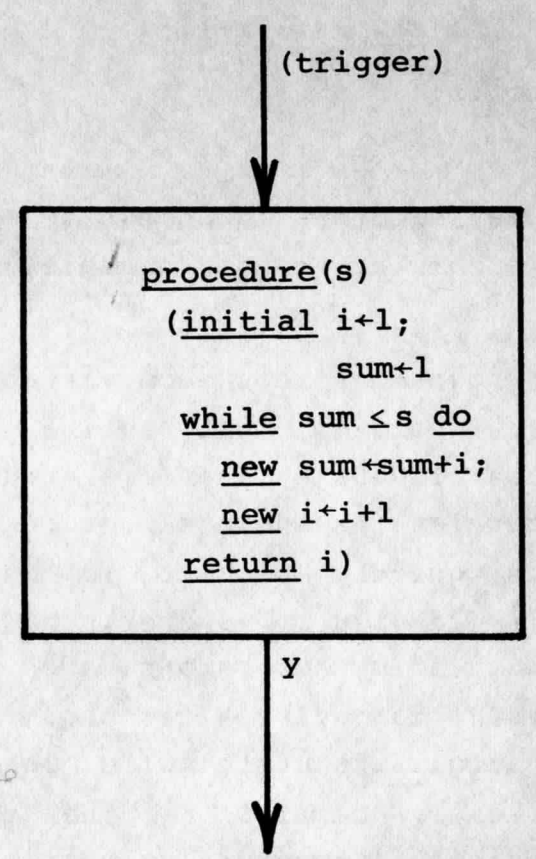

Figure 2.12
Compilation of statement (2.14)

dataflow language where any structure-carrying token still behaves as if it carried the entire structure value*. We do not discuss detailed memory mechanisms 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 need to be carried by a token.

2.5.2 Procedure definitions: An example of a procedure definition value being assigned to a variable y is

```
y <- procedure (s) (initial i <- 1; sum <- 1
                    while sum<s do
                        new sum <- sum+i;
                        new i <- i+1
                    return i)                    (2.14)
```

The specification of a procedure definition value, exemplified by (2.14), is much like a constant specification. That is, just as a numeric constant α in Id actually implies a constant function that produces α as its value, so does a procedure definition imply a function that produces that procedure as its value.

Statement (2.14) translates into Figure 2.12. Since Id variables are not typed, variable y in (2.14) is not a procedure variable, but does receive a procedure value whenever (2.14) is executed. Variable y is like any other variable and can be passed as an argument to a procedure, appended to a structure, or operated upon by any operator that is defined on the type of value carried by y. In Id, two operators are defined over procedure values: APPLY and COMPOSE. We recall from Section 2.4 that the APPLY operator applies a procedure definition to a list of actual arguments by writing

```
apply (y,x1,x2,...,xn)
```

or simply

```
y(x1,x2,...,xn)
```

Since a variable may assume any one of the several values during an execution, it cannot be guaranteed at compile-time that the value of

---

*We differ from Dennis [Dennis73] on this point, since in his language a distinction can be made between a pointer and an elementary value.

y will always be a procedure requiring exactly n input parameters. Suppose the value of y is a procedure with m formal parameters and, as above, the apply lists n actual arguments. If m<n then the first m arguments from the actual parameter list are passed to the procedure while the remaining n-m actual parameters are ignored. On the other hand, if m>n then m-n actual arguments with the special value ξ are supplied in place of the missing actual arguments. If the programmer wishes to omit an argument, its place can simply be left empty. For example, y(x1,,x3) is translated as y(x1,ξ,x3). Corresponding situations can arise where the number of outputs expected from an apply do not match the number received from the applied procedure. The compiler must be able to deduce the number of outputs to be produced by the apply; where necessary, the programmer must make the deduction unambiguous by proper parenthesization on the left-hand side of an assignment statement. For example,

$$(x,y),z \leftarrow f(a),g(b) \qquad (2.15)$$

means that f(a) must produce two outputs and g(b) only one. If the procedure value from f produces more than one output, the extraneous outputs are lost. If f produces fewer then two outputs, the value ξ is produced by apply. Section 3 discusses the implementation of APPLY in detail.

It is also possible to give a name to an Id procedure. This can be useful in writing recursive programs. The named procedure f which calculates the ubiquitous factorial function may be written

$$y \leftarrow \underline{\text{procedure}}\ f(n)\ (\underline{\text{if}}\ n=0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ n*f(n-1)) \qquad (2.16)$$

Essentially, whenever a named procedure is applied its definition is also passed as if it were a parameter, thereby making a named procedure able to reference itself. Recursive programs can be written without named procedures, but it can be a little less convenient. This is shown in (2.17) where z(3,z) is 3.

$$z \leftarrow \underline{\text{procedure}}\ (n,f)(\underline{\text{if}}\ n=0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ n*f(n-1,f)) \qquad (2.17)$$

An unnecessary danger also exists in (2.17), since when z is applied the procedure passed as a parameter need not be z. Hence, z(3,g) is a valid application, but the effect of the application may be different from what one expects. Procedure definitions such as the

one in (2.16) are more convenient than (2.17), as well as more desirable for good programming.

The remaining operator defined on procedure values is COMPOSE. This operator is actually a very simple but very powerful functional in that it accepts a procedure as input and produces a new procedure as output. 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. For example, in

```
q <- procedure (a,b,c)(a^2+b^2+c^2);
r <- compose (q,<<b:5>>)                    (2.18)
```

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

```
r <- procedure (a,c)(a^2+5^2+c^2)
```

since the argument of compose is a list of subarguments either of the form <formal-parameter-position:value> or the alternative form <formal-parameter-name:value>. As another example, we use z from (2.17) and write.

```
w <- compose (z,<<2:z>>)
```

which produces a value w with one formal parameter (the parameter n in (2.17)). The value of each instance of f in (2.17) has thus been frozen to the procedure value z, so w(3)=z(3,z)=3! . In fact, a named procedure is implemented by compose. For example, statement (2.16) 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 y, regardless of how it may be composed further, will allow procedure f to internally refer to the original definition of f in all cases. The translation of the named procedure definition f to its new form is 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
   - 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 name of the resulting procedure is f (i.e., the original name).

Named procedures are important and are used extensively in implementing programmer-defined types and in the environment feature of Id. (These application points are covered later in Section 6.) Lastly, we note that the above algorithm is the straight-forward way of handling named procedure definitions. However short-cuts are sometimes possible, for example, rather than do a compose and then immediately do an apply, the compose could be deleted and instead the procedure passed directly to itself as a parameter in the apply. Such short-cuts must be very carefully considered, and in general can be done only by a complete analysis of all possible paths through the named procedure and with the guarantee that the procedure will never be returned to an outer context and applied by another program that knows nothing about it. For example, the result of such an apply could be for procedure f to return f itself, which must be the f' version of f, but which may not be the case if it is not composed before the apply.

Only COMPOSE and APPLY are allowed to manipulate the internal representation of procedure values. To describe exactly how the COMPOSE operator works, we consider the encoding of a procedure value to be a special kind of structure which the Id programmer can neither select values from nor append values to. (A programmer can, of course, always append an entire procedure value to and select a procedure value from another structure.) In this way we can guarantee the consistency of the internals of a procedure value. The special structures that encode the procedure values at outputs q and r of (2.18) are shown in Figures 2.13a and 2.13b, respectively. In these structures, the detailed encoding of the procedure body itself is of no consequence to this discussion and is left unspecified. Concerning the other components, the "name" records the name of the procedure (if any) as a string. The "#" component specifies the number of parameters. The "formals" specifies for each parameter position, the name of the parameter as a string, while "actuals" specifies, for each parameter frozen, its actual value. We note here that the COMPOSE operator rearranges nothing but the "formals" and "actuals" components. The procedure values in Figure 2.13a and 2.13b contain enough information for the APPLY

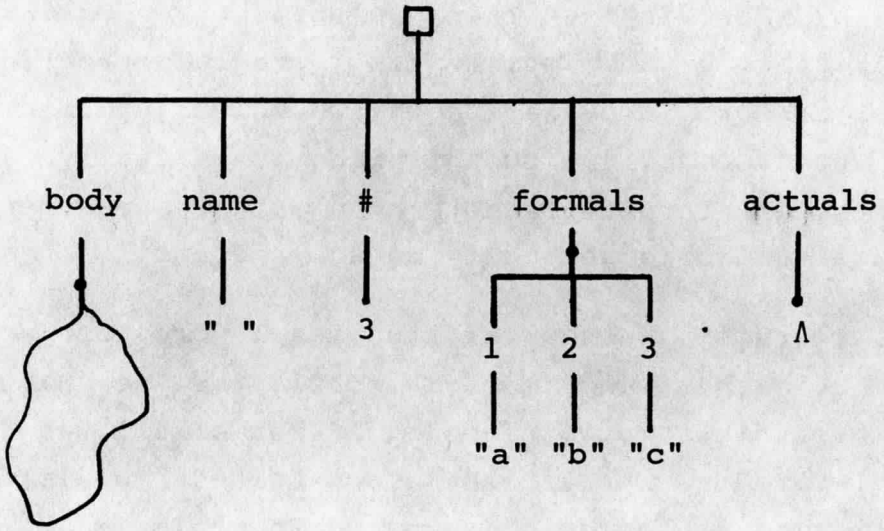

Figure 2.13a

The encoding of the procedure value on line q in (2.18)

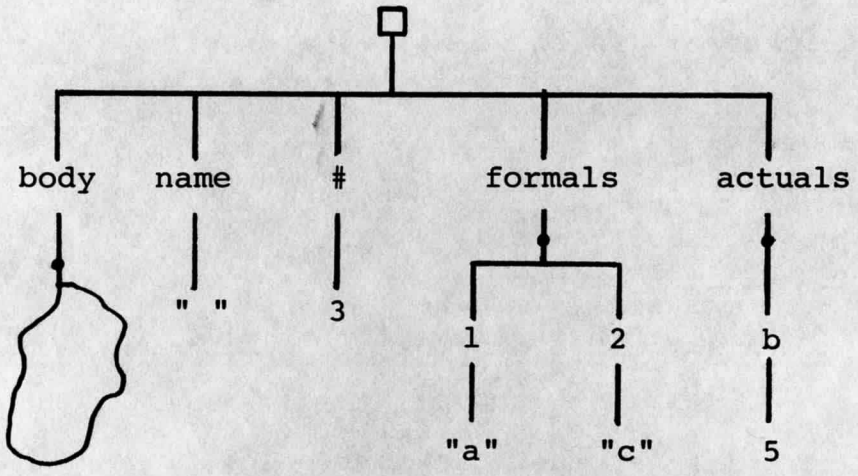

Figure 2.13b

The result of the compose function in (2.18)

operator to determine the values of all actual parameters.

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 Two sample programs: We now give two sample programs written in Id: Hoare's quicksort and matrix multiply. We have chosen conventional algorithms for two reasons. First of all, we want to show that a programmer proficient in ALGOL-60 would have no difficulty in writing programs in Id, even though Id is a dataflow language and has a different semantic base. Second, we want to show that even contentional algorithms automatically exhibit a great deal of concurrency when expressed in Id. A complete discussion of the second point must wait until Section 3 because it is related to the unravelling interpreter of the base language.

2.6.1 Hoare's quicksort: In Id we write Quicksort as

```
procedure Qucksort (a,n)
(below,j,above <-
      (initial below <- Λ; j <- 0;
               above <- Λ; k <- 0;
               middle <- a[1]
       for i from 2 to n do
             (if a[i] < middle
              then new below[j+1] <- a[i];
                   new j <- j+1
              else new above [k+1] <- a[i];
                   new k <- k+1)
       return (if j>1 then Qucksort(below,j)
                      else below),j,
                           (if k>1 then Quicksort(above,k)
                                   else above)))
    return (initial t <- below+[j+1]middle
        for i from 1 to n-j-1 do
            t <- t+[i]above[i+j+1]
        return t))                                        (2.19)
```

The time complexity for single-processor Quicksort has an average of O(n log n) and a worst case behavior of $O(n^2)$. The above Id counterpart, when compiled into the base language and executed under the unravelling 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 discussed in Section 3.

2.6.2 Matrix mutliply: The following procedure multiplies an ℓ x m matrix a by an m x n matrix in the straight-forward way.

```
procedure multiply (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)                                          (2.20)
```

The above program 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 unravelling interpreter will try to execute all of the multiplications and n of the additions in parallel, thus reducing the usual time complexity of $O(\ell mn)$ to $O(\ell+m+n)$.

The reader should not conclude from these examples that Id is just another language for writing programs. We are most interested in expressing algorithms in a way that preserves the structure of problem solutions. For many types of problems sequential languages have worked well, and we would like to keep as many of these features of sequential languages as is possible provided they are not in conflict with the semantic basis of dataflow. However, in Section 4 we will give examples of programs that one is not likely to write if restricted to von Neumann semantics. It is a well-accepted fact that language influences our thinking. We hope that thinking in terms of dataflow will remove some unnecessary constraints placed on us by languages with sequential control structures.

3. The Base Language and the Unravelling Interpreter

The unravelling interpreter (described herein) has been designed to exploit even greater asynchrony then what is normally realized in a dataflow language [Dennis73, Chamberlin71, Ashcroft & Waage76]. To see how the interpreter operates, we must examine token flow in some detail. Imagine the operator F of Figure 3.1a in the body of a loop at a time when three complete sets of values from outputs a and b are ready for processing, the results of which are to be placed at output c. According to the first principle of dataflow, since both input values x_1 and y_1 are present (from outputs a and b, respectively), the first initiation of the operator can take place and we can move to the configuration of Figure 3.1b. Immediately the second initiation of the operator can occur on the input values x_2 and y_2. However, the implication that the second initiation can take place only after the result due to the first initiation has been produced is unnecessary, since the second principle of dataflow states that the actions of each operator must be free of side-effects. To take full advantage of the freedom from side-effects, let each distinct initiation of an operator be termed an activity. Then we note that if sufficient free processors were available, and if each activity were associated with one processor, then in the case of Figure 3.1 all three activities (initiations of operator s) could be carried out concurrently. The purpose of the unravelling interpreter is to execute programs by generating activities (in fact, large numbers of activities) for execution by waiting processors. Of course, the main problem is to keep the different sets of tokens from being mixed, since for correct operation it is essential that the token carrying x_i be matched only with the token carrying y_i. The unravelling interpreter accomplishes this by appropriately tagging every token that is produced with a destination activity name. The unravelling interpreter is largely a set of rules for manipulating these activity names.

3.1 Activity names: An activity is a single execution of a base language operator. Each activity is assigned a unique activity name, and all tokens carry the name of the activity for which they are destined. The rules for generating activity names are based upon the following principles:

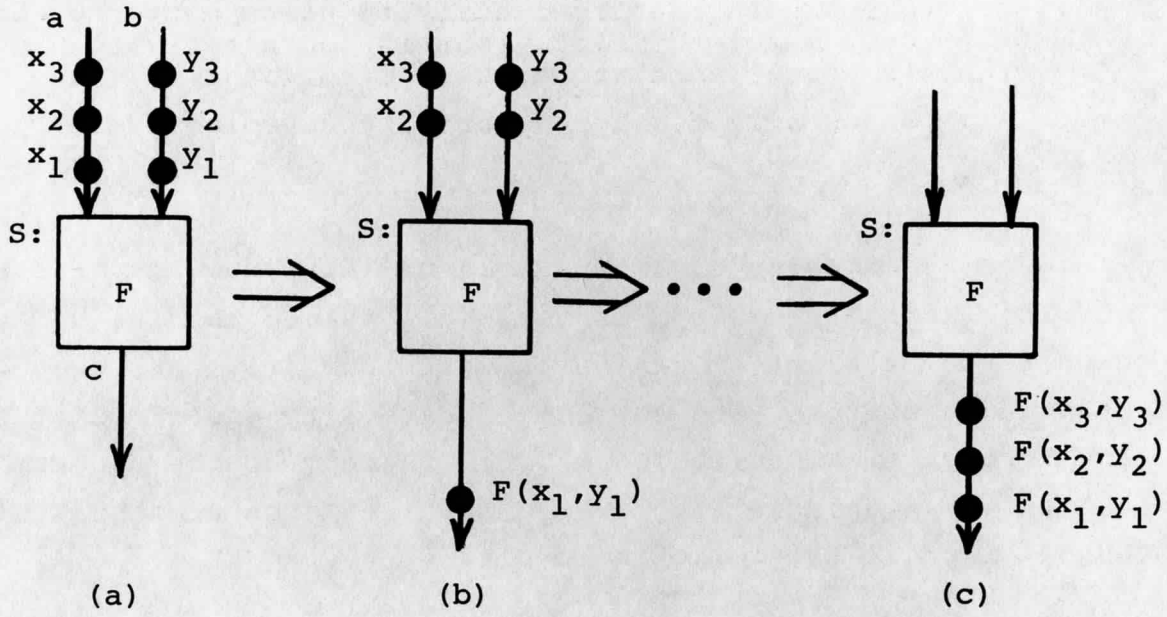

Figure 3.1
Three executions of a dataflow operator

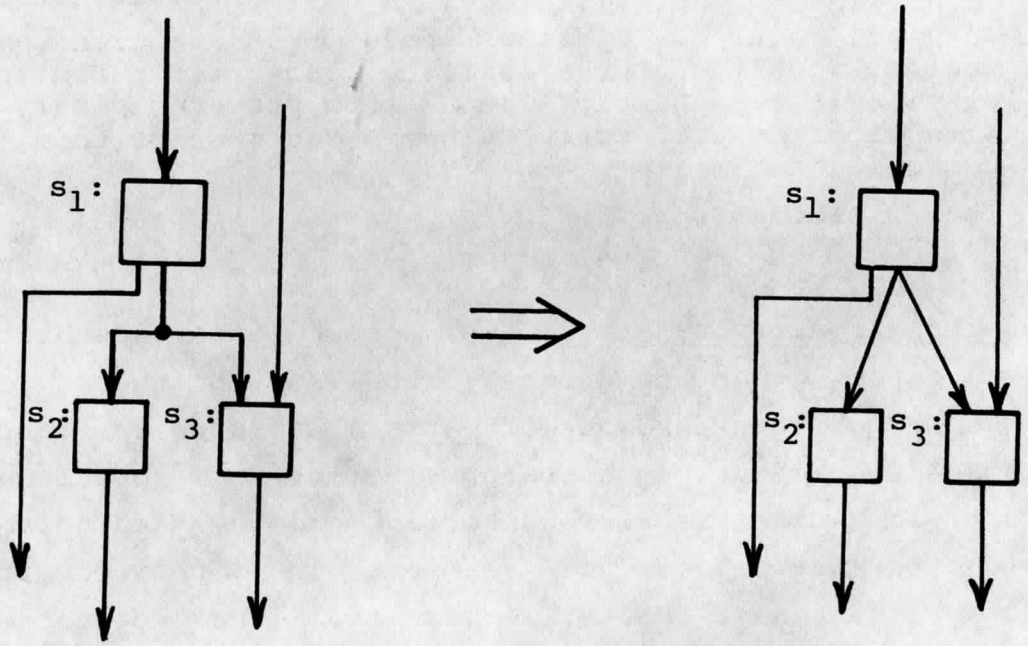

Figure 3.2
Elimination of forks

1. All tokens with identical activity names must be destined for the same activity, that is, each activity has a name that is unique throughout the system, and
2. All tokens accepted by an activity have identical activity names.

A machine composed of an ensemble of large numbers of identical processing elements (PEs) is very well suited for the unravelling interpreter. Consider a machine in which each PE can execute any base language operator and can communicate with other PEs by sending tokens through a communication medium. A copy of the program to be executed is available to every PE, and each PE repeatedly goes through basically the following cycle:

1. If a PE is free it looks for an allocation token* in the communication medium. Whenever it finds such a token, the PE acquires the activity name written on that token and thereby becomes that activity. By decoding the activity name, the PE discovers which operation in the program it is supposed to execute and how many operands are needed before it can initiate.
2. After acquiring an activity name, the PE waits for the other operands to arrive from the communication medium.
3. When all required operands have been received, the PE begins execution of the activity, and after a finite period of time it terminates. The PE then produces result tokens with appropriate activity names and inserts them into the communication medium.
4. The PE becomes free.

3.2 Generation of Activity Names: We assume each operator in a dataflow program graph is uniquely labelled, and that it has some specific number of input and output ports. As shown in Figure 3.2, a fork connection is not an operator. By moving a fork back to the output port to which it is connected and by assigning token replication to that port, we can use the operator itself to implement a fork. (Note that still no two lines converge on a single input port.) Thus, the structure of a program graph is such that each operator contains a destination list for each output port, and where that destination list contains the names of all the input

---

*Exactly one input line to each operator can be distinguished in that each token flowing along that line can be marked as an allocation token.

ports which are to receive identical output tokens. In the remainder of this section we show how activities are created and named, and how the structural integrity of the program graph is maintained in the midst of a system based on activity name manipulation.

A token is output from an operator and moves to the input of a successor operator. Each token carries both its data and its destination activity name which we write as the ordered pair

<data, destination>

where destination comprises five fields: context u, procedure code c, operator label s, initiation i, and port p. Logical groupings of the values in various fields help to identify some facets of the destination of a token. Thus fields c, s, and p specify that the token under consideration is passing along the line of the program graph connected to port p of operator s in procedure c. Any such specification must, of course, be consistent with the static structure of the program graph. The remaining fields u and i of the token destination give the context u (for example, the procedure application context) and initiation count i consistent with the dynamics of program execution. These latter fields u and i are particularly important in the case of the two operators A and L associated with procedure applications and loop expressions, respectively. Operator A dynamically creates a new program graph, while operator L copies only a portion of the executing program graph, i.e., a loop. Under such circumstances it becomes necessary to attach a unique context name to the operator labels to distinguish between the various initiations of two operators with the same label, and in general an operator is uniquely identified by fields u, c, and s instead of fields c and s only. Considering another grouping of the fields in a token destination, we may identify a logical line by u, c, s, and p, rather than fields c, s, and p only. The set of tokens associated with a logical line contains all those tokens with fields u, c, s, and p common in their destinations. These tokens are called the history of that logical line.

An activity of an operator is identified by the fields u, c, s, and i and is written as the symbol "u.c.s.i". The symbol is called

an activity name. The input tokens associated with an activity are called the input token set and are easily identified as all those tokens with fields u, c, s, and i common in their destination. The process of generating activities is described by specifying how an input token set is transformed to a set of output tokens, each token intended for some successor activity. Even though lines in the program graph have no actual physical significance, the activity name generation process for each operator (specified below) makes it clear that the structural integrity of a graph is never violated, i.e., the fields c, s, and p of a destination are easily verified correct. In general, even when logical lines are dynamically created (the A and L operators) and dynamically destroyed (the A^{-1} and L^{-1} operators) no line of the program graph is added or deleted. It is somewhat more complex to verify that the history of a line is valid, that is, that no two tokens in the history of a line have the same initiation count value i*. Brief arguments are given to show that all operators produce valid histories, provided these operators are interconnected in a proper way, i.e., according to a syntactically correct ID construct. Any other interconnection of base language operators is undefined.

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

(a->b; c->d; ...; e->f; g)

means that if a holds then token b is the result; otherwise, if c holds then token d is the result; etc., finally, if no condition holds then token g is the result. If no result token is specified (ϕ), then none is produced. Also, the following semantic specifications generally assume that u.c.s.i is the name of the activity under consideration, and that the destination of a token is port p of operator t. Because of the emphasis on activity names, we have logically grouped the fields of the token destination part in the form <activity name, port>, or <u.c.s.i, p>. Further, we often refer to a logical line by the form <u.c.s, p>.

---

*This statement does not hold for streams. A valid history of a stream line will be discussed in Section 4.

3.2.1 Block schema (functions and predicates): This category includes SELECT and APPEND as well as all arithmetic and boolean operators. The binary function F typically specifies the operators of this class:

input = {<x,<u.c.s.i,1>>, <y,<u.c.s.i,2>>}

output = {<F(x,y),<u.c.t.i,p>>}

Instead of using the above formal notation, sometimes we will express the semantics by writing the activity name and an abbreviated description of its input and output as follows:

u.c.s.i -- input: port 1 = x
port 2 = y
output: port 1 = F(x,y)

In case only one port is used, port numbers will be elided.

As we have said, the history of line <u.c.t, p> is valid if no two tokens have the same initiation count. We define a schema to be valid if given valid input histories it produces valid output histories. Since the initiation count is unchanged from input to output for function and predicate operators, and since no two lines converge on the same input port of any operator, it is clear that if a function operator receives valid input histories, then the output will be a valid history. Furthermore, any block expression (any acyclic interconnection of functions and predicates and other valid schemas) is also valid.

The interesting aspect of the activity name mechanism is that two initiations of a function box need not initiate or terminate in any particular order. Returning to the example of Figure 3.1, the result token corresponding to the input values x_2 and y_2 can be produced before the computation with tokens x_1 and Y_1 ever begins. This obviously increases asynchrony over what it might otherwise be.

3.2.2 Conditional schema: The operator needed to implement conditional schemas is the SWITCH operator. This operator copies the single input datum onto one of the two output lines depending upon the value of the boolean input. A SWITCH does not affect the initiation count field of the tokens passing through it.

Assume in Figure 3.3 that the labels of the SWITCH, f, and g operators are s, t_T, and t_F, respectively. Then the SWITCH can be described as

u.c.s.1 -- input: data-port = x
control-port = b
output: T-port = (b=true -> x; ϕ)

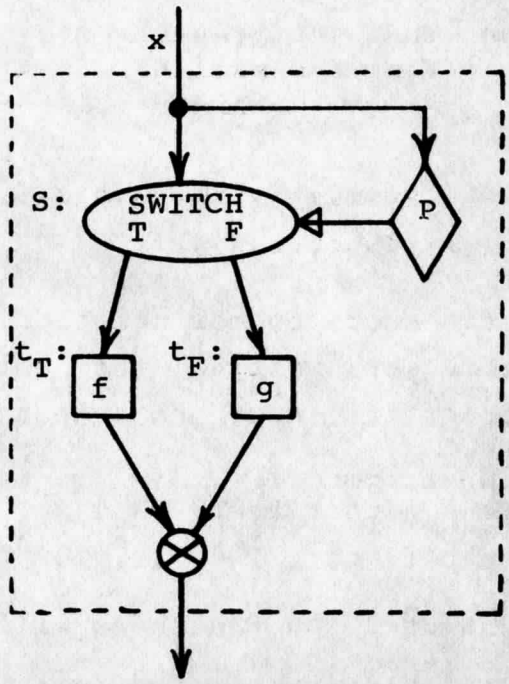

Figure 3.3
A conditional schema

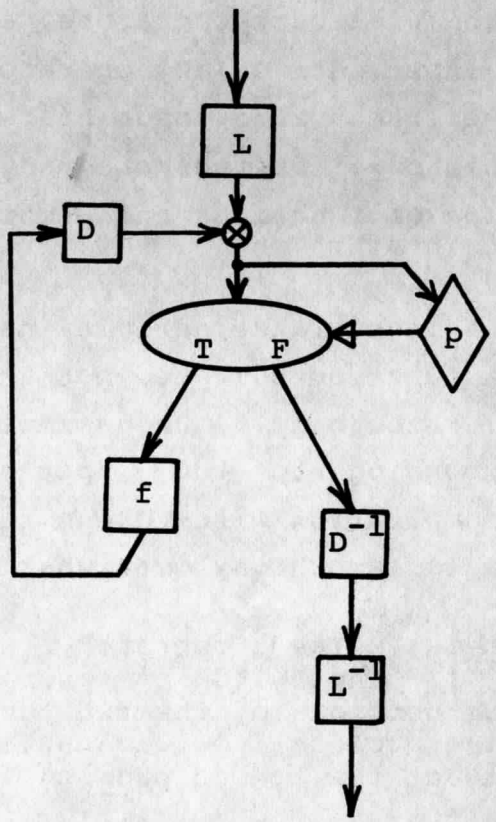

Figure 3.4
A simplified loop schema
corresponding to expression (3.1)

F-port = (b=false -> x; ϕ)

The $\otimes$ operator does not affect the initiation count field i, the context field u, or the data part of a token. In fact it does nothing except change the value of the s and p fields according to the (static) program graph connections. (Therefore the $\otimes$ operator is implemented simply by merging it with the operator following it. This operator is needed only to guarantee that lines in a program graph are not connected together in an arbitrary manner.)

Following the example in Figure 3.3, if the history of output x is valid then so will be the history of the lines going into boxes f and g. However, if a token with initiation count i exists on the line going into f, then no token with initiation count i exists on the line going into g. Hence the initiation count of tokens going into f and g are mutually exclusive. Now let us assume that f and g are valid schemas in the base language. Then, like functions and predicates discussed earlier, f and g each merely copies the initiation count from the tokens on its input lines to the tokens on its output lines. Due to the disjointedness of the initiation counts of the tokens on the lines coming out of f and g, only a proper history will result after merging the two lines via $\otimes$. Hence, if the histories of the input lines to a conditional schema are valid, so will be the history of the output. Note that the entire dotted box in Figure 3.3 behaves just like an ordinary function box.

3.2.3 Loop schema: A simplified loop schema is shown in Figure 3.4 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. All these operators are control operators and they do not affect the data portion of the tokens passing through them. Each does, however, affect the activity name of the tokens passing through them.

(a) The D operator: This operator is used if and only if there is a cycle in the base language program graph. A token going through this operator is an indication of the fact that the next iteration of the loop is underway. The D box simply increments the initiation

count of the token passing through it.

u.c.s.1 -- input = x
output = <x,<u.c.t.i+1,p>>

To explain the operation of a loop, assume the L box in Figure 3.4 produces exactly one token and that the initiation count of that token is 1 (we will show shortly that this is indeed the case). If the predicate p produces a true valued output, then the data token from L will take the T branch output from the SWITCH. If f is a valid schema then the output line of f (and the input line to D) will receive a token with an initiation count of 1. The D box then increments the initiation count of the token by one and sends it to the SWITCH and to the predicate p for the next iteration. As long as only that single first token is output by the L box, the history of each line going into the SWITCH and the predicate will remain valid. Since D always increments the initiation count by one, it can never duplicate an activity name. The operator $\otimes$ also preserves histories since the only token it receives from L has an initiation count of 1, and the tokens it receives from D all have initiation counts of two or more. If the loop executes n times and the preciate produces a false value on its $n+1^{th}$ execution, then the token coming out the F branch of the SWITCH will have initiation count n+1. Since this branch terminates the loop, the input to the D^{-1} operator can never receive more than one token. It is interesting to note that tokens need not go around a loop in any particular order unless constrained by the need for partial results. This situation was illustrated earlier by the program in Figure 2.8 where it is possible for the jth, the j+1st, and in fact all executions of box f to go on concurrently. Even if the j+1st execution of f terminates before the jth execution, no confusion or mismatch of activity names can result. The unravelling or unfolding of a loop is a very powerful feature of the unravelling interpreter. Automatic unravelling, constrained only by those data dependencies that are actually necessary, greatly increases the asynchrony of programs (for example, f could be another nested loop), many of which would otherwise be considered completely sequential.

(b) The D^{-1} operator: This operator is the inverse of the D operator, and serves to return the initiation count of the token output along with F branch of the SWITCH back to the value 1. The first token that began the loop as output from the L operator had an

initiation count of 1, the D operator incremented it for each iteration of the loop, and now the D^{-1} operator ensures that the output of the loop also has an initiation count of 1. The D^{-1} operator is

u.c.s.i -- input = x
output = <x,<u.c.t.1,p>>

Before describing the L and L^{-1} operators, we would like to point out that it is possible to define a base language without them (for example, the languages in [Dennis73] and [Kosinski73] have nothing corresponding to these operators). However, these operators were introduced after we realized that the semantics of ID loops could permit even greater asynchrony of execution. ID loops are pure functions, that is, they have no memory of previous invocations, or equivalently, an execution of a loop expression can receive information only from tokens explicitly input to it. Thus in the case of nested loops it is quite possible that the input tokens for several instantiations of the inner loop may be available at the same time. It is the L and L^{-1} operators (in conjunction with the D and D^{-1} operators) which capitalize on this fact by creating a new logical loop schema for each instantiation of that loop, and by destroying that logical schema on loop termination.

(c) The L (loop begin) operator: The L operator accomplishes the creation of a new logical loop by changing the context part of the activity name. Following is a description of an L operator with two inputs (see Figure 3.5a):

u.c.s.i -- input: port 1 = x
port 2 = y
output: port 1 = <x,<u'.c.t_1.1,p_1>>
port 2 = <y,<u'.c.t_2.1,p_2>>}
where u' = (u.s.i)

By changing the context u to u' = (u.s.i), L creates a completely new set of logical lines and operators in the loop schema for each input token set it receives. Equivalently, we may note that L puts one token with initiation count 1 on each new set of logical output lines. It is also clear that each token output by L has a unique destination name because, based on that name, we can deduce the input token which was itself assumed to be unique. It should be clear that the requirement placed by the D operator on the L box, namely that the L box produces only one token on each of its

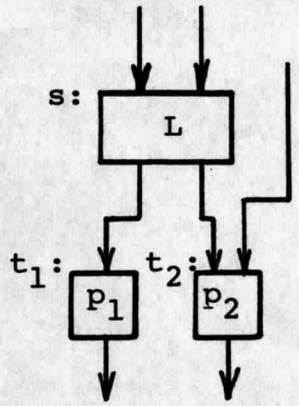

Figure 3.5a
An L operator with two inputs

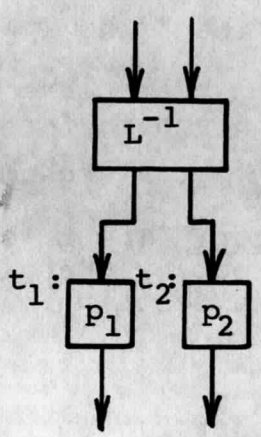

Figure 3.5b
An L^{-1} operator with two inputs

output lines, is satisfied by the definition of L. (Note that the L operator does not have to wait for all its input tokens to arrive. It may produce an output token as soon as it receives an input token.)

Since all input lines to the loop pass through the L box, all tokens involved in the i^{th} instantiation of a loop will have (u.s.i).c common to their destination activity names. This information is extremely useful in localizing the tokens belonging to a particular instantiation of a loop.

(d) The L^{-1} (loop end) operator: This operator is the inverse of the L operator. It expects only one token on each input line and changes its activity name back to the environment to which the output token belongs. The following description relates to Figure 3.5b:

u'.c.r.l where u' = (u.s.i)

input: port 1 = x
port 2 = y

output: port 1 = <x,<u.c.t_1.i,p_1>>
port 2 = <y,<u.c.t_2.i,p_2>>

Several facts about the above activity names are worth pointing out. First, only an initiation count value of 1 is valid on any input to L^{-1}. Second, statement s in u.s.i must refer to the L operator mate of the L^{-1} under discussion. Third, a close examination of the activity names generated for a given output logical line, say <u.c.t_1,p_1>, reveals that several initiations of operator L^{-1}, each under a different context, contribute tokens to that logical line. In a sense the L^{-1} operator collapses many input logical lines into new output logical line. For example, input logical lines <(u.s.1).c.r,j>, <(u.s.2)c.r,j>, ...<(u.s.i).c.r,j>..., all collapse to form the single output logical line <u.c.t_j,p_j>. Since each input logical line <(u.s.i).c.r,j> contributes exactly one token with an initiation count value of 1 to the output logical line <u.c.t_j,p_j>, the history of that output logical line is valid.

The reader may be concerned about activity names becoming arbitrarly large because the context field is recursive. Even though this is logically true, names can physically be kept within bounds by proper encoding of the information. As an example of such encoding, consider the possibility of an L operator sending the

context u' = u.s.i on a special "dummy" token directly to its mate L^{-1} operator. Then it is easy to see that u' can essentially be equal in size to u. With the help of the dummy token the L^{-1} operator will be able to generate the proper output.

Again, note that the dashed box in Figure 3.4 is a valid schema (the i^{th} set of input values produces the i^{th} set of output values). Each instantiation of a loop and the corresponding new context u' is called a loop domain, and all activities within a loop domain can proceed independent of activities outside that loop domain, including loop domains at the same and at other arbitrary nesting levels.

3.2.4 Procedure application: Procedure application is specified by the APPLY operator. Let us assume that the procedure definition value Q (recall the definition of procedure values from Section 2.5.2) is being applied to arguments x and y for the i^{th} initiation of the APPLY operator. The APPLY operator must create a new logical schema for each instance of execution of APPLY, and in this case the schema to be created is Q. We do this by breaking APPLY into two internal operators called A (activate) and A^{-1} (terminate); furthermore, every procedure must have a BEGIN and an END operator. All these operators are related as shown in Figure 3.6. The purpose of the A operator is to create, for each initiation of that operator, a new logical procedure domain similar to the way in which the L operator creates loop domains. Because of this similarity, we say that all tokens with u.c common in their activity names belong to the same logical domain, where that logical domain may be either a loop domain or a procedure domain. The major difference between a loop expression and a procedure application is that the same loop expression is always executed at the same point in a given program, but the particular procedure definition value that arrives on a token at the APPLY may vary, and in general is not known in advance. Equivalently, we can say that a loop is like a nameless procedure that is applied at only one place in the program. In Figure 3.6 the connection between the two logical domains (the calling and the called domain) is with dashed lines since the connection is known only at execution time. Note also that since APPLY is actually carried out by two disjoint operators, it is not possible for an execution of Q to keep an actual APPLY activity in execution

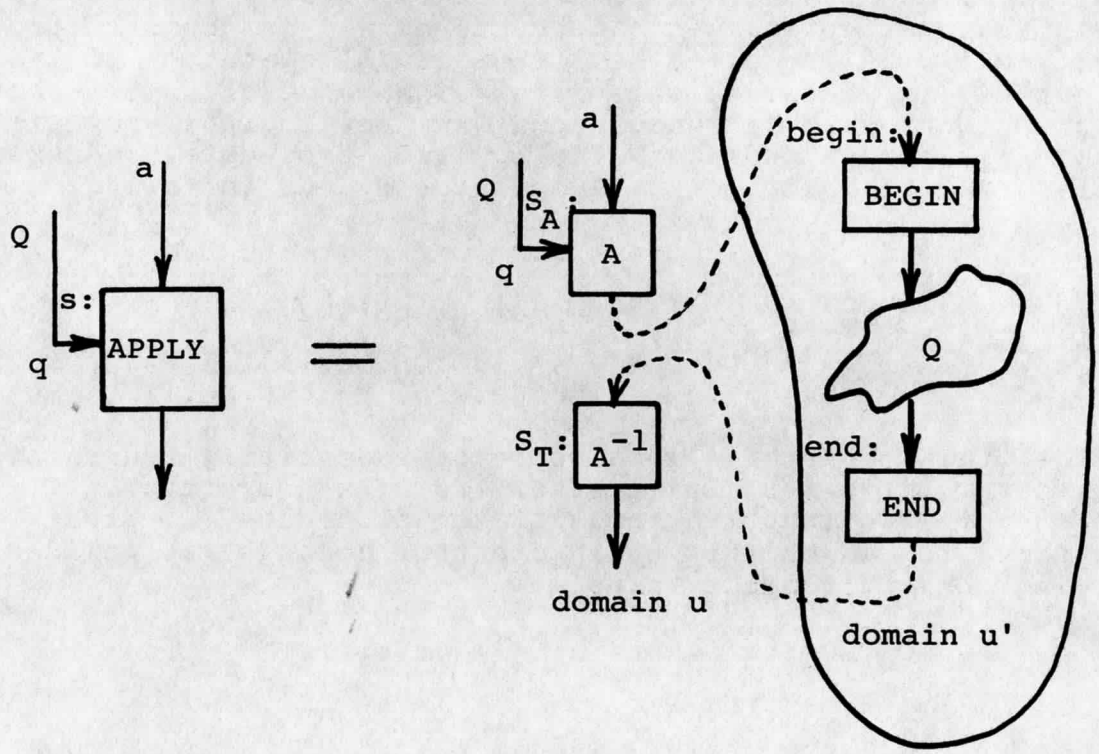

Figure 3.6
The APPLY operator and its execution

indefinitely, for example, if Q never terminates.

Activiy name generation and various other functions of A, A^{-1}, BEGIN, and END are described in the following.

(a) The A (activate) operator: This operator examines the procedure value supplied to determine how many actual parameters are needed for the execution of the procedure. It accomplishes this by matching the elements in the list of formal parameters contained in the procedure value with the actual parameters supplied to the A operator on various input ports. According to the rules explained in Section 2.5.2, a structure α containing the actual parameters input to A, and the actual parameters carried by the procedure value itself, and the environment* structure η, is constructed. This structure α is supplied to the BEGIN operator of the applied procedure. The activity name generation process for the operator A is given below. It shows that a new logical graph is created dynamically corresponding to the applied procedure. Assuming two input ports, we describe the A operator by the following:

$u.c.s_A.i$ -- input: procedure port = Q
argument port 1 = x
argument port 2 = y

output = $\langle \alpha, \langle u'.c_Q.begin.1, 1 \rangle \rangle$
where $u' = (u.c.s_T.i)$

(b) The BEGIN operator: This operator essentially tears apart the parameter list α that it receives from the A operator. It produces a token for each input variable of the procedure Q (that is, the procedure to which this BEGIN operator belongs) as well as a token carrying the environment value η.

$u'.c_Q.begin.1$ where $u' = (u.c.s_T.i)$
input = α
output: port 1 = x
port 2 = y

where x and y are selected from α based on the names of the corresponding formal parameters.

(c) The END operator: This operator first constructs a structure β containing the value received on each input port. It then sends this structure to the A^{-1} operator in the calling domain by changing the context part of the input activity name.

Assuming two inputs, we describe the END operator by the following:

$u'.c_Q.end.1$ where $u' = (u.c.s_T.i)$
input: port 1 = x
port 2 = y
output = $\langle \beta, \langle u.c.s_T.i, 1 \rangle \rangle$
where β is the structure <1:x,2:y>

---

*The environment is discussed in Section 6.

(d) the A^{-1} (terminate) operator: This operator tears the β structure apart and matches its elements with the number of output ports. In the case of extra elements in β, it discards the extra elements in β, and in the case where more outputs are needed, A^{-1} supplies an output value of ξ (see Section 2.5.2).

$u.c.s_T.i$ -- input = β

output: port 1 = x
port 2 = y
where x is either $\beta[1]$ or ξ
and y is either $\beta[2]$ or ξ

It is clear from the definition of END and A^{-1} that if the END operator were to receive two tokens on some input line (i.e., one token with initiation count 1 and another with an initiation count greater than 1) then it could generate two tokens for A^{-1} with the same activity name u.c.s.i and the history of the output lines on A^{-1} will not be valid. However, if Q is a valid ID procedure, Q cannot produce more than one token on each output line. Hence the A, BEGIN, END, and A^{-1} operators, when supplied syntactically correct procedures, will produce only valid histories on all logical lines.

The scheme discussed above for encoding the context part u' on a dummy token sent from an L operator to an L^{-1} operator, is also applicable here between the BEGIN and END operators. Again, this will allow essentially constant space to hold all context names.

3.3 Asynchrony in a sequential algorithm: In this section we analyze the procedure given in Section 2.5 for multiplying two matrices. The procedure of expression (2.20) assumes the matrices are stored by rows, so a[i] gives the structure value representing the i^{th} row of matrix a. While discussing loop schemata in Section 3.2.3 we showed that the unravelling interpreter exploits loop asynchrony in two ways: by unravelling, 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 incrementing the index k, the 1 through m values of k will be generated quickly, and the a[i,k]*b[k,j] operations will overlap. But the additions in the innermost loop must be done serially, so it will take O(m) time to generate each dot product s (each element of row a).

Since the meaning of new d[j]<-expr is actually new d<-d+[j]expr, the value of new d depends upon the old value of d (just like s above). However, many instantiations of the innermost loop might be executing concurrently because each value 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 a. Since the first append operation (i.e., new a[1]<-expr) cannot begin until the innermost loop produces an answer, the total time to generate a row a 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 $O(\ell+m+n)$; note that the total number of multiplications, i.e., the total computational flux, 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 unravelling 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. If only enabled operations are scheduled, then it can be shown that processing time is inversly related to the number of processor resources. In the case of matrix multiply, if only one processor were available rather than ℓn processors, the time complexity would be $O(\ell mn)$.

4. Programming with Streams

All variables discussed in Sections 2 and 3 are called simple variables. Another kind of variable is also possible in Id: a stream variable. Stream variables generalize the possible behavior of an activity in two ways:

1. A simple activity may input (output) a potentially unbounded number of tokens from (onto) a single stream line, and
2. Input and output of streams through an activity is asynchronous, so a single activity may be in the act of producing an output stream while still accepting tokens from an input stream.

As an example, we generate a stream FIB comprising the first k Fibonacci numbers by the statement

```
FIB ← (initial f ← 1; nextf ← 1
       for counter from 1 to k do
           new f ← nextf;
           new nextf ← f + nextf
       return all f)                    (4.1)
```

Statement (4.1) has the simple variable k and the constant 1 as its inputs, and produces a stream of tokens at its output FIB. The stream output FIB corresponds to the clause all f, i.e., the ordered sequence of values assumed by f whenever the loop predicate is true (when $1 \leq counter \leq k$). If $k<1$ initially, then FIB will receive the empty stream. Figure 4.1 illustrates the stream produced on line FIB for k=5.

As a second example, let us consider the electrical circuit shown in Figure 4.2a. Let the state of an electrical line be represented by an ordered pair giving the voltage and the point in time when that voltage is said to exist on that

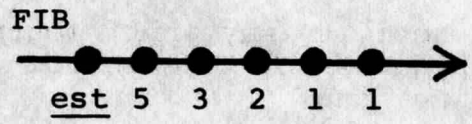

Figure 4.1
A stream on line FIB

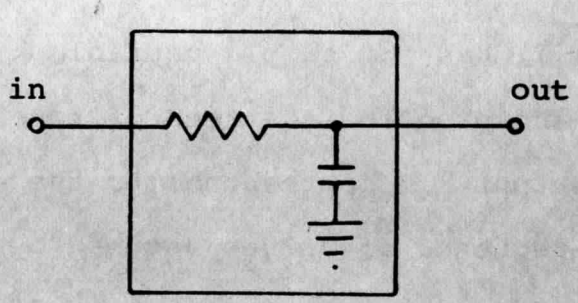

Figure 4.2a
The circuit of expression (4.2)

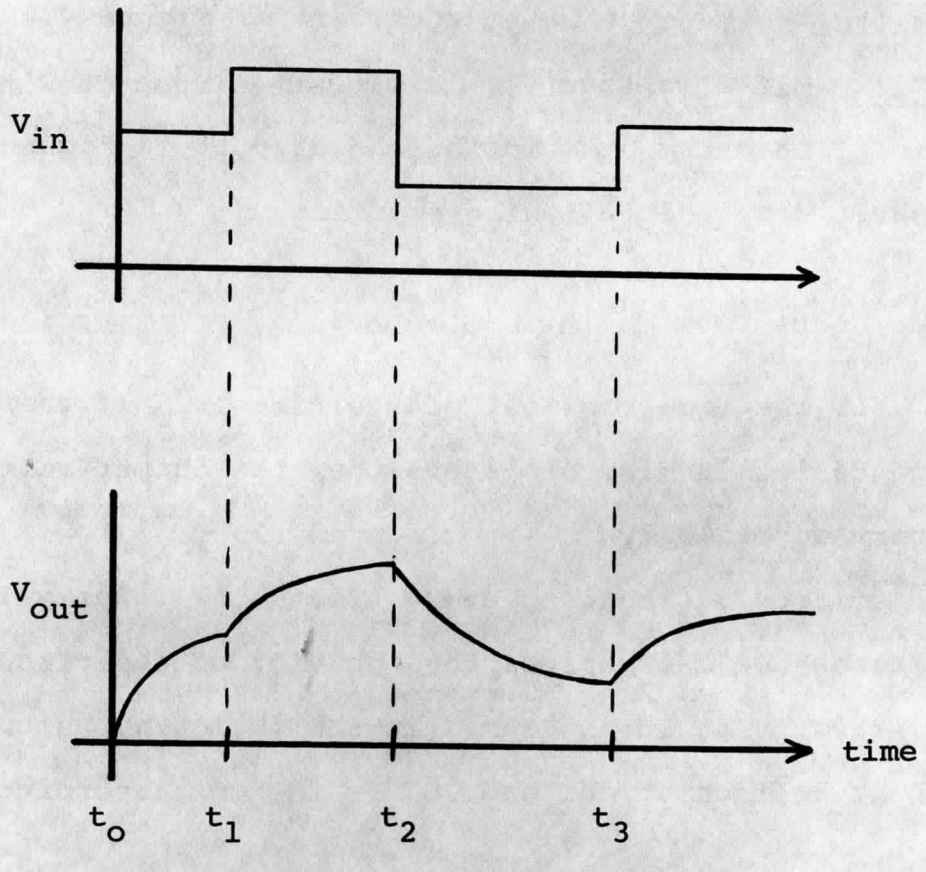

Figure 4.2b
Input to and output from the circuit
of Figure 4.2a

line. We can then represent the dynamic behavior of an electrical line by a stream of

<time: t, voltage: v>

structured values called a "voltage stream". Suppose at time t_o the voltage at the output port of Figure 4.2a is $V_{out}(t_o)$. If a voltage V_{in} is impressed upon the input port at time t then the behavior of the circuit in Figure 4.2a is described by the following equation

$$V_{out} = V_{out}(t_o) + (V_{in} - V_{out}(t_o))(1 - e^{-\tau(t-t_o)})$$

where τ is the time constant of the circuit. If the input voltage varies in discrete steps then the output voltage will also vary accordingly (see Figure 4.2b). If output voltage is recorded every time a new input voltage is impressed we can express the behavior of the circuit in Figure 4.2a by an Id program. In the following assume IN is the input voltage stream and OUT is the corresponding output voltage stream.

```
OUT ← (initial out ← V_out(t_o);
       t ← t_o

    for each status in IN do
        new t ← status.time;
        new out ← out + (status.voltage - out)
                    *(1 - e↑(-tau*(status.time - t))
    return all <time : new t,
                voltage : new out>)                    (4.2)
```

The for-each-loop above is a new construct which accepts each token as it arrives on stream line IN, transforms that

token to a simple token and places that token on the simple line "status", and finally executes the body of the loop one time. This behavior is repeated for each token arriving on the incoming stream IN. The return clause at the bottom of the loop is active on each iteration (due to the all construct) and consequently produces an output voltage stream on line OUT. Execution of (4.2) terminates when the end-of-stream is reached on IN. The function performed by (4.2) is interesting for two reasons:

1. It illustrates a "history-sensitive" function, whereby the output produced for a given input depends upon inputs previously received, and
2. Input and output is asynchronous; not all input need be defined before output can be produced.

4.1 Background

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 [Landin 65] 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 (both Landin's and ours) 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 easily modelled as a stream. However streams also have practical advantages, especially when subjected to a cascade or a pipeline of editing processes. For example in applicative languages*,

*Note that ld is an applicative language.

streams enable one to perform operations on lists (such as generating them, mapping them, concatenating them) 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 & Wise 76a,76b].

Streams were first introduced into dataflow by Weng in [Weng 75] where he gave formal rules for constructing "well-formed" dataflow schemata with streams. Weng does not allow any cyclic schema with streams except in a very limited sense; however, recursive schemas are defined in full generality. Weng did not have to extend Dennis dataflow language [Dennis 73] to implement his streams. The principal semantic difference between Id and Weng's streams is that many Id streams can appear on one line, while in Weng's language a stream is identical with the history of a line.

The remaining sub-sections are concerned with showing how streams can be implemented, and how they can be used to solve problems. Streams introduce new capabilities into dataflow that are necessary for programming certain kinds of problems, e.g. the problem of updating databases (this particular history-sensitive function is discussed in depth in Section 5). However, streams are also interesting for problems that can be solved by methods already presented in Sections 2 and 3, except that streams often introduce still another level of asynchrony

which can be very significant in exploiting machine concurrency.

4.2 Implementation of Streams

A stream is an ordered sequence of tokens, each token carrying a value, and where the last token in the stream carries the special value est (end-of-stream). Notationally we can describe the stream of Figure 4.1 in Id by writing the stream constant [1,1,2,3,5]; also, the empty stream may be written in Id as [ ] which comprises exactly one token, the est token. However, the Id programmer is never aware of the est token and cannot use it as a value except by writing [ ]. Extending the notation of Section 3, we denote the k^{th} token carrying the value X_k to activity u.c.s.i on stream line <u.c.s, p> by

$$\langle\langle X_k, k\rangle, \langle u.c.s.i, p\rangle\rangle$$

We may denote an entire stream with n tokens as a set:

$$\{\langle\langle X_k, k\rangle, \langle u.c.s.i, p\rangle\rangle \mid 1 \le k \le n\}$$

where the assumption is that the n^{th} token is est.
We denote the number of tokens in stream A (including est) by n_A.

The basic rules given in Section 3 for generating activity names are also valid for streams. Even though an activity may absorb more than one token on a port, no two tokens will have identical stream positions. Hence, each token in the input set for an activity is still uniquely identified. A further requirement on streams makes this task even simpler - a stream

cannot have missing token positions. If the est token appears in stream position n, then a token is defined for each stream position k such that $1 \leq k \leq n$.

The following shows how the four categories of Id expressions - blocks, conditionals, loops, and procedure applications - and some extensions of these expressions are implemented when stream variables are used. We assume that variables are kinded as either stream or simple and that the kind of a variable does not vary during the execution of a program. For convenience we will denote stream variables with upper case letters. This convention has been observed in both examples given earlier in this section.

4.2.1 Block expressions (functions and predicates): Several functions and predicates are defined on streams, some of which are primitive and others of which are included in the language simply because they are useful. Many of these functions and predicates will be used in implementing various Id constructs shown later. Activity names for streams in functions and predicates are manipulated in a manner identical to that for simples (see section 3.2.1); only the data parts and the stream positions are affected. Therefore we will show the manipulation of only these parts under the assumption that the input activity name u.c.s.i is transformed into the destination activity name u.c.t.i. If an operator has more than one output then it is assumed that the i^{th} output port is connected to input port p_i of statement t_i. As before, if an operator has only one port then port numbers are not shown.

(a) size (A): This function produces a simple value giving the length of the input stream A (recall that n_A is the number of tokens in stream A, including the est token).

u.c.s.i -- input: (stream) $\{<A_k, k> \mid 1 \le k \le n_A\}$

output: (simple) $n_A - 1$

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

u.c.s.i -- input: (stream) $\{<A_k, k> \mid 1 \le k \le n_A\}$

output: (simple) $(n_A = 1 \rightarrow$ true; false$)$

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

u.c.s.i -- input: (stream) $\{<A_k, k> \mid 1 \le k \le n_A\}$

output: (simple) $(n_A = 1 \rightarrow \xi;\ A_1)$

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

u.c.s.i -- input: (stream) $\{<A_k, k> \mid 1 \le k \le n_A\}$

output: (stream) $(n_A = 1 \rightarrow \{<\text{est}, 1>\};$

$\{<A_{k+1}, k> \mid 1 \le k \le n_A - 1\})$

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

u.c.s.i -- input: port 1 (simple) = x

port 2 (stream) = $\{<A_k, k> \mid 1 \le k \le n_A\}$

output: (stream) $\{<x, 1>\} \cup \{<A_{k-1}, k> \mid 2 \le k \le n_A + 1\}$

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

u.c.s.i -- input: port 1 (stream) = $\{<A_k, k> \mid 1 \le k \le n_A\}$

port 2 (simple) = x

output: (stream) $\{<A_k, k> \mid 1 \leq k \leq n_A - 1\} \cup$

$\{<x, n_A>, <\underline{est}, n_A + 1>\}$

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

u.c.s.i -- input: port 1 (stream) = $\{<A_k, k> \mid 1 \leq k \leq n_A\}$

port 2 (stream) = $\{<B_k, k> \mid 1 \leq k \leq n_B\}$

output: (stream) = $\{<A_k, k> \mid 1 \leq k \leq n_A - 1\} \cup$

$\{<B_k, n_A - 1 + k> \mid 1 \leq k \leq n_B\}$

(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.

u.c.s.i -- input: port 1 (simple) = x

port 2 (stream) = $\{<A_k, k> \mid 1 \leq k \leq n_A\}$

output: port 1 (filtered stream)

$$= \bigcup_{1 \leq k \leq n_A} (A_k \neq x \rightarrow \{<A_k, \text{count}>\}; \phi)$$

port 2 (position stream)

$$= \bigcup_{1 \leq k \leq n_A} (A_k \neq x \rightarrow \{<k, \text{count}>\}; \phi)$$

where count = $\#\{j \mid 1 \leq j \leq k \wedge A_k \neq x\}$

(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 (one of these two output remainder streams will be empty, by definition).

u.c.s.i -- input: port 1 (stream) = $\{<A_k, k> \mid 1 \leq k \leq n_A\}$

port 2 (stream) = $\{<B_k, k> \mid 1 \leq k \leq n_B\}$

output: port 1 (equalized A stream)

$= (n_A \leq n_B \rightarrow \{<A_k, k> \mid 1 \leq k \leq n_A\};$

$\{<A_k, k> \mid 1 \leq k \leq n_B - 1\} \cup \{<\underline{est}, n_B>\})$

port 2 (equalized B stream)

$= (n_B \leq n_A \rightarrow \{<B_k, k> \mid 1 \leq k \leq n_B\};$

$\{<B_k, k> \mid 1 \leq k \leq n_A - 1\} \cup \{<\underline{est}, n_A>\})$

port 3 (remainder of A stream)

$= (n_A \leq n_B \rightarrow \{<\underline{est}, 1>\};$

$\{<A_{n_B - 1 + k}, k> \mid 1 \leq k \leq n_A - n_B + 1\})$

port 4 (remainder of B stream)

$= (n_B \leq n_A \rightarrow \{<\underline{est}, 1>\};$

$\{<B_{n_A - 1 + k}, k> \mid 1 \leq k \leq n_B - n_A + 1\})$

(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 input stream is to be extended.

u.c.s.i -- input: port 1 (stream) = $\{<A_k, k> \mid 1 \leq k \leq n_A\}$

port 2 (simple) = x

port 3 (stream) = $\{<B_k, k> \mid 1 \leq k \leq n_B\}$

port 4 (simple) = y

output: port 1 (extended A stream)

$= (n_A \leq n_B \rightarrow \{<A_k, k> \mid 1 \leq k \leq n_A - 1\}$

$\cup \{<x,k> \mid n_A - 1 \leq k \leq n_B - 1\} \cup \{<\underline{est}, n_B>\};$

$\{<A_k, k>\} \mid 1 \leq k \leq n_A\})$

port 2 (extended B stream)

$$= (n_B \leq n_A \rightarrow \{<B_k, k> \mid 1 \leq k \leq n_B - 1\}$$
$$\cup \{<y,k> \mid n_B - 1 \leq k \leq n_A - 1\} \cup \{<\underline{est}, n_A>\};$$
$$\{<B_k, k> \mid 1 \leq k \leq n_B\}$$

(k) exif(A,B,x): This operator means "extend A to B by x, if necessary", and is a useful combination of the equalize and extend operators as shown in Figure 4.3. Unlike extend and equalize, exif is not symmetric. It produces a stream which is equal in length to stream B. In case A is longer than B, the output stream contains the first n_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 port.

The above functions and predicates are available to the Id programmer; some will also be used later to implement the for-each-loop and other constructs. As has already been pointed out, the general rules for activity name manipulation are also followed by stream functions. However, as opposed to simple functions, stream operators can begin producing output tokens as soon as enough input has been received to calculate any result token(s). For example rest(A) can produce tokens as soon as it receives each input token. In particular, the arrival order of tokens for rest (A) is totally immaterial. Similarly empty(A) can produce a true or a false token as soon as it receives any token belonging to stream A. But even though answers can be produced before all input has been received, the activity continues to remain in existance until it absorbs all its input tokens. (Otherwise, there would be unused tokens left to clutter up the machine.)

To understand the kind of asynchrony that is possible with streams, let us consider the block expression (4.3).

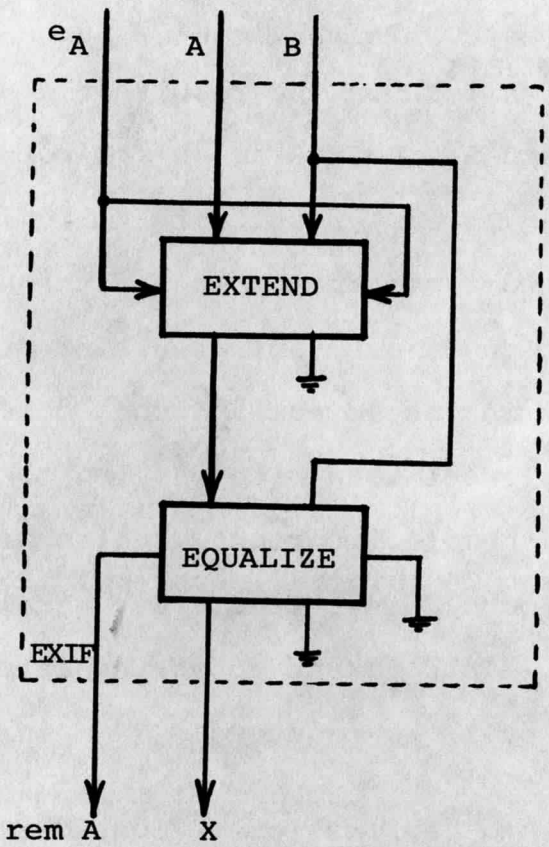

Figure 4.3
Definition of the EXIF macro operator

```
( A ← cons(3,A);
  B ← concatenate ([2,7,4],A);
 return A,B)                                    (4.3)
```

The cons function outputs the token $<A_1,1> = <3,1>$ immediately upon receipt of the simple input value 3. Thus the input stream A to cons is defined, which allows cons to produce token $<A_2,2> = <3,2>$ etc. In this manner A becomes an infinite stream of 3s. while B is the result of concatenating the stream [2,7,4] and A. Both A and B are returned as the (infinitely long stream) result of (4.3).

We indicated above that tokens within a stream may appear as input to an activity at a time out of phase with their positions in the stream. Such a condition might easily arise in an implementation with varying token communication delays. But even more important we also allow stream output tokens to be produced in a time order unrelated to stream position order. Thus tokens appearing late in physical stream position might still be produced early in time.

4.2.2 Conditional expressions: Consider the Id conditional expression

```
(if p(x,A) then f(A) else g(B))                 (4.4)
```

where f(A) and g(B) are streams. The base language translation of (4.4) is identical to the case when f(A) and g(B) are simples. The meaning is that if the predicate is true (the predicate must be a simple boolean value), then the result of the expression is stream f(A), else it is stream g(B). Any expressions may appear in the then and else clauses that satisfy the rule.

given in Section 2.2 for the case of simple value expressions (e.g. both clauses must specify the same number of results). There is only one further condition we must now impose on the expressions appearing in the then and else clauses: corresponding results in the then and else clauses must both be simples or both be streams.

To implement the above conditional on streams, we simply extend the definition of the SWITCH (Section 3.2.2) operator for the case of stream data inputs:

$$\begin{aligned} \text{u.c.s.i} \text{ -- } \quad \text{input: data port (stream)} &= \{\langle X_k, k\rangle \mid 1 \le k \le n_X\} \\ \text{control port} &= b \\ \text{output: T-port (stream)} &= (b=\underline{\text{true}} \rightarrow \{\langle X_k, k\rangle \\ &\qquad \mid 1 \le k \le n_X\}; \phi) \\ \text{F-port (stream)} &= (b=\underline{\text{false}} \rightarrow \{\langle X_k, k\rangle \\ &\qquad \mid 1 \le k \le n_X\}; \phi) \end{aligned}$$

Just like any other operator, a SWITCH with stream input is asynchronous from input to output, so if the boolean token has already arrived then each stream input token can be output immediately without waiting for further input.

4.2.3 Loop expressions: Loops on streams can be very involved expressions. However, if streams appear simply as another variable in a loop, then the schemata already discussed in Section 3.2.3 provide the appropriate semantics - we simply extend the operators D, D^{-1}, L, and L^{-1} to handle stream input and output just as we did for the SWITCH operator above by

replacing any reference to a simple variable x in the definition with the stream reference $\{\langle X_k, k\rangle \mid 1 \le k \le n_x\}$.

As we saw at the beginning of this section there are also some new loop constructs concerning streams. The following paragraphs consider these constructs.

4.2.3.1 The all construct: A loop is a natural stream generator as demonstrated by expression (4.1). A somewhat idealized loop incorporating the all construct is

```
( initial x ← f(a)
  while p(x,n) do
        y ← g(x,n) ;
     new x ← h(y)
  return all x, all y, x )                    (4.5)
```

the base language translation for which is given in Figure 4.4 where we have introduced a new operator:

(a) The E^{-1} operator: This operator is necessary for implementing the all construct. It accepts a simple token as input and changes its activity name to conform to belonging to a stream.

u.c.s.i -- input token: (simple) x
output token: (stream element) <<x,i>,<u.c.t.1,1>>

Note that every initiation of operator E^{-1} with activity name prefix u.c.s contributes to the production of the same stream, that is the stream on line <u.c.t,1>. Further note that this output stream always has an initiation count of 1.

There are two important points about Figure 4.4. First, recall that L^{-1} is asynchronous on stream input and output (all x and all y in the above example). Second, all x returns exactly those x for which p(x,n) is true, and this means that

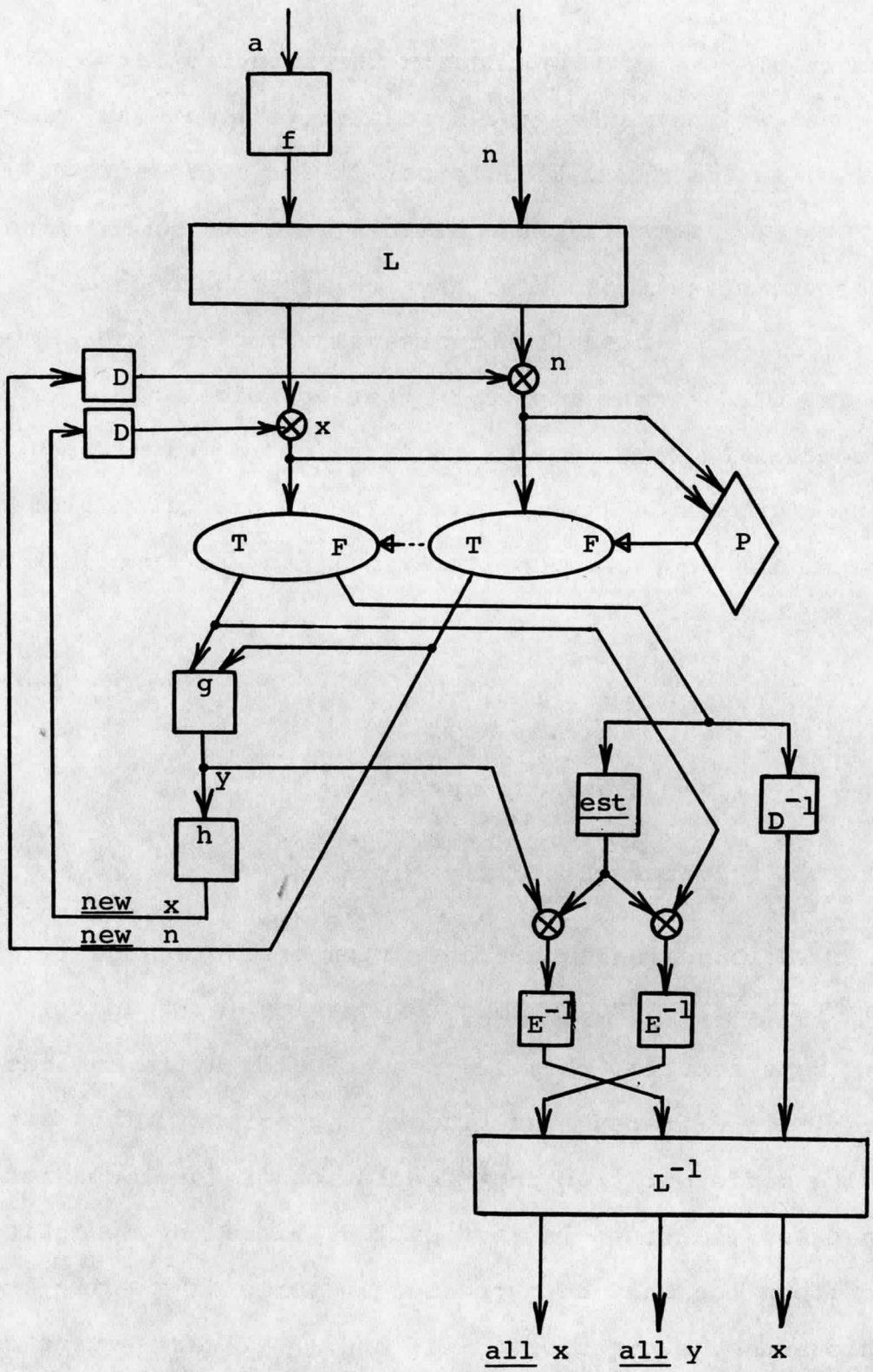

Figure 4.4

Compilation of expression (4.5)
showing the all construct

the final x is not included in the output stream. The figure also shows that since any stream is terminated by an est token, est is automatically output for every stream being generated. Note that the history of each stream line is valid and that the streams produced have no missing tokens.

The all clause is not necessary for writing stream programs in Id, but it can simplify matters considerably. Expression (4.6) produces the same results as (4.5) but involves substantially more computation (two partial streams are circulated completely around the loop on each iteration), and it functions with far less asynchrony.

```
(initial x ← f(a); X ← [ ]; Y ← [ ]
 while p(x,n) do
        y ← g(x,n);
    new Y ← consℓ(Y,y);
    new x ← h(y);
    new X ← consℓ(X,x)
 return X, Y, x)                                    (4.6)
```

The base language translation of expression (4.6) is given in Figure 4.5. Even though expression (4.5) and (4.6) produce the same results, they are semantically different because they result in different base language programs. This situation is quite different from the translation of for-loops into while-loops that was discussed in Section 2.3, since the semantics of a for-loop can only be expressed in terms of a while-loop. In this sense, expression (2.11) and (2.12) are semantically the same since they result in identical base language programs; again, this is not the case for expressions (4.5) and (4.6).

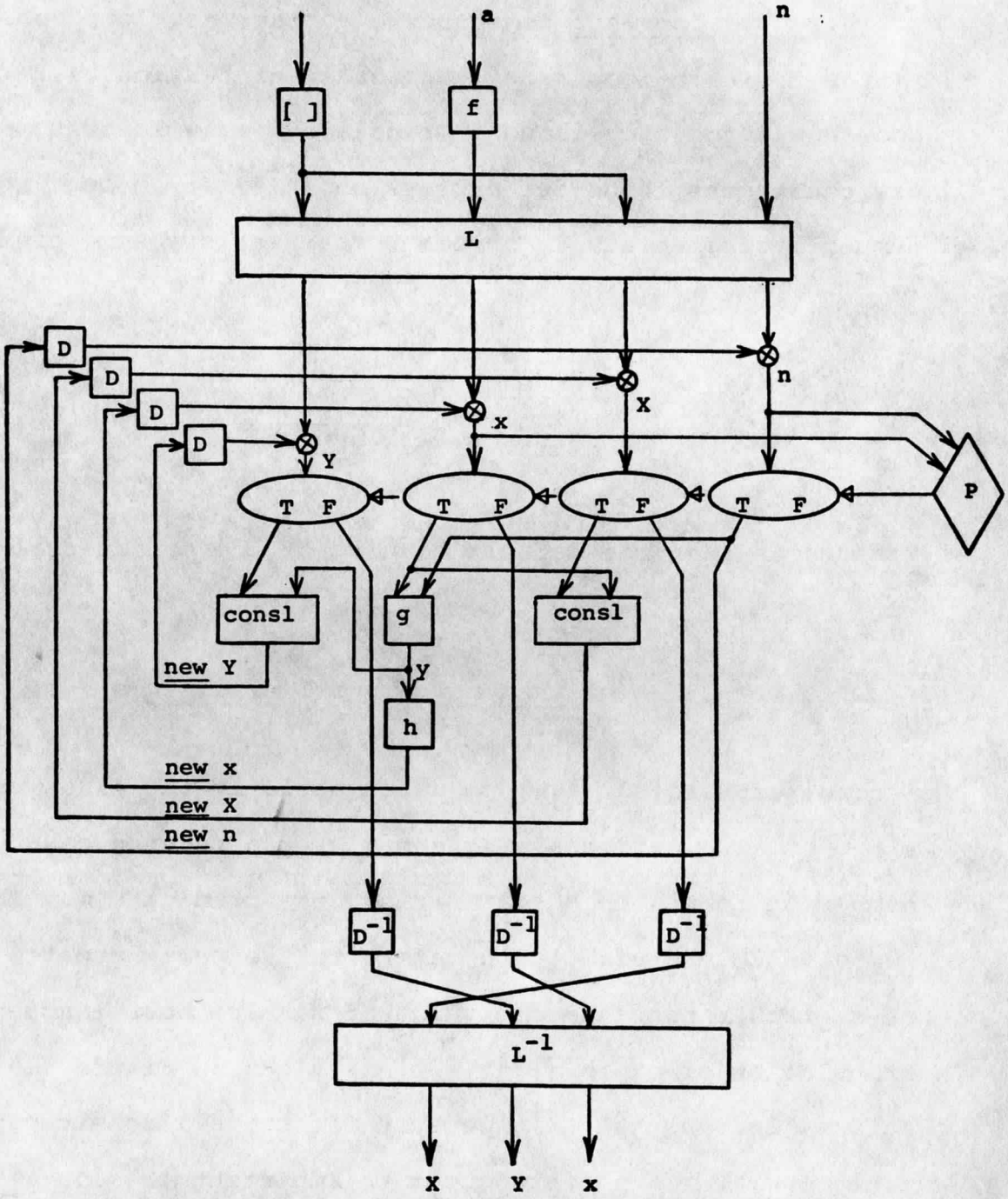

Figure 4.5
This loop (from expression (4.6)) produces results
identical to the loop of Figure 4.4,
but by circulating streams

4.2.3.2 The for-each construct: Another new loop construct is the for-each-loop shown earlier in expression (4.2). To show how a for-each-loop is translated, as well as some other new constructs, consider expression (4.7) for computing the number s of elements in stream B that satisfy some predicate q.

```
( initial s ← 0
  for each b in B do
    (if q(b) then new s ← s+1
  return s, all new s )                    (4.7)
```

First note that the default meaning of the conditional statement is

(if q(b) then new s ← s+1 else new s ← s)

Second, expression (4.7) produces not only the final value s but it also produces a stream giving a running count of the number of tokens in B that satisfy the predicate q. The phrase new s in the return clause refers to the newly created value of s, rather than the old value of s. The base language translation of expression (4.7) is given in Figure 4.6. As before, it uses the E^{-1} operator for the all construct, but a new operator E is introduced to implement the for-each construct:

(b) The E operator: Each activity of this operator takes a single stream of tokens as input and produces a sequence of simple tokens for different initiations of the operator that follows it.

u.c.s.i -- input: (stream) $\{<X_k, k> \mid 1 \leq k \leq n_X\}$

output: (simples) $\{<X_k, <u.c.t.k, p>> \mid 1 \leq k \leq n_X\}$

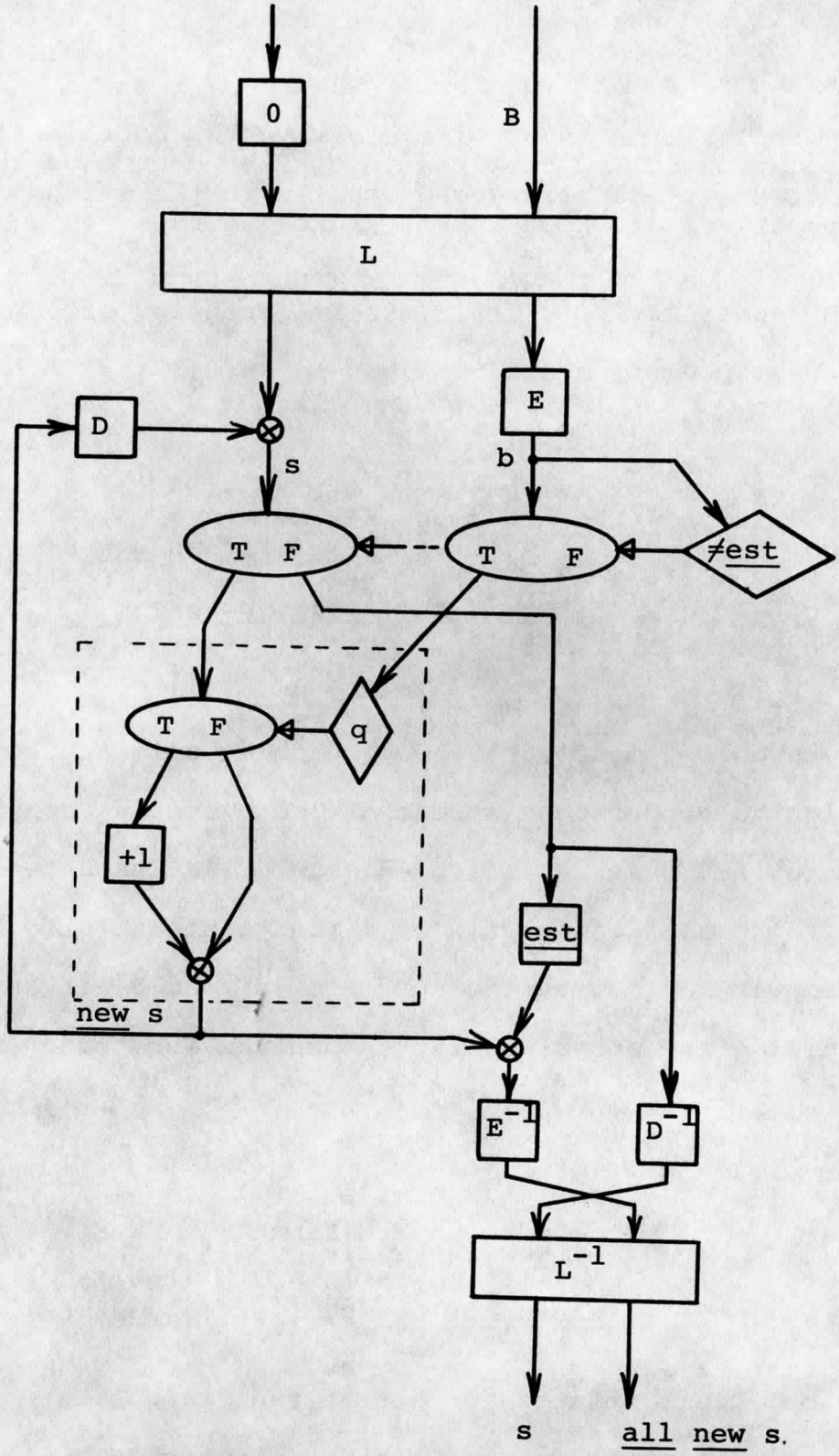

Figure 4.6
Translation of expression (4.7)
showing the for-each construct

Note that the input stream always has an initiation count of 1, and that the initiation counts of the output tokens correspond exactly to the stream positions of those tokens in the input.

So the reader may be certain of the semantics of expression (4.7), a while-loop equilvalent is given in (4.8):

```
( initial s ← 0; S ← [ ]
  while not empty (B) do
          (if q(first(B))then new s ← s+1);
        new B ← rest(B) ;
        new S ← consℓ(S, new s)
  return s, S)                                    (4.8)
```

Suppose we modify the above problem slightly so that in addition to producing a stream giving a running count of those elements in B that satisfy predicate q, let us determine the size of the smallest prefix of B that contains n such elements. We also wish to output the suffix of stream B, if any, after determining the above prefix. A program for this new problem is

```
( initial s ← 0; i ← 0
  for each b in B while s<n do
      new i ← i+1;
      (if q(b) then new s ← s+1);
  return all new s, i, remainder B )              (4.9)
```

This loop can terminate for one of two reasons: s<n and B runs out of tokens, or s=n. In the first case the final value of i will be the size of stream B, while the remainder of B will be the empty stream. In the second case, i will give the desired result and the remainder stream will contain all

those elements of B which come after the i^{th} element. Expression (4.9) can also be written as a pure while-loop by changing the loop predicate of expression (4.8) to (not empty(B) and s<n) and by introducing some detail involving i. However, since B may be an infinite stream*, we are most interested in the latter expression, the semantics of which appears in Figure 4.7. This concern is quite practical since an infinite stream should not be circulated in a loop, rather it should be taken apart one token at a time as in a for-each-loop. Figure 4.7 also shows stream B some distance from the E operator, since not all tokens in B will necessarily be used prior to loop termination. Thus a signal is necessary to indicate whether a token from stream B is needed or not. If the loop predicate turns false, then the signal stream (labelled S in Figure 4.7) is terminated by generating an est token. Also, since a token must be received from stream B before the loop can begin execution, a true valued token is put in front of the signal stream by cons. The signal stream is used as a control input to the exif operator placed between the L and the E operators. The exif operator acts like a valve and lets a token pass only if it is needed for the next iteration of the loop. Due to the extra true token in the signal stream,the last token coming out of operator E is also extra and it is destroyed by the DELETE-est operator.

---

*Infinite streams are not just a mathematical curiosity, but are very useful in modelling continuously operating systems.

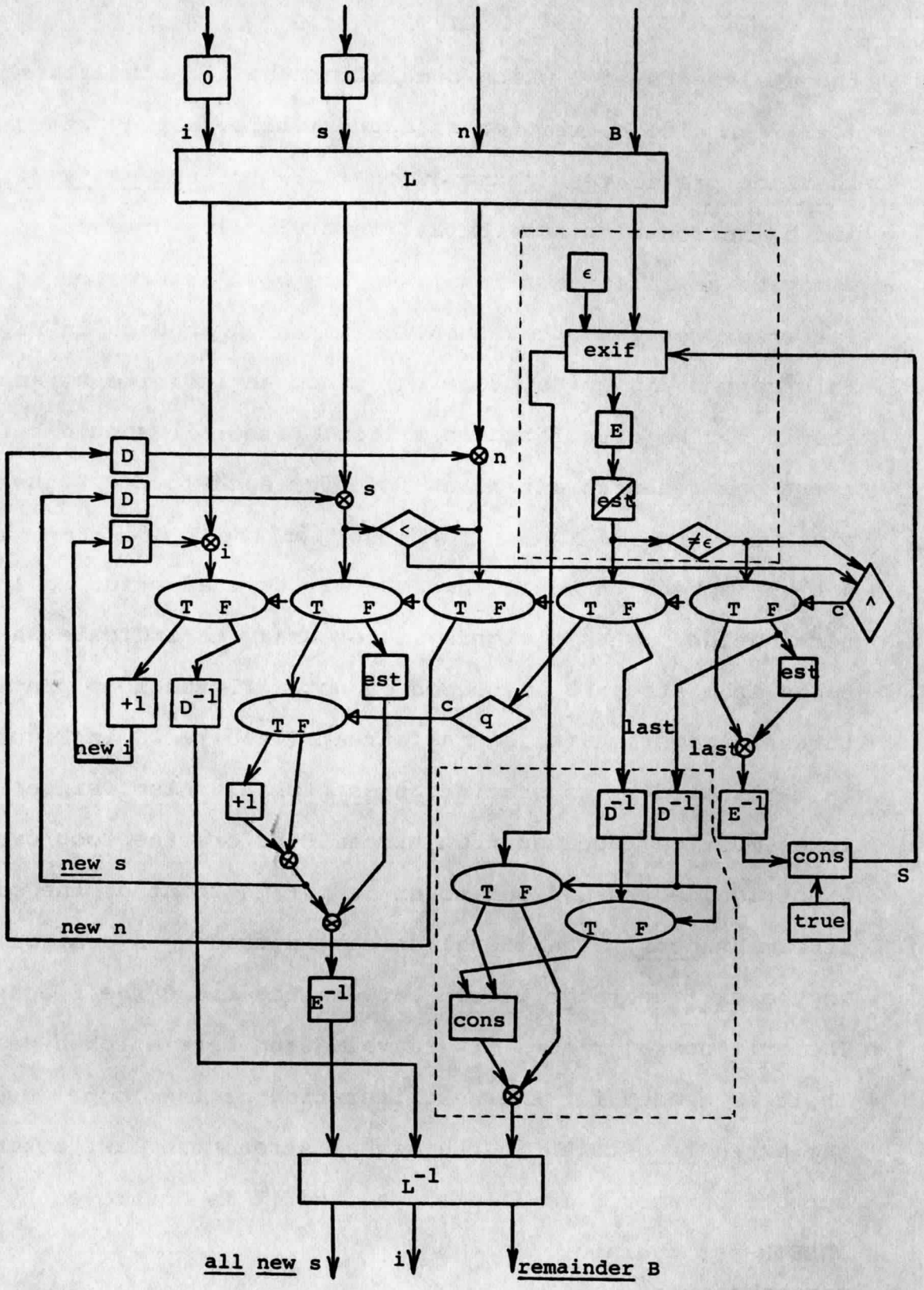

Figure 4.7

A for-each-while loop corresponding to expression (4.9)

(c) The DELETE-est operator: This operator lets all the tokens except the est token pass through. This operator is not available to the Id programmer.

u.c.s.i -- input = x

output = (x ≠ est → x;ϕ)

The graph enclosed in the lower dashed box in Figure 4.7 is needed to generate the correct remainder of stream B. When the predicate s<n turns false before tokens in stream B are exhausted, the remainder stream produced by the exif operator does not contain the first unused token of stream B(i.e., the last token to enter the loop body). The conditional construct of the dashed box is used to include this last token in the remainder stream under the appropriate condition.

The program shown in Figure 4.7 will terminate prematurely if there are any ϵ tokens in the input stream B. This problem is easily avoided either by keeping the ϵ token used by the exif operator distinct from all the other possible ϵ tokens (say, based on token names) or by generating the boolean sequence needed by the loop predicate (line c in Figure 4.7) directly on the basis of the input stream B and the signal stream S.

As a notational abbreviation in future base language program graphs, we will use the EACH macro-operator in place of both of the areas enclosed in dashed boxes shown in Figure 4.7. The EACH operator has sufficient information to generate the remainder stream also. Hence, we do not show the two ouputs

marked "last" as inputs to the EACH operator.

It is also possible to write the following type of loop:

for each b in B while p(b,x) do

where the tokens being extracted from the input stream are also part of the loop predicate. Since the last b may be an ϵ token (actually an est changed to an ϵ token), an error may be caused by evaluating p(b,x) unnecessarily; thus, the base language translation of p(b,x) is actually

(if b $\neq$ ϵ then p(b,x) else false)

Extra care has been taken to ensure that no tokens remain in the loop domain at loop termination. We note this point since it would be possible to speed-up loop execution even further if token clean-up was not a requirement.

4.2.3.3 The next construct: This mechanism has meaning only when embedded in a loop and is very useful for merging streams on the basis of some condition. Suppose we have two streams X and Y and a stream B of boolean values. A new stream may be formed by successively taking tokens from stream X and stream Y depending upon whether the token in stream B is true or false. The semantic effect we wish to achieve is described by expression (4.10).

```
(for each b in B do
 (if b then z ← first(X); new X ← rest(X)
      else z ← first(Y); new Y ← rest(Y))
 return all z,X,Y)                                (4.10)
```

Stream all z is the desired stream, and X and Y are remainder streams. The syntax described so far is inadequate for achieving the above without circulating the streams X and Y. As before, we consider the circulation of streams X and Y wasteful since at most one token is used from either stream in each iteration of the loop. We introduce the next construct to build the desired stream without circulating either X or Y. An expression equivalent to (4.10) using next is

```
( for each b in B do
    z ← (if b then next X  else next Y)
  return all z, remainder X, remainder Y)  (4.11)
```

The next construct behaves in a manner distinctly apart from our other constructs since next X takes a token from stream X only when one is needed. For example, in (4.11) a token is taken from X only when b is true, and no token is taken when b is false. To see how this works, associate a virtual counter with each stream X and a predicate with each next X operator. The counter associated with stream X is incremented when any of the predicates associated with next X is true. (For the present assume there is at most one next X .) For example, in expression (4.11) the predicate associated with next X is "b", while the predicate associated with next Y is "not b". A compiler can detect the predicate associated with each next. In order to translate next X , we generate

a stream containing the successive boolean values of the associated predicate. The length of this stream is determined by the number of times the encompassing loop is executed. Once this boolean stream has been formed, we filter out all false tokens and send the resulting stream of true tokens to the control input of an exif operator. The exif operator interprets the stream of true tokens as a stream of signals that indicate when a token should be released from stream X. The base language translation of expression (4.11) is shown in Figure 4.8. The exif operator here also behaves like a valve on a stream just as it does in a for-each-while loop. A new operator DIST (distribute) is needed in Figure 4.8 to generate the appropriate activity names for stream X tokens that are released into the loop. Again it should be pointed out that some of the complexity of the program in Figure 4.8 would vanish if self clean-up were not required.

(d) The DIST operator: The purpose of this operator is to distribute tokens only to specific initiations of another operator. This operator is available to the programmer only via the next construct.

```
u.c.s.i --      input: data port = x
                       control port = j
                output = <x,<u.c.t.j,p>>
```

As a notational abbreviation we will use the NEXT macro-operator in future base language program graphs in place of the larger network outlined in Figure 4.8. Since the exif operator does not use the values of the tokens in the signal

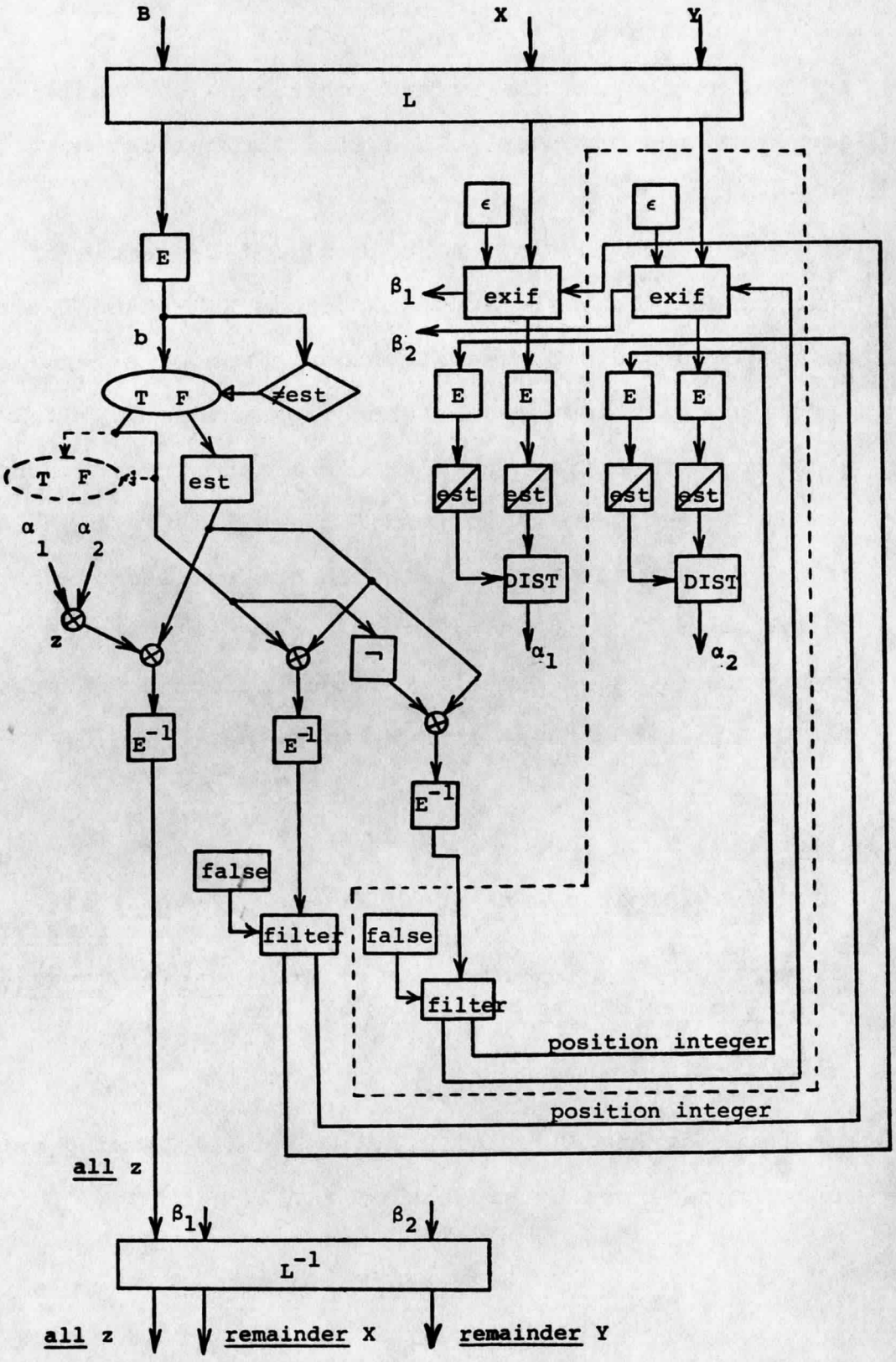

Figure 4.8
Compilation of expression (4.11),
showing the next construct

stream, we can use the stream containing the position integers for signaling purposes. This will further cut down the overhead.

The following is a more complicated example of a program that merges streams. Suppose streams X, Y, and Z are to be merged according to specifications given by stream I. Let i represent an individual token from stream I. If i=1 then a token is taken from both stream X and stream Y, and their sum is merged into the output; stream Z is left undisturbed. If i=2 then X is left undisturbed but the sum of the next tokens from Y and Z is taken. In case i=3 a token is taken from streams X and Z while Y is not affected. Expression (4.12) shows how these streams might be formed.

```
(for each i in I do
   (if i=1 then new X ← rest(X); new Y ← rest(Y);
                a ← first(X)+first(Y); b ← first(Y);
    else if i=2 then new Y ← rest(Y); new Z ← rest(Z)
                a ← first(Y); b ← first(Y)+first(Z)
    else new X ← rest(X); new Z ← rest(Z)
                a ← first(X); b ← first(Z))
 return all a, all b )                                    (4.12)
```

For input stream I = [1,1,2,1,3,2], the following output streams would be produced by expression (4.12):

$$A = [x_1+y_1, x_2+y_2, y_3, x_3+y_4, x_4, y_5]$$

$$B = [y_1, y_2, y_3+z_1, y_4, z_2, y_5+z_3]$$

Expression (4.13) instead uses the next operator to produce the same output as expression (4.12).

```
( for each i in I do
      a,b ←
            (if i=1 then next X +next Y, next Y;
             else if i=2 then next Y, next Y + next Z;
             else next X, next Z)
  return all a, all b )                                    (4.13)
```

The translation of expression (4.13) is quite involved; the following program is semantically identical to expression (4.13) but is better suited for illustrating the techniques of compiling the next construct. (Nevertheless, the programmer would probably consider (4.13) easier to write.)

```
1  ( for each i in I do
2  a,b ← (if i=1 then (y1 ← next Y ; return next X +y1,y1)
3         else if i=2 then (y2 ← next Y; return y2,y2+next Z)
4         else next X, next Z)
5  return all a, all b)
```

As noted before there may be several next constructs referencing a given stream, the associated predicates of which are not necessarily identical. For example, the predicate associated with next X in line 2 of (4.14) is "i=1" while the predidate associated with the next X of line 4 is "i≠1 and i≠2". Recall that there is exactly one counter associated with each stream regardless of the number of next clauses in which it appears. This implies that next X on line 2 is affected by next X on line 4, and vice-versa. The translation process deals with this situation in essentially the following way (below we will state the limitations of the translation technique):

1. Generate sequences of true/false tokens for every next operator in a loop (remember that only the innermost loop encasing a next affects its behavior). If the ith token in such a sequence is true, then the ith

iteration of the loop requires a token from the stream being input to the next.

2. All such true/false sequences associated with the same stream (say X) are logically ORed element by element to generate signal stream for input to the NEXT macro-operator. In this way the NEXT macro produces a token exactly when some next associated with X requires a token.

3. The remaining problem is to send the token produced by the NEXT macro to the appropriate place. Information regarding which next clause expects the token is contained in the true/false sequence associated with that next; these true/false sequences are thus used to switch the tokens to the waiting clause.

It is possible to take advantage of the fact that a given predicate may be associated with several next expressions on the same or different streams. We can, in general, reduce the number of true/false sequences that must be generated and thus considerably reduce overhead as shown in Figure 4.9.

The scheme described above operates properly when the next clauses associated with a stream are "properly nested" in conditional expressions. The following is an example where the above translation technique fails.

```
(for ...
 b ← (if(if p then next X  else f(y)) then next X
                                      else g(y))
 :
 :
 return ...)
```

It may be possible to assign some reasonable meaning to the above next clauses, but at this point we are not prepared to specify such complex semantics, nor does it appear essential to do so. As we will show in Section 5 (Resource Managers),

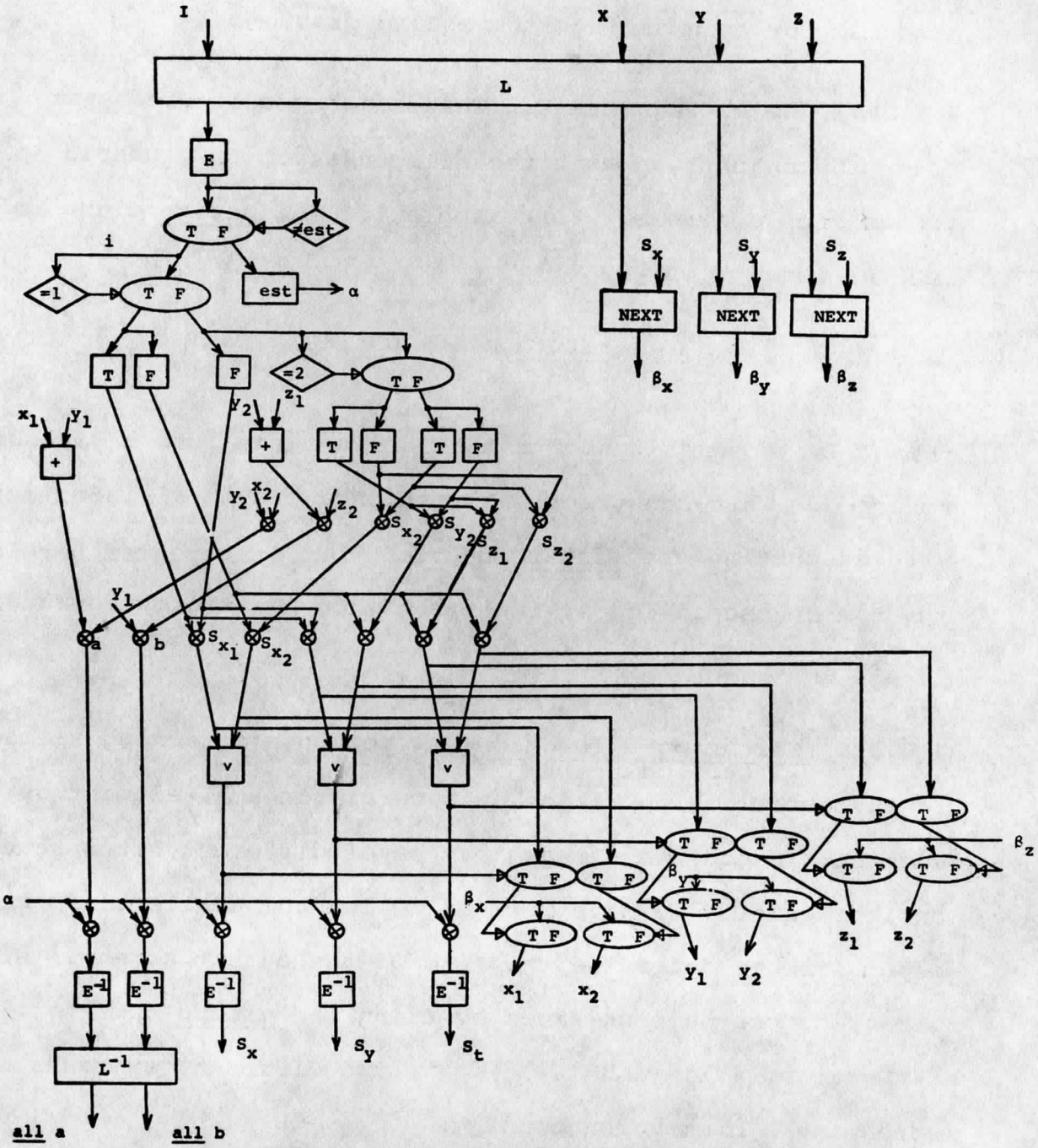

Figure 4.9
The NEXT construct and the
compilation of expression (4.13)

the next operator is already quite convenient and sufficiently powerful for programming interesting problems.

4.2.3.4 The but construct: This construct is used most often 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. This construct was mentioned previously (Section 4.2.1(h)) when discussing the filter function, but is repeated here due to its common use in loops; implementation is straightforward and is shown in Figure 4.10.

4.2.3.5 Streaming a function: A common programming situation is to perform an operation on several streams, element by element. Since the streams may be of different sizes several options exist. We have decided to let the smallest of the input streams determine the size of the output stream. Alternate semantics can be programmed by using the extend operator. Expressions (4.15) and (4.16) give two different syntaxes for streaming a binary function.

```
( for each x in X; y in Y do
  x ← f(x,y)
  return all z, remainder X, remainder Y)        (4.15)

[f(@X,@Y)]                                       (4.16)
```

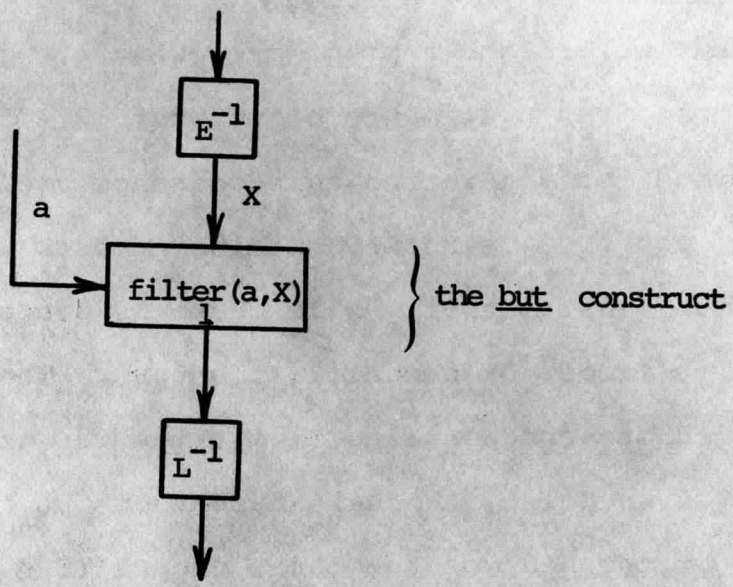

Figure 4.10
The but construct

The syntax of expression (4.16) does not permit the remainder of streams X and Y to be returned.

An example:

Now, we present a program to generate in ascending numerical order the first n elements of the set $\{2^i 3^j 5^k \mid i,j,k \geq 0\}$ [Dijkstra]. 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 i^{th} queue). If the i^{th} 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 i^{th} queue, according to the rules stated above.

Expression (4.17) produces the desired set as the stream output A. It is unusual in that it uses streams X1 and X2 as inputs to the same loop that generates them.

```
(A,X1,X2 ← (initial h1,h2,h3 ← 2,3,5
            for i from 1 to n do
            c ← min(h1,h2,h3);
            (if c=h1 then a,x1,x2 ← h1,2*h1, λ; new h1 ← next X1
             else if c=h2 then a,x1,x2 ← h2,2*h2,3*h2;
                               new h2 ← next X2
             else a,x1,x2 ← h3,2*h3,3*h3; new h3 ← 5*h3)
            return all a, all x1, all x2 but λ)                    (4.17)
```

4.2.4 Procedure application: Recall from Section 3.2.4 that procedure application actually involves the four operators A, A^{-1}, END, and TERMINATE. Like other control operators, the semantics of these operators must also be extended to handle streams. However, the change in the semantics here is more involved. We consider each operator in turn.

(a) A (activate): In Section 3.2.4 the A operator always had one output port which produced a single token carrying the value of structure α (the list of actual parameters). In order to maintain the asynchrony of streams as well as to handle infinite streams, it is necessary that a stream parameter be treated independent of structure α. Therefore the A operator has a separate output port for each stream parameter implied by the procedure definition being applied. If too few or too many input streams are actually supplied to operator A, then we follow the rules given in Section 2.5.2 and supply empty streams or delete extra streams as required. The following description is for the case when the procedure being applied requires one stream parameter and two simples.

$u.c.s_A.i$ --input: procedure port = q
argument port 1 (simple) = x
argument port 2 (simple) = y
argument port 3 (stream) = $\{< z_k, k> \mid 1 \leq k \leq n\}$

output: port 1 (simple) = $\langle\alpha,\langle u'.c_Q.\text{begin}.1,1\rangle\rangle$
port 2 (stream) = $\{\langle\langle Z_k,k\rangle,\langle u'.c_Q.\text{begin}.1,2\rangle\rangle \mid 1\leq k\leq n\}$
where $u' = (u.c_Q.s_T.i)$

(b) The BEGIN operator: The extension of the semantics given in Section 3.2.4(b) are quite straightforward and need no further explanation.

(c) The END operator: Here again, each stream output must be treated independently of the simple outputs (β). In the following we assume the END operator receives two simple inputs and one stream input.

$u'.c_Q.\text{end}.1$ where $u' = (u.c.s_T.i)$

input: port 1 (simple) = x
port 2 (simple) = y
port 3 (stream) = $\{\langle Z_k,k\rangle \mid 1\leq k\leq n\}$

output: port 1 (simple) = $\langle\beta,\langle u.c.s_T.i,1\rangle\rangle$ where β is $\langle x,y\rangle$
port 2 (stream) = $\{\langle\langle Z_k,k\rangle,\langle u.c.s_T.i,2\rangle\rangle \mid 1\leq k\leq n\}$

(d) The A^{-1} operator: This operator, in addition to doing what is described in 3.2.4(d), also adds streams or deletes extra streams. Activity name generation is extended in an obvious way from the case of simples to handle streams as well.

It is clear from the semantics given above that applying a procedure with stream arguments and stream results causes that procedure as a whole to be asynchronous from input to output.

4.4 Pipelining effect in stream programs

We illustrate the natural cascading effect of streams by the program in (4.18) to generate primes according to the Sieve of Eratosthenes algorithm. A recursive version of this procedure is given in [Weng 75].

```
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)                                    (4.18)
```

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 iteratively created LIST. Each iteration of the outer loop generates one prime number and a sieve; it is this sieve that produces a stream of integers for the next sieve if that stream is non-empty (see Figure 4.11). 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 (recall from Section 4.2.3 that filtering is completely sequential). 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 the SIEVE procedure will take O(n) time (assuming an unlimited number of processors is available).

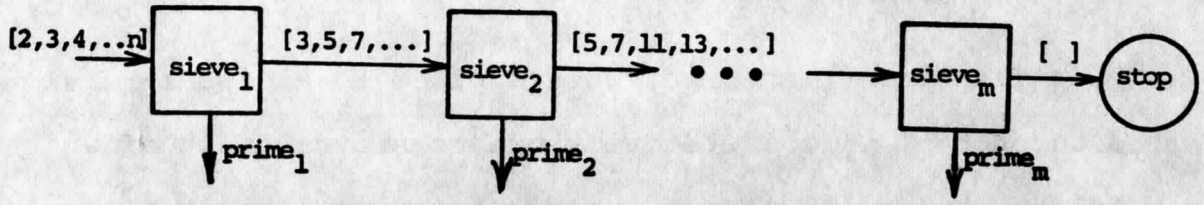

Figure 4.11
The Sieve of Eratosthenes in execution

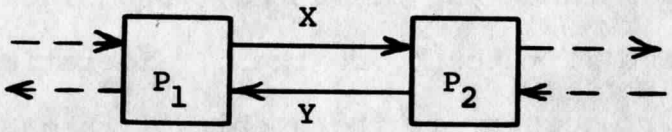

Figure 4.12
Deterministic interprocess communication channels

In order to illustrate the asynchrony of this stream procedure we compare it with the non-stream version of the Sieve of Eratosthones given in (4.19) following:

```
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[i];
    new i ← i+1;
    new list, new s ← (initial a ← Λ; k ← 1
                       for j from 1 to s do
                      (if mod (list[j], p[i]) ≠ 0
                          then new a[k] ← list[j];
                               new[k]← k+1)
                       return a,k-1)
    return p)                                                    (4.19)
```

Even though each sieve still takes $O(s_i)$ time, this procedure takes $O(\sum_{i=1}^{n} s_i)$ 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 takes place automatically. Dataflow programs generally are more asynchronous than their counterports 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 do not figure in writing programs. All dataflow programs will naturally run slower if there is a lack of resources. However no critical slow down will take place provided only enabled activities are given processors. This is primarily due to the side-effect free nature of dataflow.

4.4 Streams as interprocess communcation channels

Suppose two processes P_1 and P_2 communicate by sending messages to each other over unidirectional channels X and Y as shown in Figure 4.12. Let us further assume that communication between P_1 and P_2 is absolutely deterministic, that is, both processes send and receive messages only when certain time independent conditions hold. (We are not avoiding the case when messages may be time-dependent, that is, non-deterministic communication; we are simply postponing that discussion until Section 5.)

It is quite convenient to model such deterministic communication in Id. Expression (4.20) suggests a way in which P_1 and P_2 may send and receive messages. The outer block of expression (4.20) is of no physical consequence. It is only for making the names of channels X and Y known to each process without compromising the security of either since the normal scoping rules of Id prohibit both P_1 and P_2 from determining anything about the internal operation of the other.

```
! partial code for process P1 !
..., X, ...  ← (initial
                while true do
                  ! if q1 is true send a message to P2 !
                x ← (if q1 then m1 else  λ);
                .
                .
                .
                ! if p1 is true then receive a message from P2 !
                z ← (if p1 then next Y  else ...);
                .
                .
                .
                return, ..., all x but λ, ...);
```

```
! partial code for process P2 !
..., Y, ... ← (initial
               while true do
               ! if q2 is true send a message to P1 !
               y ← (if q2 then m2 else  λ);
               :
               ! if p2 is true receive a message from P2 !
               z ← (if p2 then next X  else ...)
               :
               return ..., all y but λ, ...);          (4.20)
```

It is easy to see that streams in Id really do behave like channels. Several processes can receive messages from the same channel, however, only one process can send messages through that channel (the single assignment rule). Since no special programming is required in order to use streams as communcation channels, the full programming power of Id is available to model interprocess communication. If processes are communicating over hardware channels then Id streams provide a good model for integrating these channels into a programming language.

Lastly, we remark again that Section 5 complements this discussion on interprocess communication by introducing the concept of a resource and by showing how general resource managers (for example, a database manager) can be written in Id.

5. Resource Managers

Any high-level language suitable for writing operating systems must include the concept of a resource, and it must also provide mechanisms for synchronizing accesses to a resource. For non-applicative languages, this is accomplished by using one or more memory cells to represent the state of each resource, so simple reading and writing (which includes P and V) of those cells is used to coordinate and to control those resources. Thus resource sharing among several processes is accomplished in an indirect way by the sharing of memory cells. Furthermore, some degree of nondeterminism is usually implied in the use of resources, for example the order in which a computer responds to two terminals, or the scheduling of two identical printers, may depend upon when those respective resources become available. Any reasonable model for resource managers must therefore incorporate a facility for nondeterministic programming.

The sharing of memory cells to effect non-determinism and access synchronization is not appropriate to Id since there are no memory cells to share. In this section we show the facilities that are present in Id for writing resource managers and how those facilities are implemented in the base language. We feel that Id provides a good framework in which the programmer can think about resource managers and produce program code that closely reflects the structure of the problem he seeks to solve. Further remarks on the approach Id takes to resource control are given in Section 5.4.

5.1 A primitive resource manager: Figure 5.1 outlines a very primitive resource manager in which the token produced by output s represents the current state of the resource being managed. The state s is part of a loop, so the next value of s is determined by the 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.

Expression (5.1) below is attempting to represent the resource manager described by Figure 5.1. However,

```
( initial s<-a        !initialize the
                       resource state!
  for each x in X do
    answer, new s <- f(x,s)
  return all answer, s )                    (5.1)
```

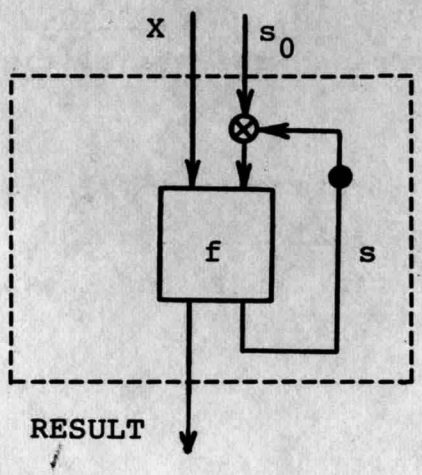

Figure 5.1
A primitive resource manager

expression (5.1) is an incomplete model for two reasons. First, it contains no provision for non-determinacy. Second, the expression could at best be a procedure being independently applied by the several contending processes. But a procedure application has no side-effects, thus no interprocess communication, and thus no successive values of s. (One possibility instead is to pass the resource to each process in turn as a parameter, a new value of which is then returned as a result. The problems here concern discovering who to send the resource to and the order of the processes to which the resource should be sent.)

To remove these difficulties, we have introduced two new semantic constructs into Id: a manager and a non-deterministic merge. Together they provide the facilities necessary to write resource managers in Id.

5.2 Dataflow managers: Statement (5.2) is a dataflow manager definition value being assigned to the variable md, where this particular manager definition will be shown to satisfy all the requirements implied by Figure 5.1.

```
md <- manager (sØ)
      (entry X do
       RESULT <- (initial s<-sØ
                  for each x in X do
                      new s, answer <- f(s,x)
                  return all answer)
       exit RESULT)                                    (5.2)
```

A manager definition value is essentially a pattern from which many instances of a manager values (i.e., managers) may be created. A given instance of a manager may then be used by any number of expressions in a program by passing the name of that manager to that program. The remainder of this section discusses Id and the underlying base language implementation of managers by following the creation, use, and destruction of a particular manager derived from statement (5.2).

Creation of a manager requires a manager definition and parameters for initializing that manager. For example, to create a particular manager from (5.2) with the value a as its initial state, we can write

```
me <- create(md,a)                                     (5.3)
```

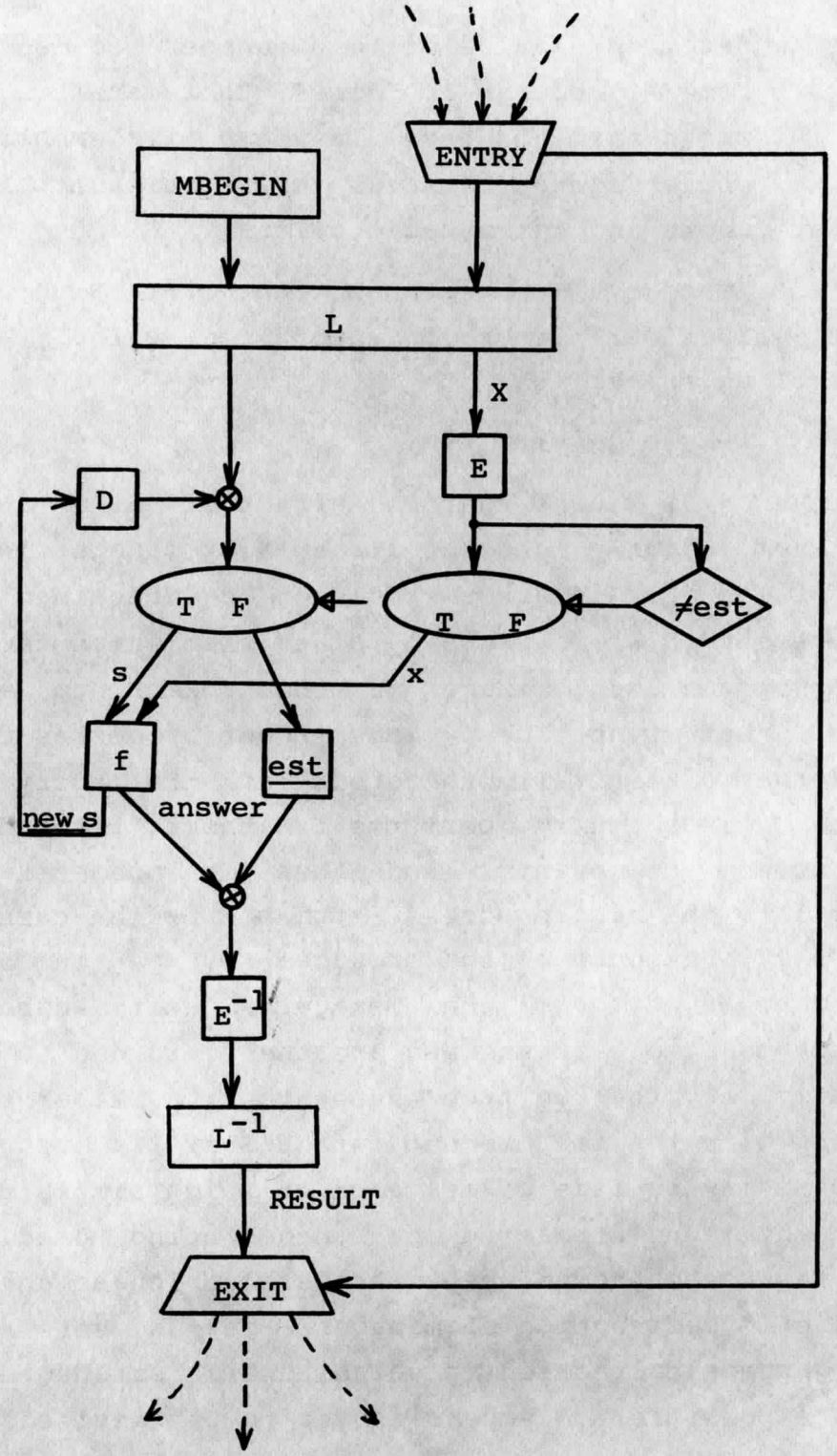

Figure 5.2

A manager created from statement (5.2)

The variable me receives a value of type manager and is the means by which the programmer refers to (names) that manager. Figure 5.2 illustrates in some detail the base language implementation of a manager value, while Figure 5.3 shows the relationships between the operators that create and use manager values.

To use the manager me requires that the programmer first acquire the value me; then to send the value y as the entry argument to me, he writes

z <- use(me,y) (5.4)

Following Figure 5.3, we see that the effect of (5.4) is to place the token from output y into the stream X of manager me; however, the exact position of y in stream X cannot be determined unless, of course, statement (5.4) is the only place from which the manager me is called. Everyone using manager me sends tokens to exactly the same manager, but since many independent processes may use that manager the order of arrival of tokens at the entry of me is indeterminate. Thus entry performs two tasks: it changes simple tokens into stream components, and then it nondeterministically merges them into one single stream (stream X in the case of manager me). Conversely, response tokens produced by me on line RESULT leave the manager through the manager's exit where they are converted back to simple tokens and are then returned to the waiting use. In particular, the use from whence the ith member of X arrived is the use to which the ith member of RESULT is returned. Any number of uses may be made of me, each of which simply supplies one component of the input stream to the manager being used. Finally, the manager may be destroyed when there are no longer any references to that manager. Destruction of managers is as yet an unsolved problem of garbage collection in which, unlike structures, circular references are possible. A scheme is yet to be devised which can decide when to destroy a set of managers that reference nothing but themselves and no one else references them*. As a final point, we

---

*I/O is not described here, but each device will be a manager that references those other managers that contain the data the device is transmitting; see references [Bic77a, Bic77b, Bic 78].

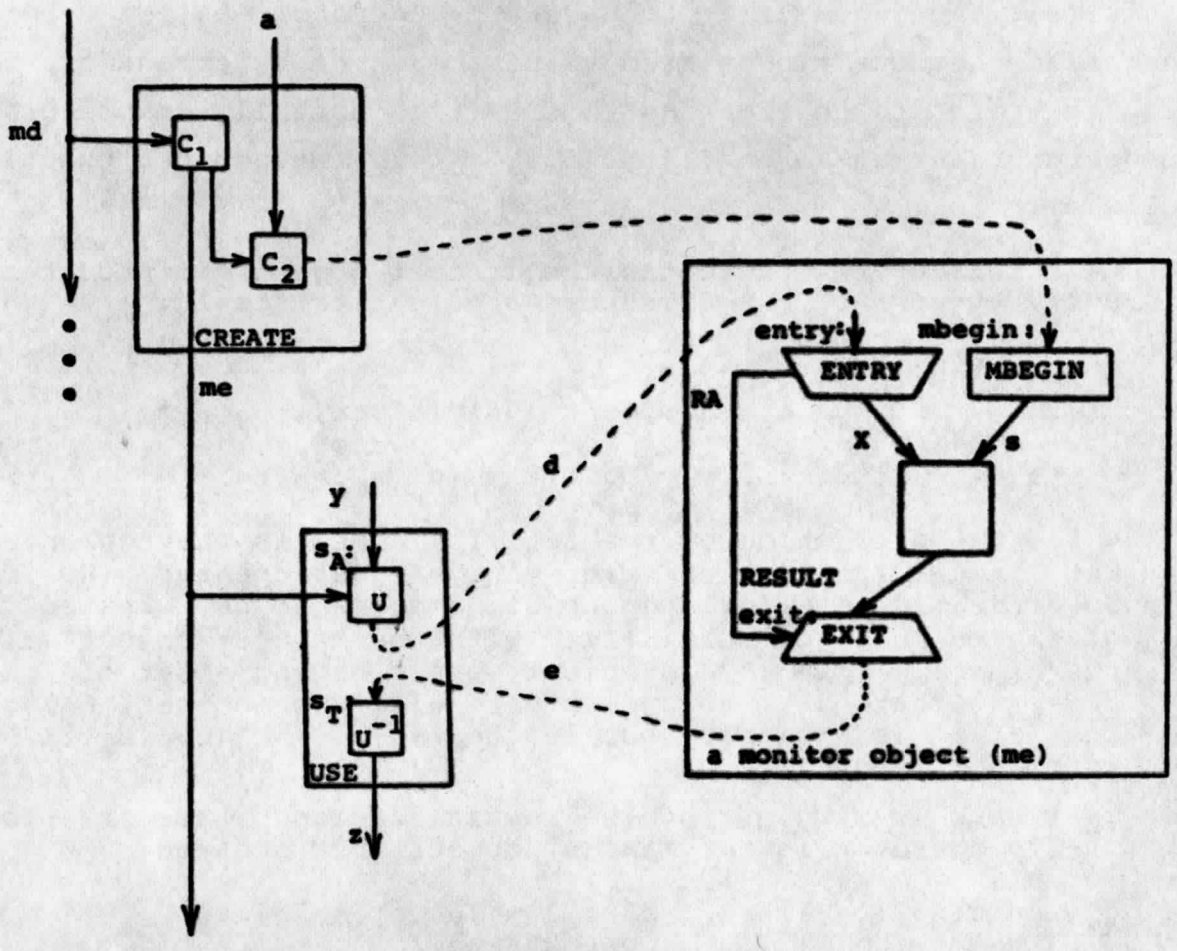

Figure 5.3
Creation and use of a resource manager

do not allow streams as initial creation arguments nor as arguments to use. All create and use arguments must be simple values.

We now consider the implementation of the constructs just described as well as the when construct which is used to control the timing of activities in a resource manager. The reader may still wish to follow Figure 5.3 to maintain perspective.

5.2.1 The create construct: Given a manager definition value and initialization arguments, the CREATE operator builds a resource manager and returns two related values, each of type manager, to be used to refer to that manager. CREATE is actually composed of two sub-operators as shown in Figure 5.3. Every manager has exactly one MBEGIN operator.

(a) C_1: This operator creates the manager object and produces the activity name of the ENTRY operator as its result:

$u.c.s_1.i$ -- input = c_m

output: port 1 = <(u'.c_m.mbegin.1), <u.c.s_2.i,1>>

port 2 = (u'.c_m.entry.1)

Note that production of the activity name is asynchronous to the initialization of the manager just created. We do this specifically to allow a group of managers to be created, each of which can be initialized with the manager object value of the others. This is convenient when creating a set of manager objects that are to communicate with one another. Also, the context u' is arbitrary but unique, so u' = ($u.c.s_1.i$) would suffice.

(b) C_2: Here we simply send the initialization parameters to the MBEGIN operator in the manager object just created:

$u.c.s_2.i$ -- input: port 1 = (u'.c_m.mbegin.1)
port 2 = x

output = <x,<u'.c_m.mbegin.1,1>>

(c) MBEGIN: Every manager has an MBEGIN operator, just as every procedure has a BEGIN. The only difference between BEGIN and MBEGIN is that MBEGIN does not receive a "return address" from CREATE, nor does it have a mate MEND operator.

u'.c_m.mbegin.1 -- input = x

output = x

5.2.2 The use construct: To use a manager object actually involves four new operators, as shown in Figure 5.3. We will let s_A be the label of the U operator, and s_T that of its mate U^{-1}.

(d) U: This operator sends a simple input token to the specified entry port of the manager object.

$u.c.s_A.i$ -- input: port 1 = $(u'.c_m.entry.1)$
port 2 = x

output = $\langle\langle x, u.c.s_T.i\rangle, \langle u'.c_m.entry.1, 1\rangle\rangle$

(e) ENTRY: This operator accepts simple tokens coming from many sources (each U operator is a source), changes them to stream components, and merges them nondeterministically into a single stream. Two streams are output: one stream contains the data part while the other stream contains the "return address" for the EXIT operator. Let t be the label of the destination of X.

$u'.c_m.entry.1$ where $u' = (u.c.s_1.i)$

input (simple) = $\langle x, u.c.s_T.i\rangle$

output: port 1 (stream element) = $\langle x, k\rangle$
port 2 (stream element) = $\langle\langle(u.c.s_T.i), k\rangle, \langle u'.c_m.exit.1, 1\rangle\rangle$

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.

(f) EXIT: The purpose of this operator is to return its data input stream, after transforming them back to simple tokens, to the activity specified in the "return address" input stream.

$u'.c_m.exit.1$ where $u' = (u.c.s_1.i)$

input: port 1 (stream element) = $\langle(u.c.s_T.i), k\rangle$
port 2 (stream element) = $\langle x, k\rangle$
where k means this is the kth such input to this EXIT.

output (simple) = $\langle x, \langle u.c.s_t.i, 1\rangle\rangle$

Notationally, we often do not show the stream RA from ENTRY to EXIT since it tends to clutter the diagrams; nevertheless, that stream is always present.

(g) U^{-1}: Similar to TERMINATE in procedure applications, this operator serves only to interface from the manager object back to the user.

$u.c.s_T.i$ -- input = x

output = x

As a final note, multiple entry-exit pairs may appear in a manager, provided each pair is named, as in

```
mon <- manager(t)
       (entry a: A,B;
              b: C
```

```
do ...
exit  a: D;
      b: F,G)                                    (5.5)
```

If the programmer writes

q <- create(mon,a)

then q will be a structure of manager object values, the selectors of which are the names of the entry-exit pairs. Thus

use(q.a,y+1,y-1)

will send y+1 and y-1 to the "a" port of q, and thus to the two streams A and B, respectively. The result from the above use will be a single value taken from stream D inside the manager.

5.2.3 The when construct: In this section we consider a construct that is needed in programming managers to provide timing signals and to release requests from queues. For example, the expression

sin(x) when t

means that sin(x) is not to be evaluated until signal t is received. The when clause behaves somewhat as an operator, and syntactically has the highest precedence of all operators. In general, to hold evaluation of the entire expression sin(x)/cos(x) until both of signals a and b have been received, we write

(sin(x)/cos(x)) when (a,b)

which is translated into the base language equivalent of

if a=a and b=b then sin(x)/cos(x) else u

where the else clause will never be executed since the predicate is a tautology.

5.3 Nondeterministic stream merge: In the previous section, entry was specified to be a nondeterministic operator. A second nondeterministic Id operator is available to the programmer and is the subject of this section.

Let A and B be streams. Then C is the result of nondeterministically merging streams A and B if

C <- merge(A,B)

Such a definition of merge allows Id to accept tokens from streams A

and B as they arrive, and output them to C subject to the restriction that the i<u>th</u> token is taken from A and output to C only if the i-1<u>th</u> token of A has already been output to C. The same behavior must hold for B. The <u>merge</u> construct is primitive and is implemented by the MERGE operator below.

(a) <u>The MERGE operator</u>: A complete operational semantics for this nondeterministic operator is very complex, so we rely on the reader's intuition and write only

$$\text{u.c.s.i -- input: port 1 (stream)} = \{\langle A_j, j\rangle \mid 1 \leq j \leq n_A\}$$
$$\text{port 2 (stream)} = \{\langle B_k, k\rangle \mid 1 \leq k \leq n_B\}$$
$$\text{output (stream)} = \{\langle C_m, m\rangle \mid 1 \leq m \leq n_A + n_B - 1\}$$

where $C_m = A_j$ or B_k and $m = j+k-1$, and where if $C_m = A_j$ and $j>1$, then there exists an $m'<m$ such that $C_{m'} = A_{j-1}$.

The behavior of the MERGE operator cannot be described in terms of a function even from the histories of the input lines to the history of the output line because output history is not uniquely determined simply by knowing input histories. The semantics of MERGE also cannot be described in terms of the set of all possible input-order-preserving histories. This latter point is rather subtle and is due to the possible presence of feedback (cycles) within a program. For example, consider expression (5.6) a semantic representation of which is given in Figure 5.4.

```
( X <- [a,b];
  Z <- merge(X,Y)
  Y <- (initial i<-1
        for each z in Z while i≤3 do
            new i <- i+1;
                y <- z*c
        return all y)
  return Z,Y )                                    (5.6)
```

Since the loop defining stream Y will execute three times, Y will receive three tokens (plus the end-of-stream token). There are also only three possibilities for the firing of the MERGE operator, which along with the final value of Y are given below:

1. The first two tokens are from X while the third token is from Y = $[ac, bc, ac^2]$.
2. The first and third tokens are from X while the second token is from Y = $[ac, ac^2, bc]$.
3. The first token is from X while the second and third are from Y = $[ac, ac^2, ac^3]$

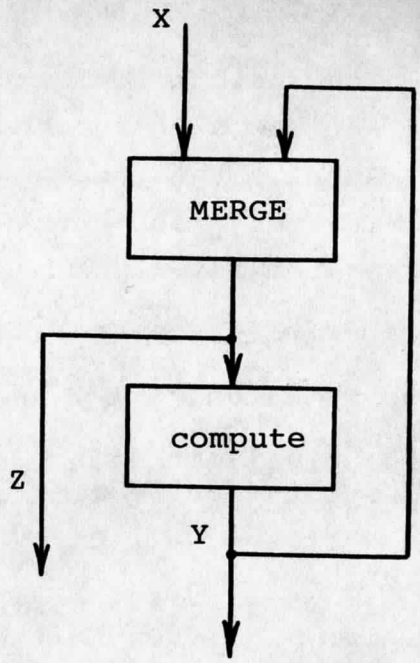

Figure 5.4
The MERGE operator in a feedback loop

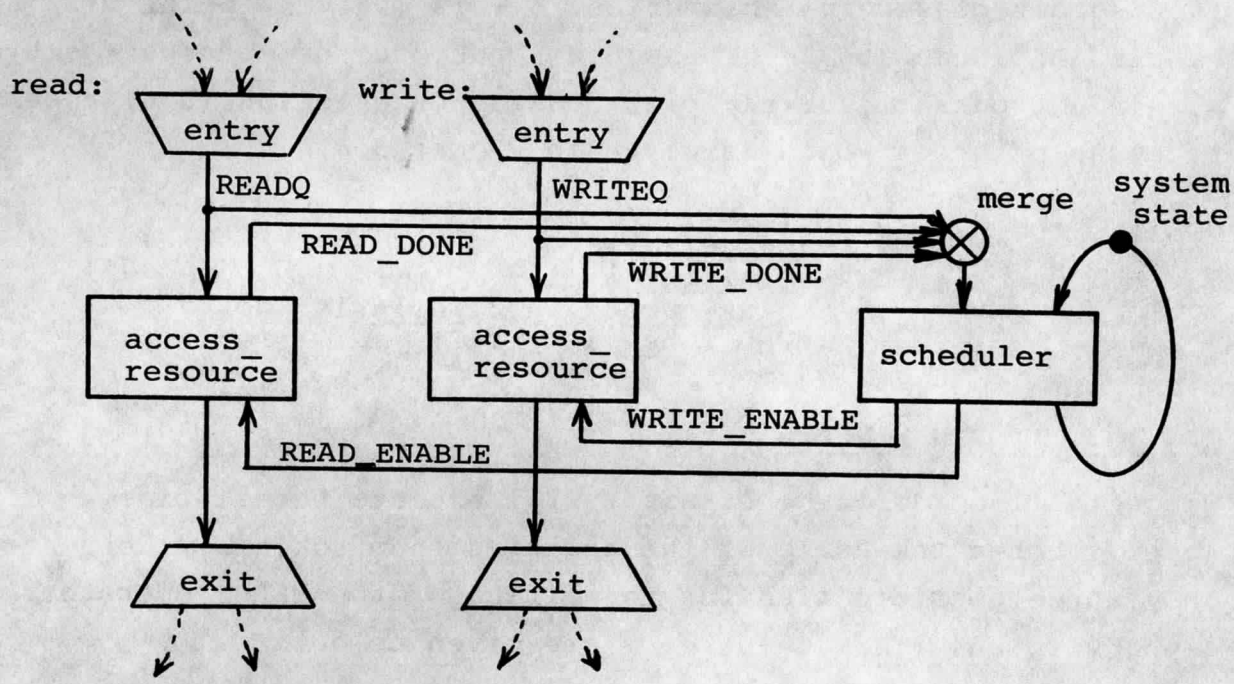

Figure 5.5
A resource manager

Let us consider the first case above in detail. Inputs to the merge are stream X = [a,b] and stream Y = [ac,bc,ac^2], and there is exactly one legal outcome for Z = [a,b,ac,bc,ac^2]. We point out that there are at least nine other input-order-preserving histories for these inputs, all of which are physically unrealizable due to the presence of feedback. A semantic specification of MERGE [Keller77, Kosinski] is beyond the scope of this paper.

5.4 Example: The problem is to devise a resource manager* which has control over a file and which accepts requests from users to read and write that file. This resource manager may permit simultaneous read accesses, but any write access must exclude all other accesses. Figure 5.5 outlines a resource manager which, with only minor changes, can implement three different scheduling policies corrsponding to three different versions of this readers-writers problem [CHP71, Hoare74]. The manager is composed of two logical parts: the agent which performs the actual computation, and the scheduler which blocks or enables individual requests within the agent. We emphasize the word "logical", in that the scheduler possesses no new primitive functions in order to carry out its work, and is programmed entirely in Id.

This particular manager has two entry-exit streams, one called "read" and the other "write" corresponding to the two kinds of requests that can be sent to the manager. For the resource manager we are now describing, each request enters the queue READQ or the queue WRITEQ according to which named entry port was used. Each queued request will match with an enabling signal from the streams READ_ENABLE or WRITE_ENABLE (generated by the scheduler) which then allow queued values to be released to the access_resource routine. This is done using the when construct discussed above. Proper operation of the resource manager requires that the scheduler be notified whenever (1) a request enters the manager or (2) a request completes its read or write access. Since these signals are nondeterministically generated, we merge them within the resource manager to form a single stream X of signals to the scheduler. Thus, nondeterminacy may appear in two ways in this manager: in the entry statements, and in the merge statement.

---

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

In the programmed solution of the resource manager shown in Figure 5.6, 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.

(a) Version 1: (Hoare [Hoare74] 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 scheme prevents indefinite exclusion ("starvation") of both the readers and the writers. The program for this version of the problem is that given in Figure 5.6.

Recall again that the semantics of an entry-exit pair assumes that the kth token in the exit stream corresponds to the kth token in the entry stream. Note that this correspondence in token positions is not related to time, that is, the time at which the k+1st result token is produced may be before the kth input request token has even been processed.

To embed the shared file file_res (another manager object) within the resource manager, we pass file_res to it at creation time:

file_manager <- create(resource_manager, file_res)

Requests to read file_res can now be performed by writing

use(file_manager.read, request)

Write requests are handled similarly.

(b) Version 2: (Problem 1 of [CHP71]) No reader is kept waiting unless a writer has already acquired the resource. Starvation of writers is possible. This means that the condition for generating an enabling signal for a "reader" is relaxed from wa=0 and ra=0 to simply wa=0. This is accomplished by deleting the code marked A from the first case condition in the scheduler of Version 1.

(c) Version 3: (Problem 2 of [CHP71]) No reader is allowed to proceed if a writer is waiting. Starvation of readers is possible. This does not affect the scheduling in the case for a "read exit" because the same condition applied in Version 1. However, for a "write exit" the scheduler must check for a

```
resource_manager ←
 monitor (file)          ! any file monitor resource such as file_res above !
 (entry read: READQ;
        write: WRITEQ do

    ! this is the agent code for a read request !
    READ_RESULT,READ_DONE ← [each r in READQ; re in READ_ENABLE:
                              (s ← access_resource(file,r) when re
                               return s, "read exit" when s)];

    ! this is the agent code for a write request !
    WRITE_RESULT,WRITE_DONE ← [each r in WRITEQ; we in WRITE_ENABLE:
                                (s ← access_resource(file,r) when we
                                 return s, "write exit" when s)];

    X ← merge(READQ,WRITEQ,READ_DONE,WRITE_DONE);


    ! the scheduler begins here --
         its function is to produce enabling signals !
    ! the input is stream X, the outputs are streams
         READ_ENABLE and WRITE_ENABLE !

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

     ! the scheduler ends here !

     exit read: READ_RESULT;
          write: WRITE_RESULT          ! end of the monitor !)
```

Figure 5.6

The reader-writer resource manager

waiting writer (ww>Ø). In particular, the code for Version 1 in position B is interchanged with that in position C, and the code in D is interchanged with that in E.

This example illustrates an advantage of dataflow. The program presents explicitly the essential components of the problem: the agent with separate reader and writer queues, and the scheduler which clearly shows the conditions under which enabling signals are sent and which thus is easily changed to implement different policies. In all the problems we have programmed, our experience has been that the scheduler policy is explicit and easily altered to suit various scheduling criteria. (Again, please note that the distinction made between agent and scheduler is only for emphasis, and that no special scheduling primitives are required.) Also, the scheme is modular, as illustrated here by the embedding of the file resource manager within the resource manager manager. Futhermore, we have found the basic structure of Figure 5.5, differing only in the number of entry-exit pairs, to be very useful in solving many resource manager problems including a distributed airline reservation system [AGP77] and a disk scheduler.

We also wish to make two final points about resource managers in Id. The first point concerns non-determinacy, which in sequential languages is usually a secondary effect resulting from a particular manner of use of shared variables. In dataflow, nondeterminacy is provided by explicit operators (MERGE and ENTRY in the base language) which allows the programmer direct control over nondeterministic behavior. The second point concerns the degree to which the requesting process, in passing through a manager (monitor [Brinch-Hansen, Hoare74]) in a sequential language say, is actually separated from that manager's internal controlling mechanisms. That is, each requesting process is actually in control and executing the code inside the manager representing the shared resource. This characteristic of managers in sequential languages 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 manager, such as enforcing conventions on the proper sequence in which semaphores are to be signaled. Id, however, implements a resource manager as a closed module which nondeterministically receives streams of

requests from other processes and acts upon these requests according to the scheduler written by the programmer that is 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 will make hardware/software module interchange easier to achieve.

6. Programmer-defined Data Types, Extensionality, and Environments

A procedure definition value is treated exactly like any other value in that it may be passed as a result of a computation, appended to a structure, etc. This general view of a procedure will allow us to incorporate some very sophisticated higher-level concepts into Id while requiring only minimally expanded lower-level mechanisms. These higher-level concepts include programmer-defined data types and operator extensionality so operators such as "+" may be interpreted at execution time in the context of the types of their arguments. These points, and discussion of an environment facility to ease programming effort, are the subject of this section.

6.1 Programmer-defined data types (pdts): In this section we show how Id allows the programmer to define his own data types and the meaning of operations on those types. We also extend the base operators in a very simple way in order to make Id an operator-extensible language.

6.1.1 Values: So far, details of value representation have not been considered other than to say that values are typed. Now that we wish to introduce programmer-defined data types, it becomes necessary to consider the internal representation of values. A value is an ordered pair comprising a type and an encoding. For example, the actual information held by a token that carries the integer value "five" is shown in Figure 6.1.

To change a value x from one type to another the programmer writes a coercion, for example

```
real: x
```

produces a real value from x (if possible). As a final remark, please note that the programmer has no access to the actual representation of values, as this is the domain of the machine's processors; the ":" is simply a binary operator.

6.1.2 Programmer-defined values: Just as the type field of a value of type integer is set to "integer", so is the type field of a value of the programmer-defined type "xyz" set to "xyz". The encoding of a value of type integer is a string of bits; the encoding of a

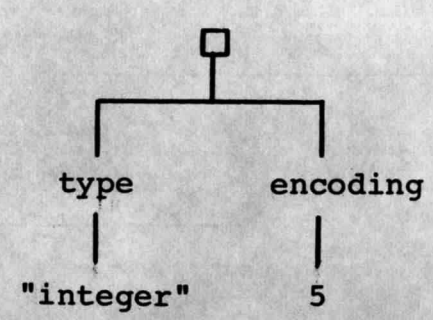

Figure 6.1
Representation of the integer value "five"

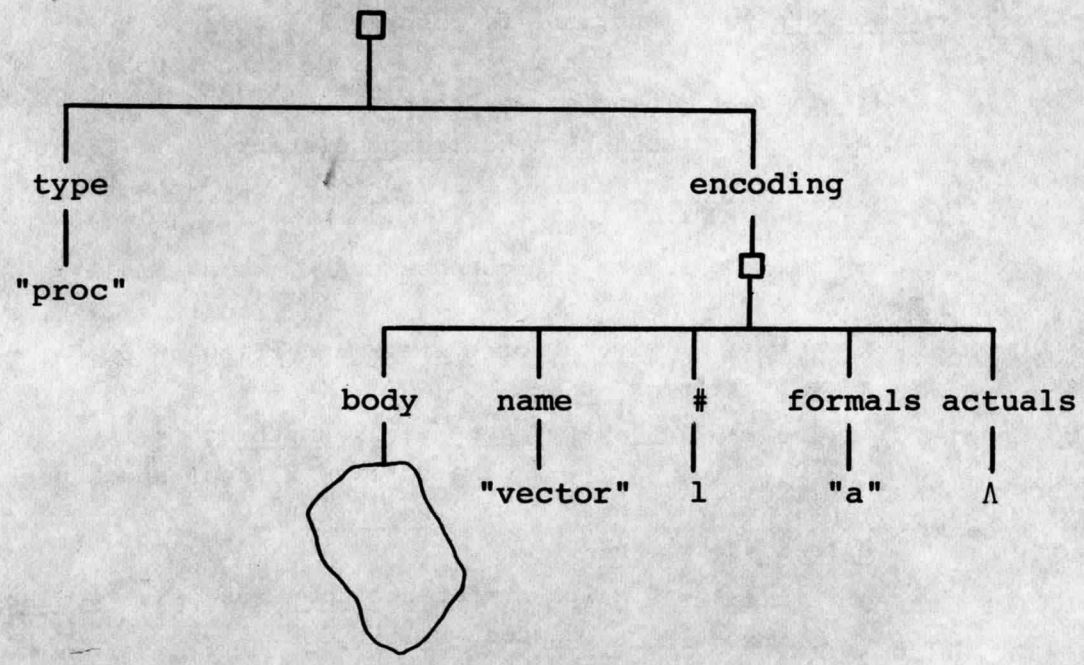

Figure 6.2a
Representation of a value of type
(named) procedure definition

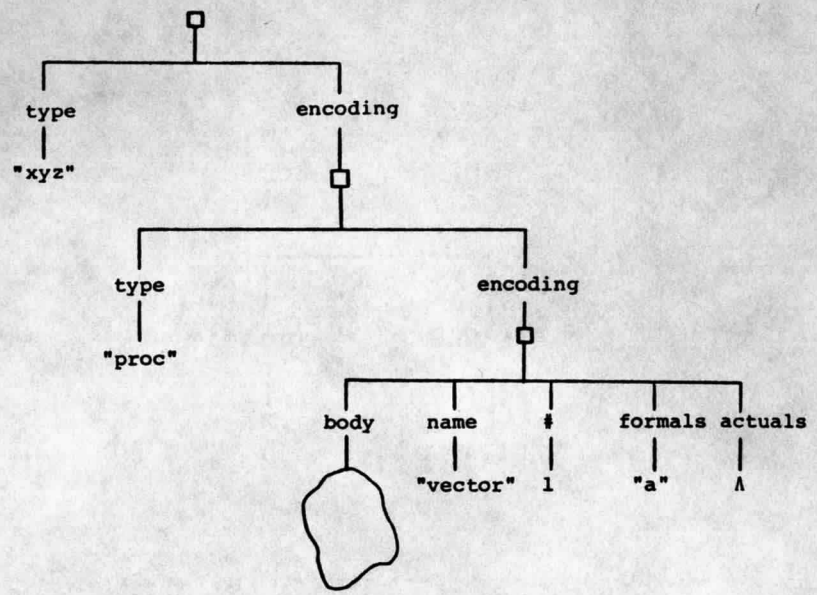

Figure 6.2b
Representation of the value in Figure 6.2a
after coercion to a pdt

```
procedure stack ( )
 ( z ← procedure sg (!function! f, !args 1,2! u,v,
                     !stack array! s, !top of stack! tos)
        (if f="size" then tos, anything
         else if f="push" then pdt:compose(stack,<<s:s+[tos+1]v>,
                                                 <tos:tos+1>>), anything
         else if f="pop" then
             (if sp≥1 then pdt:compose(stack,<<s:s+[tos]Λ>,
                                             <tos:tos-1>>), s[tos]
                      else error:<"stack underflow on pop">, anything)
         else if f="peek" then
             (if 1≤v and v≤tos then s[v], anything
                              else error:<"illegal stack peek">,
                                   anything)
         else if f="=" then
             (initial t ← (if :v="stack" then tos=|size|v else false)
              for i from 1 to tos while t do
                 new t ← s[i]=v|peek|i
              return t), anything
         else error:<"illegal stack operation">, anything);
   return stack:compose(z,<<s:Λ>,<tos:0>>))
```

Figure 6.3
The "stack" pdt

value of type "xyz" is a value of type procedure-definition. Let

x <- procedure vector(a)(...)

so x is a value of type (named) procedure definition as shown in Figure 6.2a. The result of the coercion

y <- xyz:x

is shown in Figure 6.2b.

For a detailed example of a programmer-defined type, we show the pdt "stack" with six operators and some error checking in Figure 6.3. (The reader is asked to indulge in the syntax of Figure 6.3 which is not intended to be user syntax, however we have not as yet devised a syntax for specifying pdts. The code in Figure 6.3 should be viewed as implementation code.) To generate a new stack initialized to empty, the programmer writes

a <- stack( )

where the value of a will be a pdt value of type stack. That value is produced by the procedure stack and is the result of a stack coercion on a procedure with two frozen parameters -- s=Λ and tos=0. Here s is a structure that represents the actual stack and tos is the index of s that is currently the top of the stack. Note that both the name of the pdt (i.e., "stack") and its generating procedure can be made the same*. Note also that a pdt value instance cannot be confused with the generating procedure for that type, and also that it is not possible to redefine the primitive system types.

Reviewing Figure 6.3, we can see that a pdt is actually a coerced procedure in which specific value instances of that type are produced by freezing certain of the procedure's parameters by means of the functional compose. Note that the generator of a pdt value, namely "stack", is a procedure; that procedure, when applied returns values of type stack.

6.1.3 Operators over programmer-defined types: The body of the pdt defines the operations that can be performed on the pdt. For example:

---

*Thus the pdt generator can be placed on the environment (Section 6.2) if appropriate.

```
a <- stack( );
b <- a|push|17;
c <- b|push|f(x);
```

says that a is the empty stack, b is a stack with the single item 17, and c is a stack with two items: 17 and the result of executing f(x). The extended syntax above translates directly into

```
a <- stack( );
b <- (proc:a)("push", a, 17);
c <- (proc:b)("push", b, f(x));
```

Note in the above that the instance of the data is passed to itself as an argument. This allows the operation to have access to the instance of the put data as well as its internal representation. So it, for example, an operation u|op|v were coded, which means

```
(proc:u)("op",u,v)
```

then "op" in the procedure proc:u might determine that the value v is also a put and that it must carry out "op2" and be given u as an argument. Thus we make u available to itself. To complete the notation, we let |op|u stand for unary and |op|(u,v,...,w) for n-ary operators.

The above has shown how to define new operators on programmer-defined types. There is also another kind of extensionality in Id as exemplified by the definition of the "=" operation on stacks (Figure 6.3). The definition of equality of stacks is that they are of the same length and corresponding entries in the stacks are equal. The first line of "=" in Figure 6.3 checks the argument v to be sure that it is also of type stack by asking

```
:v = "stack"
```

where the unary coercion operator ":" returns the type of value v as a string. If both arguments are thereby determined to be stacks, then the loop proceeds to check that the corresponding members of the two stacks are equal, and if so, returns true as the final result. The predicate s|i| = v|peek|i is the equality test, where "=" is evaluated in terms of the types of s[i] and v|peek|i. If, for example, s|i| and v|peek|i are also stacks, then equality is interpreted as stack equality. Since corresponding members of stacks u and v may vary each time around the loop, the definition of equality may vary each time as well. The implementation of this

extensional behavior is covered in the remaining paragraphs.

The equality predicate is primitive for several of the system-defined data types (it is not defined over monitor definition, procedure definition, or any pdt type). The configuration at the left of Figure 6.4 illustrates the pertinent base language translation extracted from Figure 6.3. However, the equality predicate (upon determining that, say, s[i] is itself of type pdt) internally "transforms" itself into the configuration at the right of Figure 6.4. In this way, whatever the type of s[i], the correct interpretation of equality will be taken (the programmer must, of course, have defined equality for his pdt).

The situation illustrated in Figure 6.4 is actually a simplification. Our example presumed that s[i] was of type pdt. In general we have an n-argument primitive operator "op", and any one (or more) of those n arguments a_j may be a pdt. If the ith argument a_i is a pdt and arguments 1 through i-1 are otherwise, then "op" transforms into

$$(\text{proc}:a_i)(\text{"op}i\text{"},a_1,a_2,\ldots,a_n)$$

where opi is a code that specifies the operation to be performed and which argument is serving as the context. This is important if the operators are non-commutative.

Lastly, we note that "op" is any operator, primitive or otherwise, so we could define "*" over the type vector to be the vector cross-product. Such flexibility seems possible in a dataflow machine since the apparently high execution overhead may be quite small when one considers the fact that an entire micro-processor could be devoted to the evaluation of just the equality predicate. The added flexibility in a language could be significant to programmer productivity. It might also allow program libraries to be more useful since they would tend to emphasize the definition of data types at an abstract level (rather than "subprograms") and provide operators specifically for manipulating those types.

As a final example of a pdt, Figure 6.5 shows the type set with the following operations:

set( )	means	ϕ
x|contains|a	means	$a \in x$
x|insert|a	means	$x \cup \{a\}$
x|union|y	means	$x \cup y$

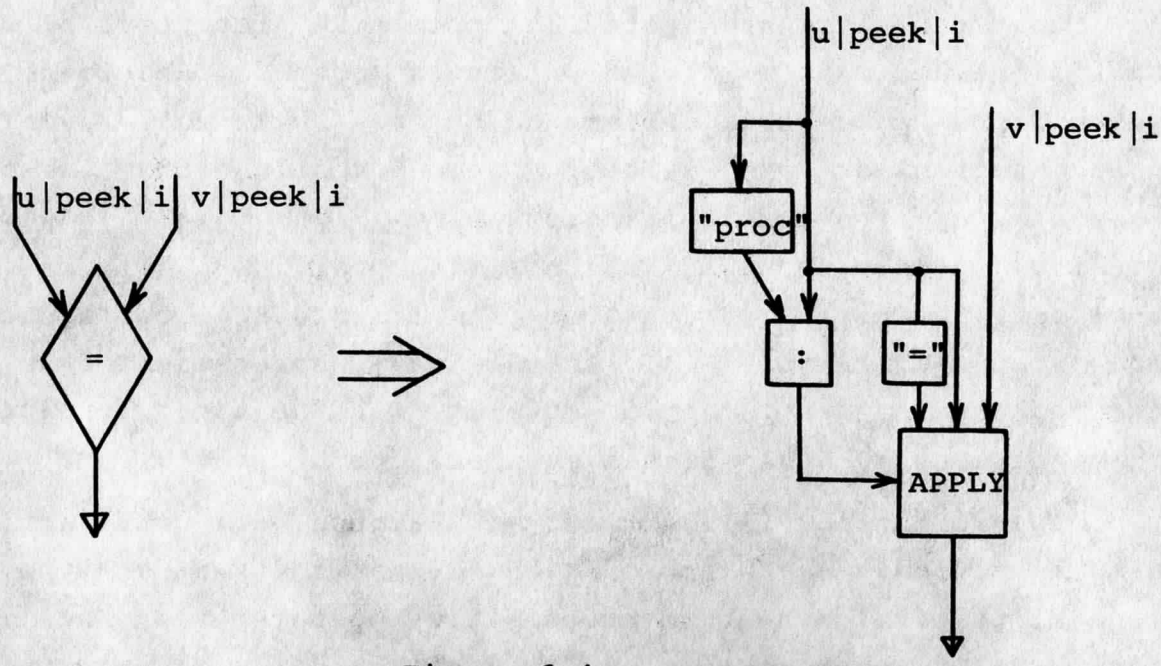

Figure 6.4
The primitive equality operator extends
to values of type pdt

```
procedure set ( )
  ( z ← procedure setg (!function! f, !args 1,2! u,v,
                        !set array! s, !cardinality! a)
        (if f="contains" then (initial t ← false
                               for i from 1 to a while  t do
                                  new t ← v=s[i]
                               return t)
         else if f="insert" then (if u|contains|v then u
                                  then u
                                  else pdt:compose(setg,<<s:s+[a+1]v>,
                                                         <a:a+1>>))
         else if f="union" then (if :v="set"
                                 then (initial x ← v
                                       for i from 1 to a do
                                          new x ← x|insert|s[i]
                                       return x)
                                 else error:<"argument is not a set">)
         else error:<"invalid set operation">)
    return set:compose(z,<<s:Λ>,<a:0>>))
```

Figure 6.5
The "set" pdt

6.2 The environment: Procedure definition expressions, both named and unnamed, have been discussed in previous sections. To review briefly, a procedure definition is simply an expression, so

```
( x <- proc(a,b)(a+b);
  y <- proc(a,b)(a-b);
  z <- x(u,v) + y(v,u)
  return z,x,y )
```

is a block returning a sum and two procedures. Note the difference between writing x and x( ), where the former means the procedure produced at output x and the latter means the result of applying x. The number of arguments passed to a procedure, or results returned from a procedure, need bear no resemblance to what was expected or required. Superfluous arguments and results are ignored, and missing arguments and results receive the value ξ. Also, a named recursive procedure is transformed at its definition into a procedure carrying the same name but possessing one new argument which is then composed with the procedure itself. A recursive procedure therefore carries a copy of itself as a frozen parameter.

In order to provide a library of procedures (such as sin, sqrt, etc.), and to allow the programmer to build a dynamic execution environment without undue parameter declaration and passing, Id automatically maintains an environment for the programmer. The environment, denoted η, is a structure of named procedures where the selectors of that structure are the names of the associated procedures. An environment is defined relative to a block and may change when execution passes from one block to some inner block. To append a named procedure to the environment, the programmer writes a procedure statement (as opposed to a procedure expression). For example, in expression (6.1),

```
A     B
( ... (x <- a+1;
       y <- proc(a,b)(f(a)+g(b))
       proc f(a)(a+1);
       proc g(b)(b+2)
       return y(x,x+1) + ( ... )   ) ... )          (6.1)
```

Block B is immediately nested in block A. Now if η_A is the environment brought into block B from A, then

```
B <- A + ["f"]proc f(a)(a+1) + ["g"]proc g(b)(b+2)    h,h
```

is the environment in block B. Thus, expression (6.1) is

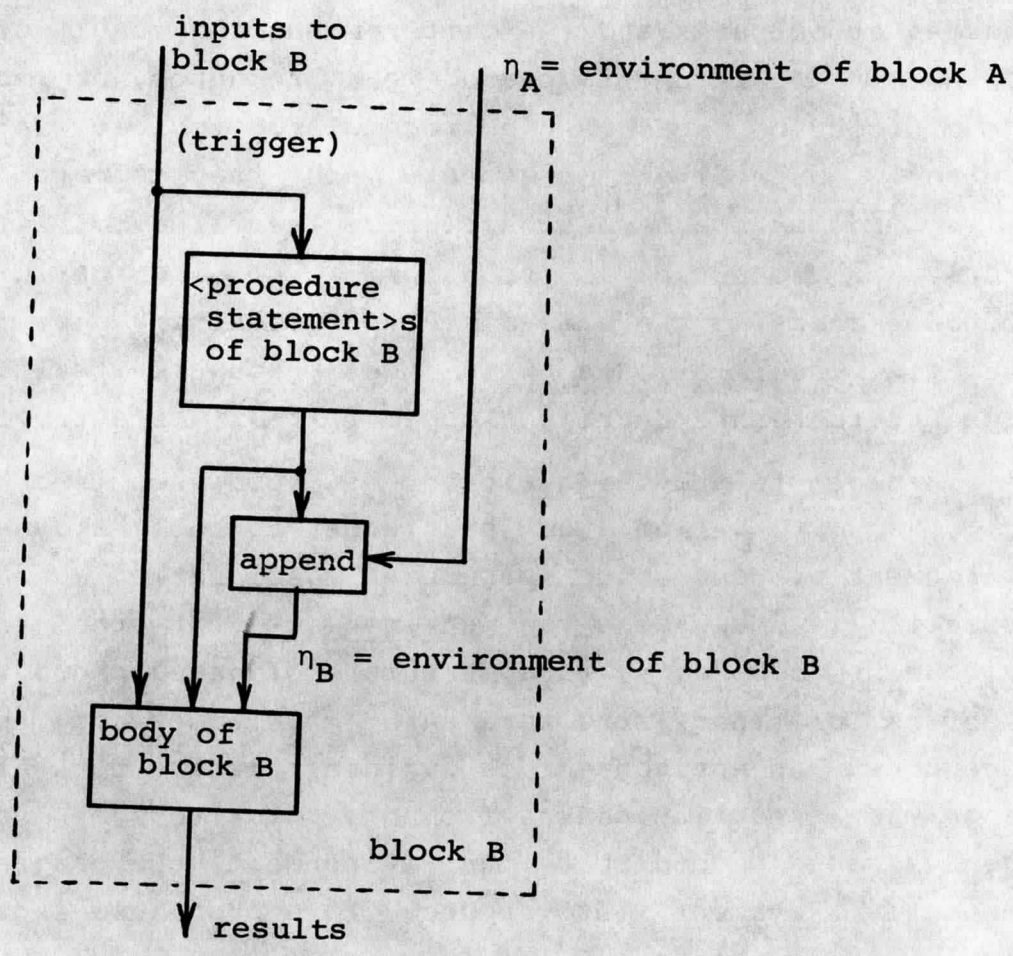

Figure 6.6
Building an environment in a block

semantically equivalent to expression (6.2) which the programmer cannot write himself, but Id does it for him:

```
A     B
( ... ( x <- a+1;
        y <- proc(a,b)(f(a)+g(b));
        f <- proc f(a)(a+1);
        g <- proc g(b)(b+2);
        η_B <- η_A + ["f"]f + ["g"]g
        return y(x,x+1) + ( ... ) ) .., )                    (6.2)
```

The above has shown how a block inherits and builds upon an environment. A procedure application can also inherit an environment. Whenever a procedure is applied the most recent environment is passed as an additional parameter. Thus both procedure definitions and applications are adjusted during the translation process to accommodate the environment parameter. To demonstrate, expression (6.3) is semantically equivalent to the compiled version of (6.2), but again the programmer can write directly neither (6.2) nor (6.3).

```
A     B
( ... ( x <- a+1;
        y <- proc(a,b,η)(f(a,η)+g(b,η));
        f <- proc f(a,η)(a+1);
        g <- proc g(b,η)(b+2);
        η_B <- η_A + ["f"]f + ["g"]g
    return y(x,x+1,η_B) + ( ... )) ... )                     (6.3)
```

Lastly, within the body of a procedure the programmer may refer to another named procedure that has been placed on the environment simply by writing that other procedure's name as if it were an Id variable. If

1. the name does not appear on the left-hand-side of an assignment in that block or any outer statically encompassing block,
2. the name is not the name of a procedure in a named procedure statement in that block or any statically encompassing block, and
3. the name is not a formal parameter nor the name of the procedure in which that block appears,

then that name is a reference to the environment and we call that name a η-parameter. In particular, the procedure assigned to variable y in (6.3) possesses two η-parameters: f and g. Thus the definition of y is semantically equivalent to

```
y <- proc (a,b,η)(η.f(a,η)+η.g(a,η))
```

though again, the programmer cannot write it this way -- Id does it

for him.

In summary, an environment grows towards inner blocks, and because appends are being used to define an environment, a name can be redefined in an inner block. Note also that <u>any</u> environment modification affects only the procedures applied in that block and the dynamic descendants of those procedures. That is, η only goes into an expression as an argument, it never comes out as a result. The environment is thus a dynamically varying value that reflects the execution structure of a program. For this reason, if a procedure A is written which depends upon a particular procedure B being available via the environment, it may be dangerous to return A from one block back to a higher-level block where A might be applied in quite a different environment. (LISP programmers might view this as a special case of the FUNARG problem.) Like most conveniences, the environment facility provided by Id depends upon assumptions, here the assumption is that the context of a procedure's application can be guaranteed by the programmer. In general, only library-type procedures or tightly controlled self-sufficient blocks of code that define their own procedures should be depended upon to properly maintain context. As an example of the kind of problem that can arise

```
proc u(x)(...v(x)...);
proc v(x)(...u(x)...);
```

are two mutually recursive procedures and both depend upon each being in the environment of the other. However, if these procedures are passed out of the environment in which they are defined, they will <u>not</u> function properly. Two ways out of the problem are possible:

1. Return the two procedures encompassed within another procedure, which when applied causes them to redefine themselves and the environment in which they need to exist.
2. Have the procedures explicitly pass themselves as parameters to one another, just as unnamed procedures would have to do.

References

[Arvind & Gostelow 77a] 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, 1978.

[Arvind & Gostelow 77b] 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.

[AGP 77] Arvind, Gostelow, K.P., and Plouffe, W.E. Indeterminacy, monitors, and dataflow. Proc. Sixth ACM Symp. on Operating Systems Principles, Nov. 1977, pp. 159-169.

[Ashcroft & Wadge 76] 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.

[Backus 73] Backus, J. Programming language semantics and closed applicative languages. Proc. ACM Symp. on Principles of Programming Languages, Oct. 1973, pp. 71-86.

[Bic 77a] Bic, L. An extended model for protection in dataflow. Dataflow Note #23, Dept. of Information and Computer Science, Univ. of California, Irvine, CA, Nov. 1977.

[Bic 77b] Bic, L. Protection in dataflow. Dataflow Note #25, Dept. of Information and Computer Science, Univ. of California, Irvine, CA, Nov. 1977.

[Bic 78] Bic, L. Security and protection in a dataflow computer system. Dataflow Note #34, Dept. of Information and Computer Science, Univ. of California, Irvine, CA, April 1978.

[Brinch-Hansen 72] Brinch Hansen, P. Structured multiprogramming. Comm. ACM 15, 7 (july 1972), 574-578.

[CHP 71] Courtois, P.J., Heymans, F., and Parnas, D.L. Concurrent control with "readers" and "writers". Comm. ACM 14, 10 (Oct. 1971), 667-668.

[Dennis 73] Dennis, J.B. First version of a data flow procedure language. Computation Structures Group Memo 93, Lab. for Computer Science, Cambridge, MA, Nov. 1973. (revised as MAC Technical Memorandum 61, May 1975).

[Dennis 74] Dennis, J.B. On storage management for advanced programming languages. Computation Structures Group Memo 109-1, Lab. for Computer Science, Cambridge, MA, Oct. 1974.

[Dijkstra 76] Dijkstra, E.W. A discipline of programming. Prentice-Hall, Englewood Cliffs, N.J., 1976. (Chapt. 17).

[Friedman & Wise 76a] 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 & Wise 76b] 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.

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

[Guttag 77a] Guttag, J. Abstract data types and the development of data structures. Comm. ACM 20, 6 (June 1977), 396-404.

[Guttag 77b] Guttag, J. Personal Communication.

[Hoare 74] Hoare, C.A.R. Monitors: An operating system structuring concept. Comm. ACM 17, 10 (Oct. 1974), 549-557.

[IPL] See [Newell & Tonge 60].

[Jones] See [Jones & Liskov 76].

[Jones & Liskov 76] Jones, A.V., and Liskov, B.H. An access control facility for programming languages. Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, PA, May 1976.

[Keller 77] Keller, R.M. Denotational models for parallel programs with indeterminate operators. In Formal Description of Programming Languages, E.J. Neuhold, Ed., North-Holland, New York, 1978.

[Kosinski 73] Kosinski, P.R. A data flow language for operating systems programming. SIGPLAN Notices (ACM) 8, 9 (Sept. 1973), 89-94.

[Kosinski 78] 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.

[Landin 65] Landin, P.J. A correspondence between Algol 60 and Church's lambda-notation: Part I. Comm. ACM 8, 2 (Feb. 1965), 98-101.

[Liskov] See [LSAS 77].

[LSAS 77] Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. Abstraction mechanisms in CLU. Comm. ACM 20, 8 (Aug. 1977), 564-576.

[McCarthy 60] McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, Part I. Comm. ACM 3, 4 (April 1960), 184-195.

[Newell & Tonge 60] Newell, A., and Tonge, F.M. An introduction to Information Programming Language V. Comm. ACM 3, 4 (April 1960), 205-211.

[Patil 70] 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.

[Shaw] See [SWL 77].

[SWL 77] 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.

[Sutherland 77] See [Sutherland & Mead 77].

[Sutherland & Mead 77] Sutherland, I.E., and Mead, C.A. Microelectronics and computer science. Scientific American 237, 3 (Sept. 1977), 210-228.

[Weng 75] Weng, K.S. Stream-oriented computation in recursive data flow schemes. MAC Technical Memorandum 68, Lab. for Computer Science, Cambridge, MA, Oct. 1975.